Analyse der Microservices eines digitalen Marktplatzes mittels ExplorViz

Masterarbeit

Felix Eichhorst

16. Oktober 2017

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Zirkelbach
M.Sc. Christian Wulf

Eidesstattliche Erklärung

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

Kiel, 16. Oktober 2017

_____

Zusammenfassung

Die moderne Welt befindet sich im stetigen Wandel. Moderne Software-Architekturen sind leistungsfähiger denn je und können komplexe Aufgaben bewältigen, wie sie vor einigen Jahren noch nicht lösbar waren. Gleichzeitig sorgt dieser Anstieg an Komplexität aber auch dafür, dass das Verständnis von Software selbst immer mehr schwindet. Herkömmliche Analyse-Methoden für Software genügen meist nicht, um sich ein Bild vom Zustand von Software zur Laufzeit zu machen.

Diesem Problem versucht das Software-Visualisierungs-Werkzeug `ExplorViz` entgegenzuwirken. Mithilfe eines Software-Monitorings, etwa durch das Monitoring-Framework `Kieker`, stellt es den Zustand von Software grafisch dar. Dies erlaubt es Entwicklerinnen und Entwicklern, Software auf eine völlig neue Weise zu betrachten und Zusammenhänge zu erkennen, die ohne Visualisierung nicht erkennbar wären.

Um die Softwareentwicklung mit dieser Visualisierung auch zukünftig unterstützen zu können, ist es wichtig, dass Tools wie `ExplorViz` oder `Kieker` sich an die schnelllebige Software-Industrie anpassen. Wo früher hauptsächlich monolithische Anwendungen vorzufinden waren entstehen heute immer öfter verteilte Architekturen, in denen Anwendungen verschiedenster Herkunft und Technologie miteinander kommunizieren. Diese Art der Kommunikation geht über die Grenzen einer einzelnen Technologie hinaus und kann daher nicht wie Kommunikation innerhalb einer Anwendung behandelt werden.

In dieser Arbeit visualisieren wir deshalb eine moderne Microservice-Architektur mit `Kieker` und `ExplorViz` am Beispiel des Online-Marktplatzes für Dienste der Informations- und Kommunikationstechnik `Symphony`. Dabei zeigen wir einerseits, welche Anpassungen für das Überwachen und Darstellen von verteilter Kommunikation nötig sind, und evaluieren andererseits, welchen Gewinn eine Visualisierung von verteilten Architekturen für die Entwicklerinnen und Entwickler bringen kann.

Abstract

The modern world is in constant change. There is hardly an area in our everyday lives that has not experienced the advent of digitalization. Modern software architectures are more powerful than ever before and can cope with complex tasks that were not possible a few years ago. At the same time, this increase in complexity also means that the understanding of software itself is dwindling. Conventional analysis methods for software are usually not sufficient to get an idea of the state of software at runtime.

The software visualization tool `ExplorViz` tries to counteract this problem. With the help of a software monitoring, for example through the monitoring framework `Kieker`, it represents the state of software graphically. This allows developers to look at software in a completely new way and to recognize interrelationships that would not have been visible without visualization.

In order to be able to support software development with this visualization in the future, it is important that tools such as `ExplorViz` or `Kieker` adapt to the fast-moving software industry. Where in the past mainly monolithic applications were found, today more and more distributed architectures are being developed in which applications of different origins and technologies communicate with each other. This type of communication goes beyond the boundaries of a single technology and therefore cannot be treated as communication within an application.

In this work we therefore visualize a modern microservice architecture with `Kieker` and `ExplorViz` using an example of the online market place for services of information and communication technology `Symphony`. On the one hand, we show which adaptations are necessary for monitoring and displaying distributed communication, and on the other hand, we evaluate what benefits a visualization of distributed architectures can bring for the developers.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziele . . . 2
1.3 Aufbau . . . 3

2 Grundlagen und Technologien **5**
2.1 Kieker . . . 5
2.2 ExplorViz . . . 6
2.3 Symphony . . . 8
2.4 Microservices . . . 9
2.5 Spring und Spring Boot . . . 10
2.5.1 Spring . . . 10
2.5.2 Spring Boot . . . 10
2.6 REST . . . 11
2.7 Java-Agenten . . . 13
2.8 Aspektorientierte Programmierung . . . 14
2.9 AspectJ . . . 14
2.10 Docker . . . 16
2.11 Groovy . . . 16
2.12 Gradle . . . 17

3 Analyse **19**
3.1 Aufbau von Symphony . . . 19
3.1.1 Übersicht über die Microservices . . . 19
3.1.2 Abhängigkeiten zwischen den Microservices . . . 21
3.1.3 Aufbau der einzelnen Microservices . . . 22
3.2 Monitoring mit ExplorViz . . . 26
3.2.1 Verwendung des Monitorings . . . 26
3.2.2 Konfiguration des Monitorings . . . 26
3.2.3 Records und Traces . . . 27
3.2.4 Nachteile des Monitorings mit ExplorViz . . . 27
3.3 Monitoring mit Kieker . . . 27
3.3.1 Verwendung des Monitorings . . . 28
3.3.2 Konfiguration des Monitorings . . . 28
3.3.3 Records und Traces . . . 28
3.3.4 Herausforderung . . . 28

3.4 Übertragung der Daten an ExplorViz . . . 29
3.4.1 Herausforderung . . . 29
3.5 Analyse und Visualisierung mit ExplorViz . . . 29
3.5.1 Verwenden des ExplorViz-Servers . . . 29
3.5.2 Verarbeiten von Traces . . . 30

4 Entwurf **33**
4.1 Monitoring mit Kieker . . . 33
4.1.1 Erfassen von verteilter Kommunikation . . . 33
4.1.2 Instrumentieren der Microservices . . . 34
4.2 Übertragen der Daten an ExplorViz . . . 40
4.2.1 Ansatz 1: Offline-Übertragung . . . 41
4.2.2 Ansatz 2: Online-Übertragung . . . 44

5 Implementierung **47**
5.1 Monitoring mit Kieker . . . 47
5.1.1 Erfassen von verteilter Kommunikation . . . 47
5.1.2 Instrumentieren der Microservices . . . 47
5.2 Übertragen der Daten an ExplorViz . . . 52
5.2.1 Ansatz 1: Offline-Visualisierung . . . 52
5.2.2 Ansatz 2: Online-Visualisierung . . . 55
5.3 Ergebnis der Implementierung . . . 56

6 Evaluation **59**
6.1 Ziele . . . 59
6.1.1 Forschungsfragen und Hypothesen . . . 59
6.1.2 Variablen . . . 60
6.2 Methodik . . . 60
6.2.1 Wahl der Methode . . . 60
6.2.2 Wahl der Probanden . . . 61
6.2.3 Einteilung der Probandinnen und Probanden . . . 62
6.2.4 Wahl des Szenarios . . . 62
6.3 Durchführung der Befragung . . . 63
6.3.1 Fragebogen . . . 63
6.3.2 Versuchsaufbau . . . 67
6.3.3 Beschreibung der Probandinnen und Probanden . . . 71
6.3.4 Durchführung des Versuchs . . . 72
6.3.5 Beschreibung der Versuchsergebnisse . . . 72
6.3.6 Diskussion der Versuchsergebnisse . . . 77
6.3.7 Validität der Versuchsergebnisse . . . 84
6.3.8 Zusammenfassung . . . 86

7 Verwandte Arbeiten **89**
7.1 Ansätze in Kieker 89
7.1.1 Spring-Interzeptoren 89
7.1.2 CXF-SOAP-Interzeptoren 89
7.2 Monitoring von verteilten Traces mit Kieker 90
7.3 Google Dapper und OpenZipkin 90
7.4 nuGEN 91

8 Fazit and Ausblick **93**
8.1 Fazit 93
8.2 Ausblick 94

A Anlage zur Befragung **97**

B Ergebnisse der Befragung **103**

C POM-Dateien für Maven **107**
C.1 POM für die Kieker-Monitoring-Komponente 107
C.2 POM für die ExplorViz-Monitoring-Komponente 108

Bibliografie **111**

Fremdwörterverzeichnis

Analysis Englisch für Analyse. In dieser Arbeit häufig im Zusammenhang mit der Analyse-Komponente von `Kieker` oder `ExplorViz`

Application Programming Interface Programmierschnittstelle für den Zugriff auf eine Anwendung von außen.

Controller Steuernde Einheit eines Systems. In der Software meist eine Klasse, die die Kontrolle über andere Klassen hat.

Extensible Markup Language Eine erweiterbare Beschreibungssprache, die häufig für das Festhalten und Versenden von Daten verwendet wird.

Framework Gerüst oder Rahmen, der bei der Erstellung einer Anwendung unterstützt. Bringt vorgefertigte Bestandteile mit, muss üblicherweise aber angepasst oder erweitert werden.

Interface Ein in der Programmierung genutztes Konstrukt, welches das Skelett einer Klasse darstellt, die das Interface implementiert. Dadurch kann sichergestellt werden, dass die Klasse bestimmte Methoden verwendet, die das Interface vorgibt.

Integrated Development Environment Integrierte Entwicklungsumgebung. Ein Werkzeug, das durch Automatisierung die Softwareentwicklung erleichtert.

Java Database Connectivity Verbindung von Java zu einer Datenbank.

Java Virtual Machine Die virtuelle Laufzeitumgebung von Java. In ihr werden Java-Anwendungen ausgeführt.

Log Ein Protokoll über eingetretene Ereignisse. Logs werden üblicherweise im Dateisystem oder einer Datenbank abgelegt, um im Nachhinein einsehbar zu sein.

Microservice Ein kleiner unabhängiger Dienst mit klar definierter Aufgabe.

Middleware Eine vermittelnde Plattform zwischen mehreren Anwendungen, die die Kommunikation verarbeitet. Beispielsweise kann eine Middleware das Format von Nachrichten verschiedener Dienste anpassen.

Monitoring Überwachen und Protokollieren des Verhaltens einer Handlung oder eines Systems. In dieser Arbeit häufig im Zusammenhang mit der Monitoring-Komponente von `ExplorViz`.

Object-Relational Mapping Abbilden von Objekten auf relationale Strukturen wie Tabellen.

Overhead Zusätzlich anfallende Zeit oder Daten, die benötigt werden, um ein Ziel zu erreichen. Beispielsweise muss für die Übertragung von Daten die Adresse des Empfängers als Overhead angehängt werden, damit diese zugestellt werden kann.

Probe Eine Sonde zum Sammeln von Daten.

Reader Lesende Einheit, hier üblicherweise aus einem Log oder einem Stream.

Remote Procedure Call Aufruf einer Methode aus der Ferne.

Simple Logging Facade for Java Wrapper für verschiedene Logging-Frameworks, welcher es einem Nutzer erlaubt eine gewünschte Implementierung zu wählen.

Stream Ein kontinuierlicher Datenstrom, der üblicherweise zur Kommunikation verwendet wird.

Tool Dienstprogramm, welches eine bestimmte Aufgabe ausführt.

Trace Ablaufprotokoll eines Programmes.

Transmission Control Protocol Ein Protokoll zur Überwachung von Nachrichtenübertragung.

Uniform Resource Identifies Ein einheitlicher Identifikator für Ressourcen.

Virtual Reality Computergenerierte Umgebung, die vorgibt die Realität zu sein.

Writer Schreibende Einheit, hier üblicherweise in ein Log oder einen Stream.

Abkürzungsverzeichnis

JVM Java Virtual Machine

XML Extensible Markup Language

RPC Remote Procedure Call

JDBC Java Database Connectivity

ORM Object-Relational Mapping

URI Uniform Resource Identifier

IKT Informations- und Kommunikationstechnologie

IDE Integrated Development Environment

API Application Programming Interface

GWT Google Web Toolkit

TCP Transmission Control Protocol

SLF4J Simple Logging Facade for Java

Kapitel 1

Einleitung

1.1. Motivation

> „I see no limit to the capabilities of machines. As microchips get smaller and faster, I can see them getting better than we are. I can visualize a time in the future when we will be to robots as dogs are to humans."
>
> — Claude Elwood Shannon, 1987

Auch wenn sich Shannons Vision bisher noch nicht bewahrheitet hat, beobachten wir im Jahr 2017 einen Einzug von Software in alltägliche Situationen mit nie da gewesener Stärke. Es gibt wohl kaum noch einen Bereich in unserem Leben, der nicht von Software gesteuert und bestimmt wird [Dodge und Kitchin 2004]. Nicht nur ist Software maßgeblich verantwortlich für Unterhaltung und Komfort wie beispielsweise in *Consumer Electronics*, längst ist sie auch essentieller Bestandteil der Grundlagen des modernen Lebens. Die Versorgung der Bevölkerung mit Lebensmitteln und Medikamenten, die Organisation von Rettungsdiensten, die Belieferung mit Treib- und Brennstoffen; sie alle sind abhängig von funktionierenden Software-Systemen.

Und doch besitzt Software für den Menschen keine greifbare Gestalt. Wir können üblicherweise nur das Ergebnis von laufender Software sehen, nicht aber die Software selbst im Betrieb. Diese Tatsache führt in der modernen Software-Entwicklung zunehmend zu Problemen, da sich durch die steigende Komplexität und Vernetzung von Systemen Fehler immer schwerer im Vorhinein durch Analyse oder Tests vorhersagen lassen und sich das tatsächliche Verhalten erst im laufenden Betrieb zeigt. Laut einer Veröffentlichung der Firma *Omnext* zum Thema Wartungskosten bei Software ist der Anteil der Wartungs- an den Gesamtkosten von 1970 bis 2014 kontinuierlich angestiegen [de Vries u. a. 2014]. Gleichzeitig bestände heute etwa die Hälfte dieser Wartungskosten aus Ausgaben für die Analyse von Software, während das Testen mit nur einem Viertel und das Design, die Implementierung und die Dokumentation gemeinsam ebenfalls mit einem Viertel auskämen. Aus dem Anstieg der Wartungskosten lässt sich ein Trend hin zu schwindendem Verständnis von Software-Systemen erkennen.

Die Arbeitsgruppe für Software Engineering um Prof. Dr. Wilhelm Hasselbring am Institut für Informatik der Christian-Albrechts-Universität zu Kiel beschäftigt sich mit

der Qualität und der Evaluierung von Software-Systemen und versucht den genannten Problemen entgegenzuwirken. Das unter der Leitung von Hasselbring entwickelte Software-Analyse-Framework `Kieker` bietet die Möglichkeit laufende Programme so zu instrumentieren, dass ihr Verhalten überwacht, analysiert und auf verschiedene Weisen ausgewertet werden kann [Van Hoorn u. a. 2012]. Daneben befindet sich seit einigen Jahren das Software-Visualisierungs-Tools `ExplorViz` in Entwicklung, welches in Verbindung mit den durch `Kieker` gesammelten Daten in der Lage ist, die Gestalt eines laufenden Programmes visuell darzustellen [Fittkau u. a. 2013]. Hierfür werden eine zweidimensionale Ansicht der Softwarelandschaft und eine dreidimensionale Ansicht der darin enthaltenen Komponenten bereitgestellt. Ein Nutzer soll damit in der Lage sein, frei in den generierten Modellen zu navigieren und das Verhalten eines Systems zu verschiedenen Zeitpunkten seiner Laufzeit verstehen zu können.

Wo vor einigen Jahren noch hauptsächlich mit monolithischen Anwendungen gearbeitet wurde, entstehen heute mehr und mehr dynamische und verteilte Systeme [Baresi u. a. 2006]. Moderne Plattformen wie der Marktplatz für Dienste der Informations- und Kommunikationstechnologie (IKT) `Symphony` setzen auf kleine eigenständige Dienste, sogenannte Microservices. Dies soll Austauschbarkeit und die Möglichkeit eines schnellen Technologiewechsels garantieren, bringt Ausfallsicherheit und erlaubt durch eine hohe Modularität eine verteilte und unabhängige Entwicklung. Dadurch ändern sich allerdings auch die Anforderungen an ein Software-Monitoring- und -Visualisierungs-Tool wie `ExplorViz`. Statt sich auf eine einzelne Technologie zu fokussieren, muss dieses zukünftig vermehrt mit mehreren Sprachen und vor allem mit verteilter Kommunikation arbeiten. Eine weitere Herausforderung ist die immer häufigere Verwendung von Frameworks zur Entwicklung von Anwendungen. Im `Symphony`-Projekt ist jeder Microservice eine eigenständige `Spring-Boot`-Applikation, wodurch eine schnelle und einfache Erweiterung der Funktionalität garantiert werden soll. Auch komplett neue Microservices können durch den *Model-View-Controller*-Ansatz problemlos hinzugefügt werden. Ein Framework wie Spring bringt allerdings einen nicht vernachlässigbaren Teil an Eigenfunktionalität mit, was je nach Komplexität den Kontrollfluss der zu überwachenden Anwendung erhöhen kann.

1.2. Ziele

Ziel dieser Arbeit ist es, die Softwarelandschaft hinter `Symphony` mithilfe von `Kieker` und `ExplorViz` zu visualisieren und zu ermitteln, inwieweit diese Visualisierung Entwicklerinnen und Entwickler beim Verständnis der Plattform unterstützen kann.

Der erste Schritt besteht für uns deshalb darin, die technischen Gegebenheiten von `Kieker` und `ExplorViz` zu prüfen und zu ermitteln, welche Möglichkeiten beide Tools für die Verarbeitung von verteilter Kommunikation mitbringen. Gleichzeitig müssen wir analysieren, welche Besonderheiten die Verwendung von `Spring-Boot` für das Instrumentieren der Microservices bringt. Im zweiten Schritt wollen wir dann mithilfe des erlangten Wissens einen Entwurf erstellen, der das Verhalten der Microservices von `Symphony` samt

übergreifender Kommunikation überwacht, die Daten an ExplorViz sendet und sie dort dann grafisch darstellt. Anschließend werden wir den Entwurf im dritten Schritt implementieren und eine lauffähige Visualisierung der Microservices von Symphony erhalten. Für die Evaluation der Nützlichkeit einer Visualisierung von Symphony durch ExplorViz besteht der vierte Schritt darin, mithilfe eines geeigneten Fragebogens die Analysemöglichkeiten in ExplorViz mit denen in einem klassischen Integrated Development Environment (IDE) zu vergleichen.

1.3. Aufbau

Um den folgenden Kapiteln folgen zu können, führt uns Kapitel 2 im Anschluss an dieses Kapitel in die wichtigsten Grundlagen und Technologien dieser Arbeit ein.

Kapitel 3 beschreibt die Analyse der technischen Gegebenheiten der verwendeten Projekte und damit den ersten Schritt zu unserem Ziel. Dies umfasst einerseits die Analyse von Symphony selbst, andererseits, wie schon erläutert, die Analyse der Tools Kieker und ExplorViz. Für den zweiten und dritten Schritt behandeln Kapitel 4 den Entwurf der Visualisierung und Kapitel 5 dessen Implementierung.

Kapitel 6 befasst sich im Rahmen der Evaluation mit dem Erstellen des Fragebogens und der Durchführung der Entwickler-Befragung. Es analysiert außerdem die Studien-Ergebnisse und präsentiert die daraus gewonnen Erkenntnisse.

In Kapitel 7 werden einige verwandte Arbeiten vorgestellt und ein Teil des wissenschaftlichen Hintergrunds dieses Themas aufgezeigt. Abschließend fasst Kapitel 8 die Ergebnisse dieser Arbeit zusammen und erklärt Schlussfolgerungen sowie mögliche zukünftige Arbeiten in diesem Bereich.

Kapitel 2

Grundlagen und Technologien

2.1. Kieker

`Kieker` ist ein `Java`-basiertes, modulares Software-Monitoring- und -Analyse-Framework, das unter der Leitung von Prof. Dr. Wilhelm Hasselbring entstanden ist [Van Hoorn u. a. 2012] und bis heute aktiv weiterentwickelt wird. Es zeichnet sich durch eine hohe Flexibilität und Erweiterbarkeit aus und erlaubt es so, eine Vielzahl an Sprachen und Architekturen zu überwachen und nahezu beliebige Analysen durchzuführen.

Für eine Idee der Funktionsweise gibt Abbildung 2.1 einen Überblick über die Struktur des Frameworks. Alle Teile der Software lassen sich in zwei Bereiche aufteilen; die links dargestellte Monitoring-Komponente und die rechts abgebildete Analyse-Komponente. Das Monitoring ist dafür zuständig, Daten aus einer zu überwachenden Anwendung zu sammeln und für die Analyse vorzubereiten. In der Analyse können dann Plugins dafür verwendet werden, um die Monitoring-Daten zu verarbeiten.

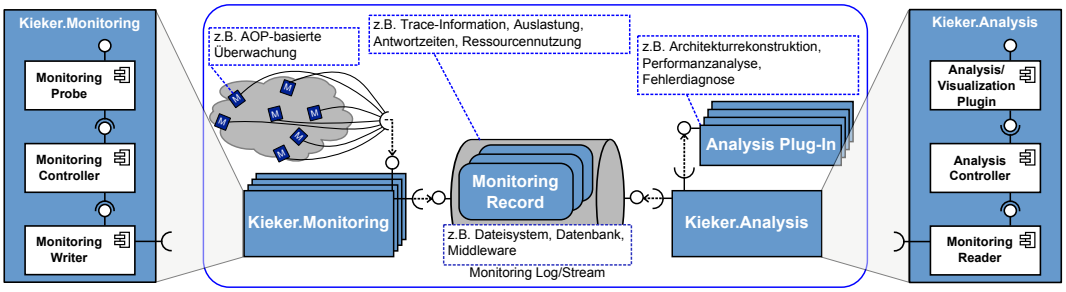


Abbildung 2.1. Überblick über die Funktionsweise des `Kieker`-Frameworks[1]

Betrachten wir die Monitoring-Komponente auf der linken Seite genauer, so erkennen wir, dass diese aus drei Subkomponenten besteht. An erster Stelle stehen die Monitoring-Probes. Eine Probe nutzt eine beliebige Methode, um Daten aus einer laufenden Anwendung abzugreifen. Als Beispiel wird in Abbildung 2.1 das aspektorientierte Abhören von Methoden-Aufrufen genannt, es sind aber je nach Technologie beliebige Möglichkeiten vorstellbar. Jede Probe schickt beim Eintreten eines zu erwartenden Ereignisses

[1] Eingedeutscht, von `http://kieker-monitoring.net/`, zuletzt besucht am 16.10.2017

eine sogenannten Record an den Monitoring-Controller. Ein Record kann, abhängig von seinem Typ, beispielsweise Informationen das Ereignis selbst oder die Systemdaten wie die Speicherauslastung enthalten. Der Monitoring-Controller hat die Aufgabe, Probes mit einem Monitoring-Writer zu verbinden, indem er diese an letzteren weiterreicht. Der Monitoring-Writer schreibt die Records im Anschluss entweder in ein Log oder in einen Stream.

Analog zum Monitoring besteht auch die Analyse-Komponente aus drei Subkomponenten. Der Monitoring-Reader liest Records entweder aus einer Log-Datei oder direkt aus einem Stream ein und gibt jedes an den Analyse-Controller weiter. Der Analyse-Controller kennt alle registrierten Analyse-Plugins und schickt jedem jeden eintreffenden Record. Damit sind dann beliebige Analysen wie etwa eine Rekonstruktion der Architektur oder eine Fehlerdiagnose möglich.

Die Verwendung von Logs und Streams durch unterschiedliche Writer und Reader macht sowohl eine Offline- als auch eine Online- oder Live-Analyse möglich. Außerdem erlaubt es das Austauschen der Writer, komplett ohne `Kieker`-Analyse zu arbeiten und die Daten zum Beispiel an einen Webserver oder ein anderes Analyse-Tool weiterzuschicken.

Im Rahmen dieser Arbeit werden wir `Kieker` dazu werden, das Monitoring der Microservices von `Symphony` durchzuführen, um die Daten mit `ExplorViz` zu visualisieren. Weitere Details zur Funktionsweise von `Kieker` werden in den folgenden Kapiteln eingeführt.

2.2. ExplorViz

`ExplorViz` ist ein Software-Visualisierungs-Tool, welches am Lehrstuhl für Software-Engineering der Christian-Albrechts-Univeristät zu Kiel unter der Leitung von Prof. Dr. Wilhelm Hasselbring entwickelt wird [Fittkau u. a. 2017]. Es ist in der Lage, `Java`-Anwendungen automatisch in Echtzeit zu instrumentieren und im Webbrowser zu visualisieren.

`ExplorViz` bietet dem Nutzer eine zweidimensionale Ansicht der Softwarelandschaft und eine dreidimensionale Ansicht der einzelnen Anwendung innerhalb dieser [Fittkau u. a. 2015b]. Abbildung 2.2 zeigt die Demo-Landschaft von `ExplorViz` in der Version 1.0. Die grauen Boxen repräsentieren einzelne Systeme. Jedes System kann aus einzelnen Knoten oder Knotengruppen bestehen, die hellgrün beziehungsweise dunkelgrün eingefärbt sind. Die lila gefüllten Kästen stellen die Applikationen der Landschaft dar. Wenn zwischen Anwendungen Kommunikation stattfindet, wird diese als orangene Linie dargestellt. Je mehr Austausch pro Zeiteinheit stattfindet, desto dicker sind die entsprechenden Linien. Zur Übersichtlichkeit können sowohl Systeme als auch Knoten und Knotengruppen durch einen Doppelklick geöffnet und geschlossen werden.

Die sogenannte Stadtansicht einzelner Applikationen besteht ebenfalls aus verschiedenen Boxen, die Pakete darstellen. Auch hier lassen sich Pakete durch einen Doppelklick öffnen oder schließen. Abbildung 2.3 zeigt eine teilweise geöffnete Ansicht einer exemplarisch visualisierten Anwendung. Neben Paketen finden wir Klassen, dargestellt als lila Blöcke. Je höher die Box einer Klasse erscheint, desto mehr Instanzen von ihr existieren.

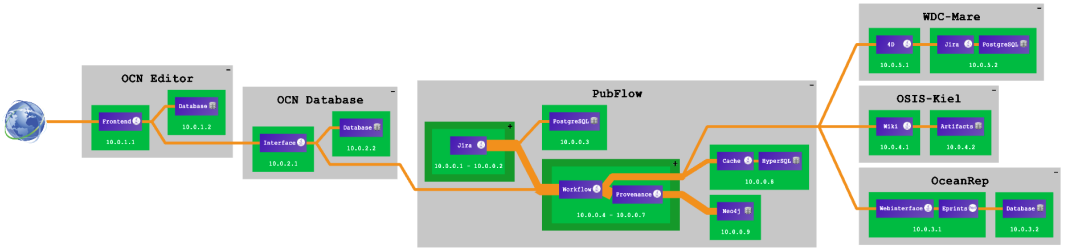


Abbildung 2.2. Zweidimensionale Ansicht der Demo-Landschaft von ExplorViz

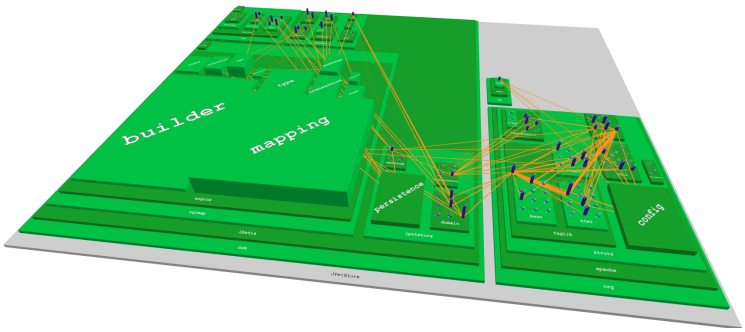


Abbildung 2.3. Dreidimensionale Ansicht einer exemplarisch visualisierten Anwendung in ExplorViz

Kommunikation ist analog zur Landschaftsansicht durch orangene Linien dargestellt.

Beide Darstellungen erlauben dem Benutzer eine genaue Analyse der Anwendungen zu einem beliebigen Zeitpunkt. Einerseits können die Modelle frei vergrößert und verkleinert, gedreht und verschoben werden, andererseits bietet eine Zeitleiste die Möglichkeit zu beliebigen Zeitpunkten der Visualisierung zu springen und die automatische Aktualisierung zu pausieren. Ergänzend stellt ExplorViz einige Analysewerkzeuge bereit, um Datenbankzugriffe zu überwachen und Performanz zu messen. Zusätzlich zur Betrachtung von Software im Browser enthält ExplorViz einen Virtual-Reality-Modus [Fittkau u. a. 2015a] und die Möglichkeit, ein Modell für den 3D-Druck zu exportieren [Fittkau u. a. 2015c].

Wie Kieker besteht auch ExplorViz aus einer Monitoring- und einer Analyse-Komponente. Zusätzlich existiert eine Visualisierungs-Komponente, die die Daten im Browser darstellt. Abbildung 2.4 zeigt Fittkaus Visualisierung von Applikationen in fünf Aktivitäten [Fittkau u. a. 2013]. Aktivität *A1* beschreibt das tatsächliche Monitoring der Anwendungen. Durch die Verwendung von Probes wird mit AspectJ erstellt es Daten über das Verhalten einer Anwendung in Echtzeit. Die Daten werden dann an die Analyse geschickt, welche wir mit dem Aktivitäten *A2* und *A3* zusammenfassen können. Darin werden die Monitoring-Daten zu vollständigen Traces zusammengefügt und Modelle daraus erstellt. Die Aktivitäten *A4* und *A5* ordnen wir der Visualisierung zu. Sie übertragen die Daten an den Browser und zeigen diese dort in den unterschiedlichen Perspektiven an. Die

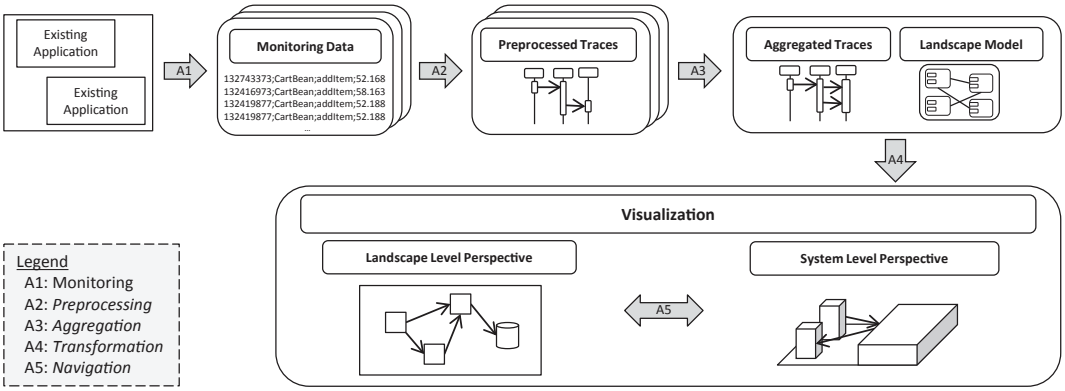


Abbildung 2.4. Aktivitäten der *live-trace*-Visualisierung großer Softwarelandschaften im `ExplorViz`-Ansatz von Fittkau und anderen [Fittkau u. a. 2013].

folgenden Kapitel werden weitere Details zur Funktionsweise von `ExplorViz` aufzeigen.

Obwohl `ExplorViz` sich bisher durch seinen geringen Overhead ausgezeichnet hat [Waller u. a. 2014], planen die Entwickler von `ExplorViz` das eigene Monitoring durch das Monitoring von `Kieker` abzulösen und sich nur auf die Visualisierung selbst zu fokussieren. Dieser Schritt erlaubt `ExplorViz` in Zukunft auch andere Technologien und Sprachen zu unterstützen und erspart eine parallele Entwicklung.

Wir werden `ExplorViz` in Verbindung mit `Kieker` im Rahmen dieser Arbeit verwenden, um die Machbarkeit einer Visualisierung der Microservices von `Symphony` zu zeigen und deren Nutzen zu evaluieren.

2.3. Symphony

`Symphony`[2] ist ein vom BMWi[3] gefördertes Forschungsprojekt, an dem verschiedene Partner aus Forschung und Industrie beteiligt sind. Ziel des Projekts ist eine Plattform für Informations- und Telekommunikationsdienste, die die Vermarktung, Verwaltung, Wartung und das Störungsmanagement verschiedener Anbieter in einer Schnittstelle vereint. Über diese Plattform sollen kleine und mittelständische Unternehmen in der Lage sein verschiedene Dienste unterschiedlicher Hersteller zu vergleichen und zu buchen.

Die Struktur des Projekts lässt sich in zwei Bereiche teilen, die, unter paluno[4] als forschendem Projektleiter, von IN-telegence[5] und adesso[6] entwickelt werden. IN-telegence setzt die Kombination der verschiedenen technischen Dienste um und sorgt dafür, dass

[2] `http://www.ikt-symphony.de/`, zuletzt besucht am 16.10.2017
[3] Bundesministerium für Wirtschaft und Energie
[4] paluno – The Ruhr Institute for Software Technology, Universität Duisburg-Essen
[5] IN-telegence GmbH, Köln
[6] adesso AG, Dortmund

diese in einer einzigen Datenhaltung erfasst werden können. Adesso übernimmt die Entwicklung der eigentlichen Plattform und des Portals. Weitere Partner werden hier nicht genannt, sind aber auf der Seite des Projekts[7] zu finden.

Für größtmögliche Flexibilität bei der Entwicklung und gute Skalierbarkeit setzt adesso bei der Entwicklung der Plattform auf eine Microservice-Architektur und folgt damit dem Trend hin zu verteilten Systemen. Neben einem *Frontendproxy*, welches die Weboberfläche bereitstellt, sind Microservices für beispielsweise die Authentifizierung, die Produktverwaltung oder für Wartungszwecke vorhanden.

Sowohl durch ihre Nähe zur Forschung als auch durch ihren Aufbau eignet sich die Plattform hinter `Symphony` gut für eine Visualisierung durch `ExplorViz`. An ihrem Beispiel werden wir die Fähigkeit von `Kieker` und `ExplorViz` prüfen, verteilte Microservice-Architekturen zu überwachen und grafisch darzustellen. Eine genauere Analyse der Plattform geschieht in Kapitel 3.

2.4. Microservices

Eine Microservice-Architektur besteht aus einer Menge einzeln laufender und sich austauschender Dienste, sogenannter Microservices. Aufgrund der zunehmenden Komplexität von Systemen und dem schnellen Technologiewechsel auf dem Softwaremarkt ist die Verwendung von Microservices sehr beliebt, wie es auch beispielsweise ein auf LinkedIn Pulse veröffentlichter Blogpost[8] bestätigt. Gegenüber klassischen monolithischen Anwendungen bringt sie erhebliche Vorteile.

Statt nur eine Anwendung auf einem Rechner zu starten, werden einzelne Komponenten des Systems auf isolierte Server verteilt. Die Kommunikation zwischen ihnen verläuft dann üblicherweise über eine HTTP-Schnittstelle, wie etwa REST. Diese Art der Modularität des Gesamtsystems erlaubt eine sonst nicht mögliche Austauschbarkeit und sogar einen Wechsel der kompletten Technologie und Sprache [Namiot und sneps sneppe 2014]. Zudem kann die Entwicklung an einzelnen Komponenten separat erfolgen, solange die Kommunikation zwischen ihnen festgelegt ist.

Weiter bieten Microservices Vorteile für die Performanz des Gesamtsystems, denn es können *Load Balancer* genutzt werden, um nach Belieben neue Instanzen eines Dienstes zu starten, wenn dieser mehr Last erfährt. Microservice-Architekturen sind damit gut skalierbar und durch die Kapselung ist eine hohe Fehlertoleranz gegeben [Hasselbring 2016]. Darüber hinaus gibt es Versuche, bei denen bei hoher Last auf ein Gesamtsystem, bestimmte Microservices abgeschaltet werden, um die Gesamtperformanz zu verbessern [Hasselbring und Steinacker 2017]. Bei einem Marktplatz im Internet könnte beispielsweise die Anzeige ähnlicher Produkte deaktiviert werden, da diese nur geringe Priorität im Gegensatz zur Preisberechnung hat.

[7] `http://www.ikt-symphony.de/#partner`, zuletzt besucht am 16.10.2017

[8] `https://www.linkedin.com/pulse/5-reasons-why-microservices-have-become-so-popular-last-sakhuja`, zuletzt besucht am 17.05.2017

2.5. Spring und Spring Boot

2.5.1. Spring

Spring ist ein quelloffenes Framework zur Entwicklung von Java-Applikationen[9]. Durch seine Leichtgewichtigkeit und Modularität lässt es sich von einem einfachen Application Programming Interface (API) bis hin zur Enterprise-Anwendung für eine große Bandbreite an Projekten nutzen. Eines der Kernprinzipien von Spring ist die sogenannte *Dependency-Injection*, welche Objekten bei Bedarf benötigte andere Objekte automatisch zuordnet, ohne dass der Entwickler einem besonderen Programmschema wie beispielsweise *Factory* oder *Service-Locator* folgen und die Instanzen seiner Klassen selbst verwalten muss [Johnson u. a. 2004]. Dazu stellt Spring unter anderem die *Bean* und die *Autowired* Annotationen bereit. Ein Bean ist ein Objekt, welches durch Spring verwaltet wird. Ist eine Methode *autowired*, dann sorgt Spring dafür, dass die in den Methodenparametern deklarierten Objekte automatisch aus dem bekannten Beans geholt werden.

