Enterprise Application Discovery and Monitoring Management with ExplorViz

Master’s Thesis

Alexander Krause

March 29, 2018

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Zirkelbach
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, 29. März 2018

__________________________________________
Abstract

Developers’ knowledge and comprehension about program internals does not scale with the complexity and scope of today’s software. The vast number of lines of source code and external dependencies are two factors, which exacerbate this observation. Static analysis approaches, such as utility tools of integrated development environments, may reduce the number of overall problems. However, only dynamic analysis and monitoring tools are able to show flaws during program runtime. Such tools use different visualizations to depict analyzed data and potential software issues. Unfortunately, the underlying monitoring approaches often lack a plausible usability and require an unnecessary long time to setup. This might discourage developers from using these tools. Therefore, the latter miss their major purpose, i.e., improving software.

In this thesis, we present an approach for an application discovery and monitoring management system, abbreviated to ADAMMS. Our developed design addresses critical features such as monitoring setup and application lifecycle or monitoring management. We implement the design in the form of an extension to the ExplorViz research tool, which explores software visualizations for dynamic runtime data. The resulting ExplorViz ADAMMS is able to discover running Java applications in Debian operating systems. The software facilitates the implementation of custom discovery mechanisms, based on a provided scaffold. Thus, our extensible system is capable of supporting different types of applications and operating systems.

In a subsequent evaluation, we successfully show the feasibility of our implementation. We further examine the usability by means of a pilot study, which includes a task-oriented experiment and a structural interview. The results indicate a good usability, but further experiments with more probands are necessary to obtain comparable results. Both experiments reveal small adjustments that would possibly enhance the implementation. Finally, we give an outlook and suggest topics for future work.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10 Conclusion and Future Work</strong></td>
<td>67</td>
</tr>
<tr>
<td>10.1 Conclusion</td>
<td>67</td>
</tr>
<tr>
<td>10.2 Future Work</td>
<td>68</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>71</td>
</tr>
<tr>
<td><strong>A Dockerfiles</strong></td>
<td>75</td>
</tr>
<tr>
<td>A.1 ExplorViz Discovery Agent &amp; Test Applications</td>
<td>75</td>
</tr>
<tr>
<td>A.2 ExplorViz Frontend &amp; Backend with Extensions</td>
<td>77</td>
</tr>
<tr>
<td><strong>B Recognition Strategies</strong></td>
<td>79</td>
</tr>
<tr>
<td>B.1 Tomcat Strategy</td>
<td>79</td>
</tr>
<tr>
<td>B.2 KiekerSampleApp Strategy</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Today’s software systems are getting increasingly complicated. The heterogeneity in terms of programming languages, internal complexity, or communication between applications, is ever-expanding. As a result, developers need to continuously renew their knowledge of developed program internals and deployment details. New developers are often in a similar position, when they must know application internals and dependencies, e.g., which classes depend on a certain single class. Comprehending large software landscapes and the included applications might be challenged by maintaining a well-documented overview. However, software system complexity and distributed developer teams require supporting tools. Research shows that software visualizations (SV) facilitate this task [Cornelissen et al. 2009; Bassil and Keller 2001].

[Bassil and Keller 2001] present a questionnaire survey regarding the usability of SV tools in the domain of software comprehension and analysis. Results show that study participants classify reliability of a tool as the most important property. The ease of using the tool, quality of the user interface (UI), and the ease of visualizing software are the subsequent most desired characteristics. These observations coincide with the results of a survey regarding the most important criteria from the viewpoint of web developers [Offutt 2002]. However, Bassil and Keller identify a deviation between the participants’ statements regarding user interfaces and ease of use and their respective realization in the examined SV tools. Nevertheless, the benefits of software visualizations remain.

The ExplorViz research tool provides a live trace visualization of large software landscapes and their included applications [Fittkau et al. 2017]. For that reason, the software utilizes the Kieker framework [van Hoorn et al. 2012] for instrumentation and monitoring aspects. Unfortunately, users must manually follow a specific procedure to start monitoring a single application. This might work with a small number of locally running applications, but is not suitable for enterprise software landscapes with a vast number of server nodes.

In this thesis, we therefore design and implement a new system for ExplorViz that essentially simplifies the application monitoring aspects. This application discovery and monitoring management system (ADAMMS) is supposed to automatically discover running Java applications. The enhancement to ExplorViz allows users to centrally handle the
1. Introduction

monitoring management of found applications, which might run on external servers. The underlying implementation does not only serve our needs, but also provides extensible mechanisms. Thus, developers are able to introduce their custom adaptations and enhance the overall applicability.

1.2 Goals

This section introduces the goals of the thesis. With respect to feasibility and the extend of this work, we will only consider supporting Java applications on a Debian-based operating system (OS). As a result, this will constrain the applicability of the system at the time of writing.

1.2.1 G1: Evaluation of Application Discovery Tools

There are different tools that can be used to discover running applications in operating systems. The first goal is to identify common tools, which provide this feature for Java applications on the Debian OS. Furthermore, we are going to evaluate each tool by means of suitability and applicability in respect to our purposes. This will allow us to identify and subsequently use a qualified tool in our design and implementation.

1.2.2 G2: Development & Integration of an Application Discovery and Monitoring Management System in ExplorViz

The second goal is the development of the ExplorViz ADAMMS. This encompasses the design, software architecture, and implementation. The discovery functionality should find and recognize running Java applications of the hosting OS. The monitoring management functionality is supposed to provide an easy to use application monitoring through our system. Developers will be able to implement their custom adaptations regarding application discovery. The integration into ExplorViz is approached by using the ExplorViz extension mechanism.

1.2.3 G3: Evaluation in an Exemplary Enterprise Situation

An evaluation of the developed system represents the last goal of this thesis. We examine the feasibility and usability of the ExplorViz ADAMMS. An experiment will be used to prove the correct system behavior and functionality in an exemplary software landscape environment. In an additional pilot study, we examine the usability of our implementation.
1.3 Document Structure

The remainder of this thesis is structured as follows. Chapter 2 introduces essential foundations and technologies. In Chapter 3, we introduce and evaluate tools for application discovery. Our design for an ADAMMS will be introduced in Chapter 4. We explain the extension mechanism of ExplorViz in detail in Chapter 5. Chapter 6 covers the software architecture that we used for our system. Then, Chapter 7 explains the implementation. Chapter 8 contains the evaluation of the implementation. In chapter Chapter 9, we discuss related work. Chapter 10 concludes this thesis and gives an outlook regarding future work.
This chapter introduces essential information for a better comprehension of the thesis. We start by explaining communication technologies, proceed with employed web frameworks, and finally introduce the monitoring framework Kieker and the visualization tool Explorviz.