Daneben stellt Spring einer Reihe von Modulen für verschiedene Anwendungszwecke bereit, welche in unterschiedliche Ebenen eingeordnet werden. Die Daten-Zugriffs-Ebene bietet Module zur Verarbeitung von Daten über beispielsweise Java Database Connectivity (JDBC) oder Object-Relational Mapping (ORM). Die sogenannte Web-Ebene stellt Eigenschaften bereit, die für die Entwicklung von Web-Anwendungen benötigt werden, unter anderem ein *Model-View-Controller*-System. Weitere wichtige Bestandteile sind zum Beispiel das Test-Modul oder die Unterstützung von aspektorientierter Programmierung, um Teile des Programms zu instrumentieren und etwa *Middlewares* zu erstellen.

Das hauptsächlich durch sein Deployment-Tool JRebel bekannt gewordene Softwareunternehmen *ZeroTurnaround* veröffentlichte im Juli 2016 seinen jährlichen Bericht „Java Tools and Technologies Landscape Report“ und gab an, dass Spring MVC das mit Abstand beliebteste Java-Web-Framework unter den befragten Java Entwicklern sei, gefolgt von der Spring-Erweiterung Spring Boot[10].

2.5.2. Spring Boot

Spring Boot ist eine Erweiterung zum Spring-Framework und soll die Entwicklung von Spring-Anwendungen erleichtern. Sie automatisiert die sonst üblichen Schritte des Integrierens verschiedener Abhängigkeiten, des Erstellens einer *WAR* Datei für das Deployment und das Einrichten eines Servers, der die Applikation ausliefern kann, über einen Befehl. Zusätzlich arbeitet Spring Boot mit der an Java angelehnten Sprache Groovy, was neben Vorteilen für die Erweiterung selbst auch die Arbeit für den Entwickler erleichtern kann.

Jeder der Microservices von `Symphony` ist als eigene Spring-Boot-Anwendung aufgebaut, beruht also auf Spring, lässt sich aber auf einfache Weise in einem integrierten Tomcat bereitstellen.

[9] `https://spring.io/`, zuletzt besucht am 16.10.2017

[10] `https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/`, zuletzt besucht am 16.10.2017

2.6. REST

Der sogenannte *Representational State Transfer*, kurz REST genannt, ist ein dem Internet und verteilten Systemen zugrundeliegende Architekturstil. Er definiert den Austausch von unterschiedlichen Ressourcen und abstrahiert von der Implementierung der beteiligten Komponenten. Perry und Wolf unterscheiden drei Arten von Elementen, die zu einer REST-Architektur gehören [Perry und Wolf 1992]:

- Daten (engl. *„data elements“*) werden bei der REST Kommunikation übertragen. Sie umfassen zum einen die Repräsentationen von Ressourcen sowie deren Identifikatoren, zum anderen aber auch Meta- und Kontrolldaten.
- Konnektoren (engl. *„connecting elements“*) bieten eine Schnittstelle für Komponenten und abstrahieren dabei von der hinter ihnen stehenden Implementierung und Technologie. Dies hat den Vorteil, dass bei einem Austausch einer Komponente bei gleichbleibenden Konnektoren keine Änderungen an anderen Komponenten vorgenommen werden müssen. Es existieren verschiedene Konnektortypen wie beispielsweise *Client* zum Senden von Anfragen und Empfangen von Antworten, *Server* zum Warten auf Anfragen und Senden von Antworten oder *Resolver* zum Übersetzen eines Identifikators für eine Ressource in eine Netzwerkadresse.
- Als Komponenten (engl. *„processing elements“*) bezeichnet man die einzelnen Parteien der Architektur, die je nach ihrer Rolle Daten über Konnektoren verfügbar machen oder anfordern. Komponententypen sind zum Beispiel *User-Agent*, der einen *Client*-Konnektor verwendet, um Anfragen zu stellen und Antworten zu erhalten, oder *Origin-Server*, der eine Schnittstelle zum Erreichen von Ressourcen bietet und auf Anfragen Antworten ausliefert.

Ein typisches Beispiel für Webanwendungen ist die Kommunikation zwischen einer *User-Agent*- und einer *Origin-Server*-Komponente, wie sie in Abbildung 2.5 abgebildet ist. Der *User-Agent* ist in diesem Fall ein Webbrowser und der *Origin-Server* ein Webserver. Beide Komponenten nutzen ihre eigene Implementierung für die geforderte Funktionalität und kommunizieren über ihre Schnittstellen miteinander. Die unterschiedlichen Uniform Resource Identifiers (URIs) bieten Zugang zu Ressourcen:

```
GET  /person      # Holt alle Personen
GET  /person/id   # Holt die Person mit der ID id
POST /person      # Erstellt eine neue Person
PUT  /person/id   # Ersetzt die Person mit der ID id
```

Diese Art der Schnittstelle nennt man auch *RESTful*. Der Browser kann nun HTTP Anfragen stellen und dabei Repräsentationen von Ressourcen mitsenden oder empfangen. Die Repräsentation der Ressourcen, deren Metadaten und Checksummen sind die Datenelemente dieser Architektur.

Da REST Schnittstellen der allgemeine Standard bei der Kommunikation im Internet ist, verwenden auch die Microservices von `Symphony` REST um sich untereinander auszutauschen. Im Rahmen dieser Arbeit muss eine Möglichkeit geschaffen werden, um die Kontrollflüsse der einzelnen Komponenten über die REST-Aufrufe zu verbinden.

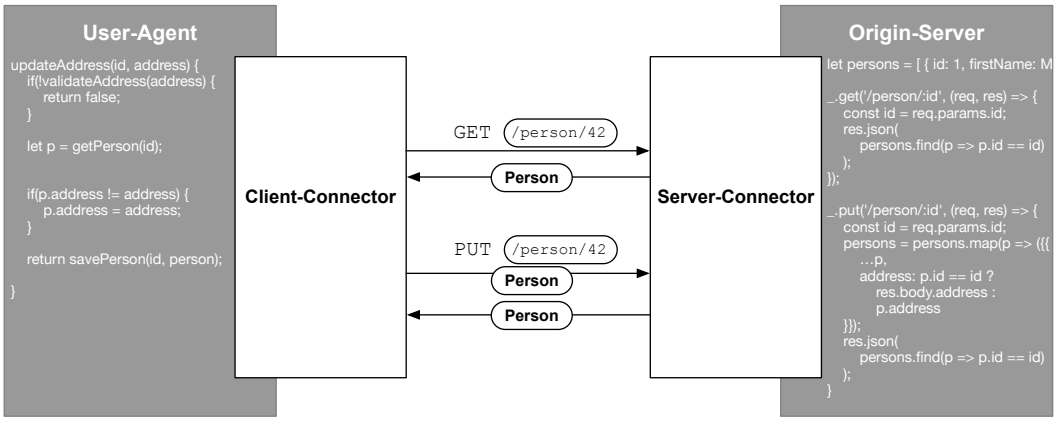


Abbildung 2.5. Beispiel von Kommunikation über REST zwischen einem *User-Agent* und einem *Origin-Server*.

2.7. Java-Agenten

Ein Java-Agent ist ein Dienst, der in der Lage ist, eine andere Java-Anwendung zur Laufzeit durch Manipulation des Byte-Codes zu instrumentieren[11]. Dieser Dienst befindet sich in einer eigenen *JAR*–Datei, welche der Java Virtual Machine (JVM) beim Starten einer Anwendung über den Parameter `-javaagent:pfad-zur-agent.jar` mitgegeben wird.

Abbildung 2.6 gibt einen groben Überblick über die Verwendung eines Java-Agenten. Die *MANIFEST* Datei in der *JAR* des Agenten gibt die Hauptklasse des Agenten mit einer `premain` Methode an, welche vor der eigentlichen `main` Methode der zu instrumentierenden Applikation ausgeführt wird. In dieser `premain` Methode können sogenannte `Transformer` angelegt werden, welche bestimmte Methoden der Applikation verändern. Wird dann eine Klasse vom Class-Loader der JVM geladen, wird diese vom Java-Agenten angepasst.

Java-Agenten werden unter anderem von AspectJ und von `Kieker` oder ExlporViz genutzt, um Java Anwendungen zu überwachen.

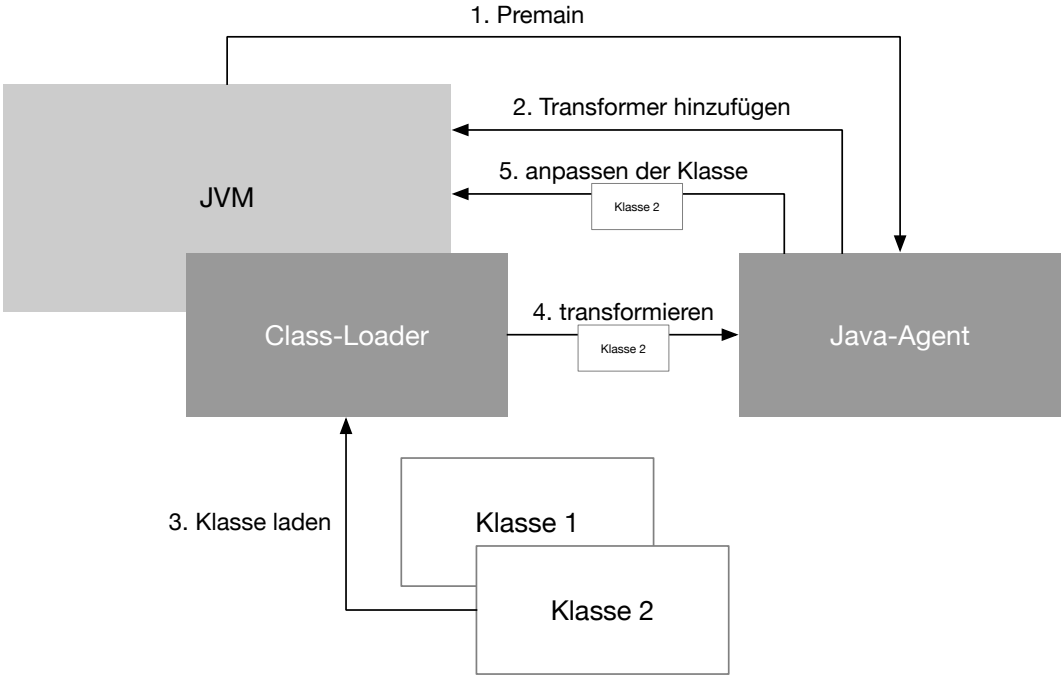


Abbildung 2.6. Grober Überblick über die Verwendung eines Java-Agenten, nach Oleg Šelajev[12].

[11] `https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html`, zuletzt besucht am 16.10.2017

[12] `http://www.barcelonajug.org/2015/04/java-agents.html`, zuletzt besucht am 25.09.2017

2.8. Aspektorientierte Programmierung

In der Software-Entwicklung wird üblicherweise versucht dem Paradigma zu folgen, Programme möglichst modular zu gestalten und eine gute Teilung der Belange (engl. *separation of concerns*) zu erreichen [Påhlsson 2002]. Im Kontrast dazu existieren aber auch Anwendungsfälle, in denen eine Anforderung übergreifend über das gesamte Programm verteilt realisiert werden muss [Kiczales u. a. 2001]. Ein Beispiel hierfür ist das Protokollieren von Aktionen, die während der Laufzeit ausgeführt werden. In der klassischen Programmierung muss jede Komponente, die einen Log-Eintrag erstellen soll, zu diesem Zweck angepasst werden. Dies bringt aber eine Vernetzung und Abhängigkeit zwischen den einzelnen Objekten, die der Modularität eines Programms entgegen wirkt. Eine kleine Änderung an der Protokollkomponente kann zur Folge haben, dass alle protokollschreibenden Objekte verändert werden müssen. Diese Art der Belange werden im Englischen auch *Cross-Cutting-Concerns* genannt.

Eine aspektorientierte Programmiersprache bietet Möglichkeiten, um übergreifende Belange, wie beispielsweise das Protokollieren, umzusetzen, ohne die Modularität eines Programms zu zerstören. Kiczales und andere beschreiben mehrere Punkte, die eine aspektorientierte Sprache erfüllen muss [Kiczales u. a. 2001]:

- Es müssen Momente im Programmablauf festgelegt werden können. Diese Momente heißen auch *Join-Points*.
- Jeder *Join-Point* muss durch die Sprache eindeutig identifizierbar sein.
- Die Sprache muss die Möglichkeit bieten, zu definieren, wie sich das Programm verhalten soll, wenn ein *Join-Point* erreicht wird.
- Es muss möglich sein, Komponenten zu erstellen, die identifizierte *Join-Points* im Programmablauf und ein definiertes Verhalten kombinieren.
- Durch die Sprache muss außerdem definiert sein, wie diese Komponenten in ein Programm integriert werden.

Der letzte Punkt wird üblicherweise durch das sogenannte „Einweben“ vorgenommen. Vor oder während des Kompilieren eines Programms verändert der *Aspect-Weaver* den Programmcode so, dass das übergreifende Verhalten bei der Programmausführung vorhanden ist.

2.9. AspectJ

AspectJ[13] ist eine aspektorientierte Erweiterung für Java, die mithilfe weniger neuer Konstrukte die Umsetzung solcher *Cross-Cutting-Concerns* möglich macht, ohne dabei die Modularität eines Systems zu stören [Kiczales u. a. 2001]:

[13] `http://www.eclipse.org/aspectj/`, zuletzt besucht am 16.10.2017

- *Join-Points* definieren bestimmte Momente im Programmablauf, die abgefangen werden sollen. AspectJ definiert eine Reihe von *Join-Point*-Typen wie etwa Methoden- und Konstruktoraufrufe, das Setzen und Auslesen von Feldern oder die Ausnahmenbehandlung. Als Beispiel könnte der folgende *Join-Point* definiert sein, der eintritt, wenn von einer Person die Adresse gefordert wird:

  ```
  calls(Address Person.getAddress())
  ```

- *Pointcuts* sind Gruppen von *Join-Points*, die mittels booelscher Logik kombiniert werden können.
  Um dem obigen Beispiel zu folgen, stellen wir uns den folgenden *Pointcut* vor, der mehrere Aktionen bündelt:

  ```
  pointcut getPersonData:
     calls(Address Person.getAddress()) ||
     calls(Date Person.getBirthdate()) ||
     calls(String Person.getJob());
  ```

  Dieser *Pointcut* wird immer dann aktiviert, wenn die Daten einer Person, also entweder die Adresse, das Geburtsdatum oder der Job, abgefragt werden.

- Mit einem *Advice* wird festgelegt, was passieren soll, wenn bestimmte *Join-Points* oder *Pointcuts* erreicht werden. Um dem Beispiel nach eine Überprüfung der Berechtigungen zu initiieren, könnte ein *Advice* wie folgt aussehen:

  ```
  before(): getPersonData() {
     if(!isAllowed) {
        throw new NotAuthorizedException("Sorry, but you cannot get this data :(")
     }
  }
  ```

- Ein *Aspect*, auch Aspekt, ist das AspectJ Pendant zur Java Klasse. Es bündelt *Pointcuts* und *Advices*.

Mithilfe von Aspekten ist es möglich ein Programm zu instrumentieren und bestimmte Aktionen auszuführen, wenn das Programm einen definierten Zustand erreicht. `Kieker` nutzt AspectJ beispielsweise um eine Sonden bereitzustellen, die jeden Methodenaufruf innerhalb eines Programmes protokolliert. Der von `Kieker` verwendete *Join-Point* `calls (* *.*(..))` betrifft jede Methode einer jeden Klasse mit beliebigem Rückgabewert und beliebigen Parametern. Mit den richtigen *Advices* kann Kieker so einen Kontrollfluss rekonstruieren.

Um `Kieker` in die Lage zu versetzen, auch Kommunikation über `HTTP` und REST zu registrieren, werden wir eigene Aspekte schreiben. `AspectJ` ist für uns deshalb von entscheidender Rolle.

2.10. Docker

Docker ist eine Software, die es ermöglicht, Anwendungen in sogenannten Containern zu starten[14]. Dabei enthält jeder davon ein virtuelles Betriebssystem. Dadurch isoliert Docker die Anwendung vom System des Hauptcomputers, da alle benötigten Abhängigkeiten in den Containern integriert sind. Es ist somit möglich, eine Anwendung unter identischen Voraussetzungen auf jedem System auszuführen.

Gerade im Hinblick auf Microservices eignet sich Docker gut für die Bereitstellung der einzelnen Dienste, da nicht immer sichergestellt werden kann, dass der Server und die Rechner der Entwickler gleich funktionieren und aufgesetzt sind. Auch bietet Docker die Möglichkeit, die Lücke zwischen verschiedenen Betriebssystemen zu überbrücken. Mithilfe der Virtualisierung durch Docker kann jeder Entwickler jeden Microservice auf seinem Arbeits-Computer nutzen.

Auch Symphony nutzt über eine Gradle-Erweiterung Docker-Container, um die Spring-Boot-Applikationen zu starten. Wir werden uns deshalb überlegen müssen, wie wir die Java-Agenten von Kieker und ExporViz in den Docker-Container bekommen.

2.11. Groovy

Groovy ist eine auf Java basierende Programmiersprache, die durch sprachliche Besonderheiten die Entwicklung von Applikationen vereinfachen soll. Listing 2.2 und Listing 2.1 zeigen einen kurzen Teil Code aus einem fiktiven Programm, einmal als konformer Java-Code und einmal als Groovy-Version, die einige dieser Besonderheiten zeigt.

Listing 2.1. Java-konformer Code

```
1 Message hello = new Message();
2 hello.setText("Hello");
3 System.out.println(hello.getText());
```

Listing 2.2. Besonderheiten von Groovy

```
4 def hello = new Message()
5 hello.text = 'Hello'
6 println hello.text
```

In Zeile 4 sehen wir, dass Groovy keine verpflichtende Typisierung fordert. Mit dem Schlüsselwort *def* kann ein Objekt erstellt werden. Zusätzlich werden bei Groovy im Gegensatz zu Java keine Seminkolons am Ende der Zeilen benötigt. In Zeile 5 ist zu erkennen, dass in Groovy auch Setter- und Getter-Methoden entfallen und Merkmale auch direkt über die Punkt-Notation gesetzt und gelesen werden können. Zeile 5 ist damit äquivalent zu Zeile 2. Die letzte an dieser Stelle vorgestellte Besonderheit ist, dass ein Methodenaufruf mit mindestens einem Argument ohne Klammern erfolgen kann. Mit der in Groovy verfügbaren Funktion *println*, die als Abkürzung für *System.out.println* fungiert, ist Zeile 6 ein Groovy-konformer Methodenaufruf, gleichbedeutend mit Zeile 3 in Listing 2.1.

[14] https://www.docker.com/what-docker, zuletzt besucht am 25.09.2017

Groovy wird für die meisten der Microservices von Symphony verwendet und ist die genutzte Sprache in den *build*-Dateien von Gradle. Auch wenn wir nicht viel Code in Groovy schreiben werden, ist eine Kenntnis darüber für das Verständnis der Gradle-Anpassungen von Vorteil

2.12. Gradle

Gradle[15] ist ein Tool für die Bereitstellung von Anwendungen, ein sogenanntes *Build-Tool*. Es wird hauptsächlich für Java-Projekte verwendet. Wie andere *Build-Tools* arbeitet auch Gradle mit *Build-Files*, die Anweisungen für das Bereitstellen beinhalten. Die *Build-Files* von Gradle enthalten ausführbaren Groovy-Code und kommen deshalb ohne Extensible Markup Language (XML) zur Konfiguration aus.

In jedem *Build-File* von Gradle stehen bestimmte Interfaces wie Project oder TaskContainer als Schnittstellen zur Verfügung. Dies sorgt dafür, dass gewisse Methoden nutzbar sind, ohne dass dies auf den ersten Blick erkennbar ist. Listing 2.3 zeigt ein einfaches Gradle-Skript.

Listing 2.3. Einfaches Gradle-Skript

```
1  apply plugin: 'java'
2
3  repositories {
4     jcenter()
5  }
6
7  dependencies {
8     compile 'net.kieker-monitoring:kieker-monitoring:1.0.0'
9  }
10
11 jar {
12    manifest { attributes 'Main-Class': 'mypackage.Application' }
13    baseName = 'service'
14    version = '1.0.0'
15 }
```

Die Zeilen 1, 3, 7 und 11 sind alle Aufrufe existierender Methoden. In Zeile 1 wird die Methode apply mit einer ein-elementigen *map* als Parameter aufgerufen, äquivalent zu apply(['plugin': 'java']). Groovy erlaubt das Weglassen der Klammern und der Anführungsstriche um die Schlüssel. Das Konstrukt in geschweiften Klammern von Zeile 3 bis 5 wird in Groovy *Closure* genannt und ähnelt einer anonym ausgeführten Methode. Wieder ist Zeile 3 also ein Methodenaufruf, der die *Closure* als Argument bekommt. Das

[15] https://gradle.org/, zuletzt besucht am 16.10.2017

`jar` in Zeile 11 ist ein sogenannter *Task*, also eine Aufgabe, die `Gradle` ausführen kann. `jar {...}` ist eine Abkürzung für die Methode `TaskContainer.getByName('jar', {...})`, welche einen *Task* zurückgibt und die optionale *closure* als Konfiguration auf diesen *Task* anwendet.

Es ist also möglich eine Aufgabe beliebig oft durch Aufruf ihres Namens, gefolgt von einer Konfiguration, zu modifizieren. Ein *Task* wird dabei stets nur erweitert und nicht komplett überschrieben. Dieses Verfahren werden wir später nutzen, um die `Kieker` Bibliothek hinzuzufügen.

Kapitel 3

Analyse

Durch die Vielzahl an Microservices, deren Projektstruktur und verwendete Frameworks ist der Aufbau der Plattform von `Symphony` nicht selbsterklärend. Um überhaupt ein Monitoring an die einzelnen Dienste anbinden zu können, müssen wir deren Struktur und Funktionsweise verstehen. Da sich die Monitoring-Tools `Kieker` und `ExplorViz` trotz ihrer ähnlichen Ansätze in einigen Punkten voneinander unterscheiden, ist es auch wichtig, diese Unterschiede zu verstehen.

Bevor wir im folgenden Kapitel 4 ein Konzept für die Verbindung beider Tools und die Visualisierung der Microservices von `Symphony` entwerfen, wollen wir uns in diesem Kapitel erst einen Überblick über die bereits bestehende Arbeit in den Projekten verschaffen.

3.1. Aufbau von Symphony

Betrachten wir zuerst den Aufbau der Softwarelandschaft der Plattform von `Symphony`. Hierbei möchten wir einen Überblick über den Aufbau der einzelnen Dienste bekommen, sowie verstehen, wie diese zusammenhängen und untereinander kommunizieren.

3.1.1. Übersicht über die Microservices

Betrachten wir zuerst den Aufbau der Microservice-Architektur, die hinter der Plattform von `Symphony` steht. Zum Zeitpunkt des Schreibens dieser Arbeit besteht diese aus acht Microservices. Diese sind in Abbildung 3.1 abgebildet. Alle Microservices laufen in der Standard-Konfiguration auf der gleichen Maschine. Es ist üblich zum Zwecke der Skalierbarkeit mehrere Instanzen eines Dienstes hinter einen *Load-Balancer* zu schalten [Namiot und sneps sneppe 2014], hierfür existieren im `Symphony`-Projekt allerdings noch keine Versuche. Um Konflikte zu vermeiden wird deshalb jedem Microservice ein fester und eindeutiger Port zugewiesen. Im nachfolgenden Abschnitt werden alle Microservices kurz vorgestellt.

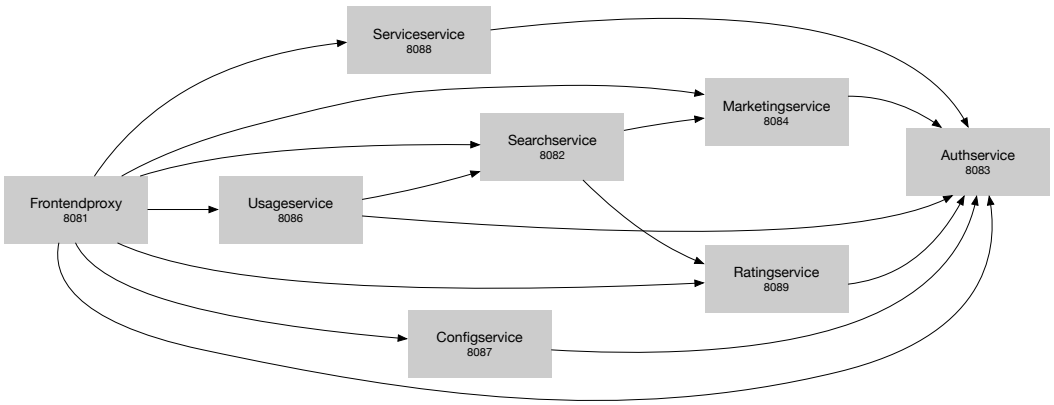


Abbildung 3.1. Die Microservices und ihre Abhängigkeiten, dargestellt als Pfeile

Frontendproxy

Der *Frontendproxy* stellt den Kern der Plattform da. Es ist zuständig für die Bereitstellung der Nutzeroberfläche und einem umfangreichen API zu allen anderen Diensten. Der *Frontendproxy* kommuniziert du diesem Zeitpunkt über den Port 8081. Um die Webseite von Symphony zu erreichen muss ein Nutzer deshalb den genannten Port aufrufen.

Authservice

Der *Authservice* ist für die Authentifizierung von Nutzern zuständig. Er wird von fast jedem anderen Diensten benötigt und ist deshalb nach dem *Frontendproxy* der wichtigste Microservice.

Marketingservice

Vom *Marketingservice* wird die Verwaltung von Informationen zu Diensten und Dienstanbietern übernommen. Er ermöglicht es einer Firma, die für die Vermarktung relevanten Daten wie Produkttexte oder -bilder einzustellen. Möchten wir Produkte oder Anbieter anzeigen lassen, so muss der *Marketingservice* laufen.

Searchservice

Als Verbindung zur Datenhaltung, indiziert der *Searchservice* Produktinformationen und bietet eine Produktsuche mittels *Elastic Search*[1]. Diese wird benötigt, um Listen von Diensten oder Anbietern anhand verschiedener Kriterien anzuzeigen.

[1] `https://www.elastic.co/de/products/elasticsearch`, zuletzt besucht am 16.10.2017

Ratingservice

Der neueste unter den Microservices ist der *Ratingservice*. Er ist zuständig für das Erfassen und Verwalten von Beliebtheitsinformationen sowie Kundenbewertungen.

Usageservice

Der *Usageservice* sammelt Nutzungsdaten der Anwender der Plattform und legt damit Nutzungsprofile an. Diese Profile werden zum einen dazu verwendet, um jedem Nutzer eine Übersicht seines Verhaltens zu präsentieren und andererseits um beispielsweise bei der Produktsuche die angezeigten Produkte automatisch nach den Anforderungen zu filtern oder die Preise für den aktuellen Konsum zu berechnen.

Configservice

Für die Konfiguration der gebuchten Dienste steht der *Configservice* zur Verfügung. Er ermöglicht es beispielsweise, Agenten hinzuzufügen oder eigene Routings festzulegen. Die Ontologie hält für diesen Zweck Konfigurationen und Regeln bereit, wie welcher Dienst verwendet werden kann.

Serviceservice

Im Gegensatz zum *Marketingservice* ist der *Serviceservice* für die Verwaltung von technischen Daten da. Über ihn können Anbieter ihre Dienste konfigurieren, zum Beispiel um Änderungen an Preismodellen oder Adapterversionen vorzunehmen.

3.1.2. Abhängigkeiten zwischen den Microservices

Je nach Aufgabe kommunizieren die einzelnen Microservices mit anderen oder sind sogar auf deren Daten angewiesen. Diese Abhängigkeiten haben einen direkten Einfluss auf die Reihenfolge, in der die Microservices gestartet werden müssen. Der *Searchservice* beispielsweise fragt direkt beim Start eine Liste der Produkte beim *Marketingservice* an. Wenn der *Marketingservice* nicht gestartet ist, kann der *Searchservice* keine Produktsuche durchführen, bis nicht eine Reindizierung manuell gestartet wurde.

Um die Abhängigkeiten zu ermitteln, können wir die einzelnen `.properties`-Dateien der Dienste analysieren. In ihnen sind, wie bereits erwähnt, alle projektrelevanten Attribute enthalten, die von der Anwendung ausgelesen und verwendet werden. Diese Attribute beinhalten auch URIs zu anderen Microservices. Die Controller der Dienste lesen diese aus den `.properties`-Dateien aus und bei einer Änderung müssen die URIs nur an einer Stelle angepasst werden. Besitzt ein Microservice A also ein Attribute für die Schnittstelle eines anderen Microservice B, so gehen wir davon aus, dass Microservice B vor Microservice A gestartet werden muss.

Die aus diesen Daten generierten Abhängigkeiten sind in Abbildung 3.1 als Pfeile eingezeichnet. Jeder Pfeil bedeutet, dass der Ursprungs-Microservice ein URI des Ziel-Microservices kennt. Sortieren wir die Knoten des Graphen topologisch, lässt sich daraus neben den Abhängigkeiten auch eine Reihenfolge ermitteln, in der die einzelnen Microservices gestartet werden können, ohne dass es zu Konflikten kommt. Abbildung 3.2 zeigt eine mögliche Startreihenfolge der Microservices von rechts nach links.

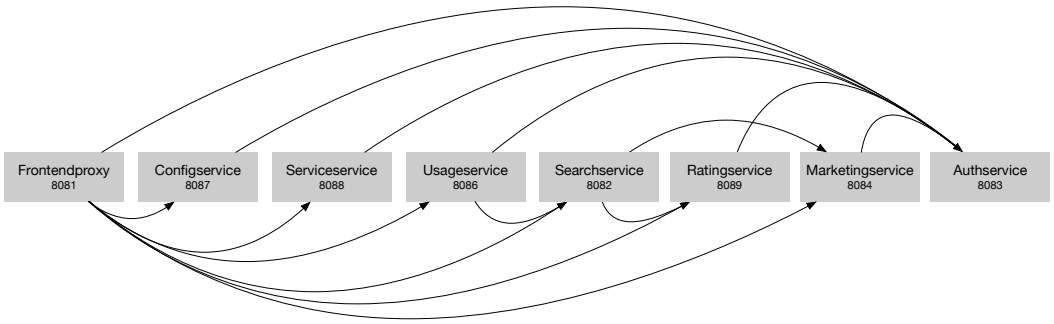


Abbildung 3.2. Konfliktfreie Startreihenfolge von rechts nach links, durch topologische Ordnung der Microservices

3.1.3. Aufbau der einzelnen Microservices

Alle Microservices in `Symphony` sind `Gradle`-Projekte und bauen auf das `Spring`-Framework mit der `Spring-Boot`-Erweiterung auf. Die Projektstruktur folgt in allen Microservices einem ähnlichen Schema. Wir wollen uns in diesem Abschnitt diese Projektstruktur und einige ihrer Besonderheiten ansehen. Abbildung 3.3 zeigt dafür exemplarisch die für uns relevanten Dateien und Verzeichnisse des *Authservice*.

Quellcode

Mit dem *Auth-*, dem *Marketing-*, dem *Service-* und dem *Usageservice* enthalten die meisten Dienste ausschließlich `Groovy`-Code. Der *Config-* und der *Ratingservice* sind dagegen in `Java` geschrieben. Nur der *Frontendproxy* und der *Ratingservice* beinhalten `Java-` und `Groovy`-Bestandteile. Entsprechend der verwendeten Sprache finden sich im `src/main/`-Verzeichnis jeweils für `Groovy`, `Java` oder beide eigene Unterverzeichnisse mit dem Quellcode der Anwendung. Im *Authservice* in Abbildung 3.3 befindet sich aller Programmcode im `main/groovy`-Verzeichnis.

Der Aufbau der Paketstruktur innerhalb der `groovy-` und `java`-Verzeichnisse ist nicht vorgegeben und auch nicht einheitlich. Sie wird je nach Bedarf angelegt. Durch die von `Spring` verwendete und bereits in Abschnitt 2.5 vorgestellte *Dependency Injection*, ist dies

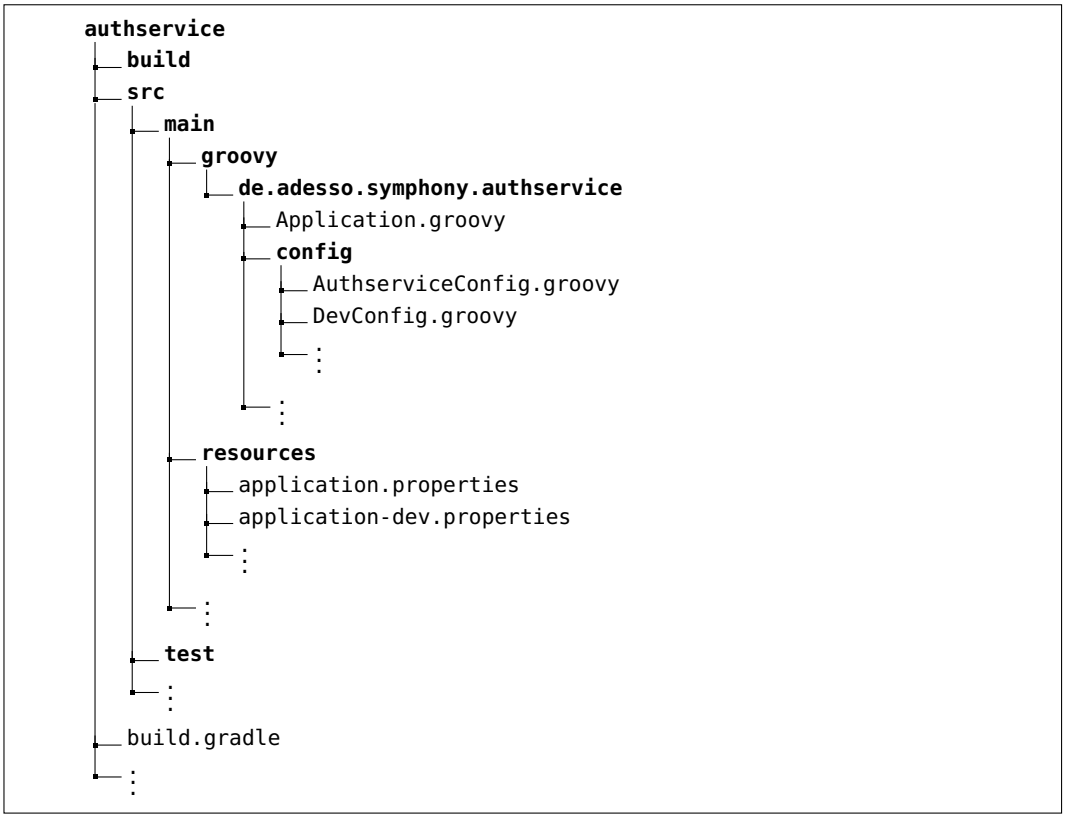


Abbildung 3.3. Exemplarischer Auszug aus der Projektstruktur eines der Microservices

auch nicht nötig. Jede Klasse wird für ihren Zweck annotiert und von `Spring` dann automatisch gefunden[2]. Die Hauptklasse des `Spring-Boot`-Projekts, üblicherweise als `Application` benannt, kann sich beispielsweise in der Wurzel des Quellcode-Verzeichnisses oder in einem Unterordner wie `app` befinden. Durch ihre Annotation `@SpringBootAplication` wird sie als Hauptklasse erkannt.

Dienstkonfiguration

Die relevanten Attribute für jedes Projekt befinden sich in einer oder mehreren `.properties`-Dateien im `src/main/resources`-Verzeichnis. In einer `.properties`-Datei wird jedes Attribut als eine Folge von Attributname, einem Gleichzeichen und dem Attributwert aufgeführt. Damit werden, je nach Bedarf, unterschiedliche Informationen wie beispielsweise der

[2] `https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-spring-beans-and-dependency-injection.html`, zuletzt besucht am 27.09.2017

Dienstname, der dem Dienst zugewiesene Port oder die Zugangsdaten zu einer Datenbank definiert. Die Attribute der `.properties`-Dateien können von der Anwendung ausgelesen werden.

Zusätzlich finden in `Spring` sogenannte Konfigurationen Verwendung. Wir finden diese in allen Microservices im Paket `de.adesso.symphony.[servicename].config`. Eine Konfiguration enthält die Annotation `@Configuration` und wird in `Spring` in erster Linie dazu verwendet, Beans[3] zu erstellen, die innerhalb der ganzen Anwendung verfügbar sein sollen.

Kommunikation über REST

Um die clientseitige Kommunikation über HTTP zu erleichtern, enthält `Spring` die Klasse `RestTemplate`[4]. Das `RestTemplate` ermöglicht es einem Entwickler, REST-Aufrufe an einen Server zu senden, ohne sich um den Aufbau einer HTTP-Verbindung und die Konvertierung von Ressourcen kümmern zu müssen. Für einen `GET`-Anfrage genügt es beispielsweise, die Methode `getForObject` mit der entsprechenden URI der Ressource und einem Objekttypen für den Rückgabewert aufzurufen, wie Listing 3.1 zeigt. Nach erfolgreicher Antwort baut die Methode automatisch aus den Daten ein Objekt des gewünschten Types und gibt dieses zurück.