2.1 Representational State Transfer

Representational state transfer (REST) is an architectural style for network-based communication. While initially proposed by Fielding in the year 2000 [Fielding 2000], it is now one of the main used architectures for developing web services [Upadhyaya et al. 2011]. The REST style is essentially a set of architectural constraints for data exchange and communication participants. For example, requests from clients to servers must contain all information that is necessary for processing (self-descriptive messages). A subset of these constraints form the uniform interface, i.e., the fundamental feature that distinguishes REST from other network-based styles. If developers correctly apply all constraints, REST facilitates low latency and scalability of web services [Fielding and Taylor 2002]. The communication in REST is based on exchanging representations of resources. Any real or virtual information that can be named, e.g., a person, a text document, or a service, can be a resource [Fielding and Taylor 2002]. Representations use different format types to depict a resource, e.g., Extensible Markup Language (XML), JavaScript Object Notation (JSON), Hypertext Markup Language (HTML), or Portable Document Format (PDF). A resource can take up multiple representations, but has at least one [Tilkov 2015]. Furthermore, they are uniquely identifiable through a resource identifier, e.g., a Uniform Resource Identifier (URI). Today, most REST (or RESTful) web services use the Hypertext Transfer Protocol version 1.1 (HTTP/1.1) for data exchange. This is no constraint defined by the architectural style. However, since HTTP dominates web communication, it has been naturalized and accepted by developers across the years. As a result, we too only consider HTTP for REST in this thesis. ExplorViz is implemented as RESTful web service. Therefore, extensions, such as our ExplorViz ADAMMS, are advised to also apply the same architectural style.
2. Foundations and Technologies

2.2 JSON API

Network-based communication requires data-interchange formats. Web services often use JSON to exchange data with their communication participants. Unfortunately, JSON does not come with pre-defined rules on how request and response messages should be structured. JSON objects, i.e., essentially JavaScript (JS) objects, can therefore appear in many different variants, e.g., nested or flat. This affects maintainability, since developers must incorporate and remember the structure of objects which also might structurally change in later development.

```json
{
  "data": [{
    "type": "shoes",
    "id": "1",
    "attributes": {
      "name": "JSON-Speculizers",
      "color": "Hazelnut-Brown",
      "launch-date": "658914060000"
    },
    "relationships": {
      "brand": {
        "data": {"id": "2", "type": "brands"}
      }
    }
  },
  "included": [
    {
      "type": "brands",
      "id": "2",
      "attributes": {
        "name": "JSON-shoes-4-u"
      }
    }
  ]
}
```

Listing 2.1. Exemplary JSON API response document as result for submitting the HTTP GET request /shoes/?include=brand

The JSON API (Application Programming Interface) specification\(^1\) aims to solve the described problems. It determines how JSON objects for resources should be structured by defining rules for JSON-based messages.\(^2\) JSON API compliant objects (JSON API

---

\(^1\)http://jsonapi.org (last retrieval February 20, 2018)

\(^2\)http://jsonapi.org/format/#document-structure (last retrieval February 20, 2018)
documents) are identified by the new media type \texttt{application/vnd.api+json}. We can see an exemplary JSON API document in Listing 2.1. It shows a response message for a request to obtain all shoes and their brands (with respect to simplicity we only included one type of shoe and one brand). The data member holds primary requested information, i.e. a JSON API representations of resources. Resource objects must contain at least the top-level member \texttt{id} and \texttt{type}. The relationship object states associations to other resources by including their type and \texttt{id}. Finally, the included member can optionally hold the representations of the relationships. At the time of writing, ExplorViz only communicates via JSON API documents. If an extension includes new resources, it must use the same media type for their representations to facilitate maintainability and extensibility.

2.3 JAX-RS and Jersey

The Java Enterprise Edition (Java EE) contains the Java API specification for RESTful web services (JAX-RS).\footnote{https://javaee.github.io/tutorial/jaxrs.html (last retrieval February 27, 2018)} JAX-RS includes interfaces for both RESTful servers and clients, thus defines a way how RESTful Java web services should be implemented. Servers define their resources through root resource classes (RRC). JAX-RS annotations determine their URI, applicable operations (e.g. HTTP GET), and consuming or producing media types, e.g., \texttt{application/vnd.api+json}. Unlike RESTful servers, JAX-RS clients can be defined with only a few lines of code. They are able to submit HTTP requests and process the response.

As already stated, JAX-RS is only an API specification. The Jersey framework (hereinafter referred to as Jersey)\footnote{https://jersey.github.io (last retrieval February 20, 2018)} implements the interfaces provided by JAX-RS. Jersey belongs to GlassFish, an open-source Java EE reference implementation.\footnote{https://javaee.github.io/glassfish (last retrieval February 18, 2018)} Because of the Java EE origin, Java Servlets build the foundation of the server-side communication technology.

ExplorViz employs Jersey to provide a RESTful web server. Extensions must therefore work with the same framework. Additionally, we utilize the JAX-RS client interface to submit HTTP requests from web servers.

2.4 Ember.js

Ember.js\footnote{https://www.emberjs.com (last retrieval March 01, 2018)} (hereinafter referred to as Ember) is an open-source client-side web framework based on the JS runtime Node.js.\footnote{https://nodejs.org/en (last retrieval March 01, 2018)} It enables developers to write component-based JS applications for web browsers. Dependency management is typically realized by using the node package manager (npm).\footnote{https://www.npmjs.com (last retrieval February 20, 2018)}
Ember’s core concepts are encapsulated as single files in a pre-defined project directory tree. The most important concepts are shown in Figure 2.1. We assume that a user navigates to http://localhost:4200/rentals, which is an URI provided by an Ember application. The Ember router will then lookup the mapping for the URI and call the respective handler, i.e., a certain route object (or simply route). For that reason, it holds a mapping of application URIs and related routes. When initialized, routes generally load their route model, i.e., the data which they display. In terms of Figure 2.1, the route rental submits a HTTP GET request to obtain all resources of type rental from a web server (network aspects are explained shortly). The returned data is then rendered via a Handlebars template. These files contain the HTML layout for a related parent file, e.g., a route. Additionally, they initialize components, i.e., reusable and encapsulated pieces of a website. Components can constitute everything from a button to a single table entry, since their scope is not pre-
defined. Therefore, the granularity of components is the developers choice. In Figure 2.1, the line \(\{\text{rental-tile rental=}\text{rental-unit}\}\) creates one rental-tile component. A single rental-unit, i.e., an entry of the previously requested resource array from the web server, is passed as parameter rental. Finally, the component’s template renders the passed rental-unit.

The introduced core concepts require a starting point, since Ember determines a project directory layout by default. As result, Ember provides a command line interface (CLI) called Ember CLI to generate scaffolds for single concepts. For example, the CLI command `ember generate route shoes` will create a route file, a related test file and add the URL mapping to the router. Files can also be removed, e.g., `ember destroy route shoes`.