Listing 3.1. Verwendung des `RestTemplates` mit `Spring`

```
1 RestTemplate restTemplate = new RestTemplate();
2 ProductInformation pI = restTemplate.getForObject(url, ProductInformation.class)
3
4 println "Produktinformationen fuer " + pI.getName() + " erhalten."
```

Die Microservices von `Symphony` nutzen alle `RestTemplates` für die Kommunikation als Client. Um zu verhindern, dass mehrere Instanzen des `RestTemplates` innerhalb eines Dienstes erstellt werden, wird ein `RestTemplate` üblicherweise als Bean in einer der Konfigurationen definiert. Dies hat auch den Vorteil, dass Einstellungen an einer einzigen Instanz vorgenommen werden müssen, und nicht an jeder Stelle, an der ein `RestTemplate` erstellt wird.

Für die serverseitige Kommunikation über HTTP und REST, bietet `Spring` ebenfalls eine einfache Lösung. Mit `@RestController` annotierte Klassen werden automatisch als Ressourcen-Controller erkannt. Die Annotation `@RequestMapping` weist Klassen und Methoden URIs zu, für die diese zuständig sind. Listing 3.2 stellt einen Ausschnitt aus dem `ProductController` des *Marketingservice* vor. In den Zeilen 1 und 2 wird die Klasse als für die URI `marketing` zuständiger Controller festgelegt. Die Annotation `@RequestMapping` von Zeile 5 bis 9 legt fest, dass eingehende `GET`-Anfragen an die URI `/marketing/products/id`

[3] Siehe Abschnitt 2.5 zur Einführung in `Spring`.

[4] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html`, zuletzt besucht am 25.09.2017

Listing 3.2. Verwendung von `RestControllern` mit `Spring`

```
@RestController
@RequestMapping(value='/marketing')
class ProductController {

    @RequestMapping(
        value='/products/{id}',
        method=RequestMethod.GET,
        produces='application/json'
    )
    Product getProduct(@ApiParam(required=false) @PathVariable String id {
        Product product = this.productRepository.findProduct(id)
        ...
        return product
    }

    ...
}
```

von der Methode `getProduct` bearbeitet werden. Darin wird das geforderte Produkt gesucht und zurückgegeben. Durch die Einstellung `produces='application/json'` wird das Produkt automatisch in ein JSON-Objekt umgewandelt.

Erstellungsprozess

Für den Build-Prozess und Starten wird in den Microservices von `Symphony` das Tool `Gradle` verwendet. `Gradle` wird über eine `build.gradle`-Datei konfiguriert. Diese finden wir im Hauptverzeichnis eines jeden Dienstes. Die `build.gradle`-Dateien definieren unter anderem die von `Gradle` zu verwendenden Plugins, Quellen für zu beziehende Abhängigkeiten, und Einstellungen bezüglich des Erstellungsprozesses.

Um die Bereitstellung der Dienste zu erleichtern wird die `Spring`-Erweiterung `Spring Boot` verwendet. Mittels `Spring Boot` ist es möglich einen Microservice über einen Befehl automatisiert in Betrieb zu nehmen:

```
$ gradle bootRun
```

`Gradle` lädt dann selbst die benötigten Abhängigkeiten aus den definierten Quellen, kompiliert mithilfe des `Java`- oder `Groovy`-Plugins alle Bestandteile des Dienstes und erstellt ein Archiv der Anwendung. In Verbindung mit dem `Docker`-Plugin, startet es im Anschluss einen `Docker`-Container, in welchem es die *JAR* der Anwendung mit `Java 8` ausführt.

Als Quelle für Abhängigkeiten steht jedem Microservice neben JCenter[5], welches verbreitete, veröffentlichte Java-Pakete hält, ein lokales Maven-Repositorium für nicht veröffentlichte Pakete zur Verfügung. Dieses liegt im Hauptverzeichnis des Symphony-Projekts. Es beinhaltet für Symphony entwickelte Bibliotheken, die dienstübergreifend verwendet werden oder Bibliotheken von Drittanbietern, die nicht aus den öffentlichen Datenbanken bezogen werden können. Wir werden das Symphony-eigene Repositorium benötigen, um den Microservices auch die Tools Kieker und ExplporViz zur Verfügung zu stellen.

3.2. Monitoring mit ExplorViz

Als nächstes betrachten wir die Monitoring-Komponente von ExplorViz. Wir werden einen kurzen Überblick darüber erlangen, wie das Monitoring verwendet wird und funktioniert und verstehen, warum ExplorViz nicht geeignet ist, um das Monitoring der Microservices zu übernehmen.

3.2.1. Verwendung des Monitorings

Das Monitoring von ExplorViz ist zuständig für das Protokollieren des Verhaltens einer laufenden Anwendung. Es enthält geeignete Probes, einen MonitoringController und einen TCPWriter, der alle Records an den ExplorViz-Server schickt. Standardmäßig verwenden die Probes AspectJ für das Instrumentieren von Java-Anwendungen [Fittkau u. a. 2015b].

Da das ExplorViz-Monitoring die Monitoring-Daten nicht für eine spätere Analyse im Dateisystem ablegt, muss der Server vor einem Monitoring gestartet sein. Für die Überwachung einer Anwendung müssen wir beim deren Start die Bibliothek des ExplorViz-Monitorings als Java-Agenten definieren:

```
$ java -javaagent:/pfad/zur/explorviz-monitoring.jar -jar /pfad/zur/anwendung.jar
```

Der Java-Agent von ExplorViz weist dann AspectJ an, die Aspekte der Probes in den Byte-Code der Anwendung einzuweben.

3.2.2. Konfiguration des Monitorings

Im META-INF-Verzeichnis der *JAR* befinden sich zwei Konfigurationsdateien für AspectJ und ExplorViz selbst. Die aop.xml legt fest, welche Teile der Anwendung instrumentiert werden sollen, und die explorviz.live_trace_processing.properties definiert Einstellungen für das Monitoring und die Analyse. Unter anderem werden in dieser Datei Informationen über die IP-Adresse und den Port des Servers hinterlegt, Metadaten zu der überwachten Anwendung deklariert oder Einstellungen bezüglich der Datenübertragung vorgenommen.

[5] https://bintray.com/bintray/jcenter, zuletzt besucht am 16.10.2017

3.2.3. Records und Traces

Für das Festhalten eingetretener Ereignisse gibt es in ExplorViz sogenannte Event-Records. Ein Event-Record enthält verschiedene Informationen wie etwa über den Aufruf einer Methode oder eines Konstruktors einer Klasse. Um die Reihenfolge der Records festzuhalten, bekommt jeder Record eine Trace-ID und einen Order-Index.

Ein Trace ist immer eine Folge von Ereignissen, die nacheinander auftreten. Ein Trace beginnt dann, wenn eine Methode der überwachten Anwendung aufgerufen wird, ohne dass die Ursache dem Monitoring bekannt ist. Der Order-Index zählt die Methodenaufrufe innerhalb eines Traces hoch und legt so deren Reihenfolge fest.

Die Analyse-Komponente von ExplorViz verwendet Traces für die Rekonstruktion des Kontrollflusses. Damit kann in der Visualisierung das Verhalten der Anwendung grafisch dargestellt werden.

3.2.4. Nachteile des Monitorings mit ExplorViz

Obgleich das Monitoring in ExplorViz dem von Kieker ähnelt, bringt es eigene Einschränkungen mit sich. Zum einen können mit ihm nur Java-Anwendungen instrumentiert werden. Dies ist für den Anwendungsfall Symphony zwar unerheblich, ist aber unpraktisch für zukünftige Vewendungen. Zum anderen erwartet ExplorViz die aop.xml- und die explorviz.live_trace_processing.properties-Datei innerhalb des *META-INF*-Verzeichnisses der *explorviz-monitoring.jar* und nicht innerhalb der zu überwachenden Anwendung. Dies erschwert das Monitoring von mehreren Applikationen erheblich, da die *JAR* entweder für jede eigens angepasst werden muss oder die Parameter beim Aufruf einer jeden über die Kommandozeile überschrieben werden müssen.

Kieker löst diese Probleme besser. Die Konfigurationsdateien werden im *META-INF*-Verzeichnis der zu überwachenden Anwendungen gesucht und müssen nicht in der kieker-1.12-aspectj.jar angepasst werden. Durch seine Erweiterbarkeit erlaubt es Kieker außerdem, eigene Komponenten wie Writer oder Reader anwendungsspezifisch in der Konfiguration zu definieren. Damit ist es beispielsweise möglich, neue Sprachen zu unterstützen und das Monitoring und die Analyse komplett zu trennen.

Aufgrund dieser Unterschiede der beiden Monitoring-Komponenten arbeitet das ExplorViz-Team auf eine komplette Ablösung des eigenen Monitorings hin und plant in Zukunft nur noch Kieker dafür zu verwenden.

Wir werden deshalb in den folgenden Abschnitten und Kapiteln jeweils das Monitoring von Kieker betrachten und von ExplorViz nur die Analyse und Visualisierung nutzen.

3.3. Monitoring mit Kieker

Als Alternative zum Monitoring mit ExplorViz kann ein Monitoring auch mit Kieker durchgeführt werden. Dieser Abschnitt listet die Gemeinsamkeiten und Unterschiede

zwischen Kieker und ExplorViz auf und definiert die Herausforderung, die im Rahmen dieser Arbeit ensteht.

3.3.1. Verwendung des Monitorings

Das Monitoring mit Kieker ähnelt dem mit ExplorViz. Auch Kieker verwendet standardmäßig AspectJ für das Instrumentieren der Anwendungen und setzt bei der Wahl der Probes auf Aspekte, die in den Byte-Code eingewebt werden. Als Beispiel existiert für ein volles Monitoring von Java-Anwendungen das FullInstrumentation-Aspekt. Dieses Aspekt schickt vor und nach Ausführung einer beliebigen Methode durch eine beliebige Klasse ein BeforeOperationEvent beziehungsweise ein AfterOperationEvent an den Monitoring-Controller.

Um das Monitoring zu starten, müssen wir wie bei ExplorViz die Monitoring-Bibliothek von Kieker als Java-Agent angeben. Da in Kieker das Monitoring und die Analyse getrennt voneinander sind, existiert kein Kieker-Server, der vor dem Beginn des Monitorings gestartet sein muss. Auf welche Weise die Analyse stattfindet, wird in der kieker.monitoring.-properties-Datei definiert.

3.3.2. Konfiguration des Monitorings

Die Konfiguration von Kieker besteht ebenfalls aus zwei Dateien; einer aop.xml-Datei für AspectJ und einer kieker.monitoring.properties-Datei für Kieker selbst. Entgegen des Monitorings mit ExplorViz, bietet Kieker mehr Einstellungsmöglichkeiten und unter anderem die Möglichkeit, einen eigenen Writer anzugeben, der die Monitoring-Daten an einen definierten Ort schreibt oder schickt. Möchten wir beispielsweise Logs im Dateisystem speichern, so können wir den AsciiFileWriter von Kieker verwenden.

Kieker erwartet die Konfigurationsdateien in der Anwendung, die überwacht wird, was einen großen Vorteil für die Nutzbarkeit bringt.

3.3.3. Records und Traces

Das Monitoring von Kieker verwendet genau wie ExplorViz Records für das Festhalten von Ereignissen, die wiederum in Traces zusammengehören. Die Records in Kieker sind zwar anders benannt und anders aufgebaut als deren Äquivalente in ExplorViz, enthalten aber dieselben Informationen, auch über den Trace, in dem sie auftreten, und über deren Order-Index.

3.3.4. Herausforderung

Das Kieker-Projekt enthält zum Zeitpunkt des Schreibens dieser Arbeit noch keine Möglichkeit Kommunikation zwischen Knoten in einer verteilten Softwarelandschaft automatisch

zu überwachen. Wir machen es uns deshalb zur Aufgabe, ein Monitoring der verteilten Dienste von Symphony zu implementieren um dessen Machbarkeit zu zeigen.

3.4. Übertragung der Daten an ExplorViz

Um die durch Kieker generierten Monitoring-Daten in ExplorViz verwenden zu können, müssen wir diese übertragen und in ExplorViz-Records umwandeln. Hierfür gibt es generell zwei Wege. Einmal ist es möglich, einen eigenen Reader auf der Seite von ExplorViz zu entwerfen, der bestehende Kieker-Logs einliest, entsprechende Records generiert und diese wie der TCPReader an die ExplorViz-Analyse gibt. Dadurch würde der ExplorViz-eigene Reader zwar nicht ausgetauscht, aber ein Verarbeiten von Kieker-Records wäre somit möglich. Ein anderer Weg ist, einen eigenen Writer für Kieker zu entwickeln, der die Umwandlung der Daten übernimmt und die ExplorViz-Records an den TCP-Reader schickt. Für beide Ideen zur Übertragung der Daten existieren bereits Ansätze, die aber nicht mehr aktuell sind und nicht gepflegt werden.

Auch wenn das Monitoring in Kieker und ExplorViz nach dem gleichen Prinzip funktioniert, sind durch die parallele Entwicklung unterschiedliche Record-Typen entstanden. Kieker verwendet beispielsweise das BeforeOperationEvent, während das Äquivalent in ExplorViz BeforeOperationEventRecord heißt. Auch die Reihenfolge, in der die Daten der Records in String-Repräsentation angegeben werden unterscheidet sich.

Eine Übertragung von Daten zwischen Kieker und ExplorViz muss alle Kieker-Records in ExplorViz-kompatible umwandeln.

3.4.1. Herausforderung

Für die Online-Übertragung der Monitoring-Daten existiert in Kieker seit Version 1.9 der sogenannte ExplorVizTcpWriter. Um die Daten offline zu übertragen, ist der *External Monitoring Logs Adapter* Teil des ExplorViz-Projekts. Im Rahmen dieser Arbeit wollen wir beide bestehenden Implementierungen prüfen und so erweitern, dass sie verteilte Kommunikation aus Kieker ebenfalls an ExplorViz leiten können.

3.5. Analyse und Visualisierung mit ExplorViz

Zuletzt beschäftigen wir uns mit der Analyse und Visualisierung durch ExplorViz. Dabei werden wir erkennen, wie die Traces in ExplorViz verarbeitet werden und was das für den Entwurf des Monitorings von verteilter Kommunikation in Kieker bedeutet.

3.5.1. Verwenden des ExplorViz-Servers

Der ExplorViz-Server ist zuständig für das Empfangen und Verarbeiten von eingehenden Monitoring-Records und für das Bereitstellen der Weboberfläche, die wiederum die Visuali-

sierung der Anwendungen übernimmt. Den ExplorViz-Server können wir entweder bereits kompiliert von der Projektseite herunterladen und dann in einem geeigneten Web-Server wie beispielsweise Apache Tomcat bereitstellen oder über das offizielle *Github*-Repository[6] beziehen und über die im Projektverzeichnis enthaltene Konfigurationsdatei direkt aus der Entwicklungsumgebung Eclipse starten.

Einmal gestartet nutzt der ExplorViz-Server zwei relevante Ports. Diese sind individuell wählbar, wir nehmen hier aber die Standardeinstellungen an. Über den Port 8888 liefert der Server die Benutzeroberfläche an einen Webbrowser aus. Das verwendete Framework Google Web Toolkit (GWT)[7] kompiliert die Oberfläche beim ersten Aufruf und bei Änderungen am Code automatisch. Der zweite verwendete Port ist 10133 und wird vom TCPReader belegt. Der TCPReader wartet auf eingehende Monitoring-Records und leitet diese an die Analyse weiter.

3.5.2. Verarbeiten von Traces

Wir werden im Rahmen dieser Arbeit nicht genauer auf das Verarbeiten der Traces zum Landschafts- und zu den Applikationsmodellen eingehen. Es ist aber wichtig zu verstehen, wie der Kontrollfluss in ExplorViz, vor allem mit dem Fokus auf verteilte Kommunikation, erfasst wird.

Abbildung 3.4 stellt grob die Verarbeitung von eingehenden Records dar. Das LandscapeRepositorySink bekommt vom ExplorViz-Framework sogenannte IRecords, welche es dann an das LandscapeRepositoryModel weitergibt. Dort werden die IRecords an das InsertionRepositoryPart geschickt, welches für die Verarbeitung derer zuständig ist. Interessant sind für uns besonders die IRecords vom Typ Trace, die selbst wiederum aus einer Menge von Event-Records bestehen. Je nach Typ der Event-Records werden diese entweder direkt auf die Softwarelandschaft angewandt oder an das für verteilte Aufrufe verantwortliche RemoteCallRepositoryPart weitergereicht.

Das RemoteCallRepositoryPart ordnet BeforeSent- und BeforeReceivedRemoteCallRecords einander zu. In Listing 3.3 wird die Methode des RemoteCallRepositoryParts gezeigt, die bei einem eingehenden BeforeSend- den zugehörigen BeforeReceivedRemoteCallRecord sucht. Dabei werden in Zeile 6 und 7 die Trace-IDs und die Order-Indizes beider Records verglichen. Diese genügen ExplorViz, um verteilte Kommunikation zu erkennen und darzustellen.

Es wird für die Visualisierung der Microservices von Symphony ausreichen, bei REST-Aufrufen BeforeSent- und BeforeReceivedRemoteCallRecords an ExplorViz zu schicken.

[6] https://github.com/ExplorViz/, zuletzt besucht am 16.10.2017
[7] http://www.gwtproject.org/, zuletzt besucht am 16.10.2017

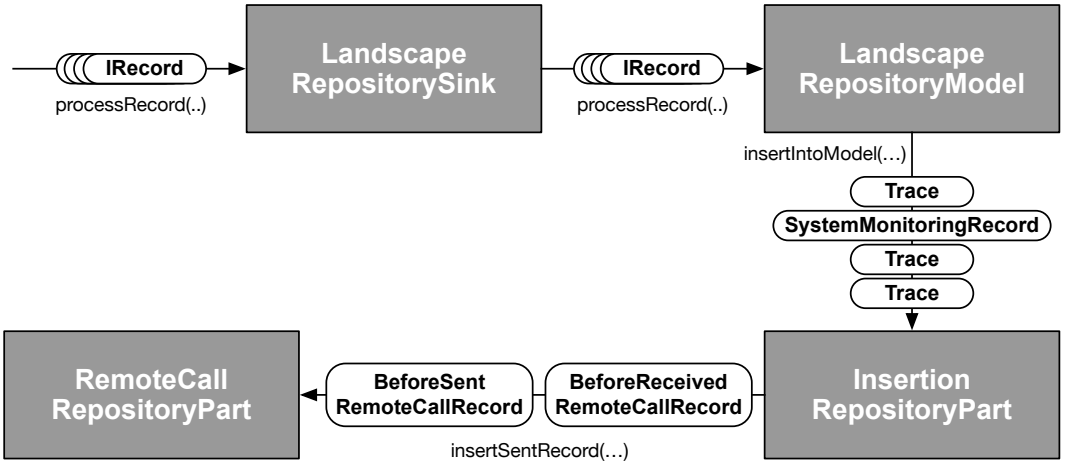


Abbildung 3.4. Verarbeiten eingehender Records in `ExplorViz`.

Listing 3.3. Erweiterung der `bootRun`-Methode um Argumente für die JVM

```
1  private BeforeReceivedRemoteCallRecord seekMatchingReceivedRemoteRecord(
2      final BeforeSentRemoteCallRecord sentRecord
3  ) {
4      for (final BeforeReceivedRemoteCallRecord receivedRemoteRecord :
           receivedRemoteCallRecordCache.keySet()
5      ) {
6          if ((receivedRemoteRecord.getCallerTraceId() == sentRecord.getTraceId())
7              && (receivedRemoteRecord.getCallerOrderIndex() == sentRecord.
                   getOrderIndex())
8          ) {
9              return receivedRemoteRecord;
10         }
11     }
12     return null;
13 }
```

Kapitel 4

Entwurf

Dieses Kapitel beschreibt den Entwurf des Monitorings und der Visualisierung der Microservices von Symphony durch Kieker und ExplorViz. Dabei baut es auf die Erkenntnis aus Kapitel 3 auf. Dort haben wir bereits erfahren, dass ExplorViz in der Lage ist, verteilte Kommunikation darzustellen. Unsere Aufgabe besteht im Entwurf deshalb lediglich darin, die Daten zur verteilten Kommunikation im Monitoring von Kieker zu sammeln und für ExplorViz verständlich an dieses zu übertragen.

4.1. Monitoring mit Kieker

4.1.1. Erfassen von verteilter Kommunikation

Strubel stellt in seiner Arbeit fest, dass für das Erfassen von verteilter Kommunikation in Kieker eigene Records für aus- und eingehende Nachrichten erstellt werden müssen, wie sie auch in ExplorViz existieren [Strubel 2017]. Da das in ExplorViz für die Darstellung von verteilter Kommunikation zuständige RemoteCallRepositoryPart ausschließlich die BeforeSent- und BeforeReceivedRemoteCallRecords verwendet, genügen auch für Kieker zwei Records, die jeweils vor dem Senden und dem Empfangen erstellt werden. Strubel nennt noch zwei weitere Records, die nach dem Senden und nach dem Empfangen von verteilter Kommunikation gespeichert werden. Diese benötigen wir für unsere Zwecke nicht.

Tabelle 4.1 listet die verwendeten Felder der beiden Remote-Call-Records von ExplorViz auf. Ruft Applikation A eine Methode der Applikation B, so werden vor dem Senden des Aufrufs dessen verwendete Technologie sowie die Trace-ID und der Order-Index von Applikation A als Record abgelegt. Applikation B speichert daraufhin sowohl die Trace-ID und den Order-Index von Applikation A ab, als auch die gleichen Informationen für sich selbst. Die Analyse von ExplorViz speichert zusätzlich ein HostApplicationMetaDataRecord in beiden Records, um sie der Applikation zuordnen zu können. Diese Informationen genügen ExplorViz, um verteilte Kommunikation zu erkennen und abzubilden.

Im ersten Schritt erstellen wir somit zwei passende Events für Kieker. Das BeforeSentEvent enthält neben einem Zeitstempel über dessen Erstellungszeitpunkt die Trace-ID und den Order-Index des jeweiligen Methodenaufrufs und die Technologie der Kommunikation. Im BeforeReceivedEvent werden ein Zeitstempel und die Trace-IDs sowie Order-Indizes

Tabelle 4.1. Felder der in ExplorViz verwendete Remote-Call-Records

BeforeSentRemoteCallRecord	BeforeReceivedRemoteCallRecord
String **technology**	*long* **callerTraceId**
long **traceId**	*int* **callerOrderIndex**
int **orderIndex**	*long* **traceId**
HostApplicationMetaDataRecord **thisHost**	*int* **orderIndex**
	HostApplicationMetaDataRecord **thisHost**

der sendenden und der empfangenden Applikation gespeichert. Damit ist es uns möglich, alle für ExplorViz benötigte Informationen in Kieker festzuhalten.

4.1.2. Instrumentieren der Microservices

Interzeptoren oder Aspekt

Um die Microservices von Symphony zu instrumentieren ergeben sich grundsätzlich zwei Möglichkeiten. Zum einen empfiehlt der Kieker *User-Guide* für die Arbeit mit Spring die Verwendung von Interzeptoren. Zum anderen bietet sich die Instrumentierung durch Aspekte an, wie sie Kieker für beliebige Java-Anwendungen durchführt und auch Spring selbst unterstützt.

Als Interzeptoren schlägt Kieker den OperationExecutionMethodInvocationInterceptor und den OperationExecutionWebRequestRegistrationInterceptor vor, die beide Teil des Kieker-Projekts sind. Ersterer erstellt wie auch das Fullinstrumentation-Aspekt ein Before- und ein AfterOperationEvent vor und nach Ausführen einer Methode. Letzterer wird bei eingehenden Nachrichten dazu genutzt, einen neuen Trace zu erstellen. Ein Problem hierbei ist, dass wir die WebRequestInterceptoren laut der Spring-Dokumentation[1] in einer der Konfigurationen des Dienstes vermerken müssen. Dies würde Anpassungen am Code der Microservices bedeuten.

Für das Instrumentieren von ausgehender Kommunikation bietet Kieker derzeit allerdings keine Lösung. Hierfür werden in Spring Interzeptoren am RestTemplate[2] genutzt. Um diese verwenden zu können, müssen wir allerdings sicherstellen, dass jeder Instanz des RestTemplates in den Microservices ein ClientHttpRequestInterceptor zugewiesen wird. Zwar verwenden die meisten Microservices für die RestTemplates Beans, die überall verwendet werden und daher nur an einer Stelle bei der Erstellung des Beans instrumentiert werden müssen, allerdings halten sich nicht alle Entwicklerinnen und Entwickler an dieses Konzept. Es hat sich herausgestellt, dass teilweise, bei Bedarf, eine neue Instanz des RestTemplates erstellt wird, welche dann keinen Interzeptor besitzt. Auch für diese Lösung

[1] https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html, zuletzt besucht am 25.09.2017

[2] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/client/ClientHttpRequestInterceptor.html, zuletzt besucht am 25.09.2017

müssten wir zusätzlichen Code in die Konfigurationsdateien der Microservices einfügen und das Erstellen von `RestTemplates` auf die Verwendung von Beans umstellen.

Aus den genannten Gründen entscheiden wir uns gegen die Verwendung von Interzeptoren für das Instrumentieren der Microservices von `Symphony` und für die Verwendung eines Aspekts. Aspekte werden sowohl von `Kieker` als auch von `Spring` verwendet. `Kieker` macht für die Instrumentierung Gebrauch von `AspectJ`, während `Spring` sogenannte *Dynamic Proxies*[3] nutzt, die bereits standardmäßig in Java enthalten sind. Letztere dienen als Fassade für Methoden anderer Klassen und können dadurch zusätzliche Funktionalität – wie beispielsweise die Instrumentierung – hinzufügen. Damit biete `Spring` gegenüber `AspectJ` allerdings den Nachteil, dass keine Feldzugriffe überwacht werden können.

Welches der beiden Tools das Aspekt verarbeitet, spielt für die Instrumentierung keine Rolle. Allerdings benötigen wir `Kieker` für das Erstellen und Verarbeiten von Event-Records. Unsere Entscheidung fällt daher auf die ausschließliche Verwendung von `Kieker`.

Anpassen der Gradle-Dateien

Standardmäßig empfiehlt `Kieker` für das Monitoring von Java-Anwendungen die Java-Erweiterung `AspectJ`. Um `Kieker`-eigene Konfigurationen zu berücksichtigen, bietet `Kieker` einen eigenen Java-Agenten als Hülle für `AspectJ`, der eine anwendungsspezifische Konfigurationsdatei einlesen und verarbeiten kann. Der *Quickstart-Guide*[4] von `Kieker` stellt den einzig benötigten Befehl für das Monitoring vor:

```
$ java -javaagent:kieker-1.12-aspectj.jar -jar MyJar.jar
```

Durch ihn wird eine neue Java-Laufzeitumgebung gestartet und die Hauptklasse des `MyJar`-Pakets geladen. Durch den Befehl `-javaagent` führt die Laufzeitumgebung noch vor der `main`-Methode der eigentlichen Hauptklasse die `premain`-Methode des Java-Agenten aus.

Für den Fall, dass `Spring Boot` für das Bereitstellen der Dienste verwendet wird, ist diese Art der Instrumentierung allerdings nicht zielführend. Die Java-Laufzeitumgebung, in der die Microservices ausgeführt werden, wird nämlich innerhalb des `Docker`-Containers erstellt und ist für uns nicht über das `Gradle`-Kommando erreichbar. Stattdessen muss `Gradle` selbst die Laufzeitumgebung in `Docker` mit den oben genannten Parametern starten. Zu diesem Zweck lässt sich die `bootRun`-Methode des `Spring-Boot`-Plugins für `Gradle` mit dem Attribut `jvmArgs` um bestimmte Argumente für die JVM erweitern. Listing 4.1 zeigt diese Erweiterung exemplarisch.

Listing 4.1. Erweiterung der `bootRun`-Methode um Argumente für die JVM

```
1 bootRun {
2    jvmArgs = [ "-javaagent:pfad/zur/kieker-1.12-aspectj.jar" ]
3 }
```

[3] `https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html`, zuletzt besucht am 13.10.2017

[4] `http://kieker-monitoring.net/documentation/quickstart-guide/`, zuletzt besucht am 25.09.2017

Wir müssen ergänzend dafür sorgen, dass statt `pfad/zur/` der echte Pfad zur `Kieker`-Bibliothek benutzt wird. Dafür können wir die Eigenschaften des `DependencyHandlers` von `Gradle` ausnutzen. Der `DependencyHandler` gruppiert Abhängigkeiten in sogenannte Konfigurationen. Eine Konfiguration ist eine benannte Menge von Dateien. Dieses Verfahren wird üblicherweise dazu genutzt, um bestimmten Phasen der Bereitstellung benötigte Pakete zuzuweisen. Listing 4.2 zeigt wie der `DependencyHandler` dazu verwendet wird, in den vorgegebenen Repositorien nach dem `Hibernate Validator` zu suchen und die Abhängigkeit dazu der *compile*-Konfiguration hinzuzufügen. Das `Java`-Plugin für `Gradle` liest aus dieser Konfiguration dann die für das Kompilieren benötigten Dateien.

Listing 4.2. Verwenden des `DependencyHandlers` in `Gradle`

```
1 dependencies {
2    compile "org.hibernate:hibernate-validator"
3 }
```

Erstellen wir eine eigene Konfiguration und fügen dieser ausschließlich das `Kieker`-Paket hinzu, so können wir über die durch das Interface `FileCollection` bereitgestellte Methode `getSingleFile()` den absoluten Pfad zur ersten Datei darin erhalten. Diese Art der Verwendung einer Konfiguration wird in Listing 4.3 gezeigt. In Zeile 1 wird eine neue Konfiguration namens *kieker* angelegt. Zeile 4 fügt das `Kieker`-Paket der neuen Konfiguration als einzige Abhängigkeit hinzu. In Zeile 5 erweitern wir die *compile*-Konfiguration um die neue Abhängigkeit. Zeile 10 verwendet nun die einzige Datei der *kieker*-Konfiguration als Pfad zum Java-Agenten.

Um nicht bei jedem Start des Dienstes `Kieker` nutzen zu müssen, ergänzt das Skript durch die Abfrage in Zeile 8 nur dann den Java-Agenten, wenn dem Projekt das Attribut *„Kieker“* mitgegeben wurde.

Aspekt für verteilte Kommunikation

Wie bereits in Abschnitt 2.9 erwähnt, verwendet `AspectJ` für das Überwachen von Verhalten von Anwendungen sogenannte Aspekte. Die Monitoring-Komponente von `Kieker` beinhaltet bereits einige dieser für verschiedene Anwendungszwecke. Unter anderem bietet sie das `FullInstrumentationNoGetterAndSetter`-Aspekt. Mit diesem wird vor Ausführen einer Methode, die weder Getter noch Setter ist, ein `BeforeOperationEvent` beim Monitoring-Controller registriert. Nach Ausführen der Methode erstellt das Aspekt ein `AfterOperationEvent`. Beide Events enthalten Informationen über deren Zeitpunkt der Erstellung, die Trace-ID und den Order-Index, und außerdem über die Klasse und die Methode selbst.

Für die Ermittlung des aktuellen Traces steht den Aspekten die `TraceRegistry` zur Verfügung. Über diese lässt sich sowohl die derzeitige Trace-ID als auch über den Trace den Order-Index ermitteln. Der Code in Listing 4.4 zeigt einen angepassten Ausschnitt

Listing 4.3. Anpassung der `Gradle`-Dateien von `Symphony` für das Monitoring durch `Kieker`

```
1  configurations { kieker }
2
3  dependencies {
4     kieker "kieker:kieker-aspectj:1.13"
5     compile configurations.kieker.dependencies
6  }
7
8  if(hasProperty("kieker")) {
9     bootRun {
10       jvmArgs = [ "-javaagent:${configurations.kieker.singleFile}" ]
11    }
12 }
```

Listing 4.4. Erstellen eines `MonitoringRecords` im `FullInstrumentationNoGetterAndSetter`-Aspekt

```
1  IMonitoringController controller = MonitoringController.getInstance();
2  TraceMetadata trace = TraceRegistry.INSTANCE.getTrace();
3
4  controller.newMonitoringRecord(
5      new BeforeOperationEvent(
6          controller.getTimeSource().getTime(),
7          trace.getTraceId(),
8          trace.getNextOrderId(),
9          operationSignature,
10         thisObject.getClass().getName()
11     )
12 )
```

aus dem `FullInstrumentationNoGetterAndSetter`-Aspekt. In Zeile 2 wird die `TraceRegistry` dazu verwendet den aktiven Trace zu erhalten. Ab Zeile 5 erstellt das Aspekt ein neues `BeforeOperationEvent`, welches in Zeile 4 an die Instanz des `MonitoringControllers` übergeben wird. Der Controller wird in Zeile 6 dazu verwendet, die aktuelle Zeit zu nehmen. Zeilen 7 und 8 lesen die Trace-ID und den nächsten Oder-Index aus dem Trace aus. Auf die Signatur der Operation und den Klassennamen in Zeile 9 und 10 gehen wir nicht weiter ein, da diese für uns nicht weiter relevant sind.

Die von `Kieker` vorgegebenen Aspekte unterstützen die verteilte Kommunikation nicht. Unser Ziel ist es deshalb, ein neues Aspekt für diesen Zweck zu erstellen. Dafür benötigen wir zwei signifikante Methoden, von denen eine bei jeder eingehenden Nachricht und eine bei jeder ausgehenden Nachricht aufgerufen wird. Die für eingehende Nachrichten

zuständige Methode sollte dabei möglichst früh verwendet werden, während wir für ausgehende Nachrichten eine möglichst späte Methode suchen.

Zusätzlich muss das Aspekt die Trace-ID und den Order-Index des Senders an den Empfänger schicken. Matthiessen stellt hierzu fest, dass bei Kommunikation über Nachrichten, die einen Header besitzen, dieser gut für die Übertragung der Trace-ID geeignet ist [Matthiessen 2014]. Auf diese Weise müssen wir den Inhalt der Nachrichten nicht verändern. Die gewählten und vom Aspekt abgefangenen Methoden müssen also als weitere Bedingung Zugriff auf den Header der Anfrage haben. Anfragen und Antworten per `HTTP` werden in Java über die Klassen `HttpServletRequest` und `HttpServletResponse` abgebildet.

Eingehende Kommunikation Das *Model-View-Controller*-Framework von `Spring` ist um ein zentrales Servlet konstruiert, das sogenannte `DispatcherServlet`. Es ist zuständig für die Verarbeitung von eingehenden Anfragen. Einen groben Überblick über die Arbeitsweise des Servlets bietet, angelehnt an die `Spring`-Dokumentation, Abbildung 4.1. Wir können erkennen, dass das `DispatcherServlet` eine Anfrage an den zuständigen Controller weiterleitet, der diese dann bearbeitet. Mit dem dort erstellten Modell wird dann eine Antwort gerendert, die das Servlet zurückgibt.

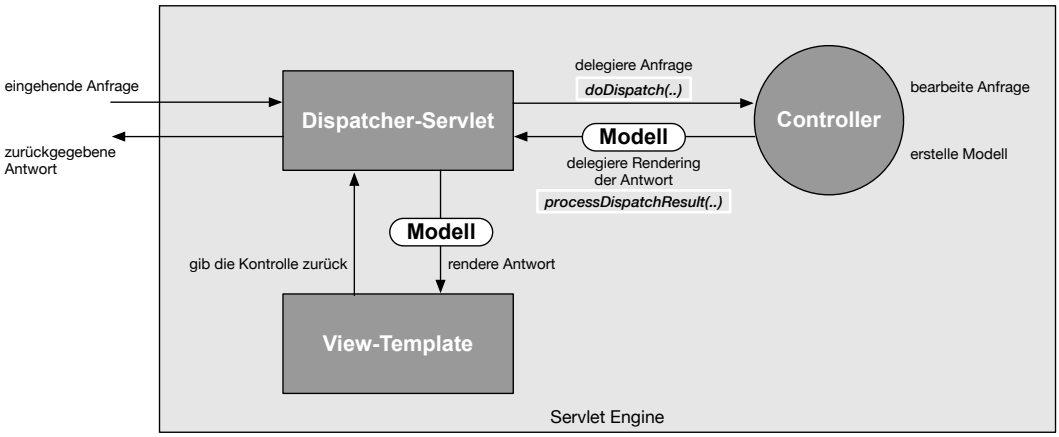


Abbildung 4.1. Funktion des `DispatcherServlets`, nach der `Spring`-Dokumentation[5]

Im Quellcode des `DispatcherServlets` finden wir die Methode `doDispatch`, welche das Weitergeben der `HTTP`-Anfragen an die Controller übernimmt. Als Parameter besitzt es unter anderem ein `HttpServletRequest`. Durch Rekonstruktion des Programmflusses stellt sich heraus, dass die Methode `doDispatch` in der Methode `doService` aufgerufen wird, welche wiederum eine Folge der Methode `processRequest` der abstrakten Klasse `FrameworkServlet` ist, von welcher `DispatcherServlet` erbt. Im `FrameworkServlet` findet `processRequest` ihren

[5] `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html`, zuletzt besucht am 13.09.2017

Ursprung in der Methode `service`. Von diesem Punkt aus finden wir keine weitere Methode des `Spring`-Frameworks, die für den Aufruf von `service` verantwortlich ist und das `HttpServletRequest`-Objekt übergibt. Auf Strubels Erkenntnis, dass sich die Standard-Java-Bibliotheken nicht mit `AspectJ` instrumentieren lassen [Strubel 2017], erübrigt sich eine Suche in diesen.