Ember uses an underlying persistence library, called Ember Data\(^{11}\) that provides all mechanics for network exchange, resource management, and storage. Resources are defined as model types, i.e., single files that contain attributes and relationships to other models. Records are runtime instances of a model type that contain actual data. Figure 2.2 shows an abstract overview of Ember Data mechanics to obtain a record for a model type. The procedure starts by calling the find function of the store object for a model type. The store contains all records and is responsible for their management, e.g., record creation. It has different functions for obtaining known, unknown, single, and multiple records. We assume we want to get the record with id 5 of model type shoe. We call `store.findRecord('shoes', 5)`. The store does not find a local record for id 5 and model type shoe. Therefore, it submits a request to the shoe adapter. An adapter builds URIs for HTTP methods and submits the request.

![Figure 2.2. Ember Data overview for finding an unknown record\(^{12}\)](https://guides.emberjs.com/v3.0.0/tutorial/ember-data(last retrieval February 20, 2018))

In this case, the result is a HTTP GET request to a web server with a resource identified by the URI .../shoes/5. Since adapters might submit an asynchronous request, every related mechanic returns a JS promise object. Finally, after retrieving the fulfilled promise object from the adapter, the store saves the new record.

One missing step from the introduced procedure is the serializing of records and deserializing of response messages, i.e., resource representations. Ember Data serializers

\(^{11}\)https://guides.emberjs.com/v3.0.0/models/#toc_architecture-overview(last retrieval 20 February, 2018)

\(^{12}\)https://guides.emberjs.com/v3.0.0/tutorial/ember-data(last retrieval February 20, 2018)
2. Foundations and Technologies

are responsible for this conversion. One specific type is the JSONAPI!Serializer. It converts records to JSON API documents and vice versa.

ExplorViz uses Ember for GUI and visualization aspects. We will extend the ExplorViz Ember project and introduce our new GUI for application discovery and monitoring management.

2.5 Kieker

Kieker is a framework for dynamic runtime analysis [van Hoorn et al. 2012]. It enables application and system performance monitoring and architecture discovery. Application monitoring requires a prior instrumentation of code. Source code can be instrumented by simply inserting code fragments, e.g., before and after the monitoring target. Byte code however requires a different approach. Kieker uses so-called probes, i.e., code fragments that are responsible for data collection. These are weaved into byte code by using AspectJ [AspectJ]. As a result, compiled applications can also be monitored. The framework is highly extensible in terms of supported programming languages and analysis features. That is why ExplorViz uses Kieker to monitor applications. Instead of using the built-in analysis, ExplorViz instructs Kieker to send monitoring data to its own analysis. This data is encapsulated in so-called records. Kieker provides two files to modify monitoring options. The aop.xml contains instructions for AspectJ, e.g., which package should be monitored. The kieker.monitoring.properties file covers the Kieker related configurations, e.g., name of the to-be monitored application. These options are subject of this thesis, since we will implement an interface to easily modify monitoring options.

2.6 ExplorViz

The ExplorViz project researches software visualizations for program comprehension [Fittkau et al. 2017] and performance analysis [Zirkelbach et al. 2015]. This includes cutting-edge input and output devices [Fittkau et al. 2015a; Fittkau et al. 2015b] for intuitive interaction. Instead of using static analysis, ExplorViz employs a dynamic application monitoring with Kieker to analyze runtime data. This gathered data enables the main functionality of ExplorViz, i.e., the live trace visualization of software landscapes and their included applications.

Figure 2.3 shows the first visualization of ExplorViz. The two dimensional landscape visualization (or perspective) renders gathered and analyzed runtime data. Users are able to inspect properties of server nodes, applications, and their network communication. Every application (purple boxes) is therefore being monitored by Kieker Section 2.5 and processed by ExplorViz’s analysis component. Semantics of pictured entities are explained in [Fittkau et al. 2017].
Figure 2.3. ExplorViz’s landscape visualization (taken from [Fittkau 2015])

Figure 2.4 shows the second visualization of ExplorViz, which is accessible via clicking a single application in the landscape visualization. The 3D application perspective is a detailed live rendering of an application’s internals, i.e., packages, classes, and their communication [Fittkau et al. 2017]. It is based on the 3D city metaphor [Wettel and Lanza 2007]. Since both visualizations depict analyzed monitoring data, users must first configure Kieker for every application they want to inspect. In terms of this thesis, we will introduce new mechanics in ExplorViz to discover running applications on different server nodes and start monitorings through the GUI.
2. Foundations and Technologies

Figure 2.4. ExplorViz’s application visualization (taken from [Fittkau 2015])
Chapter 3

Evaluation of Application Discovery Tools

In this chapter, we introduce and evaluate tools for application discovery. Due to our goals, we only consider the discovery of Java applications on a Debian-based OS. Nevertheless, some of the subsequent observations also apply to other Linux-based or common operating systems as well as to diverse applications.

3.1 General Observations regarding Application Discovery

In operating systems, a running application is represented by a running instance of a process. These instances are started by invoking an executable command with potential arguments and flags (hereinafter referred to as execution command). In terms of Java applications, processes are started by invoking the java launcher with arguments such as the main class on the CLI. This results in a Java Virtual Machine (JVM) process that handles the execution of the passed program.

For our purposes, we are going to examine tools that provide us with the execution command as well as the process ID (PID). The first parameter will enable us to both identify an application and start a process, whereas the latter is required for the monitoring management. The second parameter is a unique random numeric value that identifies a running instance of a process during runtime of the OS.

3.2 Tools to obtain Java Process Details

The JDK 8 Process API is limited in terms of controlling and managing OS processes. JDK 9 introduces an extended API with new required functionalities [Oracle 2017]. However, we don’t want to restrict the applicability and therefore use the JDK 8. Because of the JDK 8 drawbacks, developers are forced to utilize additional tools to read OS processes and their details. This section introduces and evaluates tools that allow us to obtain the required information, i.e., execution command and PID, for our ExplorViz ADAMMS.
3. Evaluation of Application Discovery Tools

3.2.1 jProcesses

jProcesses is an open-source Java library that enables developers to read details of running processes.\(^1\) The tool does not only capture Java-based processes, but also any different kind.

1 List<ProcessInfo> processesList = JProcesses.getProcessList();

Listing 3.1. Obtain all running processes in jProcesses

It contains an easy to use method to obtain all running processes at once (see Listing 3.1). The gathered data is depicted with a model class called ProcessInfo. An instance of this class contains the details of a single process, e.g., PID and execution command. Since process design differentiates between operating systems, the tool also provides a cross-platform compatibility. Variant techniques are used to read process details on different operating systems.