Wir stellen also fest, dass die Methode `service` der Klasse `FrameworkServlet` die als erstes an eingehender Kommunikation über `HTTP` beteiligte Methode ist, die das Anfrage-Objekt kennt und nicht aus den Standardbibliotheken stammt. Wir werden diese Methode mit unserem Aspekt überwachen und bei Ausführung den Header der Nachricht auf eine Trace-ID und einen Order-Index untersuchen.

Ausgehende Kommunikation Ähnlich dem Vorgehen bei eingehender Kommunikation beginnen wir bei ausgehender Kommunikation mit `Springs` zentraler Klasse für REST-Aufrufe; dem `RestTemplate`[6]. Dieses wird in allen Microservices verwendet. Das `RestTemplate` bietet Methoden für die unterschiedlichen REST-Methoden, etwa beispielsweise `getForEntity` oder `postForObject`. Diese Methoden bekommen einen URI und einen Objekt-Typen, enthalten aber noch kein Aufruf-Objekt, dessen Header manipuliert werden könnte. Wir verfolgen also den Programmfluss, ausgehend von den Hilfsfunktionen.

Jede REST-Methode der Klasse ruft die Methode `execute` auf, die wiederum die Methode `doExecute` verwendet. Innerhalb von `doExecute` wird die von `HttpAccessor` geerbte Methode `createRequest` verwendet, um ein `ClientHttpRequest`-Objekt aus dem URI und der REST-Methode zu erstellen. Der `HttpAccessor` ist die letzte Klasse, die wir finden können, die an ausgehender Kommunikation beteiligt ist und ein Anfrage-Objekt zurückliefert.

In dem von uns erstellen Aspekt werden wir für ein- und ausgehende Kommunikation je einen Pointcut und ein Advice benötigen. Für eingehende Nachrichten erstellen wir den Pointcut `frameworkServletService`, der auf die Ausführung der Methode `org.springframework.web.servlet.FrameworkServlet.service(..)` wartet. Das entsprechende Advice wird vor dem Eintreten des `frameworkServletService`-Pointcuts den Header auslesen und mit der Trace-ID sowie dem Order-Index, sofern diese vorhanden sind, ein `BeforeReceivedEvent` erstellen.

Für ausgehende Nachrichten erstellen wir einen Pointcut namens `requestCreation`, der die Ausführung von `org.springframework.http.client.support.HttpAccessor.createRequest(..)` registriert. Hier wird das Advice nach Rückgabe des `requestCreation`-Pointcuts die aktuelle Trace-ID und den Order-Index in den Header des zurückgegebenen Anfrage-Objekts schreiben.

[6] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html`, zuletzt besucht am 25.09.2017

Konfiguration von AspectJ

AspectJ verwendet für seine Konfiguration eine Datei namens aop.xml. Die als Java-Agent verwendete Kieker-Bibliothek erwartet die Datei, sofern nicht anders angegeben, im META-INF-Verzeichnis der *JAR*, die überwacht werden soll. Die aop.xml legt fest, welche Aspekte von AspectJ verwendet und in welche Klassen diese eingewebt werden sollen.

In der Dokumentation von Kieker wird das FullInstrumentationNoGetterAndSetter-Aspekt für das Monitoring empfohlen. Auch wir werden dieses Aspekt verwenden, um das Verhalten der Microservices in sich zu überwachen. Ergänzend fügen wir unser eigens erstelltes HttpRequestAspekt hinzu. Die Kombination beider Aspekte erfasst damit sämtliche Kommunikation innerhalb der Softwarelandschaft.

Als Klassen, in welche die Aspekte eingewebt werden sollen, wählen wir alle des Pakets de.adesso.∗. Das Spring-Framework wird nicht instrumentiert. Einerseits ist dies nicht Teil der Anforderung, unter anderem da Spring ein Vielfaches größer ist als die Microservices von Symphony, andererseits sind durch die Komplexität des Frameworks Fehler beim Monitoring aufgetreten, die nicht behoben werden konnten. Um dennoch ein- und ausgehende Kommunikation verbinden zu können, instrumentieren wir ausschließlich die Klassen der Pakete org.springframework.http.client.support.∗ und org.springframework.web.servlet.∗. Außerdem hat sich herausgestellt, dass auch das neue HttpRequestAspect selbst instrumentiert werden muss, damit es von AspectJ verwendet werden kann.

Neben den bereits genannten, überwachten Klassen gibt es auch Klassen, die explizit ausgeschlossen werden müssen, da es sonst zu Fehlern beim Monitoring kommt. Durch Springs eigene AOP-Unterstützung werden zur Laufzeit durch Spring erweiterte Klassen erstellt, die dem Namensschema Klassenname$$EnhancerBySpringCGLIB$$12345678 entsprechen. Versucht AspectJ diese Klassen zu instrumentieren, kommt es zu Fehlern und die Applikation wird nicht gestartet. Außerdem gibt es auch bestimmte Klassen von Symphony selbst, die zu Fehlern führen. Wir wollen in dieser Arbeit möglichst wenig am Projekt Symphony verändern und entschließen uns deswegen dazu, die betroffenen Klassen vom Einweben auszuschließen. Tabelle 4.2 listet für jeden Microservice die ausgeschlossenen Klassen auf. Es ist gut möglich, dass diese Liste nicht vollständig oder nicht genau genug ist und dass mögliche Fehler durch Änderungen an den Microservices behoben werden können. Dies soll aber nicht Teil dieser Arbeit sein.

4.2. Übertragen der Daten an ExplorViz

Für die Übertragung der Daten von Kieker zu ExplorViz existieren grundsätzlich zwei Möglichkeiten; entweder offline durch Einlesen von Log-Dateien oder online durch eine direkte Kommunikation der beiden Tools. Für beide Ansätze existieren bereits Implementierungen, die aber veraltet sind und nicht mehr gepflegt werden. Beide Ansätze haben Vor- und Nachteile und können je nach Szenario besser oder schlechter eingesetzt werden.

Tabelle 4.2. Vom Einweben ausgeschlossene Klassen

Authservice
de.adesso.symphony.searchservice.search.*
*..*BySpring*
Frontendproxy
de.adesso.symphony.marketing.data.model.*
de.adesso.symphony.ontology2.model.*
de.adesso.symphony.seachservice.search.*
*..*BySpring*
Marketingservice
de.adesso.symphony.marketing.data.model.*
de.adesso.symphony.marketing.data.classes.*
de.adesso.symphony.ontology2.model..*
*..*BySpring*
Ratingservice
*..*BySpring*
Searchservice
de.adesso.symphony.seachservice.search.*
*..*BySpring*

Bei `Kieker` wird standardmäßig ein Writer verwendet, der die Monitoring-Daten als Log-Dateien ins Dateisystem schreibt. In früheren Experimenten wurde deshalb meist die Offline-Übertragung von existierenden Monitoring-Daten verwendet. Wir beginnen daher im Folgenden mit dem Offline-Ansatz. Dieser funktioniert zu diesem Zeitpunkt bereits mit der aktuellen `ExplorViz`-Version, muss aber für verteilte Kommunikation erweitert werden. Im Anschluss betrachten wir dann den Entwurf für die Online-Visualisierung.

4.2.1. Ansatz 1: Offline-Übertragung

Wir beginnen mit einem Entwurf der Offline-Übertragung. Dabei werden die von `Kieker` erfassten Monitoring-Daten ins Dateisystem geschrieben. Standardmäßig wird für jedes Monitoring ein Ordner im `TEMP`-Verzeichnis des Betriebssystems erstellt. In diesem legt `Kieker` zwei Dateien ab; eine `.map`- und eine `.dat`-Datei. In der `.map`-Datei wird allen Event- beziehungsweise Record-Typen, die der Monitoring-Controller erhält, je eine eindeutige Nummer zugewiesen. Die `.dat`-Datei wird für das Speichern des Überwachungsprotokolls benötigt. Alle Daten der auftretenden Records werden, jeweils getrennt durch ein Semikolon, in Zeilen abgelegt. Am Anfang jeder Zeile steht die in der `.map`-Datei vergebene Nummer des jeweiligen Records.

Um Protokolle des `Kieker`-Monitorings an `ExplorViz` zu übertragen, wurde in der Vergangenheit bereits ein Tool erstellt. Der sogenannte *External Monitoring Logs Adapter*

liest Log-Dateien von Kieker aus dem Dateisystem ein und wandelt die Daten in ExplorViz-Records um. Das Tool ist zum Zeitpunkt der Schreibens mit einer letzten Änderung im Mai 2015 bereits über zwei Jahre alt und bringt für unseren Versuch zwei Probleme. Zum einen müssen die neu erstellten Records für ein- und ausgehende Kommunikation unterstützt und in die äquivalenten ExplorViz-Records umgewandelt werden. Zum anderen besitzt ExplorViz mehr Metadaten über seine überwachten Systeme als Kieker. Das bedeutet, dass Kieker weder den Systemnamen, noch die IP-Adresse, noch die Programmiersprache seiner Applikationen kennt und diese Informationen deshalb nicht aus den Logs ausgelesen werden können. Für die Offline-Übertragung der durch Kieker gespeicherten Monitoring-Daten an ExplorViz im Rahmen dieser Arbeit müssen wir den *External Monitoring Logs Adapter* an unsere Bedürfnisse anpassen.

Anpassungen am External Monitoring Logs Adapter

Bei Nutzung des *External Monitoring Logs Adapter* nach bisherigem Stand bekommt dieser zwei Argumente. Als erstes müssen wir das Verzeichnis angeben, in dem die .log- und .map-Dateien liegen. Als zweites Argument erwartet das Tool einen Applikationsnamen *appName*. Mit *appName* wird hier angegeben, nach welchem Schema die Signaturen der Before-Events umgewandelt werden sollen. Standardmäßig wird das *EPrints*-Schema verwendet. Da in ExplorViz der Begriff der Applikation eine wichtige Rolle spielt, entscheiden wir uns dazu, *appName* zu *toolName* umzubenennen. Damit erweitern wir die einzulesenden Argumente um den Applikationsnamen, den Systemnamen, die IP-Adresse und die Programmiersprache der überwachten Anwendung. Weil diese Informationen nicht verlässlich aus den Logs übernommen werden können, können sie beim Start des *External Monitoring Logs Adapters* manuell definiert werden.

Der Kern des *External Monitoring Logs Adapters* ist der KiekerImporter. Seine Funktion ist grob in Abbildung 4.2 dargestellt. Der KiekerImporter verwendet den FSReader der Analyse-Komponente von Kieker, der Logs vom Dateisystem einlesen kann, und verbindet diesen über die Pipe-und-Filter-Architektur des AnalysisControllers mit dem KiekerToExplorVizTransformFilter. Dort werden die Daten aufbereitet und an den ExplorViz-Server geschickt.

Die von uns vorzunehmenden Änderungen sind in Abbildung 4.2 in grün markiert. Wie bereits erwähnt benennen wir *appName* um und geben dann die Metadaten an den KiekerImporter und weiter an den KiekerToExplorVizTransformFilter. Der AnalysisController überträgt Objekte von einem Plugin zu einem anderen. Um neben den aufgetretenen Events auch mögliche, im Kieker-Log gespeicherte, Einträge zu Kieker-Metadaten zu berücksichtigen, ergänzen wir die Verbindung zwischen dem FSReader und dem KiekerToExplorVizTransformFilter um eine weitere Pipe für KiekerToExplorVizMetaDataRecords. Da bisher der FSReader der Kieker-Analyse verwendet wurde und wir diesen für den *Logs Adapter* nicht anpassen können, erstellen wir eine Kopie der Klasse und benennen diese KiekerToExplorVizFSReader.

Für die Übertragung der Metadaten vom KiekerToExplorVizFSReader zum KiekerTo-

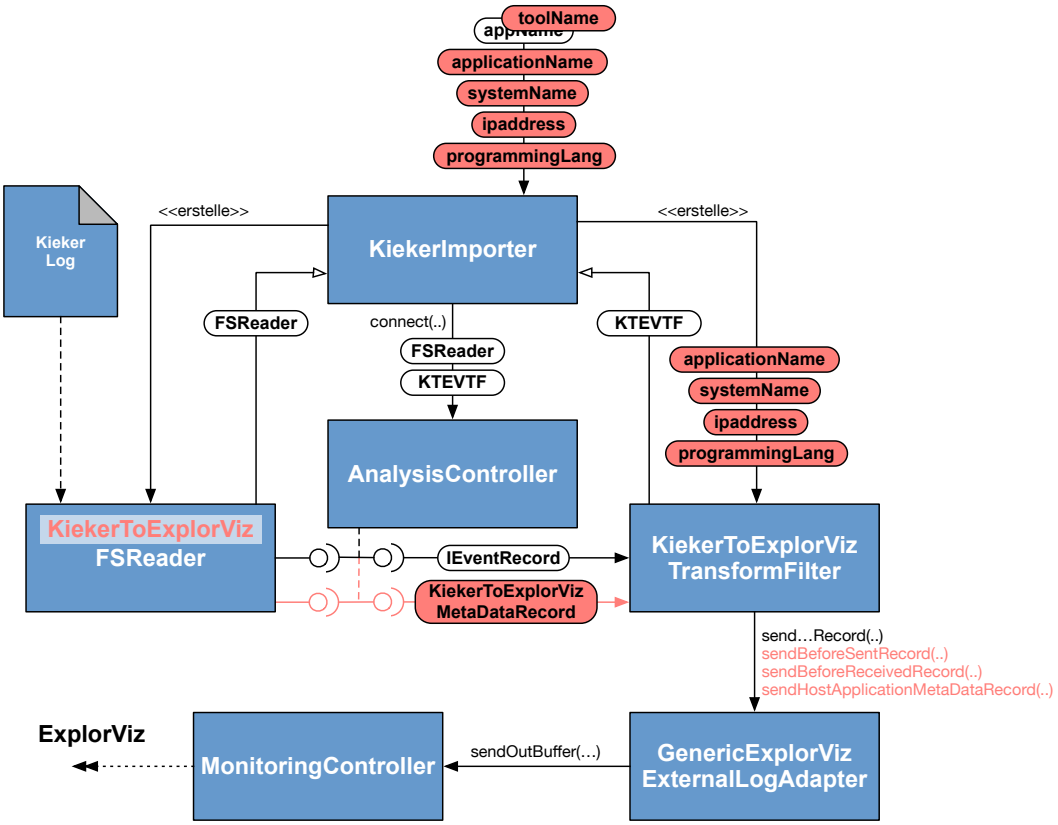


Abbildung 4.2. Grobe Funktionsweise des `KiekerImporters` im *External Monitoring Logs Adapter* mit Änderungsvorschlägen in grün.

`ExplorVizTransformFilter` eignet sich das in `Kieker` bereits vorhandene `KiekerMetadataRecord` nicht, da das `AbstractPlugin`, von welchem der `FSReader` erbt, `KiekerMetadataRecords` verwirft. Aus diesem Grund verpacken wir das `KiekerMetadataRecord` im `KiekerToExplorVizFSReader` in ein `KiekerToExplorVizMetaDataRecord`, welches identisch mit dem `KiekerMetadataRecord` ist, aber vom `AbstractPlugin` ignoriert wird und so verschickt werden kann.

Der `KiekerToExplorVizTransformFilter` wird um einen Input-Port für `KiekerToExplorVizMetaDataRecords` erweitert, welcher dessen Metadaten verarbeitet. Die über die Kommandozeile mitgegebenen Daten überschreiben dabei die Metadaten, die von `Kieker` eingelesen werden.

Als letztes passen wir den `GenericExplorVizExternalLogAdapter` so an, dass er auch `BeforeSent-`, `BeforeReceived-` und `HostApplicationMetaDataRecords` an `ExplorViz` senden kann und rufen diese Methoden im `KiekerToExplorVizTransformFilter` auf.

Vorteile einer Offline-Übertragung

Die Offline-Übertragung durch `Kieker` bietet den Vorteil, dass Daten unabhängig von einem laufenden `ExplorViz`-Server gesammelt werden können. Sie ermöglicht es, auf Basis von erstellten Log-Dateien eine Visualisierung zu starten.

Nachteile einer Offline-Übertragung

Eine Offline-Übertragung von `Kieker`-Monitoring-Daten an `ExplorViz` bringt allerdings auch erhebliche Nachteile. So ist es nicht möglich, das Verhalten einer Applikation in Echtzeit zu beobachten und es ist nicht möglich, mit dem System während der Visualisierung zu interagieren. Außerdem verwendet ExplorViz für die Aktualisierung der Modelle ein Zeitfenster von zehn Sekunden. Alle Daten, die innerhalb eines Zeitfensters eintreffen, werden in einen Systemzustand aggregiert. Wenn wir eine Log-Datei von `Kieker` senden, dann erreichen alle Records des Monitorings `ExplorViz` zur selben Zeit, was dafür sorgt, dass sämtliches Verhalten der überwachten Applikation in einem Zustand dargestellt wird und die Zeitleiste in `ExplorViz` nicht nutzbar ist.

4.2.2. Ansatz 2: Online-Übertragung

In diesem Abschnitt betrachten wir den Ansatz einer Online-Übertragung der Daten von `Kieker` zu `ExplorViz`. Online bedeutet, dass gesammelte Daten sofort an `ExplorViz` geschickt werden. Entgegen der Offline-Übertragung eignet sich eine Online-Übertragung damit für eine Visualisierung in Echtzeit – sofern von der Größe der Zeitfenster abgesehen wird.

Das Konzept eines Online-Übertragung ist weniger komplex als das der Offline-Übertragung. Wie bereits in Abschnitt 3.3.1 beschrieben, lässt sich in der Konfiguration von `Kieker` ein Writer angeben, der die anfallenden Records in ein bestimmtes Ziel schreibt. Statt des bisher verwendeten `AsciiFileWriters`, der die Monitoring-Daten als Log-Dateien ins `TEMP`-Verzeichnis schreibt, können wir einen eigenen Writer definieren, der die Daten direkt an `ExplorViz` schickt.

Auch für diesen Ansatz existiert bereits eine Implementierung; der `ExplorVizTcpWriter` des `Kieker`-Projekts. Der bestehende Writer ist genau wie die Lösung zur Offline-Übertragung nicht mehr aktuell und müsste erweitert werden, um die Metadaten und die neuen Record-Typen für verteilte Kommunikation zu unterstützen. Wir entscheiden uns an dieser Stelle aber gegen eine Erweiterung des `ExplorVizTcpWriters` und für eine Neuimplementierung eines `KiekerToExplorVizWriters`. Der Hauptgrund dafür ist, dass der alte `ExplorVizTcpWriter` das Senden der Records selbst übernimmt. Dafür müssen sowohl die Größen der `ExplorViz`-Records bekannt und im Code enthalten sein, als auch eine eigene Socket-Verbindung aufgebaut werden.

Diese Aufgaben werden aber bereits vom Monitoring-Controller von `ExplorViz` erfüllt. Der `ExplorViz`-Monitoring-Controller kennt die Größen aller Records und besitzt die Fähigkeit eine Verbindung zum `ExplorViz`-Server aufzubauen, um diesem Daten aus dem

ExplorViz-Monitoring zu senden. Dies nutzt auch der GenericExplorVizExternalLogAdapter des *External Monitoring Logs Adapters* der Offline-Übertragung aus, indem er die Daten für die ExplorViz-Records in einen Buffer schreibt und diesen dann über die sendOutBuffer-Methode an den MonitoringController von ExplorViz gibt.

Um eine gute Teilung der Belange einzuhalten und zu verhindern, dass beispielsweise bei einer Änderung der Socket-Verbindung des ExplorViz-Servers an mehreren Stellen Anpassungen vorgenommen werden müssen, entschließen wir uns dafür, auch bei der Online-Übertragung den Monitoring-Controller von ExplorViz einzubinden und zu verwenden. Abbildung 4.3 gibt einen zum Offline-Ansatz vergleichbaren Überblick über die Funktionsweise des KiekerToExplorVizWriters.

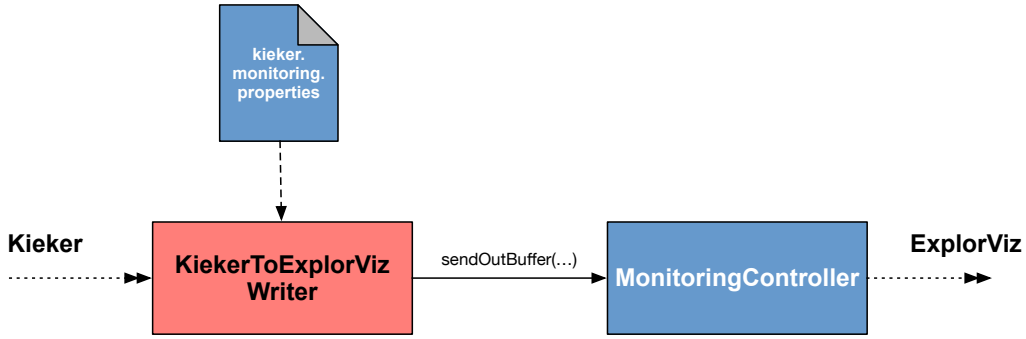


Abbildung 4.3. Grobe Funktionsweise des KiekerToExplorVizWriters.

Eine letzte Aufgabe besteht dann darin, auch die benötigten Metadaten als HostApplicationMetadataRecord an ExplorViz zu übermitteln. Statt einer Übergabe in der Kommandozeile können wir diese Daten in der kieker.monitoring.properties, der Kieker-Konfigurationsdatei, definieren. Diese ist beim Monitoring mit Kieker spezifisch für die überwachte Applikation und eignet sich deshalb gut für Daten wie den Systemnamen, die Programmiersprache oder die IP-Adresse. Bei einem eintreffenden KiekerMetadataRecord senden wir einen HostApplicationMetadataRecord an ExplorViz.

Kapitel 5

Implementierung

Aufbauend auf dem Entwurf in Kapitel 4 wird in diesem Kapitel die Implementierung beschrieben. Dazu finden wir erneut eine Unterteilung in das Monitoring mit `Kieker` und die beiden Übertragungswege.

5.1. Monitoring mit Kieker

5.1.1. Erfassen von verteilter Kommunikation

Um Daten über verteilte Kommunikation zwischen den Microservices speichern zu können, erstellen wir zwei neue Event-Records, deren Aufbau dem der äquivalenten `ExplorViz`-Records folgt. Tabelle 5.1 stellt die Felder der beiden Events dar.

Tabelle 5.1. Felder des `BeforeSent`- und des `BeforeReceivedEvent`-Records für `Kieker`

BeforeSentEvent	BeforeReceivedEvent
long **timestamp**	*long* **timestamp**
long **traceId**	*long* **callerTraceId**
int **orderIndex**	*int* **callerOrderIndex**
String **technology**	*long* **traceId**
	int **orderIndex**

Die Records haben keine weiteren Besonderheiten in ihrer Implementierung. Sie werden später in die zugehörigen `ExplorViz`-Records umgewandelt.

5.1.2. Instrumentieren der Microservices

Anpassen der Gradle-Dateien

Für die Instrumentierung der einzelnen Microservices von `Symphony` müssen diese mit dem `Kieker`-Java-Agenten gestartet werden. Dafür erweitern wir die `build.gradle`-Dateien aller Microservices um das in Abschnitt 4.1.2 bereits vorgestellte Schema. Zusätzlich machen wir einige Ergänzungen, die sich im Laufe der Implementierung als wichtig herausgestellt haben, dargestellt in Listing 5.1.

Listing 5.1. Anpassung der Gradle-Dateien von Symphony für das Monitoring durch Kieker

```
1  configurations {
2     kieker
3     all*.exclude group : "ch.qos.logback"
4  }
5
6  dependencies {
7     kieker "kieker:kieker-aspectj:1.13"
8     compile configurations.kieker.dependencies
9     compile "explorviz:monitoring:1.0"
10    compile "kieker.monitoring.symphony:symphony-monitoring:1.0"
11 }
12
13 if(hasProperty("kieker")) {
14    bootRun {
15       jvmArgs = [
16          "-javaagent:${configurations.kieker.singleFile}"
17          "-noverify"
18          "-Dkieker.monitoring.skipDefaultAOPConfiguration=true"
19       ]
20    }
21 }
```

Zum einen verwenden sowohl Symphony als auch Kieker unterschiedliche Implementierungen der Simple Logging Facade for Java (SLF4J)[1], was zwar nicht zu einem Fehler aber zu einer Warnung beim Start der Microservices führt. Indem wir in Zeile 3 die Gruppe ch.qos.logback von allen Phasen des Programms ausschließen, können wir diese Problem lösen.

Da wir in unser Implementierung Teile des ExplorViz-Monitorings benötigen, fügen wir auch die ExplorViz-Monitoring-Bibliothek in Zeile 9 als Abhängigkeit hinzu. Alle von uns in der Implementierung erstellten Klassen befinden sich in einem Paket namens symphony-monitoring, welches wir in Zeile 10 ebenfalls als Abhängigkeit einfügen.

Durch einen Fehler in der verwendeten AspectJ-Version traten Probleme bei der Verifizierung des Byte-Codes durch die JVM auf. Um die Microservices dennoch starten zu können, verwenden wir in Zeile 17 den Parameter *noverify*, der die Byte-Code-Verifizierung unterbindet.

Das Spring-Framework besitzt bereits einen eigenen *AspectJ loader*, welcher dessen Konfigurationsdateien findet und lädt. Deshalb wird der in Kieker enthaltene *AspectJ loader* nicht benötigt und wird in Zeile 18 abgeschaltet.

[1] https://www.slf4j.org/, zuletzt besucht am 16.10.2017

Wir können `Gradle` sogenannte Projekteigenschaften über die Kommandozeile mitgeben. Wir verwenden eine solche Eigenschaft in Zeile 13, um abzufragen, ob der Dienst mit den `Kieker`-spezifischen Einstellungen gestartet werden soll oder nicht. Projekteigenschaften werde über den Kommandozeilenparameter -P eingegeben werden. Für die Eigenschaft *kieker* würde man Gradle also über den Befehl `gradle bootRun -Pkieker` starten.

Aspekt für verteilte Kommunikation

Um die verteilte Kommunikation erfassen zu können, erstellen wir ein eigenes Aspekt namens `HttpRequestAspect`. Mittels `AspectJ` ist dieses in der Lage, die für das Senden und Empfangen von Nachrichten zuständige Methoden zu überwachen. Die Klasse `HttpRequestAspect` bekommt zwei Pointcuts, die in Listing 5.2 dargestellt sind. Der erste Pointcut beschreibt die Ausführung der Methode `createRequest` des `HttpAccessors` bei ausgehender Kommunikation über das `RestTemplate`. Der zweite Pointcut spiegelt die Methode `service` des `FrameworkServlets` wieder.

Listing 5.2. Pointcuts des `HttpRequestAspects`.

```
1 @Pointcut("execution(protected * org.springframework.http.client.support.
      HttpAccessor.createRequest(..))")
2 private void requestCreation() {}
3
4 @Pointcut("execution(protected * org.springframework.web.servlet.FrameworkServlet.
      service(..))")
5 private void frameworkServletService() {}
```

Bei einer ausgehenden Verbindung fügen wir dem Header der Nachricht die Trace-ID und den Order-Index des aktuellen Threads hinzu. Die Methode `createRequest` wird vom `RestTemplate` verwendet, um das Anfrage-Objekt zu erstellen. Durch das Advice `doAfterRequestCreation` können wir das zurückgegebene Objekt der Methode manipulieren. Listing 5.3 zeigt, wie das durch die Annotation *AfterReturning* markierte Advice die `TraceRegistry` verwendet, um die Trace-ID und den Order-Index abzufragen und dann die Header ergänzt. Zusätzlich wird dem `MonitoringController` in Zeile 11 ein `BeforeSentEvent` mit den benötigten Daten geschickt.

Geht eine Nachricht eines anderen Microservice ein, so müssen wir die Header überprüfen und gegebenenfalls die Trace-ID und den Order-Index des Anderen verarbeiten. Dazu erstellen wir das in Listing 5.4 abgebildete, mit *Before* annotierte Advice `doBeforeDispatcherService`. Dieses erhält als Parameter diejenigen Parameter, die auch die Methode `service` erhält und kann deshalb die Header des Anfrage-Objekts auslesen. Sofern die Überprüfung der *traceId* in Zeile 5 positiv ist, senden wir ein `BeforeReceivedEvent` an den `MonitoringController`, der sowohl die Informationen über den Trace des aktuellen Dienstes als auch die des sendenden Dienstes enthält.

Listing 5.3. Advice nach Rückgabe der Methode `createRequest` bei ausgehender Kommunikation.

```
1  @AfterReturning(pointcut = "requestCreation()", returning = "request")
2  public void doAfterRequestCreation(ClientHttpRequest request) {
3     long traceId = TRACEREGISTRY.getTrace().getTraceId();
4     int orderIndex = TRACEREGISTRY.getTrace().getNextOrderId();
5
6      HttpHeaders headers = request.getHeaders();
7
8      headers.add("kieker-traceId", Long.toString(traceId));
9      headers.add("kieker-orderIndex", Integer.toString(orderIndex));
10
11     CTRLINST.newMonitoringRecord(new BeforeSentEvent(TIME.getTime(), traceId,
           orderIndex, "REST"));
12 }
```

Listing 5.4. Advice vor Ausführung der Methode `service` bei eingehender Kommunikation.

```
1  @Before("frameworkServletService() && args(request, ..)")
2  public void doBeforeDispatcherService(HttpServletRequest request) {
3      String traceIdStr = request.getHeader("kieker-traceId");
4      String orderIndexStr = request.getHeader("kieker-orderIndex");
5      if (traceIdStr != null) {
6          try {
7              traceId = Long.parseLong(traceIdStr);
8              int orderIndex = Integer.parseInt(orderIndexStr);
9
10             if(TRACEREGISTRY.getTrace() == null) {
11                 TRACEREGISTRY.registerTrace();
12             }
13
14             CTRLINST.newMonitoringRecord(new BeforeReceivedEvent(TIME.getTime(),
                   traceId, orderIndex, TRACEREGISTRY.getTrace().getTraceId(),
                   TRACEREGISTRY.getTrace().getNextOrderId()));
15         } catch (final NumberFormatException exc) {}
16     }
17 }
```

Konfiguration von AspectJ

Die Konfiguration von AspectJ wurde bereits ausführlich in Abschnitt 4.1.2 beschrieben. Obwohl der *AspectJ loader* von Spring die aop.xml-Datei auch außerhalb des META-INF-Verzeichnisses eines Microservice findet, legen wir die Datei aus Konventionsgründen dort ab. Listing 5.5 stellt die aop.xml des *Authservice* dar. Unter dem Knoten *weaver* werden alle *includes* und *excludes* definiert, in die AspectJ einweben soll. Mit *aspects* definieren wir die Aspekte, die AspectJ verwenden soll. Neben dem von Kieker empfohlenen Standard-Aspekt geben wir dort das von uns erstellte HttpRequestAspect an.

Listing 5.5. aop.xml des *Authservice*.

```
1  <aspectj>
2      <weaver options="">
3          <include within="de.adesso..*"/>
4          <include within="kieker.monitoring.symphony.*"/>
5          <include within="org.springframework.http.client.support.*"/>
6          <include within="org.springframework.web.servlet.*"/>
7          <exclude within="de.adesso.symphony.searchservice.search.*"/>
8          <exclude within="*..*BySpring*"/>
9      </weaver>
10
11     <aspects>
12         <aspect name="kieker.monitoring.probe.aspectj.flow.operationExecution.
               FullInstrumentationNoGetterAndSetter"/>
13         <aspect name="kieker.monitoring.symphony.HttpRequestAspect"/>
14     </aspects>
15 </aspectj>
```

Bereitstellen der Bibliotheken

Das Symphony-Projekt nutzt neben Online-Quellen auch ein lokales *libs* Repositorium im Projektverzeichnis für benötigte Abhängigkeiten. Um die Bibliotheken von Kieker und ExplorViz verwenden zu können, werden diese in den build.gradle-Dateien der Microservices als Abhängigkeiten angegeben. Damit Gradle diese nun finden kann, müssen wir beide Tools in den definierten Repositorien verfügbar sein. Zwar ist Kieker in der Version 1.12 über *JCenter* verfügbar, allerdings möchten wir die neueste Version 1.13 verwenden. ExplorViz ist gar nicht über eine Online-Quelle beziehbar.

Um Kieker und ExplorViz im lokalen Repositorum von Symphony zu installieren, erweitern wir die beiden Projekte um je eine pom.xml-Datei, wie in Anhang C abgebildet. Dann können wir Maven verwenden um über den Befehl mvn install:install-file die Installation vorzunehmen. Mittels Maven stellen wir auch die von uns im Rahmen dieser Arbeit

erstellten Dateien im `symphony-monitoring`-Paket für `Gradle` zur Verfügung. Die POM-Datei dieses Pakets befindet sich im Projektverzeichnis im Repositorium zu dieser Arbeit.

5.2. Übertragen der Daten an ExplorViz

Wie in Kapitel 4 teilen wir auch in der Implementierung die Übertragung der Daten von `Kieker` an `ExplorViz` in zwei Ansätze auf, die nachfolgend getrennt voneinander beschrieben werden.

5.2.1. Ansatz 1: Offline-Visualisierung

Main.java Wir beginnen bei der Implementierung der Offline-Übertragung mit der Anpassung der `Main`-Klasse des *External Monitoring Logs Adapter*s. Dort ergänzen wir die bereits erwähnten Metadaten. Wie Listing 5.6 zeigt, sollen bei einer Mindestzahl von sechs Argumenten auch der Applikationsname, der Systemname, die IP-Adresse und die Programmiersprache eingelesen werden. Danach können wir die Daten an den `KiekerImporter` weitergeben.

Listing 5.6. Ausschnitt aus der angepassten `Main`-Klasse des *External Monitoring Logs Adapter*s.

```
1 if (args.length > 5) {
2     applicationName = args[2];
3     systemName = args[3];
4     ipaddress = args[4];
5     programmingLanguage = args[5];
6 }
7
8 final AbstractImporter currentImporter = new KiekerImporter();
9 currentImporter.importLogs(inputFolder, toolName, applicationName systemName,
      ipaddress, programminglanguage);
```

KiekerImporter.java Um die neuen Metadaten im `KiekerImporter` verarbeiten zu können, müssen wir dessen `importLogs`-Methode anpassen. Zuerst einen benennen wir das vorher verwendete *appName* zu *toolName* um. Dann übergeben wir die Metadaten an den `KiekerToExplorVizTransformFilter`, den wir später auch anpassen werden. Wie bereits in Abschnitt 4.2.1 beschrieben, tauschen wir den `FSReader` gegen den `KiekerToExplorViz-FSReader` aus, weil wir den `FSReader` aus `Kieker` nicht verändern können. Für die Übertragung von `KiekerToExplorVizMetaDataRecords` ergänzen wir wie in Listing 5.7 abgebildet den `AnalysisController` um eine weitere Verbindung. Zeile 1 zeigt die bereits bestehende Verbindung zwischen dem `FSReader` und dem `KiekerToExplorVizTransformFilter`. In Zeile 3 ist die Verbindung zum neuen Port für `KiekerToExplorVizMetaDataRecords` zu sehen.

Listing 5.7. Ausschnitt aus der angepassten `KiekerImporter`-Klasse des *External Monitoring Logs Adapter*s.

```
1 analysisInstance.connect(reader, KiekerToExplorVizReader.OUTPUT_PORT_NAME_RECORDS,
      transformFilter, KiekerToExplorVizTransformFilter.INPUT_PORT_NAME_KIEKER);
2
3 analysisInstance.connect(reader, KiekerToExplorVizReader.OUTPUT_PORT_NAME_RECORDS,
      transformFilter, KiekerToExplorVizTransformFilter.
     INPUT_PORT_NAME_KIEKER_META_DATA);
```

KiekerToExplorVizFSReader.java Im nächsten Schritt erstellen wir eine Kopie des `FSReaders` von `Kieker` im `symphony-monitoring`-Paket. Diese nennen wir `KiekerToExplorVizFSReader`. Wir müssen in dieser Klasse lediglich eine kleine Ergänzung vornehmen. `Reader`-Plugins wie der `FSReader` versenden ihre Records über ihre Ausgabe-Ports mit der geerbten `deliver`-Methode des `AbstractReaderPlugin`s. Diese Methode verwirft `KiekerMetadataRecords` allerdings, sodass wir Metadaten aus `Kieker` nicht in deren üblicher Gestalt übertragen können. Wir ergänzen vor dem Aufruf von `deliver` deshalb eine Abfrage, die `KiekerMetadataRecords` in `KiekerToExplorVizMetadataRecords` verpackt. Die Abfrage wird in Listing 5.8 dargestellt. Die Methode `deliver` weiß nun nicht mehr, dass es sich um Metadaten handelt und schickt diese als normalen Record an den `KiekerToExplorVizTransformFilter`.

Listing 5.8. Ausschnitt aus der `KiekerToExplorVizFSReader`-Klasse des *External Monitoring Logs Adapter*s.

```
1 if (record instanceof KiekerMetadataRecord) {
2     record = new KiekerToExplorVizMetadataRecord( ((KiekerMetadataRecord) record).
          toArray());
3 }
4
5 super.deliver(OUTPUT_PORT_NAME_RECORDS, record);
```

KiekerToExplorVizMetaDataRecord.java Das `KiekerToExplorVizMetaDataRecord` unterscheidet sich bis auf den Namen nicht vom `KiekerMetadataRecord`. Es genügt deshalb eine einfache Kopie in unser `symphony-monitoring`-Paket.

KiekerToExplorVizTransformFilter.java Auch am `KiekerToExplorVizTransformFilter` müssen wir einige Änderungen vornehmen. Zuerst soll der Konstruktor die Metadaten vom `KiekerImporter` entgegennehmen. Wir ergänzen private Felder für alle neuen Metadaten und beschreiben diese beim Erstellen einer Instanz der Klasse. Damit der `KiekerToExplorVizTransformFilter` die von uns erstellten `KiekerToExplorVizMetaDataRecords` emp-

fangen kann, erweitern wir diesen um einen zusätzlichen Input-Port. Wie Listing 5.9 zeigt, geben wir in den Zeilen 11 bis 14 Standard-Werte für die Metadaten an und nutzen den *hostName* des eingehenden Records als Applikationsnamen, für den Fall, dass keine Metadaten über den Konstruktur übergeben wurden. Sofern der `KiekerToExplorVizTransformFilter` aber mit Metadaten erstellt wurde, funktioniert ein eingehendes `KiekerToExplorVizMetaDataRecord` nur als Auslöser für das Senden der Metadaten an `ExplorViz`. Letzteres geschieht in Zeile 17.

Listing 5.9. Ausschnitt aus der `KiekerToExplorVizTransformFilter`-Klasse des *External Monitoring Logs Adapter*s.

```
1  @InputPort(
2      name = KiekerToExplorVizTransformFilter.INPUT_PORT_NAME_KIEKER_META_DATA,
3      description = "",
4      eventTypes = {KiekerToExplorVizMetadataRecord.class}
5  )
6  public void inputKiekerMetaDataRecord(final KiekerToExplorVizMetadataRecord
       kiekerRecord) {
7      final KiekerToExplorVizMetadataRecord kiekerMetaDataRecord = kiekerRecord;
8
9      long timestamp = kiekerMetaDataRecord.getLoggingTimestamp();
10     String hostName = kiekerMetaDataRecord.getHostname();
11     String systemName = this.systemName != null ? this.systemName : "";
12     String application = applicationName != null ? applicationName : hostName;
13     String ipaddress = this.ipaddress != null ? this.ipaddress : "127.0.0.1";
14     String programmingLanguage = this.programmingLanguage != null ?
15         this.programmingLanguage : "Java";
16
17     GenericExplorVizExternalLogAdapter.sendHostApplicationMetaDataRecord(
18         timestamp, systemName, ipaddress, hostName, application,
               programmingLanguage
19     );
20 }
```

Zum Schluss erweitern wir die Fallunterscheidung hinter dem zweiten Input-Port um einen Fall für ein eingehendes `BeforeSentEvent` und einen für ein eingehendes `BeforeReceivedEvent`. In beiden Fällen wird wie sonst auch der `GenericExplorVizExternalLogAdapter` zum Senden der Records an `ExplorViz` verwendet.

GenericExplorVizExternalLogAdapter Die letzte Klasse, die wir für die Offline-Übertragung der Monitoring-Daten anpassen müssen, ist der `GenericExplorVizExternalLogAdapter`. Sie ist zuständig für das Senden der Informationen an den `ExplorViz`-Server. Um `BeforeSent-` und `BeforeReceivedEvents` sowie `HostApplicationMetaDataRecords` verschi-

cken zu können, ergänzen wir die Klasse um drei Funktionen; `sendBeforeSentRecord`, `sendBeforeReceivedRecord` und `sendHostApplicationMetaDataRecord`. Die Methode `sendBeforeSentRecord` ist in Listing 5.10 abgebildet. Wir beschreiben den Buffer mit den Daten in der Reihenfolge, in der sie `ExplorViz` erwartet, und senden diesen ab. Die anderen Methoden funktionieren auf die gleiche Weise.

Listing 5.10. Ausschnitt aus der `GenericExplorVizExternalLogAdapter`-Klasse des *External Monitoring Logs Adapter*s.

```
1 public static void sendBeforeSentRecord(final long timestamp, final long
      ownTraceId, final int ownOrderId,
2 final String technology) {
3     explorVizBuffer.put(BeforeSentRemoteCallRecord.CLAZZ_ID);
4     explorVizBuffer.putLong(ownTraceId);
5     explorVizBuffer.putInt(ownOrderId);
6     explorVizBuffer.putInt(MonitoringStringRegistry.getIdForString(technology));
7
8     sendBufferIfHasElements(timestamp);
9 }
```

5.2.2. Ansatz 2: Online-Visualisierung

Für den Ansatz zur Online-Visualisierung genügt es, einen Writer zu `ExplorViz` zu erstellen und diesen im Monitoring von `Kieker` zu verwenden.

Anpassen des KiekerToExplorVizWriters

Als Grundlage für den Writer, den wir erstellen wollen, dient der `ExplorVizTcpWriter` von `Kieker`. Einen großen Teil des alten Codes wird allerdings bereits vom `MonitoringController` von `ExplorViz` übernommen und eine parallele Pflege beider Komponenten scheint nicht sinnvoll. Darum kopieren wir die Klasse als `KiekerToExplorVizWriter` in unser `symphony-monitoring`-Paket. Wir entfernen die `send`-Methode und allen Code, der mit der Socket-Verbindung zu `ExplorViz` zu tun hat. Die drei Methoden `onNewRegistryEntry`, `onStarting` und `onTerminating` werden vom `AbstractMonitoringWriter` geerbt, bleiben aber leer, da wir weder zum Start noch zum Ende noch bei einem Eintrag der String-Registrierung etwas zu tun haben. Einzig füllen wir die geerbte Methode `writerMonitoringRecord`, die eingehende Records an `ExplorViz` schicken soll.

Zu diesem Zweck führen wir eine Fallunterscheidung ein, wie sie auch im `KiekerToExplorVizTransformFilter` verwendet wird und kombinieren diese mit der Verwendung des `MonitoringControllers` von`ExplorViz`, wie es im `GenericExplorVizExternalLogAdapter` geschieht. Wie Listing 5.11 zeigt, beschreiben wir beispielsweise bei Eingehen eines `BeforeSentEvents` den Buffer und senden diesen ab. Die Methode `sendBufferIfHasElements`

Listing 5.11. Ausschnitt aus der `KiekerToExplorVizWriter`-Klasse.

```
} else if (kiekerRecord instanceof BeforeSentEvent) {
    final BeforeSentEvent beforeSentEvent = (BeforeSentEvent) kiekerRecord;

    explorVizBuffer.put(BeforeSentRemoteCallRecord.CLAZZ_ID);
    explorVizBuffer.putLong(beforeSentEvent.getTraceId());
    explorVizBuffer.putInt(beforeSentEvent.getOrderIndex());
    explorVizBuffer.putInt(
        MonitoringStringRegistry.getIdForString(beforeSentEvent.getTechnology())
    );

    sendBufferIfHasElements(beforeSentEvent.getTimestamp());
} ...
```

übergibt den Buffer an die Methode `sendOutBuffer` des `ExplorViz-MonitoringControllers`. Um die Metadaten an `ExplorViz` senden zu können, lesen wir diese wie in Listing 5.12 gezeigt im Konstruktor der Klasse aus der `Kieker`-Konfiguration wie folgt aus.

Listing 5.12. Auslesen der `Kieker`-Konfiguration im `KiekerToExplorVizWriter`.

```
appName = configuration.getStringProperty(CONFIG_APPLICATION_NAME, "<UNKNOWN-
    APPLICATION>");
```

Konfiguration von Kieker

Für die Verwendung des von uns erstellten `KiekerToExplorVizWriters` müssen wir diesen zusammen mit den Metadaten in der `kieker.monitoring.properties`-Datei der jeweiligen zu überwachenden Applikation definieren. Wir legen jedem der Microservices die Vorlage dieser Datei ins `META-INF`-Verzeichnis im `resources`-Ordner und passen diese wie in Listing 5.13 für den *Authservice* abgebildet an. Als `kieker.monitoring.writer` wählen wir unseren neuen Writer. Die Metadaten werden mit dem Präfix `kieker.monitoring.writer.exporviz.KiekerToExplorVizWriter` angelegt, welches wir ebenfalls im `KiekerToExplorVizWriter` definiert haben. Wichtig ist, dass der vorige Writer aus der Konfigurations-Datei entfernt wird, da `Kieker` nur einen Writer zur Zeit verwenden kann.

5.3. Ergebnis der Implementierung

Nach abgeschlossener Implementierung können wir sowohl die Online- als auch die Offline-Übertragung verwenden, um Monitoring-Daten von `Kieker` zu `ExplorViz` zu übertragen und diese dort zu visualisieren. In beiden Fällen erhalten wir dasselbe Ergebnis. Dieses wird in

Listing 5.13. Exemplarische `Kieker`-Konfiguration des *Authservice*.

```
1 kieker.monitoring.writer=kieker.monitoring.writer.explorviz.
      KiekerToExplorVizWriter
2 kieker.monitoring.writer.explorviz.KiekerToExplorVizWriter.flush = true
3 kieker.monitoring.writer.explorviz.KiekerToExplorVizWriter.hostname = 8083
4 kieker.monitoring.writer.explorviz.KiekerToExplorVizWriter.applicationName =
      AuthService
5 kieker.monitoring.writer.explorviz.KiekerToExplorVizWriter.systemName = Symphony
6 kieker.monitoring.writer.explorviz.KiekerToExplorVizWriter.language = Java
7 kieker.monitoring.writer.explorviz.KiekerToExplorVizWriter.ipAddress = 127.0.0.1
```

Abbildung 5.2 und Abbildung 5.1 abgebildet. Abbildung 5.1 stellt einen Ausschnitt der 3D-Visualisierung des *Ratingservice* dar. Am hinteren rechten Ende der Applikation erkennen wir den ausgehenden Port zum *Authservice*. Abbildung 5.2 zeigt ausgewählte Microservices von `Symphony` in der Landschaftsansicht. In dem gezeigten Szenario kommuniziert der *Frontendproxy* mit dem *Ratingservice*, der wiederum mit dem *Authservice* kommuniziert.

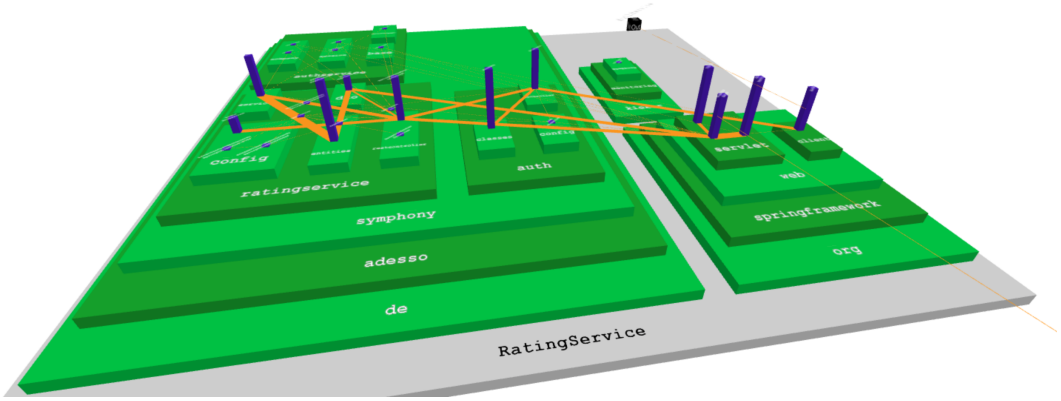


Abbildung 5.1. Ergebnis der Visualisierung des *RatingService* von `Symphony`.

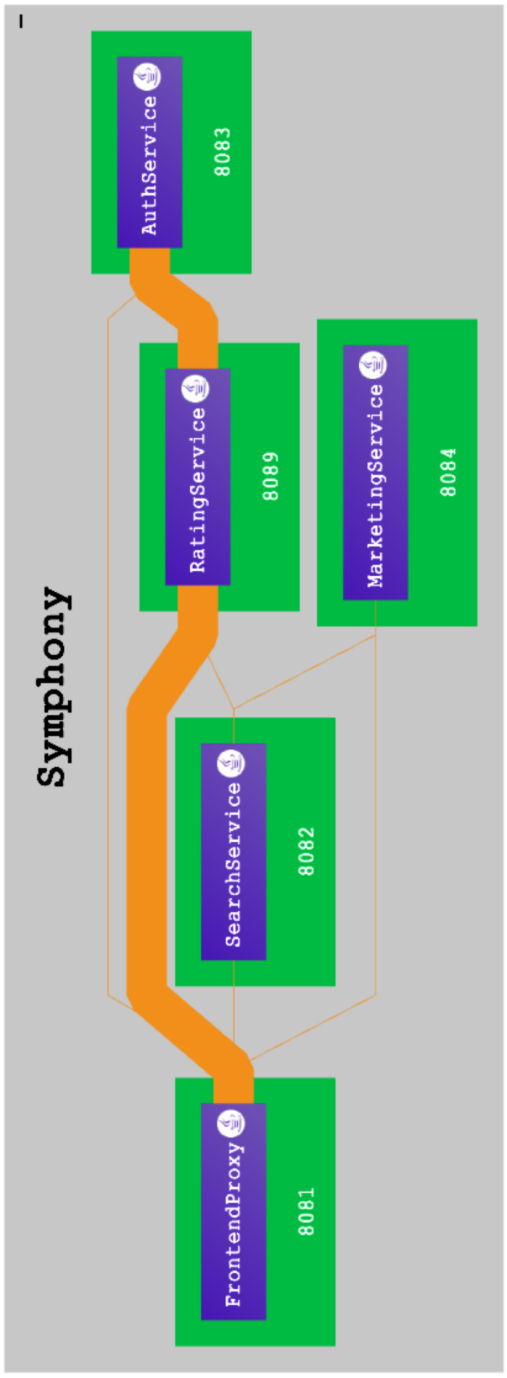


Abbildung 5.2. Ergebnis der Visualisierung der Softwarelandschaft von Symphony.

Kapitel 6

Evaluation

Dieses Kapitel beschreibt die Evaluation der Implementierung. Wir wollen ermitteln, ob eine Visualisierung der Microservices von `Symphony` mit `Kieker` und `ExplorViz` zum Verständnis von `Symphony` beitragen kann. Im Folgenden werden deshalb unsere Ziele und die gewählte Methodik definieren, bevor wir den Versuchsaufbau und dessen Durchführung beschreiben, sowie die Ergebnisse diskutieren.

6.1. Ziele

Das Ziel der Evaluation ist es, herauszufinden ob die durch `Kieker` und `ExplorViz` entstandene Visualisierung der Microservices von `Symphony` einen Mehrwert für das Verständnis, und damit für die Entwicklung, der Microservices von `Symphony` bringt. Vor allem Entwicklerinnen und Entwicklern, die mit einem Projekt nicht vertraut sind, fehlt es häufig an essentiellem Wissen über das System, an dem sie arbeiten sollen [Zirkelbach u. a. 2015]. Wir möchten deshalb auch wissen, ob `ExplorViz` besonders der Gruppe der unerfahrenen Entwickler helfen kann, in neue Software einzusteigen.

Da wir in dieser Arbeit besonders an der Visualisierung von verteilter Kommunikation interessiert sind, unterscheiden wir dabei zwischen einer Verbesserung des Verständnisses der gesamten Softwarearchitektur durch die Landschaftsansicht und einer Verbesserung des Verständnisses der einzelnen Microservices durch die Applikationsansicht.

6.1.1. Forschungsfragen und Hypothesen

Hieraus ergeben sich für uns die folgenden Forschungsfragen:

FF 1 Unterstützt die Landschaftsansicht der Visualisierung durch `ExplorViz` Entwicklerinnen und Entwickler beim Verständnis der Softwarelandschaft von `Symphony`?

FF 2 Unterstützt die Anwendungsansicht der Visualisierung durch `ExplorViz` Entwicklerinnen und Entwickler beim Verständnis der einzelnen Microservices von `Symphony`?

FF 3 Welche Entwickler-Gruppe profitiert am meisten von einer Software-Visualisierung durch `ExplorViz`?

FF 4 Unterstützt die Visualisierung durch `ExplorViz` Entwicklerinnen und Entwickler beim Verständnis der Microservices von `Symphony`?

Hypothesen

Aufbauend auf diese Forschungsfragen formulieren wir die folgenden Hypothesen, die wir in diesem Kapitel beantworten wollen:

H 1 Entwicklerinnen und Entwickler können mit der Landschaftsansicht von `ExplorViz` Fragen zur Softwarelandschaft von `Symphony` schneller beantworten als Entwicklerinnen und Entwickler, die eine IDE verwenden.

H 2 Entwicklerinnen und Entwickler können mit der Anwendungsansicht von `ExplorViz` Fragen zu den einzelnen Microservices von `Symphony` schneller beantworten als Entwicklerinnen und Entwickler, die eine IDE verwenden.

H 3 Die zeitliche Verbesserung durch Nutzung von `ExplorViz` gegenüber der Nutzung einer IDE fällt bei unerfahrenen Entwicklerinnen und Entwicklern am größten aus.

6.1.2. Variablen

Als unabhängige Variable wählen wir die für den Versuch gewählte Arbeitsumgebung. Zur Auswahl stehen das Software-Visualisierungs-Werkzeug `ExplorViz` und die oft verwendete Entwicklungsumgebung `IntelliJ IDEA`. Die Probandinnen und Probanden sollen unterschiedliche Aufgaben mit der zugewiesenen Arbeitsumgebung bearbeiten und Fragen dazu beantworten. Die abhängigen Variablen sind dann die Beantwortungszeit und die Korrektheit der gestellter Aufgaben.

Störvariablen sind das Vorwissen in den Arbeitsumgebungen sowie die allgemeine Erfahrung im Bereich der Software-Entwicklung. Kennt eine befragte Person bereits die Arbeitsumgebung `ExplorViz`, hat diese einen Vorteil gegenüber anderen Nutzerinnen und Nutzern, die dieses Vorwissen nicht haben. Andersherum könnte auch die Nicht-Kenntnis von `ExplorViz` die Ergebnisse stören, da die Verwenderinnen und Verwender von `IntelliJ IDEA` mit hoher Wahrscheinlichkeit Erfahrung mitbringen. Die Vorerfahrung als Software-Entwicklerin oder -Entwickler lässt sich schwer bestimmen und es ist unmöglich, Probandinnen und Probanden mit gleichen Erfahrungen zu finden. Eine weitere Störvariable ist die Zahl der Teilnehmerinnen und Teilnehmer, die für das Experiment zur Verfügung stehen. Ist die Stichprobe zu klein, ist das Ergebnis unter Umständen nicht aussagekräftig.

6.2. Methodik

6.2.1. Wahl der Methode

Um eine Verbesserung des Software-Verständnisses von Entwicklerinnen und Entwicklern durch `ExplorViz` feststellen zu können, benötigen wir einen Vergleich zwischen der Visualisierung und einer geeigneten herkömmlichen Analyse-Methode. Für letztere eignet

sich eine Entwicklungsumgebung wie `Eclipse` oder `IntelliJ IDEA`, da für die Arbeit mit Software häufig nur textuelle Werkzeuge zur Verfügung stehen.

Dafür entscheiden wir uns für die Durchführung einer Studie mit Probandinnen und Probanden. Der Vergleich der Ergebnisse einer Versuchsteilnehmerin oder eines Versuchsteilnehmers in beiden Arbeitsumgebungen ist für unsere Studie nicht geeignet [Charness u. a. 2012]. Zum einen lassen sich die Aufgaben im ersten Versuchsaufbau nicht unter gleichen Bedingungen im zweiten wiederholen, da die Probandin oder der Proband beim ersten Durchlauf bereits ein Vorwissen erlangt. Zum anderen besteht die Gefahr, dass eine Probandin oder ein Proband einen der beiden Versuchsaufbauten favorisiert, was das Messergebnis beeinflussen würde. Stattdessen teilen wir die Teilnehmerinnen und Teilnehmer in zwei Gruppen ein, von denen die eine als Versuchsgruppe ausschließlich mit `ExplorViz` und die andere als Kontrollgruppe nur mit `IntelliJ IDEA` arbeitet.

Wahl des geeigneten Messwerts

Für den Versuch stellen wir allen Probandinnen und Probanden eine Reihe von Fragen zu den Microservices von `Symphony`, die sie mit der ihrer Gruppe zugeordneten Arbeitsumgebung beantworten sollen. Wir nehmen an, dass eine Probandin oder ein Proband mit einem besseren Verständnis der Softwarelandschaft oder einer ihrer Komponenten, Fragen dazu schneller beantworten kann. Als Messwert eignen sich dann die Bearbeitungszeiten der einzelnen Fragen. Zusätzlich spielt die Richtigkeit der Ergebnisse für den Vergleich eine Rolle. Sind Fragen komplett oder teils falsch beantwortet, so muss dies beim Vergleich in Betracht gezogen werden.

Wahl der Befragungsart

Damit die Teilnehmerinnen und Teilnehmer des Versuchs die Fragen möglichst ohne störende Einflüsse beantworten können und die gemessenen Zeiten repräsentativ sind, wählen wir einen Fragebogen für die Befragung. Ein Interview ist nicht geeignet für diese Art der Datensammlung, da der Interviewleiter nur schwer sicherstellen kann, dass er die Befragten nicht durch die Beantwortung von Zwischenfragen oder durch Hinweise beeinflusst.

6.2.2. Wahl der Probanden

Rekrutierung

Da die Software hinter `Symphony` noch nicht veröffentlicht ist, stehen für die Befragung ausschließlich Mitarbeiterinnen und Mitarbeiter der Firma adesso zur Verfügung. Mögliche Probandinnen und Probanden für den Versuch wurden zusammen mit dem zu diesem Zeitpunkt leitenden Entwickler der `Symphony`-Plattform ausgewählt und in einer Rundmail angeschrieben. Um weitere Teilnehmer zu finden, wurden auch Entwickler in der Hamburger adesso-Filiale direkt angesprochen.

Ein- und Ausschlusskriterien

Das Hauptkriterium für die Teilnahme an der Befragung ist die Rolle als Software-Entwickler. `ExplorViz` soll Entwicklerinnen und Entwickler im Verständnis von Software unterstützen, kann aber kein Grundverständnis ersetzen. Zusätzlich sollten die befragten Probandinnen und Probanden möglichst schon Erfahrung mit `Symphony` gemacht haben, um eine Aussage über eine Verbesserung des Verständnisses und nicht nur über den Aufbau dessen machen zu können.

Ausgeschlossen sind diejenigen Mitarbeiterinnen und Mitarbeiter, die täglich außerhalb der Filiale in Kundenprojekten arbeiten und deshalb nicht für eine Befragung zur Verfügung stehen. Außerdem ausgeschlossen werden mögliche Probandinnen und Probanden, die zwar in der Hamburger Geschäftsstelle arbeiten, denen es aber aufgrund von zu hoher Projektbelastung zeitlich nicht möglich ist, an dem Versuch teilzunehmen.

6.2.3. Einteilung der Probandinnen und Probanden

Ein Ziel unserer Befragung ist es, zu ermitteln, ob bestimmte Gruppen von Entwicklern einen höheren Nutzen gegenüber anderen haben. Aufgrund der geringen Zahl von Teilnehmern eignet sich besonders der Vergleich von unerfahrenen Entwicklern und erfahreneren Entwicklern.

Um eine möglichst ausgewogene Gruppeneinteilung zu erhalten, teilen wir die Probanden in zwei Gruppen ein und verteilen diese dann zufällig in die Versuchs- und in die Kontrollgruppe. So garantieren wir, dass mindestens zwei unerfahrene Trainees in beiden Gruppen landen und das Ergebnis so repräsentativ wie möglich ist.

6.2.4. Wahl des Szenarios

Für die Befragung müssen nicht unbedingt alle acht Microservices von `Symphony` gezeigt werden. Wichtiger ist es, dass die Architektur und das abgebildete Szenario verständlich sind. Unter Szenario verstehen wir hier die Menge der betrachteten Aktionen der Software.

Abbildung 6.1 zeigt die für das Szenario der Produktauswahl benötigten Microservices in topologischer Ordnung. Als Zentrum des Versuchs-Szenarios ist der *Ratingservice* gut geeignet. Er ist zum Zeitpunkt der Befragung der jüngste Microservice in `Symphony` und deshalb nur wenigen Entwicklerinnen und Entwicklern bekannt. Der *Ratingservice* sorgt unter anderem dafür, dass bei einem Produktaufruf dessen gespeicherte Aufrufzahl erhöht wird. Diese Zahl der Aufrufe trägt zum Ermessen der Beliebtheit eines Produktes bei. Der Aufruf eines Produkts erfolgt über der *Frontendproxy*. Bevor ein Produkt aus der Liste aller Produkte ausgewählt werden kann, fragt der *Frontendproxy* beim *Searchservice* eine Liste von Produkten an. Der *Searchservice* wiederum benötigt bei seinem Start den *Marketingservice*, um Informationen über alle Produkte zu bekommen. Bis auf den *Searchservice* benötigen alle genannten Dienste den *Authservice* für verschiedenste Interaktionen.

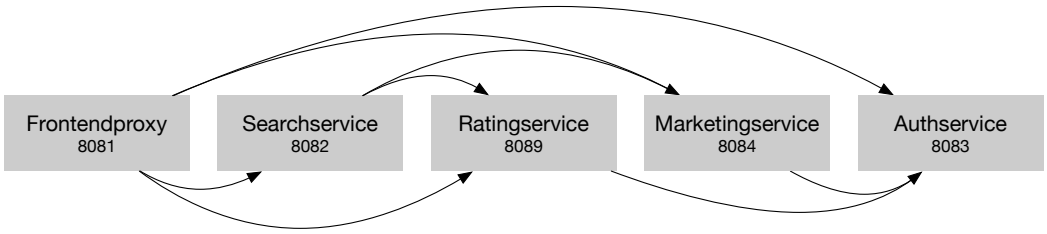


Abbildung 6.1. Auswahl der Microservices in topologischer Ordnung

6.3. Durchführung der Befragung

Der nun folgende Abschnitt beschreibt die Durchführung der Befragung im Detail. Wir betrachten den Aufbau des Fragebogens, die Suche und Wahl der Probandinnen und Probanden und die Ergebnisse der Befragung.

6.3.1. Fragebogen

Für die Zeitmessung erstellen wir einen digitalen Fragebogen, der von den Probandinnen und Probanden im Browser ausgefüllt wird. Da für die Befragung keine zusätzlichen Bildschirme zur Verfügung stehen, müssen die Teilnehmerinnen und Teilnehmer zwischen ihrer Arbeitsumgebung und dem Fragebogen wechseln. Dies geschieht durch Wechseln des Fensters über die Taskleiste.

Der Fragebogen besteht aus drei Abschnitten, die nachfolgend vorgestellt werden.

Persönliche Daten

Im ersten Abschnitt erfragen wir persönliche Daten der Probandin oder des Probanden. Wir sind interessiert an dem Alter und Geschlecht, sowie an einer bestehenden Sehschwäche, da die Farbgestaltung in `ExplorViz` für dessen Bedienung von Relevanz ist.

Zur Einschätzung der Erfahrung möchten wir die Zahl der Jahre erfahren, die die Person bereits in der Softwareentwicklung gearbeitet hat, und die Rolle, die sie derzeit bei adesso belegt. Als Rollen stehen *Trainee*, *Software-Entwickler*, *Software-Architekt* und *Projektleiter* zur Auswahl, sowie die Senior-Formen der letzten drei Rollen.

Um die Fähigkeiten der Teilnehmerinnen und Teilnehmer in der späteren Auswertung berücksichtigen zu können, verwenden wir eine Likert-Skala [Likert 1932] und fragen nach der eigenen Einschätzung in den folgenden Themen:

- **IntelliJ IDEA**
  Die Kontrollgruppe arbeitet mit der IDE. Es ist wichtig für die Auswertung, ob jemand den Umgang damit schon kennt oder nicht.

- **Eclipse**
  Für den Fall, dass eine Probandin oder ein Proband noch nicht mit *IntelliJ IDEA* gearbeitet hat, ist ihre oder seine Erfahrung mit der anderen sehr bekannten IDE `Eclipse` von Bedeutung. In diesem Fall wären lediglich die Befehle anders, das Grundprinzip aber das gleiche.

- **Java**
  Da die Mircoservices von `Symphony` in Java geschrieben wurden, spielt die Java-Erfahrung für das Zurechtfinden innerhalb des Projekts eine entscheidende Rolle.

- **Groovy**
  Einige der Microservices sind zum Teil oder komplett in Groovy geschrieben. Teilweise verwendet Groovy eine andere Syntax, die bei Nicht-Wissen das Verständnis des Programmcodes schmälern kann.

- **Spring**
  Die Microservices von `Symphony` sind als Spring-Boot-Projekte angelegt. Die Erfahrung mit dem Framework kann das Verständnis der Software stark beeinflussen.

- **Microservices**
  Das Verständnis der Zusammenhänge der einzelnen Dienste ist möglicherweise abhängig vom generellen Wissen über Microservices.

- **ExplorViz**
  Die Versuchsgruppe wird mit `ExplorViz` arbeiten. Der Erfahrungsstand mit dem Tool spielt ebenfalls eine große Rolle für den Umgang mit ihm.

Neben der Erfahrung in den oben genannten Themen ist für das Verständnis des `Symphony`-Projekts wichtig, ob die Versuchsteilnehmerin oder der Versuchsteilnehmer bereits daran mitgearbeitet hat. Wir fragen deshalb im Anschluss an die Fähigkeiten nach einem Mitwirken an einem der fünf für den Versuch ausgewählten oder anderen Microservices. Die Entwicklerin oder der Entwickler kann hier mehrere Auswahlen treffen. Sollte an mindestens einem Teil von Symphony gearbeitet worden sein, interessiert uns noch die Dauer, die sie oder er insgesamt an `Symphony` gearbeitet hat.

Ergänzend möchten wir in Erfahrung bringen, ob der Teilnehmer oder die Teilnehmerin schon Erfahrung mit anderen Software-Visualisierungs-Werkzeugen außer `ExplorViz` gemacht hat. Einerseits beeinflusst dies vielleicht den Umgang mit `ExplorViz`, andererseits hilft uns diese Antwort vielleicht dabei, einen Überblick über verwandte Projekte zu gewinnen.

Analyse der Microservices

Mit dem zweiten Abschnitt des Fragebogens beginnt der eigentliche Versuch. Die Probandinnen und Probanden haben die Aufgabe, die gestellten Fragen mithilfe ihrer Arbeitsum-

gebung, also entweder der Visualisierung durch `ExplorViz` oder der Nutzung der IDE, zu beantworten.

Den Hypothesen folgend unterscheiden wir zwischen Fragen nach der Softwarelandschaft und Fragen nach einzelnen Microservices. Außerdem wollen wir einen Überblick über die Software getrennt von einem tieferen Verständnis des Programmablaufs betrachten. Die letzte Unterscheidung ist nicht immer klar trennbar und überschneidet sich bei manchen Fragen. Wir unterteilen die gestellten Fragen deshalb in vier Bereiche, den Überblick über die Softwarelandschaft, den Überblick über die einzelnen Microservices, das Verständnis der Kommunikation innerhalb der Microservices und das Verständnis der verteilten Kommunikation.

Überblick über die Softwarelandschaft Der erste Fragenbereich fordert einen Überblick über die Softwarelandschaft von `Symphony`. Es werden drei Fragen gestellt, die sich alle auf die Port-Belegung der einzelnen Dienste beziehen. Die Probandin oder der Proband muss verstehen, welche Microservices existieren und welche Position diese im Netzwerk haben. Die Nummerierung der folgenden Fragen richtet sich nach der Reihenfolge im Fragebogen und beginnt, anschließend an die Fragen zu den persönlichen Daten, mit 11.

11. Über welchen Port kommuniziert der *Authservice*?
12. Gibt es Microservices, die über den gleichen Port kommunizieren?
13. Stellen Sie sich vor, Sie würden einen neuen Microservice erstellen. Welchen Port würden Sie wählen?

Überblick über die einzelnen Microservices Mit dem zweiten Fragenbereich wollen wir ermitteln, wie gut der Proband oder die Probandin in der Arbeitsumgebung einen Überblick über die einzelnen Dienste erlangen kann. Dazu fragen wir nach dem Paket einer Klasse und den Klassen eines Pakets.

14. Betrachten Sie bitte den *Searchservice*. In welchem Paket liegt die Klasse `ReindexTaskQueue`?
15. Betrachten Sie bitte den *Frontendproxy*. Welche Klassen befinden sich im Paket `de.adesso.symphony.frontendproxy.product.model.search`?

Verständnis der Kommunikation innerhalb der Microservices Getrennt vom Überblick interessiert uns im dritten Bereich die Möglichkeit, dem Programmfluss zu folgen und zu verstehen, was die vorgelegte Software macht. Im ersten Schritt prüfen wir dafür, ob die Entwicklerin oder der Entwickler durch ihre oder seine Arbeitsumgebung feststellen kann, was die Aufgabe einer Klasse ist. Im Anschluss folgen zwei Aufgaben, für die der Kontrollfluss innerhalb je eines Services ein Stück weit verfolgt werden muss.

16. Betrachten Sie bitte den *Ratingservice*. Versuchen Sie kurz zu beschreiben, was die Aufgabe vom `AuthWrapper` ist.

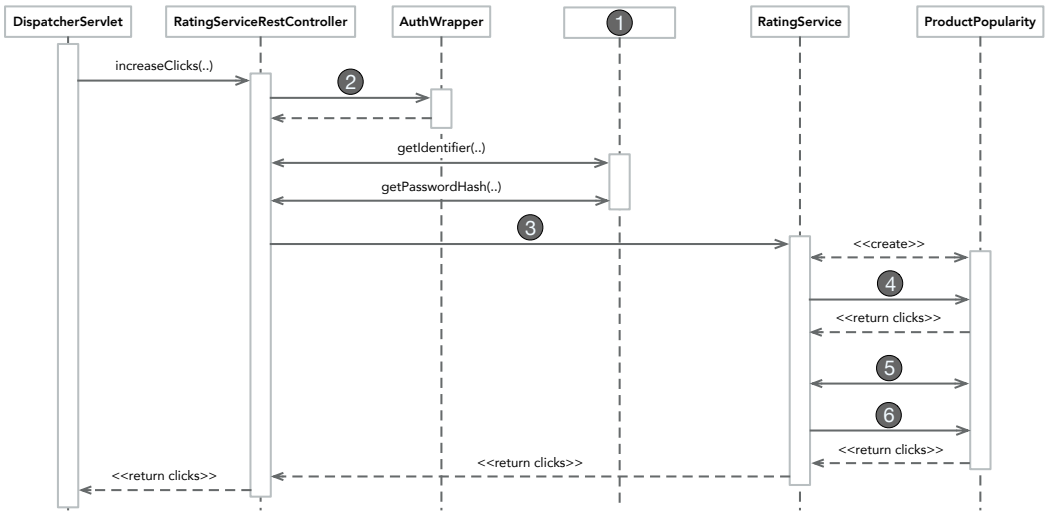


Abbildung 6.2. Sequenzdiagramm zur Frage 18 des Fragebogens

17. Betrachten Sie bitte den *Authservice*. Bei einem Login werden unter anderem die folgenden Klassen verwendet. Sortieren Sie die Klassen in die Reihenfolge, in der sie beim Login das erste Mal auftreten durch Verschieben der Listeneinträge.
    `Company` `Department` `LogHandler` `Principal` `SessionController` `SessionData`

18. Eine Aufgabe des *Ratingservice* ist es, die Aufrufzahlen der Produkte zu verwalten. Wird eine Produktseite geöffnet, so erhöht er die Klick-Zahl des Produkts um 1. Das unten abgebildete Sequenzdiagramm bildet einen Teil des Verhaltens des *Ratingservice* ab. Versuchen Sie, die Methoden- und Klassennamen an den mit Nummern gekennzeichneten Stellen zu ermitteln und tragen Sie diese in die zugehörigen Textfelder ein.

    Die in der Frage als unten stehend genannte Grafik ist in Abbildung 6.2 abgebildet.

Verständnis der verteilten Kommunikation Im vierten Fragenbereich betrachten wir die verteilte Kommunikation zwischen den Microservices und stellen vier Fragen dazu. Die Versuchsteilnehmerin oder der Versuchsteilnehmer muss miteinander kommunizierende Microservices in der Softwarelandschaft analysieren, aber auch ein- und ausgehende Kommunikation innerhalb der Applikationen untersuchen.