The relevant Linux approach of jProcesses employs the Java ProcessBuilder\(^2\) to invoke Linux command-line tools. Therefore, the actual reading of process details is delegated to these tools. jProcesses simply reads their raw output and puts the process details in instances of its data model. Unfortunately, ExplorViz uses a Java library to convert Java objects to JSON API documents. This requires specific annotations in the related classes of every to-be converted object. As a consequence we would need to fork the jProcesses project and apply the required changes to its source code. However, we could use the same method to obtain process details and depict them with our self-defined annotated model classes. This would minimize the dependency count and coupling of the final project. These observations therefore argue against the utilization of jProcesses.

3.2.2 jps

The Java Development Kit (JDK) comes with a tool for analyzing the JVM. The JVM Process Status Tool (jps) [Oracle 2018] is a command-line tool that displays details of running JVM processes for which the user has the access permission.

1 # jps -lvm
2 29267 sun.tools.jps.Jps -lvm -Dapplication.home=/usr/lib/jvm/jdk1.8.0 -Xms8m
3 29223 sampleApplication.jar

Listing 3.2. Exemplary execution and output of jps

Listing 3.2 shows an exemplary execution of the jps. The flags in line 1 instruct the tool to show all details of processes that the it can obtain. The following output (see lines 3 and 4) shows a table of PIDs and passed parameters to the JVM or the main class.

\(^1\)https://github.com/profesorfalken/jProcesses\(\text{last retrieval March 12, 2018}\)
\(^2\)https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html\(\text{last retrieval March 12, 2018}\)
3.2. Tools to obtain Java Process Details

jps is an experimental Java tool. The related documentation states that these kind of tools are unsupported and might not be available in future JDK versions. Furthermore, it is recommended that the output of jps should not be parsed, because the format might change [Oracle 2018]. However, this would be necessary, since we might use a ProcessBuilder to invoke jps and subsequently read the details of processes, i.e., execution command and PID. jps is additionally constrained in terms of availability, since it is only contained in JDKs. Therefore, operating systems containing only the Java Runtime Environment (JRE) cannot use this tool. This would restrict the applicability of the final ExplorViz ADAMMS. Therefore, we decide against using jps.

3.2.3 ps and grep

ps is a Linux command-line tool that displays information of running processes in operating systems. Without any flag, it shows details of all processes that share the same executing user and the same terminal (tty). The extended command `ps -e -U $(whoami)` displays every process of the current user. It utilizes the result of the command-line tool whoami, which returns the name of the executing user. However, execution commands of JVM processes are simplified. This can be resolved by using `ps -e -U $(whoami) -o pid,args`. Now, ps displays not only the full execution command, but also trims the output by only showing this parameter and the PID.

The output of the introduced ps command contains details of every running process of the current user. Since we are only interested in JVM processes, we need to filter this output. Therefore, we pipe it to the Linux command-line tool grep. With the additional string parameter java we are able to filter any process that does not contain this string in its execution command.

```
# ps -e -U $(whoami) -o pid,args | grep java
```

```
13 java -cp . -jar sampleApplication.jar
```

```
```

Listing 3.3. Exemplary output for the CLI command `ps -e -U $(whoami) -o pid,args | grep java` (adjusted in terms of human readability)

---

3https://docs.oracle.com/javase/8/docs/technotes/tools/#monitor (last retrieval March 12, 2018)
3. Evaluation of Application Discovery Tools

Listing 3.3 shows the final `ps` and `grep` command in line 1. The output, i.e., PID and execution command of two running JVM processes, can be seen in line 3 and 5. As with `jps`, a Java application can use a `ProcessBuilder` to run this command. The output can then be read and processed by means of string operations.

Listing 3.3 also presents that `ps` highlights the used Java launcher for a single process. Therefore, we can detect if different Java versions have been used. This benefits the functionality, since we do not need to require a set `JAVA_HOME` environment variable. The simplicity and availability in Linux distributions are the main factors that speak for these tools. Therefore, we are going to use the `ps` and `grep` commands for our ExplorViz ADAMMS. For convenience, the application of these tools is hereinafter also referred to as the PG-method.

3.2.4 Graphical Tools for JVM Analysis

The previously introduced tools can be invoked and processed from source code. That is mandatory for the ExplorViz ADAMMS. However, there are also graphical tools that provide similar features. For the sake of completeness, we introduce two of these tools, namely `jconsole` and `jvisualvm`, in this section.

![Figure 3.1. jvisualvm GUI showing details of a running Tomcat web server](https://docs.oracle.com/javase/8/docs/technotes/tools/#jconsole (last retrieval March 12, 2018))
3.2. Tools to obtain Java Process Details

Both tools are primarily used to monitor and analyze JVM performance and resources. However, certain details for our purposes can be seen as well. ❶ shows the PID of the selected JVM process. ❷ presents the main class of the Java application, whereas ❸ displays the class path of the program.
Design of an Application Discovery and Monitoring Management System

The PG-method was selected as the basis for application discovery. However, further steps are necessary to provide the overall functionality. This chapter presents our design for an application discovery and monitoring management system.

4.1 Application Discovery Design

In this section, we introduce the application discovery design. It is based on the previously introduces PG-method (see Section 3.2.3).

4.1.1 Recognition Mechanism

Any previously mentioned tool for application discovery is not sufficient on its own to fulfill our introduced goal (see Section 1.2.2). Since the discovery functionality is also supposed to recognize applications, a further feature is required.

```
<<Interface>>
RecognitionStrategy
+isDesiredApplication() : boolean
+applyStrategy() : void
```

Figure 4.1. Interface for rule-based recognition strategies

Figure 4.1 depicts a UML class diagram showing the RecognitionStrategy interface. The contained method isDesiredApplication is supposed to recognize an application process. If an application was recognized, the applyStrategy method is called. Here, an application might be labeled with a pre-defined name such that ExplorViz users are easily able to identify it. This design enables developers to implement their custom recognition strategies.

In this thesis, we are going to examine the execution command to recognize applications. These commands contain information that allow us to define strategies based on their
4. Design of an Application Discovery and Monitoring Management System

contained substrings. Listing 3.3 shows an exemplary output of the PG-method. In line 3 we can detect the execution command of the process with PID 13. We perceive a relative path for the jar file. This is due to the fact that `ps` shows only the command the way it was launched. This means that the process was initially started from a directory that also contained the jar file on the same level. These relative paths would affect our recognition strategies, since these are based on substring patterns in the execution command. We can resolve this drawback by using another Linux command-line tool, namely `pwdx`.

```
1 java /kiekerSampleApp/sampleApplication.jar
```

Listing 4.1. Enriched execution command based on `pwdx` output

The tool `pwdx` returns the path of the working directory, i.e., the directory from where the process was started. As we can see in Listing 4.1, it is possible to enrich the execution command with the output string from `pwdx` (blue segment). The command of the process is now an absolute path. Therefore, the recognition strategies have more information to examine.