19. Bitte nennen Sie drei Microservices, mit denen der *Searchservice* kommuniziert.

20. Über welche Technologie kommuniziert der *Ratingservice* mit dem *Authservice*?

21. Gibt es Microservices, die nur aufgerufen werden, selbst aber nie einen anderen aufrufen?

22. Betrachten Sie bitte den *Searchservice*. Versuchen Sie herauszufinden, welche Klasse als letzte an ausgehender Kommunikation beteiligt ist.

Persönliche Meinung

Der dritte Abschnitt fragt die Teilnehmerin oder den Teilnehmer nach ihrer oder seiner persönlichen Einschätzung, wie leicht ihr oder ihm der Umgang mit der Arbeitsumgebung und die Beantwortung der Fragenbereiche gefallen ist.

23. Wie leicht fällt Ihnen der Umgang mit der Arbeitsumgebung?

24. Wie leicht fiel Ihnen das Lösen der gestellten Aufgaben?

25. Wie leicht fiel es Ihnen mit der gegebenen Arbeitsumgebung einen Überblick über die Microservices zu bekommen?

26. Wie leicht fiel es Ihnen mit der gegebenen Arbeitsumgebung sich in den Microservices zurechtzufinden?

27. Wie leicht fiel es Ihnen mit der gegebenen Arbeitsumgebung den Zusammenhang der Microservices zu verstehen?

28. Wie leicht fiel es Ihnen mit der gegebenen Arbeitsumgebung dem Programmablauf innerhalb der Microservices zu folgen?

Für die Beantwortung dieser Fragen wird eine Bewertungsskala mit den Auswahlmöglichkeiten *Sehr leicht*, *Leicht*, *Eher leicht*, *Eher schwer*, *Schwer* und *Sehr schwer* gegeben. Wir entscheiden uns für eine gerade Zahl an Möglichkeiten, um zu erzwingen, dass Teilnehmerinnen und Teilnehmer Stellung beziehen und wir auch bei Unentschiedenheit Tendenzen erkennen können.

6.3.2. Versuchsaufbau

Termine

Die Versuche wurden an zwei Tagen durchgeführt. Als erster Termin wurde der Tag eines Mitarbeiterfrühstücks gewählt, an dem viele Mitarbeiterinnen und Mitarbeiter, die normal auswärts beim Kunden arbeiten, in der Geschäftsstelle waren. Der zweite Termin fungierte als Alternative, für den Fall dass nicht alle Probandinnen und Probanden an einem Tag befragt werden konnten. Auch er wurde genutzt.

Arbeitsplatz

Der Versuch fand in den Räumlichkeiten der Firma adesso in Hamburg statt. Zur Befragung standen zwei benachbarte Telefonkabinen zur Verfügung. Die Kabinen sind schallabgedichtet und sollen ein möglichst ungestörtes Arbeiten ermöglichen.

Jede der Kabinen beinhaltet einen Arbeitsplatz mit Schreibtisch und Stuhl, sowie einem Telefon, welches für den Versuch aber nicht von Belang ist. Für den Versuch wurden zwei Laptops vorbereitet und je in einer der Kabinen aufgebaut. Die Laptops sind ähnlich aber nicht gleich ausgestattet. Tabelle 6.1 listet die unterschiedlichen Konfigurationen auf.

Tabelle 6.1. Ausstattung der Laptops zur Befragung

	Laptop 1	Laptop 2
Betriebssystem	Windows 10 (64 Bit)	Windows 10 (64 Bit)
Prozessor	Intel Core i5-4300 1.9 Ghz	Intel Core i5-3340M 2.7 Ghz
Grafikkarte	Intel HD Graphics 4400	Intel HD Graphics 4000
Arbeitsspeicher	8 GB	16 GB
Bildschirm	Internes 14-Display, 1920x1080	Internes 15.6-Display, 1920x1080
Zubehör	USB-Maus	USB-Maus

Da im Versuch keine rechenintensiven Operationen gefordert werden, spielt der Leistungsunterschied beider Rechner keine merkliche Rolle für die Ergebnisse. Beide Laptops laufen mit den gegebenen Arbeitsumgebungen flüssig. Die unterschiedlichen Bildschirmgrößen können dagegen die Arbeitsqualität und somit auch die Qualität der Ergebnisse beeinflussen. Für den Versuch standen weder externen Monitore noch externe Tastaturen zur Verfügung.

Software Beide Rechner laufen mit `Windows 10` und bieten daher die gleiche Oberfläche. Für den Versuch haben wir alle Verknüpfungen aus der Task-Leiste entfernt, sodass nur noch die für den Versuch benötigten und geöffneten Programme dort sichtbar waren. Für die Befragung und die Nutzung von `ExplorViz` installierten wir `Google Chrome` in Version 60.0 auf beiden Rechnern. Im normalen Betrieb aktualisiert `ExplorViz` die Visualisierung alle zehn Sekunden. Um zu verhindern, dass das für die Befragung verwendete Szenario nach zehn Sekunden nicht mehr sichtbar ist, wurde `ExplorViz` für den Versuch so manipuliert, dass es durchgehend dasselbe Szenario zeigt.

Die Rechner wurden für die Kontrollgruppe mit der *Community Edition* der `IntelliJ IDEA` in Version 2017.2.2 ausgestattet. Darin haben wir ein neues Projekt für `Symphony` erstellt und die fünf Microservices als Module in das Projekt importiert. Damit lassen sich alle Microservices in der Seitenleiste einer `IDEA`-Instanz verwalten.

Fragebogen

Für die Befragung entwickeln wir eine eigene Web-App, die die Bearbeitungszeiten der einzelnen Fragen ermittelt. Der Fragebogen wird im `Chrome`-Browser der Arbeitsrechner geöffnet. Vor jedem Durchlauf löschen wir den Cache des Browsers um vorige Eingaben aus der Auto-Fill-Funktion zu entfernen.

Den Nutzerinnen und Nutzern wird je eine Frage zur Zeit angezeigt. Unter den Fragen befinden sich zwei Buttons um zur nächsten oder zur vorigen Frage zu navigieren. Zusätzlich zeigt eine Fortschrittsleiste den Fortschritt der Befragung an, um den Befragten einen Anhaltspunkt zu bieten, wie lange die Befragung noch dauert. Letztere Funktion könnte die Entwicklerinnen und Entwickler dazu verleiten, gegen Ende der Befragung flüchtig zu antworten, wir gehen aber davon aus, dass die Mitarbeiterinnen und Mitarbeiter durch ihren Projektdruck entspannter sind, wenn sie ein Ende der Befragung abschätzen können.

Die Applikation misst für jeden Durchlauf die Zeiten, die die Verwenderinnen und Verwender auf den einzelnen Frage-Seiten bleiben. Auch beim Navigieren zu vorigen Fragen wird die Bearbeitungszeit ergänzt. Nach Beantwortung aller Fragen wird die oder der Befragte gebeten, die Daten über einen Button abzusenden. Bei einem Fehler erscheint eine Meldung mit der Bitte, dem Experimentleiter Bescheid zu geben.

Die Ergebnisse der Befragung werden an einen `NodeJS`-Server geschickt, der auf einem Rechner im Netzwerk der Geschäftsstelle läuft und im Netzwerk erreichbar ist. Dieser Server speichert jeden Ergebnissatz in eine CSV-Datei, die ebenfalls auf dem Rechner liegt. Nach jeder Befragung wird vom Experimentleiter kontrolliert, ob alle Daten übertragen wurden, bevor die Web-App für den nächsten Befragten zurückgesetzt wird und die Ergebnisse auf den Experiment-Rechnern verworfen werden.

Abbildung 6.3 zeigt einen Screenshot des Fragebogens.

Erzeugung des Szenarios

`ExplorViz` aktualisiert standardmäßig in regelmäßigen Abständen das angezeigte Modell. Alle in dem letzten Zeitfenster angefallenen Monitoring-Daten werden während des nächsten Zeitfensters visualisiert. Um es der Nutzerin oder dem Nutzer zu ermöglichen, zu vorigen Zeitpunkten springen zu können, werden die Modelle aller Zeitfenster lokal im Heimverzeichnis des Rechners abgelegt. Der Dateiname eines jeden Modells besteht aus einem Zeitstempel, gefolgt von einem Bindestrich, gefolgt der Zahl der Aktivitäten innerhalb des Zeitfensters.

Damit die Teilnehmerinnen und Teilnehmer des Experiments das gespeicherte Szenario einer Produktauswahl angezeigt bekommen, versetzen wir `ExplorViz` in den sogenannten *Replay*-Modus. Dieser erlaubt es, ein bestimmtes Modell festzulegen, welches dann statt der neu eingegangenen Daten in jedem Zeitfenster angezeigt wird. Zu diesem Zweck passen wir die Klasse `LandscapeExchangeServiceImpl.java` an. In ihr werden die privaten Felder `timestamp` und `activity` normalerweise mit *null* initialisiert. Definieren wir einen

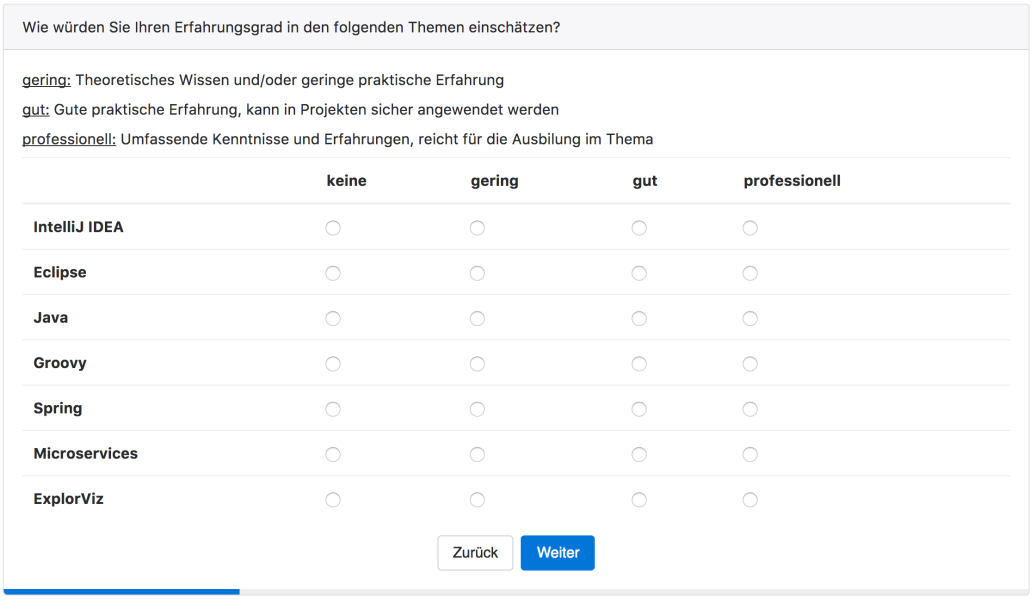


Abbildung 6.3. Screenshot des Online-Fragebogens

Zeitstempel und eine Aktivitätszahl, wird beim Laden der Landschaft in jedem Zeitfenster das vorgegebene Modell gesucht und verwendet und nicht das zuletzt generierte. Statt im Verzeichnis `.explorviz/landscapeRepository` müssen Modelle für den *Replay*-Modus im Verzeichnis `.explorviz/replay` des Heimverzeichnisses liegen.

Um die Daten zu generieren, wird das in Abschnitt 6.2.4 beschriebene Szenario nachgestellt. Wir öffnen die Produktübersicht und warten einige Zeit ohne Aktivität ab, damit `ExplorViz` einige Modelle ohne Aktivität generiert. Dann öffnen wir eines der Produkte und warten wieder einige Zeit ab. Wir erreichen dadurch, dass unter den zuletzt generierten Modellen genau ein Modell mit einer Aktivitätszahl über Null existiert, welches exakt unser Szenario wiedergibt. Wir kopieren die Datei des Modells `1502266486164-827.expl` in das oben erwähnte `replay`-Verzeichnis und vergeben in der classLandscapeExchangeServiceImpl die Werte *1502266486164* und *827*. Öffnen wir nun die Web-Ansicht von `ExplorViz`, sehen wir immer dasselbe Modell des gewünschten Szenarios.

In `IntelliJ IDEA` ist es nicht möglich, ein solches Szenario zu stellen. Die Nutzerinnen und Nutzer müssen selbst die verwendeten Klassen und Methoden finden. Hierfür stehen ihnen aber die Projektordner der fünf für den Versuch verwendeten Microservices in einem Fenster zur Verfügung.

6.3.3. Beschreibung der Probandinnen und Probanden

Von den ursprünglich gefragten 19 Mitarbeiterinnen und Mitarbeitern konnten zehn für den Versuch gewonnen werden. Die Zahl der ausgeschlossenen Probandinnen und Probanden und deren Ausschlussgründe werden in Tabelle 6.2 aufgeführt.

Tabelle 6.2. Ausschlussgründe der Probandinnen und Probanden

Gesamtzahl der eingeladenen Probandinnen und Probanden	19
Ausschluss aufgrund von	
Einsatz beim Kunden ohne absehbaren Besuch der Geschäftsstelle	-7
zu hoher Projektbelastung	-2
Teilnehmende Probandinnen und Probanden für die Stichprobe	10

Von den befragten Personen sind acht männlich und zwei weiblich. Ein Proband ist jünger als 25 Jahre, vier der Befragten sind sogar älter als 35, keiner aber ist älter als 50.

Tabelle 6.3 zeigt die Verteilung der Rollen und Erfahrung in der Softwareentwicklung unter den Probandinnen und Probanden. Fünf und damit die meisten befragten Entwicklerinnen und Entwickler sind zum Zeitpunkt der Befragung als Trainee bei adesso tätig. Zwei Mitarbeiter arbeiten als Software-Entwickler und drei als Senior Software-Entwickler. Der Großteil der Trainees arbeitet weniger als drei Jahre im Bereich der Softwareentwicklung. Beide Software-Entwickler gaben an, drei bis fünf Jahre Erfahrung zu haben. Die Senior Software-Entwickler haben alle über elf Jahre Software-Erfahrung, zwei von ihnen sogar über 20 Jahre.

Tabelle 6.3. Verteilung der Rollen und Erfahrung in der Softwareentwicklung

Rolle insgesamt in der Softwareentwicklung	Anzahl
Trainee	**5**
0-2 Jahre	4
11-20 Jahre	1
Software-Engineer	**2**
3-5 Jahre	2
Senior Software-Engineer	**3**
11-20 Jahre	1
über 20 Jahre	2

6.3.4. Durchführung des Versuchs

Vor der Durchführung des Versuchs wurde ohne den kompletten Versuchsaufbau eine Vorstudie mit dem leitenden Entwickler des `Symphony`-Projekts gemacht. Dabei wurden Erkenntnisse über die ungefähre Dauer eines Durchlaufs gewonnen und es konnten Fragen geklärt, sowie Fehler ausgebessert werden. Zwei Fragen, die ursprünglich verwendet werden sollten, wurden gestrichen, da sie zu schwer waren oder zu lange gedauert hätten.

Pro Versuch benötigte eine Entwicklerin oder ein Entwickler etwa eine Stunde für die Einführung und die Bearbeitung aller Fragen. Um möglichst viele Probandinnen und Probanden an einem Tag schaffen zu können, wurden pro Durchlauf zwei von ihnen gleichzeitig in je einer der beiden Telefonkabinen befragt.

Die Durchführung des Versuchs bestand aus zwei Teilen. Im ersten Teil erhielten die Entwicklerinnen und Entwickler eine Einführung durch den Versuchsleiter. Diese Einführung umfasste eine Vorstellung des Themas und den Grund für die Befragung, sowie eine Erklärung des Versuchsablaufs. Die Teilnehmerinnen und Teilnehmer werden darauf hingewiesen, dass es eine Versuchs- und eine Kontrollgruppe gab und ihre Daten geschützt werden würden. Der Versuchsleiter führte die Probandinnen und Probanden dann in ihren Arbeitsplatz ein und erklärt deren Gruppenzugehörigkeit. Zuletzt zeigte der Versuchsleiter die für die Beantwortung der Fragen wichtigsten Funktionen der jeweiligen Arbeitsumgebung.

Während der Befragung standen den Teilnehmerinnen und Teilnehmern Informationsblätter zur Verfügung, die die Formalitäten des Versuchs und Informationen zu den Fragen sowie der Arbeitsumgebung enthalten. Diese sind Anhang A zu finden.

6.3.5. Beschreibung der Versuchsergebnisse

Wir betrachten nun im folgenden Abschnitt die Ergebnisse der Befragung. Dabei gliedern wir den Fragebogen wie in Abschnitt 6.3.1. Abhängig von den gegebenen Antworten vergleichen wir entweder die durchschnittliche Bearbeitungsdauer oder die durchschnittliche Richtigkeit der Antworten beider Gruppen. Nachfolgend verwenden wir für den Mittelwert oder Durchschnitt der Ergebnisse das Symbol $\varnothing$ und für die statistische Stichproben-Standardabweichung (ab hier auch nur Standardabweichung genannt) das Symbol σ.

Die vollständigen Versuchsergebnisse befinden sich in Anhang B und im Repositorium zu dieser Arbeit.

Persönliche Daten

Sehschwäche In den persönlichen Angaben hat keine Person angegeben, unter einer Sehschwäche zu leiden, die die Arbeit am Rechner beeinflusst. Dennoch bemerkte einer der Teilnehmer der Versuchsgruppe schon zu Beginn, dass der 14-Bildschirm für ihn sehr klein wäre. Eine Beeinflussung des Ergebnisses kann an dieser Stelle nicht ausgeschlossen werden.

Erfahrung in der Softwareentwicklung Die Versuchsgruppe arbeitet im Durchschnitt 9,2 Jahre im Bereich der Softwareentwicklung, während die Kontrollgruppe im Schnitt 7,2 Jahre Erfahrung mitbringt. Zwei der Teilnehmerinnen und Teilnehmer der Versuchsgruppe gaben an, bereits länger als 20 Jahre in der Softwareentwicklung zu arbeiten. Für die Durchschnitts- und Standardabweichungsberechnung werden für Tabelle 6.4 nur 20 Jahre angenommen. Längere Erfahrungsdauern würden lediglich den Durchschnittswert der Versuchsgruppe erhöhen.

Tabelle 6.4. Durchschnittliche Jahre an Erfahrung in der Softwareentwicklung

	ExplorViz		IntelliJ IDEA	
Frage	∅	σ	∅	σ
5. Jahre in der Softwarentwicklung	9,2	9,9	7,2	7,2

Fähigkeiten Wie die Verteilung der Rollen sind auch die Fähigkeiten in den abgefragten Themengebieten gemischt. Tabelle 6.5 übersetzt die Auswahlmöglichkeiten der Likert-Skala in numerische Werte, mit denen der Durchschnitt ermittelt werden kann.

Tabelle 6.5. Übersetzung der Fähigkeits-Level in numerische Werte

Wert der Likert-Skala	keine	gering	gut	professionell
Numerischer Wert	0	1	2	3

Aus den Ergebnissen können wir für jede Fähigkeit den Mittelwert und die Standardabweichung der Probandengruppen berechnen, wie es in Tabelle 6.6 gezeigt ist.

Tabelle 6.6. Mittelwerte und Standardabweichungen der Fähigkeits-Level beider Gruppen

	ExplorViz		IntelliJ IDEA	
Fähigkeit	∅	σ	∅	σ
IntelliJ IDEA	1,00	0,71	1,60	1,14
Eclipse	2,20	0,45	2,20	0,45
Java	2,20	0,45	2,40	0,89
Groovy	1,00	0,71	1,40	0,55
Spring	1,60	0,55	2,00	0,71
Microservices	1,40	0,55	2,00	1,00
ExplorViz	0,20	0,45	0,20	0,45

Mitarbeit an Symphony Von den zehn befragten Personen haben drei noch keine Erfahrung mit den Microservices von `Symphony` gemacht. Fünf der anderen Teilnehmerinnen und Teilnehmer waren in die Entwicklung von mindestens einem der im Versuch gezeigten

Microservices involviert. Zwei Befragte haben am Ratingservice mitentwickelt und haben deshalb gute Voraussetzungen, dessen Funktion zu verstehen.

Da wir nicht nach allen Bestandteilen von `Symphony` explizit gefragt haben, können wir keine durchschnittliche Zahl pro Gruppe angeben. Ohne Projekterfahrung von über einem Jahr zu berücksichtigen, beträgt die durchschnittliche Arbeitsdauer der Mitglieder beider Gruppen etwa vier bis fünf Monate. Tabelle 6.7 listet die Erfahrungen der Probandinnen und Probanden in Spalten auf.

Tabelle 6.7. Erfahrung der Probanden und Probandinnen mit den Microservices von `Symphony` und die Gesamtzeit an Projekterfahrung

	ExplorViz					IntelliJ IDEA				
Erfahrungen										
Frontendproxy	x								x	
Authservice	x									
Marketingservice	x									
Searchservice	x								x	
Ratingservice					x	x				
Andere	x		x	x				x	x	
Arbeitserfahrung im Symphony-Projekt in Monaten:										
Monate	12+	0	0-3	0-3	4-6	4-6	0	7-12	7-12	0
Durchschnitt	4					4,8				

Analyse der Microservices

Im Folgenden werden wir die Erbnisse der Analyse-Aufgaben betrachten. Hierfür ergänzen wir zusätzlich zum Durchschnitt und der Standardabweichung noch die Korrektheit der Antworten c. Die Korrektheit ist der Anteil an korrekten Fragen innerhalb der Gruppe. Sollten Fragen nur halb falsch beantwortet worden sein, wird dies im Wert berücksichtigt.

Überblick über die Softwarelandschaft Für den Überblick über die Softwarelandschaft mussten die Teilnehmer und Teilnehmerinnen der Befragung die Ports der einzelnen Microservices ermitteln. Tabelle 6.8 listet die durchschnittlichen Bearbeitungszeiten der beiden Gruppen für die Aufgaben 11 - 13 auf. Die Bearbeitungszeiten aller Teilnehmerinnen und Teilnehmer werden in Abbildung 6.4 visualisiert.

Bis auf eine Ausnahme haben alle Teilnehmerinnen und Teilnehmer alle drei Aufgaben korrekt beantwortet. Eine oder einer gab als Port für einen neuen Dienst *noch nicht belegten Port* als Lösung an und hat für die Antwort 139 Sekunden gebraucht.

Überblick über die einzelnen Microservices Für einen Überblick über die Struktur der einzelnen Microservices mussten die Probandinnen und Probanden eine Klasse finden und

Tabelle 6.8. Durchschnittliche Bearbeitungszeiten der Fragen 11 bis 13 und die Standardabweichungen

	ExplorViz			IntelliJ IDEA		
Frage	∅	σ	c	∅	σ	c
11. Port des `Authservice`	50 Sek	56 Sek	100%	56 Sek	37 Sek	100%
12. Dienste mit selbem Port	34 Sek	39 Sek	100%	96 Sek	19 Sek	100%
13. Port für neuen Dienst	53 Sek	32 Sek	100%	88 Sek	44 Sek	90%

das Paket angeben, sowie die Klassen eines bestimmten Pakets eintragen. In Tabelle 6.9 werden die durchschnittlichen Bearbeitungszeiten und deren Standardabweichungen der beiden Aufgaben aufgeführt.

Tabelle 6.9. Durchschnittliche Bearbeitungszeiten der Fragen 14 bis 15 und die Standardabweichungen

	ExplorViz			IntelliJ IDEA		
Frage	∅	σ	c	∅	σ	c
14. Paket von `ReindexTaskQueue`	194 Sek	64 Sek	100%	118 Sek	60 Sek	100%
15. Klassen im Paket `search`	78 Sek	20 Sek	100%	69 Sek	15 Sek	90%

Auch in Aufgabe 15 hat eine oder einer der Probandinnen und Probanden der Kontrollgruppe eine ungenügende Lösung eingetragen. Statt den zu findenden drei Klassen wurde nur eine angegeben. Die durchschnittliche Bearbeitungszeit und deren Standardabweichung verändert sich aber durch das falsche Ergebnis aber nicht merklich. Selbst bei Ignorieren der falschen Antwort, würden die gerundeten Zeiten gleich bleiben.

Verständnis der Kommunikation innerhalb der Microservices Um zu zeigen, wie gut die Versuchsteilnehmer und -teilnehmerinnen dem Kontrollfluss innerhalb eines Dienstes folgen können, wurden drei Aufgaben gestellt.

Frage 16 erwartet einen Freitext, der beschreibt, was die Aufgabe des `AuthWrappers` im *RestTemplate* ist. Drei der Teilnehmer und Teilnehmerinnen gaben dabei nicht ausreichende Erklärungen ab, durch den Freitext lässt sich aber schlecht einen Prozentwert der richtigen Antworten angeben. Stattdessen betrachten wir einmal die durchschnittlichen Zeiten unter Einbezug aller Antworten und einmal unter Ausschluss der nicht ausreichenden Antworten. Diese werden in Tabelle 6.10 abgebildet.

Die Fragen 17 und 18 erforderten es von den Probandinnen und Probanden, die Reihenfolge von Klassen und Methoden zu erkennen. In diesen Fragen wurden am meisten Fehler gemacht. Wir werten deshalb nicht die Bearbeitungszeiten der Teilnehmerinnen und Teilnehmer aus, sondern die Zahl der Fehler. Tabelle 6.11 listet die durchschnittlichen Fehlerzahlen und deren Standardabweichungen auf.

Tabelle 6.10. Durchschnittliche Bearbeitungszeiten der Frage 16 und die Standardabweichungen

	ExplorViz		IntelliJ IDEA	
16. Aufgabe des `AuthWrappers`	∅	σ	∅	σ
unter Einbezug aller Antworten	351 Sek	138 Sek	193 Sek	88 Sek
unter Ausschluss schlechter Antworten	392 Sek	119 Sek	229 Sek	82 Sek

Tabelle 6.11. Durchschnittliche Fehlerzahl und deren Standardabweichungen in den Fragen 17 und 18.

	ExplorViz			IntelliJ IDEA		
Frage	∅	σ	c	∅	σ	c
17. Reihenfolge beim Login	481 Sek	227 Sek	60%	555 Sek	524 Sek	30%
18. Sequenzdiagramms	551 Sek	165 Sek	86,7%	531 Sek	336 Sek	66,7%

Verständnis der verteilten Kommunikation Im letzten Fragenbereich wurden den befragten Personen vier Fragen zu verteilter Kommunikation gestellt. Tabelle 6.12 zeigt die durchschnittlichen Bearbeitungszeiten und deren Standardabweichungen für diese Fragen.

Tabelle 6.12. Durchschnittliche Bearbeitungszeiten der Fragen 19 bis 22 und die Standardabweichungen

	ExplorViz			IntelliJ IDEA		
Frage	∅	σ	c	∅	σ	c
19. Partner des `Searchservice`	58 Sek	27 Sek	100%	103 Sek	59 Sek	95%
20. Technologie	36 Sek	40 Sek	100%	53 Sek	65 Sek	80%
21. Dienst, der nur empfängt	137 Sek	73 Sek	80%	114 Sek	158 Sek	80%
22. Letzt Klasse	289 Sek	246 Sek	80%	182 Sek	140 Sek	40%

Bei Frage 21 hat je ein Mitglied beider Gruppen falsch geantwortet und angegeben, dass es keinen Microservice gibt, der nur Empfänger von Kommunikation ist. Weiter ist zu beachten, dass bei Frage 22 drei der Mitglieder der Kontrollgruppe keine Lösung angegeben haben und eine Mitglied der Versuchsgruppe eine falsche Lösung angegeben hat.

Persönliche Meinung

Die Fragen 23 bis 28 bitten die Befragten um eine Bewertung ihrer Erfahrung während der Befragung. Dafür stehen für jede Frage *Sehr leicht*, *Leicht*, *Eher leicht*, *Eher schwer*, *Schwer* und *Sehr schwer* als Auswahl zur Verfügung. Im Folgenden werden wir die durchschnittlichen Bewertungen der beiden Gruppen ausrechnen und dafür die Begriffe in numerische Werte übersetzen, wie es in Tabelle 6.13 gezeigt ist. Tabelle 6.14 listet die durchschnittlichen Bewertungen für alle sechs Fragen auf.

Tabelle 6.13. Übersetzung der Bewertungen in numerische Werte

Bewertung	Sehr leicht	Leicht	Eher leicht	Eher schwer	Schwer	Sehr schwer
Wert	1	2	3	4	5	6

Tabelle 6.14. Durchschnittliche Bewertungen der Versuchsteilnehmerinnen und -teilnehmer

	ExplorViz	IntelliJ IDEa
Arbeitsumgebung	3	2,6
Lösen der Aufgaben	3	3,4
Überblick über die Microservices	1,8	3,8
Zurechtfinden in den Microservices	2,6	2,6
Verstehen des Zusammenhangs zwischen den Microservices	3	4
Folgen des Programmablaufs in den Microservices	3,2	3,4

Im Durchschnitt empfanden die Nutzer und Nutzerinnen der IDE den Umgang mit ihrer Arbeitsumgebung etwas leichter als diejenigen, die mit `ExplorViz` gearbeitet haben. Der Unterschied ist allerdings gering. Im Gegenzug fiel das Lösen der Aufgaben den `ExplorViz`-Nutzern und -Nutzerinnen etwas leichter als den Mitgliedern der Kontrollgruppe mit `IntelliJ IDEA`.

Ein deutlicher Unterschied ist beim Überblick über die Microservices zu sehen. Der Gruppe, die mit `ExplorViz` gearbeitet hat, fiel dies im Durchschnitt deutlich leichter als der IDE-Gruppe. Ebenso fiel es der Versuchsgruppe eher leicht, den Zusammenhang zwischen den Microservices zu verstehen, während es der Kontrollgruppe mit `IntelliJ IDEA` eher schwer gefallen ist.

Beim Zurechtfinden innerhalb der Microservices gibt es im Durchschnitt keinen Unterschied zwischen der Versuchs- und der Kontrollgruppe. Das Folgen des Programmablaufs innerhalb der Microservices ist Verwenderinnen und Verwendern von `ExplorViz` im Schnitt aber etwas leichter gefallen, als Teilnehmerinnen und Teilnehmern, die mit der IDE arbeiten mussten.

6.3.6. Diskussion der Versuchsergebnisse

Wir werden nun die Versuchsergebnisse diskutieren und auf Muster analysieren.

Persönliche Daten

Die Erfahrungen der befragten Personen sind gemischt. Dennoch zeigen sich einige Auffälligkeiten. Die Rolle bei adesso korreliert in neun von zehn Fällen mit den Jahren, die die Teilnehmerinnen und Teilnehmer bereits Erfahrung im Bereich der Softwareentwicklung gesammelt haben. Trainees besitzen in fast allen Fällen erst 0-2 Jahre Erfahrung, Software-Entwickler 3-5 Jahre und alle Senior Software-Entwickler arbeiten bereits mehr als 10 Jahre

in dem Bereich. Nur einer der Befragten gab an, trotz seiner Rolle als Trainee bereits 11-20 Jahre Erfahrung in der Softwareentwicklung zu haben.

Betrachten wir die Fähigkeiten der Entwicklerinnen und Entwickler, so fällt auf, dass die Kontrollgruppe ihre Fähigkeiten in den Bereichen *Java*, *Grooy*, *Spring* und *Microservices* im Schnitt mit 0,4 um knapp eine halbe Stufe besser einschätzt als die Versuchsgruppe. Im Kontrast dazu liegt die Arbeitserfahrung der Kontrollgruppe im Bereich der Softwareentwicklung durchschnittlich mindestens zwei Jahre unter der Arbeitserfahrung der Versuchsgruppe. Ihre Fähigkeiten in der Verwendung von `Eclipse` bewerten beide Gruppen im Schnitt gleich gut. Die Probandinnen und Probanden der Kontrollgruppe, die mit `IntelliJ IDEA` arbeiten mussten, schätzten ihre Fähigkeiten im Durchschnitt mit 1,6 als gering bis gut ein. Nur eine oder einer der Befragten aus jeder Gruppe hat überhaupt geringe Erfahrungen mit `ExplorViz`, während alle anderen Mitglieder keine Fähigkeiten darin mitbringen. Die Angaben zur Mitarbeit an `Symphony` sagen allein nichts weiter über die Entwicklerinnen und Entwickler aus.

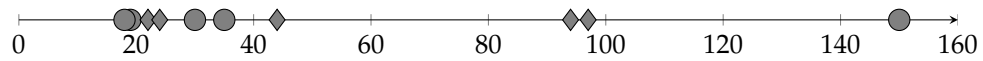
Analyse der Microservices

Überblick über die Softwarelandschaft In der ersten Aufgabe zum Überblick über die Softwarelandschaft galt es für die Teilnehmerinnen und Teilnehmer, den Port des *Authservice* zu finden. Mit `ExplorViz` genügt es hierzu, den Port im Namen des Knotens, in welchem der *AuthService* liegt, zu lesen. Allerdings müssen die Verwenderinnen und Verwender von `ExplorViz` erkennen, dass die Beschriftungen der Knoten Ports sind. Ein Hinweis dazu findet sich in der dem Experiment beiliegenden Beschreibung. Um den Port eines Microsercice in `IntelliJ IDEA` zu finden, müssen dessen `.properties`-Dateien nach der Portangabe durchsucht werden. Auch hierfür findet sich in der Beschreibung ein Hinweis.

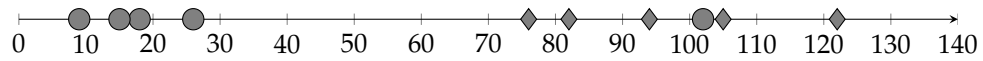
Die Versuchsgruppe konnte mit 50 Sekunden Bearbeitungszeit diese erste Frage im Durchschnitt sechs Sekunden schneller als die Kontrollgruppe beantworten. Allerdings ist die Streuung der Bearbeitungszeiten in der Versuchsgruppe größer. Ein Senior Software-Entwickler der Versuchsgruppe mit der meisten `Symphony`- und einzigen `ExplorViz`-Erfahrung hat die Frage am zweitschnellsten beantworten können. Sonst gibt es aber keine erkennbaren Muster bezüglich der Fähigkeiten oder Erfahrungen der Entwicklerinnen und Entwickler. Wir gehen davon aus, dass sich die Mitarbeiterinnen und Mitarbeiter erst an die Arbeitsumgebung gewöhnen mussten.

In der zweiten Frage mussten die Probandinnen und Probanden herausfinden, ob es zwei Microservices gibt, die über den selben Port kommunizieren. Für die Beantwortung der Fragen kann dasselbe Verfahren wie in der vorigen Aufgabe verwendet werden. Dennoch unterscheiden sich die Antwortzeiten der beiden Gruppen erheblich. Durchschnittlich brauchten die Mitglieder der Versuchsgruppe mit `ExplorViz` nur 34 Sekunden, während Personen der Kontrollgruppe im Schnitt 96 Sekunden beschäftigt waren.

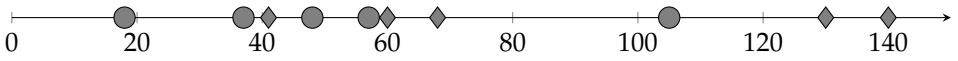
Auch die dritten Frage zum Überblick über die Softwarelandschaft konnten Teilnehmerinnen und Teilnehmer mit `ExplorViz` mit einer Differenz von durchschnittlichen 35 Sekunden schneller als diejenigen beantworten, die mit der IDE gearbeitet haben.



(a) Bearbeitungszeiten von Aufgabe 11 in Sekunden



(b) Bearbeitungszeiten von Aufgabe 12 in Sekunden



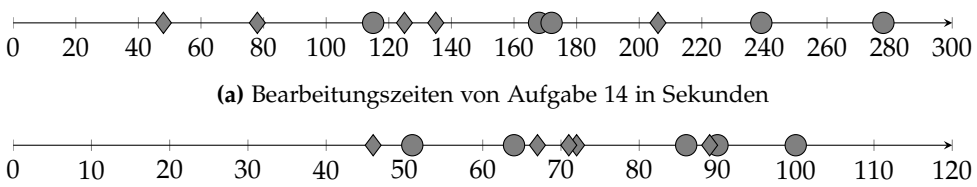
(c) Bearbeitungszeiten von Aufgabe 13 in Sekunden

Abbildung 6.4. Visualisierung aller Bearbeitungszeiten der Aufgaben zum Überblick über die Softwarelandschaft. Mitglieder der Versuchsgruppe sind als Kreise dargestellt, Mitglieder der Kontrollgruppe als Rauten.

Die Standardabweichungen allein können ohne Berücksichtigung von Ausreißern ein falsches Bild der Ergebnisse vermitteln. Abbildung 6.4 visualisiert deshalb die Bearbeitungszeiten aller Mitglieder der beiden Gruppen in Sekunden. Als Kreise dargestellte Zeiten sind die der Nutzerinnern und Nutzer von `ExplorViz`, während die Dreiecke für Verwenderinnen und Verwender der IDE stehen. In allen drei Grafiken sind die Mittelwerte der beiden Gruppen etwa ablesbar, die Grafik zu Aufgabe 12 zeigt aber auch, dass der Mittelwert und die Standardabweichung der Versuchsgruppe nur durch den Ausreißer mit 105 Sekunden erhöht sind. Auch in der Grafik zu Aufgabe 11 sehen wir, dass die Mitglieder der Versuchsgruppe ohne einen Ausreißer wesentlich mehr gruppiert sind, als diejenigen der Kontrollgruppe. Der Großteil der Entwicklerinnen und Entwickler mit `ExplorViz` waren deutlich schneller.

Es zeigt sich in den Ergebnissen der ersten Fragen deutlich, dass die Landschaftsansicht von `ExplorViz`, wenn sie verstanden wurde, wesentlich besser geeignet ist, um die Verteilung der Microservices in die Softwarelandschaft einzuordnen, als vergleichbare Methoden in einer IDE wie `IntelliJ IDEA`.

Überblick über die einzelnen Microservices Die zwei Fragen zum Überblick über die einzelnen Microservices erforderten es von den Teilnehmerinnen und Teilnehmern, Klassen zu finden. Eine solche Funktion ist essentiell für das Verständnis und die Wartung von Software. Für die Beantwortung der ersten Frage muss das Paket gefunden werden, in dem eine vorgegebene Klasse liegt. Diese Aufgabe ist in `IntelliJ IDEA` durch die Suchfunktion einfach lösbar. In `ExplorViz` existiert eine solche Suchfunktion nicht. Daher muss die Klasse manuell gesucht und dann das Paket ermittelt werden. Die unterschiedlichen Herangehensweisen spiegeln sich in den Bearbeitungszeiten der Aufgabe wieder. Mitglieder der Versuchsgruppe brauchten mit `ExplorViz` durchschnittlich 194 Sekunden, während die



(a) Bearbeitungszeiten von Aufgabe 14 in Sekunden

(b) Bearbeitungszeiten von Aufgabe 15 in Sekunden

Abbildung 6.5. Visualisierung aller Bearbeitungszeiten der Aufgaben zum Überblick über die Microservices.

Kontrollgruppe mit `IntelliJ IDEA` im Schnitt 118 Sekunden benötigte. Die Streuung der Ergebnisse ist mit 64 und 60 Sekunden etwa gleich. Daraus können wir ableiten, dass wir zwei etwa gleich große Punktwolken haben, die nur unterschiedliche Mittelpunkte haben.

Im Rahmen der zweiten Frage besteht die Aufgabe der Probandinnen und Probanden darin, alle Klassen innerhalb eines Pakets aufzulisten. Wichtig ist zu erwähnen, dass in der Visualisierung von `ExplorViz` nur zwei Klassen gefunden werden konnten, während der Dateibaum der IDE drei Klassen zeigt. Die durchschnittlichen Bearbeitungszeiten und Standardabweichungen sind ähnlich. Befragte der Kontrollgruppe waren etwa acht Sekunden schneller.

Tragen wir die Bearbeitungszeiten wie in Abbildung 6.5 auf Zeitachsen auf, können erkennen, dass in Frage 14 trotz der stark unterschiedlichen Mittelwerte viele der Befragten in der Nähe des Mittels aller Antwortzeiten liegen und die unterschiedlichen Mittelwerte eher durch die äußeren Zeiten zustande kommen. Ob diese Ausreißer sind oder aufgrund der geringen Probanden-Zahl nur vereinzelt auftreten, lässt sich aber nicht feststellen.

Die Ergebnisse des zweiten Fragenblocks zeigen eine Schwäche von `ExplorViz`. Es gibt keine Funktion, um Klassen schnell zu finden. Das Auffinden von Paketen dagegen ist nur unwesentlich langsamer als in der IDE.

Verständnis der Kommunikation innerhalb der Microservices Für das Beantworten der Fragen zum Verständnis der Kommunikation innerhalb der Microservices mussten die Nutzerinnen und Nutzer der jeweiligen Arbeitsumgebungen den Programmablauf verfolgen. Aufgrund der hohen Fehlerzahl betrachten wir nicht gesondert die Antwortzeiten der einzelnen Fragen sondern beschäftigen uns mit den Fehleranteilen.

In der ersten zugehörigen Frage sollte die Aufgabe der Klasse `AuthWrapper` im *Ratingservice* als Freitext beschrieben werden. Sieben der Antworten der Probandinnen und Probanden werten wir als korrekt, drei dagegen sind nicht aussagekräftig genug oder enthalten falsche Informationen. Dennoch zeigt Tabelle 6.10, dass sowohl unter Einbezug als auch unter Ausschluss der ungenügenden Antworten die Kontrollgruppe im Schnitt etwa 160 Sekunden schneller als die Versuchsgruppe war. Über den Grund für diesen Unterschied können wir nur mutmaßen. Vielleicht fällt es Nutzerinnen und Nutzer in der

textuellen Ansicht leichter, einen Überblick über die Methoden einer Klasse und deren Inhalte zu bekommen. Dazu könnte kommen, dass `ExplorViz` es zwar ermöglicht, dem Programmfluss zu folgen, `IntelliJ IDEA` aber das Navigieren zwischen Methoden durch verschiedene Funktionen erleichtert. So ermöglicht es die IDE, von einem Methodennamen direkt zu deren Deklaration zu springen oder die Referenzen auf eine Methode an anderen Stellen zu finden.

Die zweite Aufgabe dieses Fragenbereichs stellt eine Liste von Klassennamen des *Authservice*. Die Befragten hatten die Aufgabe, die Klassen per Drag-and-Drop in die Reihenfolge zu bringen, in der sie das erste mal bei einem Login vorkommen. Mit `ExplorViz` ist diese Aufgabe leicht gelöst. Es genügt, die Trace-Verfolgung auf einer beliebigen Kommunikationslinie zu aktivieren und dem Trace vom Anfang bis zum Ende zu folgen. In dem vorliegenden Szenario gibt es nur Login-Anfragen im *Authservice*. Alle Klassen, die im Fragebogen genannt sind, werden in der Reihenfolge ihres Auftretens geordnet.

In `IntelliJ IDEA` ist das Finden des Traces etwas schwieriger. Die Nutzerin oder der Nutzer muss den `SessionController` als erste Klasse ausfindig machen. Dies kann sie oder er beispielsweise über eine Suche nach *login(* bewerkstelligen. Die einzige Methode mit dem Namen `login` befindet sich im `SessionController`. Von dort aus muss die Teilnehmerin oder der Teilnehmer lediglich dem Programmcode folgen und alle Klassen der Frage finden. Zeitlich ist dieses Verfahren etwas aufwendiger, da nicht wie bei `ExplorViz` über eine Schaltfläche von Methode zu Methode gesprungen werden kann. Die Versuchsgruppe war in der Beantwortung der Frage im Schnitt 74 Sekunden schneller.

Neben dem zeitlichen Unterschied existiert ein erheblicher Unterschied in der Richtigkeit der Antworten. Während `ExplorViz`-Verwenderinnen und -Verwender bis auf einen Probanden alle dieselben drei und damit durchschnittlich 2,4 Fehler von sechs möglichen gemacht haben, liegt die Fehlerzahl der Kontrollgruppe im Schnitt bei 4,2. Es fällt außerdem auf, dass bei drei der fünf Mitgliedern der Kontrollgruppe für alle der sechs Klassen eine falsche Position angegeben wurde.

Als mögliche Klassen für diese Frage haben wir Klassen ausgesucht, die sowohl im Trace von `ExplorViz` als auch im Programmcode von `IntelliJ IDEA` auffindbar sind. Die Schwierigkeit bei der Untersuchung des Programmcodes besteht darin, dass Klassen teilweise innerhalb von Methoden verwendet werden, bevor diese im `SessionController` auftreten. Listing 6.1 präsentiert einen Ausschnitt aus dem Code des `SessionControllers`.

In Zeile 1 tritt die Klasse `SessionData` das erste Mal auf. Zeile 2 verwendet dann die Klasse `Principal`. Zeile 6 erweckt nun den Eindruck, die Klasse `LogHandler`, von der `log` eine Instanz ist, würde nach der Klasse `Principal` verwendet. Tatsächlich aber wird der `LogHandler` schon in der Methode `findSession` in Zeile 1 genutzt. Die Methode `findSession` gehört dem `BaseController`, von dem der `SessionController` erbt.

`ExplorViz` scheint die Verfolgung des Traces an dieser Stelle leichter zu machen. Seine Schwäche liegt aber darin, dass es nicht erkennbar macht, dass die Methode `debug` eine Methode der Super-Klasse des `SessionControllers` ist. Hätten wir nach der Reihenfolge der Klassen gefragt, die direkt in der Klasse `SessionController` auftreten und nicht in geerbten

Listing 6.1. Gekürzter Auszug aus der SessionController.groovy

```
1 SessionData sessionData = findSession(null)
2 Principal principal = principalRepo.findByEmail(identifier)
3
4 if(principal != null && passwordHash != null) {
5   if(passwordHash.equals(principal.passwordHash)) {
6     log.debug('Login succeeded')
7   }
8 }
```

Methoden, so wäre die Frage in ExplorViz nicht beantwortbar gewesen.

Ein ähnliches Bild zeigt sich auch in den Antworten der letzten Aufgabe des Fragenbereichs zur Kommunikation innerhalb der Microservices. Mit durchschnittlich 0,8 Fehlern fiel es den Nutzerinnen und Nutzern von ExplorViz leichter die Frage korrekt zu beantworten, als den Verwenderinnen und Verwendern von IntelliJ IDEA mit 2 Fehlern im Durchschnitt. Die Zeiten beider Gruppen unterscheiden sich nur gering, sind aber angesichts der Fehlerzahl auch nicht repräsentativ, da nicht ausgeschlossen werden kann, dass Teilnehmerinnen und Teilnehmer sich wenig Mühe gegeben haben. Das Hauptproblem bei der Code-Analyse scheint zu sein, dass die Entwicklerinnen und Entwickler Methoden übersehen. Daraus können wir eine Schwäche in der Verständlichkeit von textuellem Programmcode ableiten.

Verständnis der verteilten Kommunikation Der letzte Bereich der Analyse der Microservices prüft das Verständnis der verteilten Kommunikation und des Zusammenhangs der Microservices. Die erste Frage erwartet deshalb drei Microservices, mit denen der *Searchservice in Kommunikation steht*. Da in der Landschaftsansicht von ExplorViz nur ein bestimmtes Szenario abgebildet wird, können die Ergebnisse beider Gruppen voneinander abweichen. In ExplorViz sind der *Frontendproxy*, der *Marketingservice* und der *Ratingservice* zu erkennen.

Bei der Verwendung einer IDE lässt sich feststellen, dass auch zwischen dem *Authservice* und dem *Searchservice* Kommunikation stattfinden kann. Mit einer durchschnittlichen Zeit gab auch ein Benutzer von ExplorViz statt des *Frontendproxys* den *Authservice* an. Es ist nicht klar, wie der Entwickler darauf gekommen ist, zumal er keine Erfahrungen mit ExplorViz angegeben hat.

Mit einer Zeit von 58 Sekunden gelang es Teilnehmerinnen und Teilnehmern der ExplorViz-Gruppe im Mittel fast doppelt so schnell die Frage zu beantworten, wie den Mitgliedern der Kontrollgruppe.

Ebenfalls bei der zweiten Frage zur für die Kommunikation zwischen zwei Diensten verwendeten Technologie konnte die Versuchsgruppe die Aufgabe im Vergleich zur Kon-

trollgruppe deutlich schneller beantworten. Es ist aber zu erwähnen, dass die Streuungen um den Durchschnitt sehr groß sind, was die Aussagekraft der Ergebnisse dieser Frage verringert.

In der dritten Frage war gefragt, ob es einen Microservice gibt, der nur Nachrichten anderer Dienste empfängt, aber nie selbst sendet. Hierzu genügte es in `ExplorViz`, sich die In- und Out-Ports aller Microservices anzusehen und man stieß auf den *Authservice* und den *Marketingservice*. Letzterer sendet nur im dargestellten Szenario keine Nachrichten an andere Dienste. In der IDE mussten dafür die `.properties`-Dateien untersucht werden. Erstaunlicherweise gelang es Entwicklerinnen und Entwicklern der Kontrollgruppe die dritte Frage, mit durchschnittlich 115 Sekunden, trotz der hohen Standardabweichung von 158 Sekunden etwa 22 Sekunden schneller als jene der Versuchsgruppe zu beantworten.

Die letzte Frage des eigentlichen Versuchs fragt nach der letzten Klasse, die an ausgehender Kommunikation beteiligt ist. Drei der fünf Mitglieder der Kontrollgruppe konnten die Frage mit *IntelliJ IDEA* nicht beantworten. In der Versuchsgruppe hat ein Befragter falsch geantwortet. Alle anderen Teilnehmerinnen und Teilnehmer haben mit dem `RestTemplate` die richtige Antwort gegeben. Aufgrund der hohen Fehlerzahl lassen sich die Zeiten beider Gruppen schlecht vergleichen. Stattdessen erkennen wir aber, dass diejenigen, die die Frage richtig beantworten konnten, alle Bereits am Projekt `Symphony` mitgearbeitet haben. Lediglich ein Proband der `ExplorViz`-Gruppe konnte ohne Erfahrung die richtige Antwort finden.

Um die Frage mit `ExplorViz` zu beantworten genügt es, die letzte Klasse eines ausgehenden Programmflusses zu finden. Dies kann rein visuell geschehen. Mit `IntelliJ IDEA` ist die Beantwortung schwieriger. Mann muss durch Folgen der Methodenaufrufe feststellen, welche die letzte Klasse ist, bevor der Kontrollfluss das Java-Programm verlässt.

Durch die Erfahrung mit `Symphony` lässt sich schlecht sagen, wie viel Vorwissen die Probandinnen und Probanden in die Antwort konnten einfließen lassen. Gerade weil die beiden korrekten Antworten der Kontrollgruppe die erst- und drittschnellsten waren, müssen wir in Betracht ziehen, dass die beiden Entwickler die Antwort gewusst und deshalb eingetragen haben. Ob `ExplorViz` bei der Beantwortung dieser letzten Frage geholfen hat, lässt sich nicht sagen.

Persönliche Meinung

Werten wir die persönlichen Meinungen zur Befragung aus, so spiegeln diese die Ergebnisse wieder. Die Verwendung der Arbeitsumgebung und das Bearbeiten der Aufgaben wurde von beide Benutzergruppen ungefähr gleich bewertet. Verwenderinnen und Verwender der IDE gaben im Schnitt an, besser mit der Arbeitsumgebung zurechtzukommen, als die Befragten, die mit `ExplorViz` arbeiteten. Obgleich die Stichprobe sehr klein ist, können wir davon ausgehen, dass eine Entwicklungsumgebung der Probandinnen und Probanden vertrauter als die Visualisierung von `ExplorViz` ist. Das Beantworten der Fragen fiel allen `ExplorViz`-Nutzerinnen und -Nutzern eher leicht, während einige Mitglieder der Kontrollgruppe die Arbeit mit ihrer Arbeitsumgebung als eher schwer empfanden.

Deutlich besser schneidet ExplorViz beim Überblick über die Microservices und beim Verstehen von deren Zusammenhang ab. Dies bestätigt die Ergebnisse der Fragen zur Analyse. Während das Zurechtfinden innerhalb der Microservices von beiden Gruppen gleich bewertet wurde, fiel es besonders einem Senior Software-Entwickler, der ExplorViz schon mal verwendet hatte, sehr leicht, dem Programmablauf zu folgen. Es ist möglich, dass nach einer längeren Lern- und Einarbeitungsphase die Verwendung von ExplorViz einen noch größere Unterstützung bei der Bearbeitung der Fragen bringt.

Durchschnittliche Verbesserung der Probandinnen und Probanden

In jeder der beiden Gruppen befinden sich mindestens zwei Trainees, die bereits Erfahrung mit Symphony haben, aber noch nicht lange in der Softwareentwicklung arbeiten, ein Software-Entwickler, mit einigen Jahren Erfahrung, der ebenfalls bereits mit Symphony gearbeitet hat und ein Senior Software-Entwickler, der noch nie mit den Microservices von Symphony zu tun hatte.

Wir können deshalb für die drei Benutzergruppen errechnen, welche sich durch die Verwendung von ExplorViz im Durchschnitt am meisten verbessert hat. Dazu subtrahieren wir die durchschnittliche Bearbeitungsdauer der IntelliJ-IDEA-Gruppe von der durchschnittlichen Bearbeitungsdauer der ExplorViz-Gruppe für jede Aufgabe und bilden dann über alle Differenzen den Mittelwert. Positive Werte bedeuten eine Verbesserung der Dauer durch ExplorViz, negative das Gegenteil.

Es zeigt sich, dass nur die Gruppe der Trainees mit ExplorViz insgesamt einen zeitlichen Vorteil von 67 Sekunden gegenüber IntelliJ IDEA hat. Sowohl die befragten Software-Entwickler mit als auch die Senior Software-Entwickler ohne Symphony-Erfahrung schnitten mit 123 und 50 Sekunden im Gesamtdurchschnitt langsamer mit ExplorViz als mit der IDE ab. Aufgrund der kleinen Stichprobe ist dieses Ergebnis nicht repräsentativ. Zwar könnte sich diese Regel bei einer größeren Zahl an Probandinnen und Probanden bestätigen, aber die Werte könnten ebensogut durch die starken Ausreißer in Frage 17 und 18 zustande kommen.

6.3.7. Validität der Versuchsergebnisse

Wir werden im Folgenden die Validität der Versuchsergebnisse betrachten, die durch die unterschiedlichen Bestandteile des Versuchs beeinflusst werden kann.

Versuchsteilnehmerinnen und -teilnehmer Der wohl stärkste Einfluss auf die Validität der Ergebnisse ist die geringe Zahl der Probandinnen und Probanden. Zusammen mit den unterschiedlichen Fähigkeiten und Erfahrungen ist es schwierig, Muster in den Ergebnissen zu erkennen. Bei einer höheren Anzahl an Teilnehmerinnen und Teilnehmern wäre es wesentlich besser möglich, Aussagen darüber zu treffen, ob Personengruppen mit bestimmten Merkmalen eine Aufgabe leichter fällt als anderen. Mit einer Zahl von zehn

Personen müssen wir die individuellen Erfahrungen meist ignorieren und können nur eine Aussage über die gesamte Versuchs- beziehungsweise Kontrollgruppe treffen.

Gerade im Hinblick auf die vierte Forschungsfrage, welche Entwickler-Gruppe am meisten von einer Software-Visualisierung durch `ExplorViz` profitiert, spielt die geringe Zahl eine entscheidende Rolle. Es ist kaum möglich, die Teilnehmerinnen und Teilnehmer zu gruppieren. Das Ergebnis ist nicht repräsentativ für die Gesamtmenge an Entwicklerinnen und Entwicklern.

Fragestellung Eine entscheidende Rolle für die Validität der Ergebnisse spielt die Fragestellung. Werden vermehrt Fragen gestellt, die auf die Überlegenheit einer der beiden Arbeitsumgebungen abzielen, so kann dies den Eindruck erwecken, dass im Mittel eine Arbeitsumgebung der anderen überlegen ist.

Um eine derartige Beeinflussung der Testergebnisse zu verhindern, haben wir die Fragen nach unseren Forschungsfragen und in vier Bereiche gegliedert gestellt. Dennoch ist die Validität dadurch nicht unbedingt gegeben.

Ein weiterer Faktor, der die Fragestellung beeinflusst ist deren Alltagstauglichkeit. Es wäre möglich, dass einige der Fragen zu konstruiert sind und für eine Entwicklerin oder einen Entwickler im normalen Arbeitsleben keine Bewandtnis hat. Dann könnten die Ergebnisse zwar einen Vorteil einer Arbeitsumgebung zeigen, die aber in der Praxis keine Auswirkungen hat.

Versuchsaufbau Ein weiterer Faktor, der die Versuchsergebnisse verfälschen kann, ist der Versuchsaufbau. Die Probandinnen und Probanden wurden in Telefonkabinen befragt, die zwar gegen Geräusche gedämmt, aber durch eine Glasscheibe visuell verbunden sind. Durch die Befragung von zwei Teilnehmerinnen oder Teilnehmern gleichzeitig ist es möglich, dass die die Befragten unter Druck gesetzt fühlten und deshalb nicht mit voller Konzentration an den Aufgaben arbeiten konnten. Um Nervosität als Faktor ausschließen zu können, hätten man diese vor dem Versuch noch abfragen können.

Arbeitsumgebungen Betrachten wir die Erfahrungen der Entwicklerinnen und Entwickler mit ihren Arbeitsumgebungen, so ist schnell ersichtlich, dass diese in der Kontrollgruppe deutlich höher sind als in der Versuchsgruppe. Zwar haben die Befragten weniger Erfahrung mit `IntelliJ IDEA` als mit `Eclipse`, allerdings ist in der Versuchsgruppe bis auf eine Ausnahme gar keine Erfahrung mit `ExplorViz` vorhanden. Schon im Versuchsaufbau ist es schwierig bis unmöglich, gleiche Grundkenntnisse für beide Gruppen zu schaffen. Wir müssen in Betracht ziehen, dass die Kontrollgruppe bestimmte Aufgaben allein durch ihre Grunderfahrung mit der Arbeitsumgebung schneller bearbeiten konnte als die Versuchsgruppe.

Ergänzend sollten wir beachten, dass auch das Interesse an der Arbeitsumgebung für veränderte Antwortzeiten sorgen kann. Während die Versuchsgruppe die Arbeit mit einer

IDE bereits kennt, ist es möglich, dass die Kontrollgruppe nur aus Neugier mehr Zeit für die Beantwortung der Fragen benötigt.

Szenraio Nicht zuletzt kann auch die Wahl des Szenarios die Versuchsergebnisse beeinflussen. Um einen Versuch mit `ExplorViz` durchzuführen wird ein Szenario benötigt, das visualisiert wird. In einer IDE wie `IntelliJ IDEA` ist ein solches Szenario nicht gegeben. Wir müssend deshalb beachten, dass die gegebenen Umstände sich nicht unbedingt vergleichen lassen. Während der Prozess eines Logins in der Visualisierung bereits dargestellt wird, müssen Nutzerinnen und Nutzer der IDE diesen erst suchen, um ihm folgen zu können.

6.3.8. Zusammenfassung

Abschließend fassen wir die Ergebnisse der Mitarbeiterbefragung noch einmal zusammen und beantworten die Forschungsfragen mit unseren Ergebnissen. Durch die unterschiedlichen Fragenbereiche können wir ableiten, in welchen Bereichen `ExplorViz` einen Vorteil und in welchen es einen Nachteil gegenüber einer herkömmlichen Entwicklungsumgebung hat.

H 1 **Entwicklerinnen und Entwickler können mit der Landschaftsansicht von `ExplorViz` Fragen zur Softwarelandschaft von `Symphony` schneller beantworten als Entwicklerinnen und Entwickler, die eine IDE verwenden.**
Die Landschaftsansicht unterstützt die Entwicklerinnen und Entwickler beim Verständnis der Architektur von `Symphony`. Vor allem die Kommunikation zwischen den Microservices ist schnell verständlich. Meta-Daten wie Ports oder Protokolle können, sofern diese korrekt eingepflegt werden, leicht eingesehen werden, ohne dass dafür Code oder Konfigurationen durchsucht werden muss.

Die Hypothese H1 beurteilen wir als korrekt.

H 2 **Entwicklerinnen und Entwickler können mit der Anwendungsansicht von `ExplorViz` Fragen zu den einzelnen Microservices von `Symphony` schneller beantworten als Entwicklerinnen und Entwickler, die eine IDE verwenden.**
Die Anwendungsansicht in `ExplorViz` hilft Entwicklerinnen und Entwicklern bedingt. Eine Übersicht über die Paketstrukturen ist ähnlich derer in einer IDE. Es fehlt allerdings eine Suchfunktion, die es ermöglicht schneller innerhalb einer Applikation zu navigieren. Während der direkte Kontrollfluss mit `ExplorViz` gut verfolgbar ist und es Nutzerinnen und Nutzern erleichtert, zu verstehen wann welche Methode welcher Klasse aufgerufen wird, fällt es Entwicklern durch mangelnde Flexibilität schwerer, wirklich zu verstehen, was eine Klasse tut. Dazu tragen zwei Probleme bei. Erstens zeigt `ExplorViz` keine Super-Klassen oder Interfaces an, weshalb es schwierig ist, den Fluss in beiden Arbeitsumgebungen zu vergleichen. Zweitens verkompliziert die Verwendung von `Spring` durch Konzepte wie die *Dependency Inception* den Kon-

trollfluss ungemein. Ohne das Spring-Framework zu visualisieren, erscheinen die Methodenaufrufe in ExplorViz oft willkürlich.

Die Hypothese H2 werten wir als falsch. Es gibt keinen signifikanten Vorteil der Verwendung von ExplorViz gegenüber der Verwendung von IntelliJ IDEA.

H 3 **Die zeitliche Verbesserung durch Nutzung von ExplorViz gegenüber der Nutzung einer IDE fällt bei unerfahrenen Entwicklerinnen und Entwicklern am größten aus.** Die Gruppe der Trainees mit wenig Erfahrung in der Softwareentwicklung, die aber bereits mit Symphony gearbeitet haben, profitierten im Schnitt am meisten von einer Visualisierung durch ExplorViz. Es ist aber zu beachten, dass dieser Durchschnitt weder repräsentativ für alle Fragen ist, noch für eine größere Zahl an Probanden.

Hypothese H3 können wir bestätigen.

Fazit

Eine Visualisierung durch ExplorViz unterstützt Entwicklerinnen und Entwickler beim Verständnis der Microservices von Symphony. Vor allem ist diese hilfreich beim Überblick über die Softwarelandschaft. Es hat sich gezeigt, dass hauptsächlich unerfahrene Entwicklerinnen und Entwickler ihr Verständnis der Software durch eine Visualisierung erweitern können.

Kapitel 7

Verwandte Arbeiten

7.1. Ansätze in Kieker

Die Monitoring-Komponente von Kieker kommt bereits mit zwei verwandten Lösungen zu dieser Arbeit. Beide sind in keinem Paper dokumentiert und können nur über den *User-Guide* von Kieker, den Source-Code, sowie die Dokumentation zu Spring nachvollzogen werden. Wir wollen trotzdem beide Lösungen betrachten und diskutieren.

7.1.1. Spring-Interzeptoren

Die offizielle Dokumentation von Kieker beschreibt die Möglichkeit, Spring-eigene Mittel zu verwenden, um damit erstellte Anwendungen zu überwachen. Dies wird als Alternative zum Monitoring mit AspectJ genannt, allerdings nicht so umfangreich beschrieben. Demnach stellt Kieker sogenannte Interzeptoren (engl. Interceptors) bereit, die für verschiedene Zwecke in eine Spring-Applikation eingebunden werden können.

Interzeptoren sind Teil von Springs Lösung für aspektorientierte Programmierung. Dadurch können, wie auch in den von Kieker normalerweise verwendeten Aspekten, Pointcuts und Advices definiert werden. Aufgrund der in Kapitel 4 erwähnten Nachteile bei Verwendung von Interzeptoren für ausgehende Verbindung, entscheiden wir uns gegen eine Verwendung der Spring-Interzeptoren-Lösung. Zusätzlich wäre die Wiederverwendbarkeit einer Spring-spezifischen Lösung nur bedingt gegeben. Um ein nachhaltiges Ergebnis schaffen zu können, sollte die Lösung so universell wie möglich sein. Es spricht aus unserer Sicht auch nichts dagegen, die Aspekte von Kieker zu verwenden, da diese wie bei jeder Java-Anwendung funktionieren.

7.1.2. CXF-SOAP-Interzeptoren

Der zweite Ansatz, den Kieker dokumentiert ist die Verwendung des *Apache*-CXF-Frameworks, mit welchem Interzeptoren für auf SOAP basierende Web-Aufrufe umgesetzt werden können. Dabei nutzen die Interzeptoren auch den Header der SOAP-Nachrichten für den Austausch zwischen zwei Anwendungen. Die Trace-ID und der Order-Index des Senders werden vor dem Absenden in den Header geschrieben und beim Empfangen ausgelesen. Über den MonitoringController können dann, ähnlich der von uns verwendeten

Methode, entsprechende Event-Records erstellt werden. Die Technik lässt sich nicht auf die Microservices von Symphony übertragen, da diese REST für die Kommunikation nutzen.

Weiterhin unterscheidet sich die genannte Lösung zu unserem Ansatz dadurch, dass bei Empfangen einer Nachricht die Trace-ID in der Empfangenen Applikation ersetzt wird, so dass ein Trace über die Applikationen hinweg entsteht. Diese Methode funktioniert mit ExplorViz nicht, da dort die Traces getrennt übertragen und über BeforeSent- und BeforeReceivedRemoteCallRecords verbunden werden.

7.2. Monitoring von verteilten Traces mit Kieker

Strubel befasst sich in seiner Arbeit mit der allgemeinen Unterstützung von verteilten Traces in Kieker [Strubel 2017]. Dabei beschränkt er sich auf das Überwachen von Kommunikation über Transmission Control Protocol (TCP). Er analysiert sowohl die Monitoring- als auch die Analsyse-Komponente von Kieker und erläutert, was benötigt wird, damit Kieker verteilte Kommunikation registrieren und daraus verteilte Traces zusammenbauen kann.

In seiner Ausarbeitung kommt Strubel zu dem Schluss und bemängelt, dass es nicht möglich sei, eine einheitliche Lösung für das Monitoring aller Technologien zu finden. Vor allem die Beschränkung von AspectJ auf Klassen, die nicht aus den Standard-Bibliotheken von Java stammen, sorgt dafür, dass je nach System eigene Aspekte definiert werden müssten.

Dennoch legt Strubel mit seinem Vorschlag, vier Records für vor und nach dem Aus- und Eingehen von Nachrichten zu erstellen und die Trace-ID des MonitoringControllers als Identifikator über die Systeme hinweg zu nutzen, die Grundlage für die Implementierung dieser Arbeit.

7.3. Google Dapper und OpenZipkin

Google Dapper ist ein Werkzeug für das Monitoring von großen verteilten Infrastrukturen in C++ und Java [Sigelman u. a. 2010]. Dazu baut es einen Baum von Remote Procedure Calls (RPCs) auf, in dem alle miteinander verbundenen Knoten eine mit großer Wahrscheinlichkeit eindeutige Trace-ID teilen. Aus den Monioring-Daten generiert Dapper eine Tabelle, in der jede Zeile einen Trace abbildet und die Spalten mit den RPCs gefüllt sind. OpenZipkin verwendet den Entwurf von Google Dapper und bietet durch eine Reihe von Bibliotheken Unterstützung für viele beliebte Programmiersprachen.

Der Ansatz von Dapper und OpenZipkin verfolgt ein ähnliches Ziel wie das Monitoring von Kieker, lässt sich aber nicht auf die Weise mit ExplorViz verwenden, da automatisch Trace-Daten generiert werden und nicht die Möglichkeit besteht, bei Eintreten eines Events einen Record an ExplorViz zu schicken. Einzig könnte man einen Reader schreiben, der Logs von OpenZipkin einliest und für ExplorViz aufbereitet. Hier müsste geprüft werden,

ob die Bibliotheken von `OpenZipkin` in der Lage sind, die von `ExplorViz` benötigten Daten zu sammeln.

7.4. nuGEN

Die Firma *NuPSYS* verspricht mit ihrer Software `nuGEN` Echtzeit-Visualisierung von IoT-Netzwerken, also Infrastrukturen von vernetzten Alltagsgeräten. Die Software wird kommerziell vertrieben und es lassen sich keine Arbeiten oder Informationen zur Vorgehensweise der Entwickler finden. Dennoch zeigen die auf der offiziellen Website[1] vorgestellten Produktbilder vernetzte Knoten und Verbindungen zwischen ihnen. Die Kommunikation wird gerichtet und mit Protokoll abgebildet. Knoten geben Informationen über die aktuellen Datenübertragung in Bytes an.

Neben einer frei beweglichen 3D-Ansicht lässt sich aus einer Liste von Layouts wählen. Damit ähnelt die Aufgabe von `nuGEN` derer von `ExplorViz`, mit einem direkten Fokus auf verteilte Systeme. Es wäre interessant zu erfahren, auf welcher Basis die Daten über die Systeme gesammelt werden.

[1] `http://www.nupsys.com/nugen.html`, zuletzt besucht am 26.09.2017

Kapitel 8

Fazit and Ausblick

8.1. Fazit

In dieser Arbeit haben die Microservices von `Symphony` mit `Kieker` und `ExplorViz` visualisiert und mithilfe eines Fragebogens in einem Kontrollversuch evaluiert, inwieweit diese Visualisierung das Entwickler-Verständnis über die Microservices verbessern kann. Der Fokus des Entwurfs und der Implementierung lag dabei auf der Darstellung von verteilter Kommunikation zwischen den Diensten, da diese Funktion mit den beiden Tools bisher noch nicht möglich war.

Im Entwurf haben wir festgestellt, dass `Kieker` Event-Records besitzt, die äquivalent zu den für verteilte Kommunikation erstellten Records in `ExplorViz` sind. Daher haben wir zwei für diesen Zweck vorgesehene Event-Records erstellt, sowie ein Aspekt, welches vor dem Senden einer Anfrage den Header der Nachricht mit der Trace-ID und dem Order-Index des Sender befüllt und bei Empfangen einer Anfrage diese Daten aus dem Header ausliest. Für die Übertragung der Monitoring-Daten von `Kieker` nach `ExplorViz` konnten wir zwei bestehende Lösungen anpassen und wiederverwenden. Die Offline-Übertragung ermöglicht das Einlesen von `Kieker`-Logs und konvertiert die `Kieker`-Events zu `ExplorViz`-Records. Mithilfe eines geeigneten Writes ist es außerdem möglich, `Kieker` die Monitoring-Daten in einer Online-Übertragung direkt an `ExplorViz` schicken zu lassen. Mit beiden Lösungen ist uns eine Visualisierung inklusive der Kommunikation zwischen den Microservices gelungen.

Für die Evaluation haben wir Entwicklerinnen und Entwickler in eine Versuchs- und eine Kontrollgruppe eingeteilt und bestimmte Fragen zu den Microservices gestellt. Die Versuchsgruppe sollte diese Fragen mithilfe von `ExplorViz` bearbeiten, der Kontrollgruppe stand dafür die herkömmliche Entwicklungsumgebung `IntelliJ IDEA` zur Verfügung. Es hat sich gezeigt, dass `ExplorViz` vor allem in der Landschaftsansicht einen Vorteil gegenüber einer textuellen Ansicht des Projekts bringt. Beim Verfolgen von Kommunikation ist die Nutzung von `ExplorViz` häufig schneller als mit einer IDE, beim Verständnis der Funktion von bestimmten Klassen und Methoden konnte `IntelliJ IDEA` dennoch bessere Ergebnisse erzielen. Beim Vergleich der verschiedenen Entwickler-Gruppen hat sich gezeigt, dass die Gruppe der unerfahrenen Entwickler im Durchschnitt am meisten Vorteil aus einer Software-Visualisierung durch `ExplorViz` zieht.

8.2. Ausblick

Studie mit Trainingsphase

Die beiden größten Probleme der in dieser Arbeit durchgeführten Studie sind die geringe Anzahl von Teilnehmerinnen und Teilnehmer und die fehlende Vergleichbarkeit des Vorwissens der Befragten. Dadurch, dass Entwicklerinnen und Entwickler grundsätzlich den Umgang mit einer IDE gewohnt sind, ist es schwer zu sagen, ob die fehlende Erfahrung im Umgang mit `ExplorViz` dazu beigetragen hat, dass die IDE an einigen Stellen besser abgeschnitten hat. Interessant wäre ein vergleichbares Experiment mit mehr Personen und einer größeren Einarbeitungsphase in die Arbeit mit `ExplorViz`.

Verbesserung der Darstellung von Kommunikation in ExplorViz

Einer der Befragten merkte nach dem Ausfüllen des Fragebogens an, dass es sehr hilfreich wäre, wenn `ExplorViz` auf über die Applikation hinaus das Verfolgen des Traces zuließe. In der Tat hört die Trace-Verfolgung spätestens bei ein- bzw. ausgehenden Verbindungen auf und obwohl deren Ports eine Verbindung zu einer anderen Applikation andeuten, ist es nicht möglich, dem Fluss dahin zu folgen. Um in Zukunft mehr mit verteilter Kommunikation in `ExplorViz` arbeiten zu können, müssen Funktionen wie diese funktionieren. Außerdem ist bei der Implementierung aufgefallen, dass entgegen der Legende in der Landschaftsansicht, Nachrichten zwischen den Microservices nicht gerichtet angezeigt werden.

Als weitere Anmerkungen nannten befragte Probandinnen und Probanden Schwächen bei der Darstellung auf kleinen Bildschirmen. So war es teilweise nicht möglich, den Scroll-Balken im Trace-Fenster zu sehen, weil dieser außerhalb des Bildes lag. Einer der Befragten bemängelte außerdem, dass die Zoom-Stufe und der Drehwinkel beim Weg- und Zurückwechsel von Applikationen nicht bestehen bliebe. Zum Zeitpunkt des Schreibens arbeitet das Team um `ExplorViz` bereits an einer Neuauflage der Web-Applikation. Die genannten Probleme könnten dort besser gelöst sein.