4.1.2 Design

![Application Discovery (Single Iteration)](image)

Figure 4.2. Single iteration of the application discovery design

Figure 4.2 displays a UML Activity Diagram showing the design of the process flow for the application discovery functionality. The PG-method is summarized in ➊. The Operating System Java Process List (OSJPL) represents the current details of running Java processes. This list includes the pairs of PIDs and execution commands. It is enriched by means of finding the absolute paths for the execution commands with `pwdx` (see ➋). The recognition strategies are subsequently applied to every process of the list. Therefore, properties such as the name of the application can be set (see ➌). Finally, the application discovery functionality merges the new and old process lists. As a consequence, we are able to detect unforeseen process loss.
4.2 Monitoring Management Design

This section presents our design for an monitoring management feature. In general, the monitoring in ExplorViz is outsourced to Kieker. Such tools for Java application monitoring are commonly based on the Java Instrumentation API.\(^1\) An application that follows the API guidelines can be used as javaagent, therefore enables the instrumentation and monitoring of JVM processes. To instrument and monitor applications with Kieker, users must essentially start their application on the CLI with the javaagent option. The latter points to the kieker-monitoring.jar file, which weaves in the instrumentation code and handles monitoring (see Section 2.5). Therefore, processes must be restarted to enable monitoring with Kieker. In our terminology restarting a process is essentially stopping an old and starting a new instance with the same execution command.

As a side note we want to highlight that the Java Instrumentation API does provide a mechanism to start a javaagent after the desired JVM process has been started. This was previously implemented in Kieker [Flaig 2014]. However, it is not included in the main Kieker repository at the moment.

We can start a new instance of a process by passing the execution command to the Java ProcessBuilder. Since we enrich the commands with absolute paths using pwdx, starting a process instance from any directory is possible.

\[\text{Operation}\]

\[\text{Monitor process X}\]

\[\text{ADAMMS}\]

\[\text{Kill process}\]

\[\text{Update model}\]

\[\text{Insert identifier}\]

\[\text{Internal model}\]

\[\text{Find process}\]

\[\text{Get new OSJPL}\]

\[\text{Run exec cmd}\]

\[\text{Insert Monitoring}\]

\[\text{Operations on data}\]

\[\text{Next action of the ADAMMS}\]

**Figure 4.3.** Design for monitoring management of the ADAMMS

\(^1\)https://docs.oracle.com/javase/8/docs/technotes/guides/instrumentation/index.html (last retrieval March 13, 2018)
4. Design of an Application Discovery and Monitoring Management System

Figure 4.3 shows the design for the monitoring management of a process in our ADAMMS. Triggering the related operation starts the procedure. The system proceeds by killing the running process in the OS. This can be realized by using an OS-dependent command-line tool for killing processes, e.g., `kill -9 <PID>` in Linux. Afterwards, it inserts an identifier to the execution command. This is necessary for mapping the new process instance to the internal model. Furthermore, if we do not find a process with the same identifier in new process lists, we can assume a failure during process startup. The monitoring options such as the javaagent are subsequently added to the execution command.

```
java
-javaagent:/kieker.jar -Dexplorviz.agent.model.id=2
-cp . -jar /kiekerSampleApp/sampleApplication.jar
```

Listing 4.2. Exemplary final execution command for a process (adjusted in terms of human readability)

Listing 4.2 shows an exemplary execution command that was processed by the ADAMMS. The blue segment presents the pdwx enrichment of the encompassing application discovery feature. The red segment contains the identifier that is used to retrieve the new process instance. Finally, the violet segment shows the javaagent with the related Kieker jar file.

The designed ADAMMS in Figure 4.3 continues by executing the new execution command. It subsequently obtains the new OSJPL and retrieves the marked process from this list. At the end, the internal model is updated with the new PID.

4.3 Design for a Multi-server Operation

The introduced features are supposed to work on a single local machine. However, our design is also considering a multi-server operation. This allows our system to discover and manage monitoring aspects of applications running on distributed server nodes. Section 4.3 visualizes the underlying star topology for multi-server operation of our design. As we can see, all discovery and monitoring management nodes communicate with a single centralized server called backend. In summary, an implementation for our design must realize this aspect.

![Figure 4.4. Star topology used in our design](image-url)
4.4 Design for a Visualization

The previously mentioned features are supposed to be accessible by means of a graphical user interface. This section introduces our related design.

[Holzinger 2005] states the five essential usability characteristics that software projects should realize for general user interfaces.

1. **Learnability** Users can rapidly start to work
2. **Efficiency** Trained users are able to gain a high level of productivity
3. **Memorability** Casual users do not need to relearn
4. **Low error rate** Provide functionalities such that users make fewer errors, errors must be rectifiable
5. **Satisfaction** The system is pleasant to use

Figure 4.5 shows our design that is supposed to realize the introduced characteristics.
4. Design of an Application Discovery and Monitoring Management System

A web site’s usability and usefulness is primarily influenced by the ease with which information can be obtained [Lee 2000]. Therefore, we decide to employ a clearly understandable graph-based visualization. It is supposed to show the multi-server operation design. We can instantly see the ADAMMS and application topology. Furthermore, colors are used to depict which ADAMMS is available (blue), offline (gray), or handled monitoring for a found application (green). The graph layout is based on a tree design. For our purposes, this common type of hierarchy is supposed to facilitate learnability and memorability, since semantics of the visualization can rapidly perceived. We furthermore enriched our design with the results of examined user preferences for graph layouts by [Purchase et al. 2001]. Therefore, the layout does not contain bends and the text direction is horizontal. These modifications might facilitate the satisfaction of using the system.

Monitoring configurations of a single application are supposed to be visualized in a separated dialog. A click on an application might show the designated window. Here, the monitoring can be adjusted, started, or stopped. Since this essentially results in a restart of a process, errors during this procedure must be highlighted to the user. Furthermore, the modifications must also be visible in Figure 4.5.
Chapter 5

Toward an ExplorViz Extension

In the previous chapters, we talked about integrating an ADAMMS in ExplorViz without describing architectural details. Therefore, we now introduce the ExplorViz extension mechanism, which enables us to develop self-contained extensions (hereinafter also referred to as expansion or addon) [Zirkelbach et al. 2018].

5.1 Overview

ExplorViz is essentially divided into two components. The first component is called the ExplorViz core. It comprises already introduced capabilities of the tool (see Section 2.6).

Figure 5.1. Unified Modeling Language (UML) component diagram of ExplorViz’s extension mechanism
5. Toward an ExplorViz Extension

As we can see in Figure 5.1, it is composed of two subcomponents. The backend (see 1) handles the analysis of monitoring data, user authentication, and data persistence. It is based on Jersey (see Section 2.3) and therefore includes a web service that provides the analyzed data as resources to clients (see 2). The ExplorViz frontend (see 3) is such a client. This Ember-based (see Section 2.4) subcomponent visualizes the obtained resources for users, e.g., landscape visualization, and provides the general GUI.

The second component of ExplorViz encompasses every extension. We can see an exemplary extension in Figure 5.1. A single extension with backend and frontend functionality always includes two subcomponents. Backend extensions (see 4) are able to obtain objects and use methods of the core by using the provided API (see 5). They can read, but not write core objects, i.e., underlying Java objects for resources. Additionally, backend extensions can define and instantiate a separate data model. Resources are then provided to clients as web service (see 6). Therefore, developers can extend the ExplorViz frontend to obtain those resources. Frontend-related processing of obtained resources, e.g., visualization of a new data model, must now be defined by extension developers. For that reason the ExplorViz frontend core provides a collection, including layout algorithms and generic services to obtain data from the backend (see 7).

5.2 Integration into employed Frameworks

We now introduce the integration of both ExplorViz backend and frontend extensions into their core counterparts. These mechanisms are entirely based on the employed frameworks.

**Jersey Framework** Jersey applications are able to scan folders for JAX-RS annotated classes on startup. ExplorViz exploits this feature to insert new RRCs (see Section 2.3) from extensions to the comprising web service. On the one hand developers can add their source code to a specific package of the ExplorViz Java project. The now extended project can be build into a war file and deployed in a web server, e.g., Apache Tomcat. Alternatively, developers can take the compiled Java class files of their extension and insert them into the same folder of an already deployed ExplorViz. On the other hand the ExplorViz team distributes a sample extension1 that contains the ExplorViz core as dependency. This approach differs from the first one, because the extension and the core are both single JAX-RS applications. Developers don’t operate in a package of the core, but define their own application. However, due to the dependency the ExplorViz backend API (see 4) can be used. Again, this project can be build into a deployable war file containing both backend and extension. It is also possible to build a jar file that only contains the extension code and related dependencies. Therefore, this jar file can be inserted in an already deployed ExplorViz backend.

---

1 https://github.com/ExplorViz/explorviz-backend-extension-dummy (last retrieval March 16, 2018)
5.2. Integration into employed Frameworks

**Ember Framework**  The Ember framework already ships with an addon mechanism. Developers can generate addon scaffolds by using Ember CLI (see Section 2.4). More precisely, the command `ember addon explorviz-extension` will create a folder `explorviz-extension` that includes the source code and dependencies of the addon. Ember uses a convention over configuration approach. Thus, Ember addons look alike most of the time. In terms of functionality, Ember addons usually provide a single object that encapsulates the addon’s stated feature. This object can then be used in the main source code, e.g., a component is included in a main application template file (see Section 2.4). The feature requires the inclusion of an addon in the main application at first. Developers therefore insert the desired dependency in the `package.json` file, i.e., a file used by `npm` for dependency management. After downloading the dependency, the main application and extension namespace will be merged on startup. As a result the application will now know about the extension during runtime.

ExplorViz uses the introduced Ember addon mechanism. Related extensions are restricted on the convention that every addon must use an Ember route, i.e., by ExplorViz convention the entry point to the extension, to encompass their functionalities. If developers now include the extension in the core’s `package.json` file, the route is accessible in ExplorViz by inserting its URI in a web browser. In terms of usability, ExplorViz furthermore provides functionalities to include an extension in the provided GUI. For that reason the frontend core offers two Ember services (see §). Services are singleton objects that live for the duration of the Ember application. Their use cases essentially refer to globally shared data such as session management or authentication. However, developers are not restricted to this definition and can therefore include their custom data and functions in Ember services.

The first service is used to add clickable extension entry points to the navigation bar of the ExplorViz GUI. Extensions can include this service and add the name of their entry route. Every added route is automatically included in the core’s navigation bar on application startup. As a result users can click on a visible button instead of manually inserting a URI to invoke an extension. The second service is called the configuration service. It is used to provide developer-defined options to users. For example a developer wants to include a color-blind mode in his extension. By ExplorViz convention, extensions must provide one single Ember component that handles user interactions with with every modifiable option (hereinafter referred to as configuration component). In terms of the introduced color-blind mode, the configuration component must therefore contain for example a button to toggle the mode and trigger necessary functions to process this interaction. Extensions can display their options by adding the unique name of their configuration component to the includable core configuration service. The configuration dialog of the ExplorViz frontend core finally embeds every added configuration component, therefore displays extension options to users in a single location.

---

---

2 https://guides.emberjs.com/v3.0.0/applications/services(last retrieval March 9, 2018)
In this chapter, we present the software architecture for the following implementation of an ADAMMS in ExplorViz (hereinafter referred to as ExplorViz ADAMMS).

### 6.1 Architectural Drivers

This section contains the architectural drivers that are used to define the final component design of the system. We explain the requirements on the software architecture and its constraints.

#### 6.1.1 Requirements

**Deployment** The purpose of the overall system is application discovery and monitoring management. Based on our design in Section 4.3, we expect from our architecture to perform these two features on multiple servers. Therefore, we should be able to easily monitor applications on external servers.

**Extensibility** In this thesis, we only consider Java-based applications. However, there is a vast amount of different programming languages and resulting applications. ExplorViz users might want to analyze the runtime data of these non-Java processes. Since the monitoring itself is outsourced to Kieker, users depend on its capabilities. Nonetheless, we expect that the architecture provides the ability to include new discovery and management approaches.

The application discovery feature contains an recognition mechanism (see Section 4.1.1). ExplorViz users might want to recognize applications with a new strategy. Therefore, the resulting architecture must facilitate developers to implement new recognition strategies.

**Loose coupling** ExplorViz’s core components communicate through endpoints provided by the web service, which is contained in the ExplorViz backend. Components do not know about each other’s internals and only exchange representations of resources. This enables a loose coupling of the components [Zirkelbach et al. 2018]. As a result, both the current frontend and backend are easily exchangeable. We expect from our architecture to incorporate this property.
6. Architecture

**Fault Tolerance** The design for our monitoring management (see Section 4.2) reveals that we must restart, i.e., kill and start, process instances. Executing these operations might result in diverse errors, e.g., process start fails due to an invalid execution command. We require from our architecture to provide an adequate fault tolerance feature, so that issues can be rectified after they occurred.