Nicht zuletzt hatten einige Befragte Probleme damit, zu verstehen, dass die dünnen Kommunikationslinien vergangener Zeitfenster nicht anklickbar und damit verfolgbar waren. Dieses Konzept müsste bei einer späteren Studie genauer erklärt werden. Vielleicht wäre es aber auch möglich, wenn durch einen Klick auf eine Kommunikationslinie frühere Zeitfenster angezeigt würden, in denen diese aktiv war. Ohne eine solche Funktion muss ein Nutzer sich durch die Zeitleiste klicken, bis die besagte Linie Informationen preisgibt.

Verbesserung der Verarbeitung von eingehenden Records in ExplorViz

Bei der Offline-Übertragung von `Kieker` zu `ExplorViz` ist eine weitere Schwäche des Visualisierungs-Tools aufgefallen. `ExplorViz` berücksichtigt beim Empfangen von Monitoring-Daten keine Zeitstempel der Records. Dies sorgt dafür, dass alles in einem Zeitfenster

übertragene Verhalten einer Applikation gleichzeitig zu passieren scheint. Bei einer Online-Übertragung ist dieses Verhalten nicht störend, bei einer Offline-Übertragung hingegen wird so ein über längere Zeit aufgezeichnete Monitoring fälschlicherweise in ein Zeitfenster geworfen. Sollte das Einlesen von `Kieker`-Logs eine in Zukunft unterstützte Funktion von `ExplorViz` bleiben, so muss `ExplorViz` die Daten auch auf Zeitfenster aufteilen, sodass die Zeitleiste verwendet werden kann.

Verbesserung der Übertragung der Monitoring-Daten

`Kieker` verwendet für die Übertragung der Monitoring-Daten zwischen der Monitoring- und der Analyse-Komponente eine eigene Pipe-und-Filter-Architektur. Auch die Offline-Übertragung des *External Monitoring Logs Adapters* macht Gebrauch vom `AnalysisController` von `Kieker`, um den `FSReader` mit dem `KiekerToExplorVizTransformFilter` zu verbinden.

Um in Zukunft mehr Geschwindigkeit bei der Übertragung zu erreichen und zu verhindern, dass die Pipe-und-Filter-Architektur extra für `Kieker` und `ExplorViz` weiterentwickelt werden muss, wird das Team hinter `Kieker` diese ab der Version 1.14 durch die Infrastruktur von TeeTime ersetzen [Wulf u. a. 2017]. Durch die Unterstützung multipler Output-Ports und Verzweigungen erweitern sich dadurch die Möglichkeiten von `Kieker` noch. So wäre es beispielsweise möglich, mehrere Writer in `Kieker` zu verwenden, um einerseits eine Visualisierung mit `ExplorViz` durchzuführen und gleichzeitig die `Kieker`-Analyse zu verwenden.

Erweiterung von ExplorViz um die Visualisierung von Daten

Um den Nutzen von `ExplorViz` in Zukunft weiter auszuweiten, ist es sinnvoll, neben der bisherigen Visualisierung neue Perspektiven auf Applikationen zu schaffen. Aktuell betrachtet `ExplorViz` lediglich das Verhalten von Software, nicht aber die darin ausgetauschten Daten. Für eine bessere Analyse wäre es hilfreich, eine Datenfluss-basierte Analyse und ein geeignetes Monitoring zu schaffen. Außerdem arbeitet Zirkelbach an einer Erweiterung von `ExplorViz` um ein Monitoring, die Analyse und die Visualisierung von Datenbanken [Zirkelbach 2017].

Anpassungen bezüglich Spring

Das Monitoring des Spring-Frameworks konnte nicht fehlerfrei durchgeführt werden. Außerdem bedurfte es der *noverify*-Argumentes, damit der von `AspectJ` erzeugte Code nicht von der JVM abgelehnt wurde. Um `ExplorViz` produktiv im `Symphony`-Projekt einzusetzen, wäre es hilfreich, wenn diese Probleme gelöst würden.

Universell nutzbares Monitoring

Zuletzt ist es wichtig zu erkennen, dass mit der in dieser Arbeit gefundenen Implementierung keine universelle Lösung für das Monitoring von verteilter Kommunikation

gefunden wurde. Genau wie für die verschiedenen Technologien, mit denen Dienste in einer Microservice-Architektur aufgebaut sein können, scheint auch für jede Art der Kommunikation eine eigene Lösung gefunden werden. An dieser Stelle kann und sollte in jedem Fall mehr Forschung stattfinden, denn die Diversität von Sprachen und Technologien wächst stetig.

Anhang A

Anlage zur Befragung

Informationen zur ExplorViz-Befragung

Vielen Dank für Ihre Teilnahme an dieser Befragung.
Mit dem Ausfüllen und Absenden erklären Sie sich damit einverstanden, dass die Angaben für statistische Zwecke ausgewertet und weiterverwendet werden können.

Ziel

Mit dieser Befragung wird der Nutzen einer interaktiven Visualisierung durch ExplorViz[1] für das Projekt Symphony getestet. Ziel ist es, zu evaluieren, inwieweit ExplorViz beim Verständnis von Software unterstützen kann sowie eine Abgrenzung gegenüber herkömmlichen, textuellen Analysemethoden bringt.

Ablauf

Ihre Aufgabe in dieser Befragung ist es, einen Ausschnitt der Microservice-Architektur von Symphony zu analysieren. Dazu erhalten Sie nach einer Einführung in Ihre Arbeitsumgebung einen Arbeitsplatz mit einem Rechner. Der Rechner wird einen digitalen Fragebogen und Ihr Arbeitswerkzeug bereitstellen.

Die Befragung selbst besteht aus drei Teilen. Im ersten Teil werden Sie gebeten, einige persönliche Angaben über sich und Ihre Erfahrungen in der Software-Entwicklung zu machen. Der zweite Teil bezieht sich auf die Analyse der Microservices. Zum Beantworten der Fragen dürfen Sie ausschließlich das zur Verfügung gestellte Werkzeug verwenden. Abschließend werden Sie im dritten Teil nach Ihrer persönlichen Einschätzung bezüglich der Analyse der Microservices mithilfe des von Ihnen verwendeten Werkzeugs gefragt.

Nehmen Sie sich für die Beantwortung der Fragen die Zeit, die Sie benötigen. Für ein aussagekräftiges Ergebnis der Befragung sind Sie dazu angehalten, alle Fragen sorgfältig und nach bestem Wissen zu beantworten.

Datenschutz

Die in dieser Befragung gesammelten Daten werden ohne Verbindung zu Ihrer Person abgespeichert und anonym verarbeitet. Bitte beachten Sie auch, dass außer Ihren Angaben im Fragebogen und dessen Bearbeitungszeiten keine weiteren Daten gesammelt werden. Alle Daten werden ausschließlich im Rahmen der Auswertung dieser Befragung verwendet und anschließend gelöscht.

[1] ExplorViz ist ein Software-Visualisierungs-Tool, welches am Lehrstuhl für Software-Engineering an der CAU Kiel entwickelt wird.

Einführung in ExplorViz

ExplorViz visualisiert den Zustand von Software grafisch. Mit Zustand ist immer das Verhalten innerhalb eines festen Zeitintervalls gemeint. In dem in diesem Versuch betrachteten Beispiel ist dieses Intervall zehn Sekunden lang.

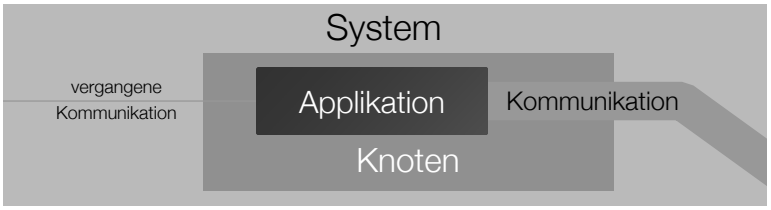
Darstellung

Für die Visualisierung von Software verwendet ExplorViz zwei verschiedene Ansichten unterschiedlichen Detailgrades. Die zweidimensionale Landschaftsansicht zeigt die Applikationen einer Softwarelandschaft. Um einzelne Applikationen zu betrachten, bietet ExplorViz die dreidimensionale Anwendungsansicht.

Landschaftsansicht

Die Landschaftsansicht zeigt die Anwendungen einer Softwarelandschaft. Anwendungen befinden sich auf Knoten, die sich wiederum in Systemen befinden.
Die im Rahmen der Befragung vorgegebene Visualisierung zeigt ein System, welches fünf Knoten mit je einer Applikation, den Microservices, beinhaltet. Üblicherweise stellen die Knoten in einer verteilten Architektur unterschiedliche Rechner dar. In unserem Fall betrachten wir aber die einzelnen Ports als Knoten in einem System.

Kommunikation wird als orangene Linien dargestellt. Je dicker eine Kommunikationslinie gezeigt wird, desto mehr Kommunikation pro Zeiteinheit findet statt. Ganz dünne Linien, wie sie auf der linken Seite der oberen Grafik zu sehen ist, zeigen vergangene Kommunikation, die vor dem betrachteten Zeitintervall stattgefunden hat.

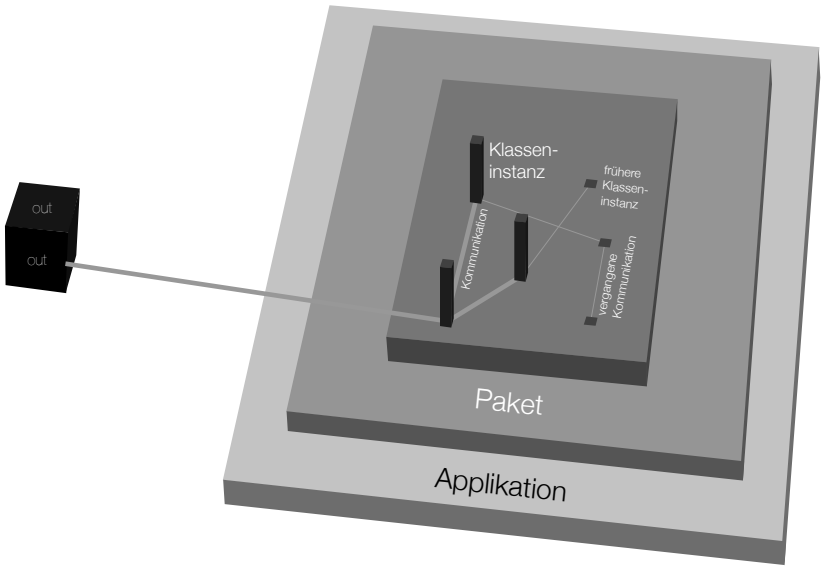


Anwendungsansicht

Die Anwendungsansicht zeigt die Pakete und Klassen innerhalb einer Applikation. Klassen werden als lila Blöcke dargestellt. Je höher der Block einer Klasse ist, desto mehr Instanzen ihrer haben in dem angezeigten Intervall existiert. Klassen, die zwar aufgetreten sind, aber im aktuell betrachteten Intervall keine verwendeten Instanzen besitzen, werden lediglich als lila Flächen gezeigt.

Auch in der Anwendungsansicht zeigt die Dicke der Kommunikationslinien die Stärke der Kommunikation. Ganz dünne Linien stehen wie in der Landschaftsansicht für vergangene Kommunkation.

Gibt es Kommunikation über Applikationen hinweg, so verlässt oder kommt diese über einen schwarzen Würfel mit der Beschriftung *out* oder *in*.



Bedienung

Normalerweise zeigt die Visualisierung in ExplorViz immer das zuletzt aufgezeichnete Zeitintervall an. Die Zeitleiste am unteren Rand des Fensters kann für das Pausieren und für das Springen in den Intervallen verwendet werden. Für diesen Versuch ist die Visualisierung allerdings so modifiziert, dass immer dasselbe Zeitintervall angezeigt wird. Sie brauchen die Zeitleiste während der Befragung nicht zu beachten.

Landschaftsansicht

Benutzen Sie das **Mausrad**, um das Modell zu vergrößern oder zu verkleinern.
Halten Sie die **linke Maustaste** gedrückt, um das Modell mit der Maus zu bewegen.
Bewegen und halten Sie die Maus über Komponenten, um Informationen über diese zu erhalten.

Durch einen **Doppelklick** auf eine Applikation kann von der Landschaftsansicht zur Anwendungsansicht gewechselt werden.

Hinweis: Am rechten unteren Rand des Fensters befindet sich eine Legende, die angibt, dass Kommunikation gerichtet als Pfeil dargestellt wird. Dies ist in der Ihnen vorliegenden Visualisierung nicht der Fall. Diese Art der Hilfestellung steht Ihnen nicht zur Verfügung.

Anwendungsansicht

Benutzen Sie das **Mausrad**, um das Modell zu vergrößern oder zu verkleinern.
Halten Sie die **linke Maustaste** gedrückt, um das Modell mit der Maus zu bewegen.
Bewegen und halten Sie die Maus über Komponenten, um Informationen über diese zu erhalten.
Halten Sie die **rechte Maustaste** gedrückt, um das Modell mit der Maus zu drehen.
Ein **Doppelklick** auf ein Paket öffnet oder schließt dieses.

Ein **Klick** auf den Button **Back to Landscape** an der oberen linken Seite wechselt zurück zur Landschaftsansicht.

Um alle Komponenten auszuklappen existiert an der oberen rechten Seite ein Button mit der Beschriftung **Open All Components**.

Ein Klick auf eine der in dem gezeigten Intervall verwendeten Klassen blendet alle Kommunikation, die nicht mit dieser Klasse zu tun hat, aus. Zusätzlich zeigen Pfeile auf den Kommunikationslinien deren Richtungen an.

Ein Klick auf eine Kommunikationslinie zeigt eine Liste aller darin enthaltenen Methodenaufrufe. Über den Button *Choose*, kann die Kommunikation ab dieser Stelle weiter verfolgt werden.

Hinweis zur Verfolgung des Kontrollflusses: Durch einen Klick auf den *Choose*-Button eines Methodenaufrufs ist es möglich den Trace weiterzuverfolgen. Es öffnet sich ein Fenster, welches Informationen zur aktuell gezeigten Methode anzeigt und die Möglichkeit bietet, mittels der Buttons *Next* und *Previous* zum nächsten oder vorigen Methodenaufruf zu gehen. Diese Funktion ist hilfreich, um die Reihenfolge von Methodenaufrufen zu überprüfen.

Einführung in IntelliJ IDEA

IntelliJ IDEA ist eine bekannte Entwicklungsumgebung (von jetzt an IDE), hauptsächlich für Java-Projekte.

Aufbau

Die Ansicht der Ihnen zur Verfügung gestellten IDEA-Instanz ist zweigeteilt. Auf der linken Seite finden Sie die Projektübersicht und eine Strukturansicht, die rechte Seite steht für geöffnete Dateien zur Verfügung.

Projektübersicht

Das geöffnete Projekt enthält die fünf für die Befragung relevanten Microservices als Module. Jedes der Module ist an dem Ordnersymbol mit blauem Quadrat zu erkennen. Die einzelnen Microservices sowie alle enthaltenen Pakete und Ordner lassen sich aus- und einklappen.

Jeder Microservice ist als Spring-Boot-Projekt angelegt und wird mit dem Build-Tool Gradle bereitgestellt. Auf der untersten Ebene eines jeden Microservice befindet sich deshalb die **build.gradle**-Datei. Diese beinhaltet alle Anweisungen und Abhängigkeiten für die Bereitstellung des Dienstes. Der geschriebene **Programmcode** befindet sich im Verzeichnis `src/main/java` bzw. `src/main/groovy`.

Jeder Microservice besitzt bestimmte **Projekt-Eigenschaften**. Diese werden in den *.properties*-Dateien im Verzeichnis `src/main/resources` definiert. Diese Eigenschaften beinhalten Informationen über den Dienstnamen, den zu verwendenden Port und zu anderen benötigten Microservices.

Dateistruktur

Ist eine Datei geöffnet, so kann über den Button *Structure* am linken Fensterrand zur **Strukturansicht** der Datei gewechselt werden. Diese gibt einen Überblick über die enthaltenen Felder und Methoden.

Ein **Klick** auf die Schaltfläche *Project* am linken Fensterrand schaltet wieder auf die Projektübersicht um.

Dateiansicht

Sind Dateien geöffnet, werden diese im rechten Teil des Fensters gezeigt. Bei mehreren Dateien, kann am oberen Rand der Dateiansicht über die Tableiste zwischen geöffneten Dateien gewechselt werden.

Bedienung

Projektübersicht und Dateistruktur

↕ Benutzen Sie das **Mausrad**, um innerhalb der Ansicht zu scrollen.

Klicken Sie auf das Dreieck vor einem Order- oder Paketnamen, um diesen oder dieses aus- bzw. einzuklappen.

Um Klassen, Dateien, Felder oder Methoden zu **suchen**, bietet IDEA die Möglichkeit durch zweimaliges schnelles Tippen der Umschalt-Taste (Shift-Taste) ein Suchfenster zu öffnen. Alternativ stehen über das Menü am oberen Fensterrand *Navigate* die Punkte *Class*, *File* oder *Symbol* für Suchen zur Verfügung.

Um innerhalb eines in der Projektübersicht markierten Verzeichnisses nach einer Zeichenfolge zu suchen, kann die Tastenfolge Strg+Umschalt+F (Ctrl+Shift+F) gedrückt werden.

Klicken Sie auf das Fadenkreuz am oberen Rand der Projektübersicht, um die aktuell geöffnete Datei in der Projektstruktur anzuzeigen.

Klicken Sie mit der rechten Maustaste auf eine Klasse in der Projektübersicht und dann auf den Menüpunkt *Find Usages*, um sich alle Stellen anzeigen zu lassen, in denen die gewählte Klasse verwendet wird.

Hinweis: Wenn Sie eine Suche durchführen, öffnet sich möglicherweise im unteren Teil des Fensters ein Bedienfeld. Um dieses wieder zu schließen, genügt ein Klick auf die Schaltfläche *3: Find* am unteren Fensterrand.

Dateiansicht

↕ Benutzen Sie das **Mausrad**, um innerhalb der Ansicht zu scrollen.

Halten Sie die Umschalt-Taste (Shift-Taste) gedrückt und klicken Sie auf einen Klassen- oder Methodennamen, um direkt zu dessen Deklaration zu springen.

Klicken Sie mit der rechten Maustaste auf einen Klassen- oder Methodennamen und dann auf den Menüpunkt *Find Usages*, um sich alle Stellen anzeigen zu lassen, in denen die Klasse oder die Methode verwendet wird.

Hinweis zur Verfolgung des Kontrollflusses: Um den Kontrollfluss innerhalb der Microservices nachverfolgen zu können, ist etwas Suchen nötig. Häufig hilft es aber schon, mithilfe des Shift-Klicks von Methode zu Methode zu springen. Möchte man rausfinden, woher ein Kontrollfluss gekommen ist, muss man versuchen die Methodenaufrufe rückwärts nachzuvollziehen. Dafür ist die oben beschriebene *Find-Usages*-Funktion hilfreich.

Anhang B

Ergebnisse der Befragung

Persönliche Daten

Verwendete Arbeitsumgebung		Sehschwäche		Geschlecht		Alter		Jahre in der Softwareentwicklung		Rolle bei adesso	
Antwort	Dauer	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer
ExplorViz	5223		3728	männlich	2536	35-50	3176	20+	21888	Senior SE	9648
ExplorViz	3885		4232	männlich	3400	35-50	3808	20+	3456	Senior SE	5880
ExplorViz	108916		5712	männlich	5952	18-24	17624	0-2	13815	Trainee	4714
ExplorViz	22270		10703	männlich	2633	25-34	3271	3-5	14872	SE	5929
ExplorViz	9060		4296	weiblich	3511	25-34	3536	0-2	6817	Trainee	4272
IntelliJ IDEA	6421		8127	männlich	3840	25-34	3415	0-2	5392	Trainee	5418
IntelliJ IDEA	32164		4248	männlich	2760	35-50	3416	11-20	4240	Senior SE	5640
IntelliJ IDEA	4069		6273	weiblich	4120	35-50	4632	0-2	8312	Trainee	4384
IntelliJ IDEA	4076		5016	männlich	2529	25-34	3631	3-5	5785	SE	8807
IntelliJ IDEA	3887		4408	männlich	3424	25-34	5015	11-20	7376	Trainee	14848

Fähigkeiten							
IntelliJ IDEA	Eclipse	Java	Groovy	Spring	Microservices	ExplorViz	Dauer
keine = 0	**gering = 1**	**gut = 2**	**professionell = 3**				
2	2	2	2	2	2	1	41768
1	3	3	1	1	1	0	20416
1	2	2	1	1	1	0	38437
0	2	2	0	2	1	0	46225
1	2	2	1	2	2	0	19272
0	2	3	1	2	1	0	21966
2	2	3	1	2	2	0	22263
1	2	1	1	1	1	0	48866
3	3	3	2	3	3	1	19335
2	2	2	2	2	3	0	27130

Mitarbeit an Symphony							Monate Arbeit an Symphony		Anderes Visualisierungs-Tool		
Frontend	Auth	Marketing	Search	Rating	Andere	Dauer	Antwort	Dauer	Antwort	Name	Dauer
x	x	x	x		x	37895	12+	13089		---	10556
						6311	---	---	x	W3L Activity Lab	25048
					x	25768	0-3	7400		---	18208
					x	9975	0-3	8104		---	9793
				x		10784	4-6	6248		---	6455
				x		13987	4-6	7641		---	12383
						4561	---	---		---	6127
					x	19528	7-12	32927		---	13547
x			x		x	14089	7-12	8416	x	AppDynamics	13512
						6729	---	---	x	UMLet FreeMind	48872

Versuch	Port des Authservice		Zwei Dienste mit selbem Port		Port für neuen Dienst		Paket der Klasse ReindexTaskQueue	
	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer
	8083		**nein**				**de.adesso.symphony.searchservice.index**	
	8083	19309	nein	17953	8085	18301	de.adesso.symphony.searchservice.index	115486
	8083	17880	nein	102319	8090	57192	de.adesso.symphony.searchservice.index	239013
	8083	29552	nein	25735	8090	37224	de.adesso.symphony.searchservice.index	277916
	8083	150189	nein	9280	8090	105207	de,adesso.symphony.searchservice.index	167585
	8083	35446	nein	15297	8090	48222	de.adesso.symphony.searchservice.index	172232
	8083	44129	nein	122405	8090	40752	de.adesso.symphony.searchservice.index	124710
	8083	93599	nein	104534	8090	59824	de.adesso.symphony.searchservice.index	134518
	8083	96533	nein	94211	noch nicht belegten Port	139653	de.adesso.symphony.searchservice.index	205678
	8083	22071	nein	75503	8090	68232	de.adesso.symphony.searchservice.index	78494
	8083	23681	nein	82001	8090	129875	de.adesso.symphony.searchservice.index	48471

Klassen im Paket .frontendproxy.product.model.search				Aufgabe des Authwrappers	
Klasse 1	Klasse 2	Klasse 3	Dauer	Antwort	Dauer
SearchResult	**Facets**	**DidYouMean** (nur IDE)			
SearchResult	Facets	---	85653	Der AuthWrapper verbirgt den AuthService vor den anderen Klassen innerhalb des RatingServi	319788
SearchResult	Facets	---	89887	Leitet Authentifizierungsanfragen an den AuthService weiter.	266941
SearchResult	Facets	---	100304	Der AuthWrapper nimmt login-Anfragen innerhalb des Ratingservice entgegen und wickelt dies	523282
SearchResult	Facets	---	64097	Dient als Proxy für das Login durch den RatingService. Speichert die Authentifizierungsdaten in	457511
SearchResult	Facets	---	51495	Bekommt Anfragen vom RestController im Ratingservice und leitet sie weiter an den web client	187974
SearchResult	Facets	DidYouMean	45822	Man fragt beim AuthWrapper ab ob ein Service Methode XY aufrufen darf	75266
SearchResult	Facets	DidYouMean	89135	Prüft, ob ein Nutzer eingeloggt ist oder Aktionen für den User ausgeführt werden können und l	278933
SearchResult	---	---	66689	Registriert Ratingservice in AuthentifizierungService	199974
SearchResult	Facets	DidYouMean	71944	Weiterleitung von Anmeldeanfragen und Rechteabfragen an den AuthService	273782
SearchResult	Facets	DidYouMean	70674	Der AuthWrapper bietet mit der Methode isPermitted(...) die Abfrage der Userrechte zu einer l	134786

Reihenfolge der Klassen beim Login						
Company	Department	LogHandler	Principal	SessionController	SessionData	Dauer
5	**4**	**2**	**3**	**0**	**1**	
1	4	2	3	5	0	241825
5	4	2	3	0	1	661393
1	4	2	3	5	0	494770
1	4	2	3	5	0	744193
1	4	2	3	5	0	265885
1	3	4	5	2	0	262051
5	4	3	2	0	1	361756
3	5	0	2	4	1	1484785
4	5	2	3	0	1	408547
2	3	4	1	5	0	259095

Sequenzdiagramm						
Platzhalter 1	Platzhalter 2	Platzhalter 3	Platzhalter 4	Platzhalter 5	Platzhalter 6	Dauer
LoginProperties	**login()**	**increaseClicks()**	**getClicks()**	**setClicks()**	**getClicks()**	
LoginProperties	login()	increaseClicks()	getClicks()	setClicks()	getClicks()	498247
LoginProperties	login()	increaseClicks	getClicks()	setClicks()	getClicks()	811999
LoginProperties	login()	findProductPopularity()	getClicks()	getReviews()	setClicks()	594594
RestTemplate	login()	increaseClicks()	getClicks()	setClicks()	getClicks()	473042
LoginProperties	login()	increaseClicks(..)	getId(..)	getClicks(..)	setClicks(..)	376516
LoginProperties	login()	increaseClicks(productID	setClicks(productPopularity.getClicks()	productPopularity.getClicks()	popularityRepository.save(productPopularity)	289141
LoginProperties	login()	increaseClicks(productID	getClicks()	setClicks()	getClicks()	631313
LoginProperties	login()	getLogger()	findProductPopularity()	getClicks()	setClicks()	1079074
LoginProperties	login()	increaseClicks(...)	setClicks(...)	getClicks()	save(...)	347420
AuthService	login()	findProductPopularity()	getClicks	setClicks	save	309372

Searchservice Kommunikationspartner				Technologie	
Service 1	Service 2	Service 3	Dauer	Antwort	Dauer
FrontendProxy / AuthService	**MarketingService**	**RatingService**			
FrontendProxy	MarketingService	RatingService	42472	REST	106847
AuthService	MarketingService	RatingService	59103	Rest	19584
Frontendproxy	MarketingService	RatingService	47375	Rest	25097
FrontendProxy	MarketingService	RatingService	37199	REST	8992
FrontendProxy	MarketingService	RatingService	103950	REST	18522
AuthService	MarketingService	RatingService	37344	REST	11287
AuthService	MarketingService	RatingService	91711	REST	56864
Ontology	MarketingService	RatingService	176868	Spring	163680
AuthService	MarketingService	RatingService	59615	REST	8344
AuthService	MarketingService	RatingService	148252	http / REST	24369

Dienst, der nur empfängt			Letzte Klasse	
Antwort	Name	Dauer	Antwort	Dauer
ja	**AuthService (Marketing bei ExplorViz)**		**RestTemplate**	
nein	---	154253	RestTemplate	239558
ja	AuthService, MarketingService	106159	RestTemplate	234758
ja	AuthService, MarketingService	255180	RestTemplate	130271
ja	AuthService	67746	AuthWrapper	717440
ja	MarketingService, AuthService	100837	RestTemplate	124656
ja	Authservice	393346	RestTemplate	59928
ja	Authservice	73494	---	84975
nein	---	73009	---	373328
ja	AuthService	10256	RestTemplate	104511
ja	AuthService	23936	---	287084

Meinung	Arbeitsumgebung		Aufgaben		Überblick über die Dienste		Zurechtfinden innerhalb der Dienste	
	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer	Antwort	Dauer
	3	4888	3	7760	2	8400	3	7104
	3	14504	3	8328	2	5952	3	5472
	3	28408	3	7897	1	11847	2	10087
	4	12527	3	6377	2	22709	3	15987
	2	46465	3	6680	2	11887	2	7530
	3	15640	3	11128	4	14304	2	7608
	2	6384	4	8456	5	8006	3	10201
	4	13664	4	6720	3	18607	4	10144
	1	19000	4	5128	5	5712	3	13575
	3	11314	2	6336	2	10695	1	13127

Zusammenhang der Dienste		Programmablauf folgen		
Antwort	Dauer	Antwort	Dauer	Ende
3	15208	1	12215	25.08.17 11:47
3	7119	4	8000	25.08.17 11:57
3	11265	4	37631	25.08.17 13:03
3	9689	4	5240	25.08.17 13:06
3	12982	3	14759	25.08.17 14:46
4	8328	3	8915	25.08.17 13:54
4	8057	3	6303	25.08.17 14:03
4	9883	4	9079	25.08.17 15:26
6	8560	3	5697	29.08.17 10:29
2	8201	4	12448	29.08.17 10:30

Anhang C

POM-Dateien für Maven

Im folgenden sind die POM-Dateien für Kieker und ExplorViz aufgelistet. Man beachte den Platzhalter jeweils in Zeile 19, der durch den Pfad zum `libs`-Verzeichnis des `Symphony`-Projekts ersetzt werden muss.

C.1. POM für die Kieker-Monitoring-Komponente

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
       .org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
3  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>kieker</groupId>
6    <artifactId>kieker</artifactId>
7    <version>1.13</version>
8
9    <description>
10   Kieker Monitoring Library
11   </description>
12
13   <properties>
14     <groupId>${project.groupId}</groupId>
15     <artifactId>${project.artifactId}</artifactId>
16     <version>${project.version}</version>
17     <packaging>${project.packaging}</packaging>
18     <file>build/libs/kieker-1.13-SNAPSHOT-aspectj.jar</file>
19     <localRepositoryPath><!-- libs-Ordner von Symphony --></localRepositoryPath>
20   </properties>
21
22   <build>
23     <plugins>
24       <plugin>
25         <groupId>org.apache.maven.plugins</groupId>
```

```
26        <artifactId>maven-install-plugin</artifactId>
27        <executions>
28          <execution>
29            <phase>install</phase>
30            <goals>
31              <goal>install-file</goal>
32            </goals>
33          </execution>
34        </executions>
35      </plugin>
36    </plugins>
37  </build>
38
39 </project>
```

C.2. POM für die ExplorViz-Monitoring-Komponente

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
       .org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
3  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>explorviz</groupId>
6    <artifactId>monitoring</artifactId>
7    <version>1.0</version>
8
9    <description>
10   ExplorViz
11   </description>
12
13   <properties>
14     <groupId>${project.groupId}</groupId>
15     <artifactId>${project.artifactId}</artifactId>
16     <version>${project.version}</version>
17     <packaging>${project.packaging}</packaging>
18     <file>dist/explorviz-monitoring.jar</file>
19     <localRepositoryPath><!-- libs-Ordner von Symphony --></localRepositoryPath>
20   </properties>
21
22   <build>
```

```
23      <plugins>
24        <plugin>
25          <groupId>org.apache.maven.plugins</groupId>
26          <artifactId>maven-install-plugin</artifactId>
27          <executions>
28            <execution>
29              <phase>install</phase>
30              <goals>
31                <goal>install-file</goal>
32              </goals>
33            </execution>
34          </executions>
35        </plugin>
36      </plugins>
37    </build>
38
39  </project>
```

Literaturverzeichnis

[Baresi u. a. 2006] L. Baresi, E. Di Nitto und C. Ghezzi. Toward open-world software: Issues and challenges. *Computer* 39.10 (2006), Seiten 36–43. (Siehe Seite 2)

[Charness u. a. 2012] G. Charness, U. Gneezy und M. A. Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization* 81.1 (2012), Seiten 1 –8. URL: http://www.sciencedirect.com/science/article/pii/S0167268111002289. (Siehe Seite 61)

[De Vries u. a. 2014] J. de Vries, C. Burki und B. de Vries. How to save on software maintenance costs. Omnext, 2014. URL: http://asq.org/public/wqm/how-to-save-on-software-maintenance-costs.pdf. (Siehe Seite 1)

[Dodge und Kitchin 2004] M. Dodge und R. Kitchin. Code, space and everyday life (Mai 2004). (Siehe Seite 1)

[Fittkau u. a. 2013] F. Fittkau, J. Waller, C. Wulf und W. Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In: *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE. 2013, Seiten 1–4. (Siehe Seiten 2, 7, 8)

[Fittkau u. a. 2015a] F. Fittkau, A. Krause und W. Hasselbring. Exploring Software Cities in Virtual Reality. In: *IEEE 3rd Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, 2015, Seiten 130–134. URL: http://eprints.uni-kiel.de/29388/. (Siehe Seite 7)

[Fittkau u. a. 2015b] F. Fittkau, S. Roth und W. Hasselbring. ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes. In: *23rd European Conference on Information Systems (ECIS 2015)*. 2015. URL: http://eprints.uni-kiel.de/28067/. (Siehe Seiten 6 und 26)

[Fittkau u. a. 2015c] F. Fittkau, E. Koppenhagen und W. Hasselbring. Research Perspective on Supporting Software Engineering via Physical 3D Models. In: *IEEE 3rd Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, 2015, Seiten 125–129. URL: http://eprints.uni-kiel.de/29387/. (Siehe Seite 7)

[Fittkau u. a. 2017] F. Fittkau, A. Krause und W. Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology* 87 (2017), Seiten 259 –277. URL: http://www.sciencedirect.com/science/article/pii/S0950584916301185. (Siehe Seite 6)

[Hasselbring 2016] W. Hasselbring. Microservices for Scalability: Keynote Talk Abstract. In: *International Conference on Performance Engineering (ICPE 2016)*. Herausgegeben von A. Avritzer und A. Iosup. ACM, 2016, Seiten 133–134. URL: http://eprints.uni-kiel.de/31829/. (Siehe Seite 9)

[Hasselbring und Steinacker 2017] W. Hasselbring und G. Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *Proceedings 2017 IEEE International Conference on Software Architecture Workshops*. IEEE, Apr. 2017. URL: http://eprints.uni-kiel.de/37489/. (Siehe Seite 9)

[Johnson u. a. 2004] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack u. a. The spring framework–reference documentation. *Interface* 21 (2004). (Siehe Seite 10)

[Kiczales u. a. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm und W. G. Griswold. An overview of AspectJ. In: *European Conference on Object-Oriented Programming*. Springer. 2001, Seiten 327–354. (Siehe Seite 14)

[Likert 1932] R. Likert. A technique for the measurement of attitudes. *Archives of psychology* (1932). (Siehe Seite 63)

[Matthiessen 2014] N. Matthiessen. Monitoring Remote Procedure Calls - Concepts and Evaluation. Bachelorarbeit. Kiel University, 2014. URL: http://eprints.uni-kiel.de/24149/. (Siehe Seite 38)

[Namiot und sneps sneppe 2014] D. Namiot und M. sneps sneppe. On Micro-services Architecture. 2 (Sep. 2014), Seiten 24–27. (Siehe Seiten 9 und 19)

[Påhlsson 2002] N. Påhlsson. Aspect-oriented programming. *Topic Report for Software Engineering* (2002), Seiten 11–03. (Siehe Seite 14)

[Perry und Wolf 1992] D. E. Perry und A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), Seiten 40–52. (Siehe Seite 11)

[Sigelman u. a. 2010] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan und C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technischer Bericht. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf. (Siehe Seite 90)

[Strubel 2017] H. Strubel. Monitoring Distributed Traces with Kieker. Magisterarbeit. Kiel University, 2017. (Siehe Seiten 33, 39 und 90)

[Van Hoorn u. a. 2012] A. Van Hoorn, J. Waller und W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, Seiten 247–248. (Siehe Seiten 2 und 5)

[Waller u. a. 2014] J. Waller, F. Fittkau und W. Hasselbring. Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability. In: *Symposium on Software Performance (SOSP 2014)*. 2014. (Siehe Seite 8)

[Wulf u. a. 2017] C. Wulf, W. Hasselbring und J. Ohlemacher. Parallel and Generic Pipe-and-Filter Architectures with TeeTime. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, Seiten 290–293. URL: http://eprints.uni-kiel.de/37563/. (Siehe Seite 95)

[Zirkelbach 2017] C. Zirkelbach. Juggling with Data: On the Lack of Database Monitoring in Long-Living Software Systems. In: *4th Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems (EMLS)*. Softwaretechnik-Trends 2. 2017, Seiten 62–65. URL: http://eprints.uni-kiel.de/37440/. (Siehe Seite 95)

[Zirkelbach u. a. 2015] C. Zirkelbach, W. Hasselbring und L. Carr. Combining Kieker with Gephi for Performance Analysis and Interactive Trace Visualization. In: *Symposium on Software Performance 2015: Joint Developer and Community Meeting of Descartes/Kieker/Palladio*. Band 35. 3. Softwaretechnik-Trends, 2015, Seiten 26–28. URL: http://eprints.uni-kiel.de/30101/. (Siehe Seite 59)