6.1.2 Constraints

**Forwarding instead of Cache Coherence** ExplorViz’s data flow can be reduced on exchanging representations of resources (see Section 2.1) between its two core components, i.e., frontend and backend. Extensions of ExplorViz are generally instructed to realize the same behavior. Based on Section 6.1.1, we can derive that the architecture must encompass a third component, which essentially contains the application discovery and monitoring management features. The frontend extension is only allowed to know about the related backend extension. Therefore, the latter has the responsibility to communicate with the third component.

![Communication flow between Source, Pcp. 1, and Pcp. 2](image)

**Figure 6.1.** Communication flow between Source, Pcp. 1, and Pcp. 2 through Source

Figure 6.1 shows an abstract communication data flow between three communication participants. Here, the source (see Source) is the object that gathers data. The first participant (see Pcp. 1) knows about the source. It provides an interface such that the second participant (see Pcp. 2) can anonymously obtain the data from the source. There are two common ways to realize data exchange.

The first approach involves an actively communicating source. It can send data to the first participant if necessary, e.g., data changed. The latter can cache the data and provide it to the remaining participant. Source can therefore obtain the data and possibly change it. We now assume that data modifications must be synchronized with the source. The data is anonymously send through the first participant to the source. Source processes the request and returns the updated instance to Source, since it must keep the cache coherent.

In the other approach, the source does not act on an internal event, but is triggered from the outside (passive method). Here, the second participant triggers the start of the communication. He anonymously requests data from the unknown source through the first participant. Source forwards the request to the source and returns the response. We can see that this method requires less logic, since we don’t cache any data, but simply forward requests. Therefore, the communication itself is free of cycles. If we however use it synchronously
with hundreds of sources in a single-threaded endless loop, we will get a bottleneck with high probability. Nevertheless, high performance and throughput are no mandatory requirements of our architecture. Therefore, our architecture uses the passive method to keep networking simple.

**No Data Persistence**  Data Persistence provides failure safety, since data can be retrieved, if the system fails. This is mandatory for applications that work with self-contained data. In our case, important data comes from the underlying OS. Therefore, our architecture does not persist process data, because we can easily re-obtain it.

**Limitations regarding Operating Systems**  We already stated that the PG-method (see Section 3.2.3) depends Linux operating systems. Therefore, we will restrict the applicability of the resulting system. To compensate this, our software architecture must facilitate developers to implement new mechanisms, which enable the execution on different operating systems.

### 6.2 Component Design

Based on the requirements and constraints, we define our component design.

![Diagram](image)

**Figure 6.2.** UML component diagram showing the internals and interfaces (gray elements are subjects of this thesis)

Figure 6.2 displays a UML diagram that shows the resulting component design. Due to the ExplorViz extension mechanism, it resembles Figure 5.1. The grey colored elements
6. Architecture

are derived from the architectural requirements. We now explain each of these three components in terms of functionality and communication.

**Agent** The ExplorViz Discovery Agent (hereinafter also referred to as agent or Discovery Agent) contains the application discovery and monitoring management features. Therefore, it is responsible for executing the PG-method and analyzing its data, e.g., find new processes. Potential extensions regarding the discovery functionality can be implemented in this component. The monitoring is outsourced to Kieker, which independently sends its data to the monitoring interface of the backend.

The agent provides its resulting resources (see Section 2.1) by means of the data interface. Furthermore, the interface also accepts requests that modify resources, e.g., another component changes a value.

**Backend Extension** The agent’s data must be available to ExplorViz frontend users. This is the purpose of the backend extension. It provides an interface that an agent uses to register. As a result, the backend knows about the existence and can submit requests through the agent’s data interface. These request are initially triggered from the frontend extension and then forwarded by the backend component. Therefore, it also provides an interface to send and obtain data.

**Frontend Extension** The frontend extension is responsible for the GUI and visualization of the agent’s data. It (automatically) submits requests to the backend and does not know about the existence of an agent.

### 6.3 Deployment Details

The design in Section 4.3 and the architectural requirements reveal a multi-server operation. Therefore, we must deploy the introduced components on different servers.

Figure 6.3 presents an exemplary deployment of the components with two servers. The physical target server contains a web server that hosts the agent. Therefore, application discovery and monitoring management are bound to processes running in the encompassing operating system. The agent communicates with the ExplorViz backend extension, which is contained in the explorviz-backend.war. Additionally, the frontend extension (explorviz-frontend.zip) requests resources from the backend extension.
6.3. Deployment Details

Figure 6.3. Exemplary deployment of ExplorViz and the discovery agent
Chapter 7

Implementation

This chapter describes the implementation of the application discovery and monitoring management system in ExplorViz. The ExplorViz ADAMMS comprises three components. In the following, we independently explain the implementation of these components. We start with an introduction of Jersey-related details. This is due to the fact that both the Discovery Agent and the backend extension are based on the Jersey framework. Then, we introduce particular details of these two components and eventually present the implementation of the ExplorViz frontend extension.

7.1 Details on the Jersey Framework

In this section, we introduce common details of JAX-RS and the Jersey framework, which both the agent and backend extension use. In JAX-RS, (RESTful) resources are defined with root resource classes (RRC). At runtime, these RRCs are plain old Java objects. Jersey can either create a RRC object for every related HTTP request or can be instructed by the @Singleton annotation to use a single instance during the runtime of the application. Further JAX-RS annotations determine attributes of the encompassing RRC, e.g., applicable operations (e.g., HTTP GET) for the resource. We can see an exemplary RRC in Listing 7.1. The @Path annotations at class and method level (lines 1 and 12) are used to define the URI for this RRC. Jersey maps a unique path to a single Java method. Therefore, getAgentById in line 14 is invoked, if the assigned URI is used inside of a HTTP request. Line 12 and 14 show how we can obtain a value from an URI's path parameter. The @Consumes annotation in line 13 determines the media type, e.g., application/vnd.api+json for JSON API documents, that every HTTP requests must use to successfully submit the request. Line 6 presents a JAX-RS unrelated feature of the Jersey framework. The @Inject annotation instructs the program to automatically pass (hereinafter also referred to as inject) a held instance of the demanded class to the constructor. This feature is called dependency injection (DI) and is provided by the employed HK2 DI library.\footnote{https://javaee.github.io/hk2/lastreleaseforFebruary19,2018} Therefore, instead of managing and holding necessary parameters, dependencies are externally passed. In summary, RRCs can be seen as entry point to the provided data of the web service.
7. Implementation

```java
@Path("agent")
//@Singleton
public class AgentResource {
  private final InternalRepository repo;

  @Inject
  public AgentResource(InternalRepository repo) {
    this.repo = repo;
  }

  @GET
  @Path("{id}")
  @Consumes(MEDIA_TYPE)
  public Response getAgent(@PathParam("id") String id) {
    Agent agent = this.repo.getAgentById(id);
    return Response.status(200).entity(agent).build();
  }
}
```

Listing 7.1. Exemplary RRC for obtaining agents

In the past, a standard web application deployment descriptor (WADD) was necessary to register web components, e.g. resources, filters, and listeners, at the web application. These definitions were placed in the `web.xml` file that is commonly located in the `WEB-INF` directory. However, since the release of the Java Servlet API 3.0 there is an alternative procedure to define necessary components. Instead of the WADD, we can use Java annotations on class level, e.g. `javax.servlet.annotation.Weblistener`. These annotated classes are then programmatically added to the runtime configuration of the web application. Additionally, there is an automatic discovery feature that uses classpath scanning and subsequently adds the found classes to the configuration. All of these configurations are done inside of the JAX-RS Application class, i.e., the starting point for JAX-RS applications. It is comparable to the `main` method in common Java programs. Jersey applications use the `ResourceConfig` class, i.e., a Jersey extension of the JAX-RS Application class, as starting point.
Listing 7.2. Exemplary ResourceConfig

Listing 7.2 shows an exemplary ResourceConfig class. Its constructor is automatically executed on application startup. First of all, it implements an AbstractBinder (lines 6-11), i.e., the object that configures the dependency injection. This object essentially enables us to obtain required dependencies by means of the @Inject annotation, for example in the previously introduced sample RRC (see Listing 7.1, line 6). It is registered at the application in line 13, hence the availability in RRCs. Injectable objects can be stored in-memory as singleton objects. Alternatively, we could use the RequestScoped class instead of the Singleton class in line 9 to create a new instance of our injectable object for each client request. The parameter of the bind method is the type that we inject for the parameter of the to method, i.e., the requested type in source code. Therefore, it must be compatible to the requested type. Most of the time the implementation of an AbstractBinder will be outsourced to another class. However, we included the implementation into the ResourceConfig to show this important feature. In summary, the DI feature and its underlying pattern keep the coupling between software modules or classes minimal with respect to the single responsibility principle.

Line 14 presents the previously mentioned package scanning feature. In this case, the org.dummy.server.resources package directory is scanned for classes with JAX-RS-annotations. Finally, we define the application path with the ApplicationPath annotation at the top of the ResourceConfig class (line 1). This string acts as the base URI for the JAX-RS application. An application path v1 will result in an application that is accessible at http://IP:Port/v1/. Therefore, the single method of the introduced sample RRC in Listing 7.1 is accessible at http://IP:Port/v1/agents/id, whereby id is a placeholder for a valid resource identifier.
7. Implementation

7.2 Data Model

Naturally, we use classes to encapsulate cohesive attributes in object-oriented programming. This includes data models, which are used to represent the underlying data of a system. This section briefly introduces the data models of the ExplorViz ADAMMS, which are used by all three components. In-depth explanations of the encompassing features follow in Section 7.3.

```
BaseModel
-ID_GENERATOR : AtomicLong
#id : long
#name : String
#lastDiscoveryTime : long
#hidden : boolean
#errorOccured : boolean
#errorMessage : String

Agent
-ip : String
-port : String
+Agent(agentIP : String, agentPort : String)
+updateProcezz(newProcezz : Procezz) : void

Procezz
-pid : long
-workingDirectory : String
-osExecutionCommand : String
-userExecutionCommand : String
-proposedExecutionCommand : String
-aopContent : String
-procezzManagementType : String
-programmingLanguage : String
-monitoredFlag : boolean
-webserverFlag : boolean
-stopped : boolean
-restart : boolean
+Procezz() : Procezz
+Procezz(newPID : long, newCommand : String)

Figure 7.1. Data model of the ExplorViz ADAMMS (without getter and setter methods)
```

Figure 7.1 illustrates a UML class diagram of our model. It is implemented by means of Java classes and Ember Data models.

**Agent** The Agent class is used to encapsulate properties of a single deployed Discovery Agent. Its ip and port attributes are primarily used by the backend extension to submit requests based on the agent’s address. The class contains a list of discovered application processes, which are depicted by the Procezz type (in the following, we notate an instance of this type with procezz). The updateProcezz method adds the passed procezz to the agent’s list. Furthermore, it uses the id of the procezz to find and remove an old instance with the same id.
Procezz  As mentioned, the Procezz class is used to encapsulate attributes of a discovered application process. We use the ExplorViz naming convention to circumvent potential problems with the otherwise same-named Java Process class. The model class contains attributes for the process details, e.g., PID, working directory, and execution command. The latter is saved in the osExecutionCommand attribute. The userExecutionCommand represents a user-defined command. It has a higher priority than the osExecutionCommand. Therefore, the Discovery Agent employs the user instead of the os execution command, if it is instructed to restart an application process. The proposedExecutionCommand is used by the recognition feature of the discovery feature. A recognition strategy that identified a procezz might propose an alternative execution command and therefore set this attribute. The aopContent represents the stringified content of the application process’ aop.xml. The procezzManagementType attribute contains a string value that is used to identify the associated management type. These classes handle data gathering and process lifecycle management, e.g., killing of an application process. The programmingLanguage, monitoredFlag, and webserverFlag attributes of the Procezz class are self-explaining. Finally, stopped and restart are used to trigger operations for an application process. For example, the Discovery Agent will restart a process, if restart is set to true. We do not provide URIs that essentially represent an operation, since this is a Remote Procedure Call style and does not belong in RESTful web applications.2

Base Entity  Both introduced data models extend the BaseModel class. It contains the unique id attribute that is mandatory for JSON API documents (see Section 2.2). The hidden attribute is used in the GUI to determine, if an entity should be visualized. Remaining attributes are self-explaining.

7.3 Discovery Agent

Due to the ExplorViz extension mechanism (see Chapter 5), backend extensions are commonly advised to use Jersey. We follow this advise and use the same framework for the Explorviz Discovery Agent, since on the one hand we do not want to unnecessarily extend the software stack of ExplorViz. On the other hand Jersey and JAX-RS enable us to define interfaces, which can also be used by other programs. As a result, we do not restrict the applicability of the Discovery Agent and the application discovery and monitoring management functionalities can be used outside of ExplorViz. If we closely examine our previously introduced design in Chapter 4, we can see that these two functionalities are each constituted of separate steps, e.g., data gathering, recognition or starting new application processes. It is necessary to implement each of the steps, so that the overall features can be provided. In this section, we explain the implementation of the ExplorViz Discovery Agent.

2http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post(last retrieved February 27, 2018)
7. Implementation

7.3.1 Root Resource Classes

The introduced data model classes (see Section 7.2) are instantiated by the Discovery Agent. In terms of RESTful design, these classes represent the data, which is accessible and modifiable by submitting requests to the associated RRCs. Table 7.1 presents the RRCs of the Discovery Agent.

<table>
<thead>
<tr>
<th>RRC</th>
<th>Class Name</th>
<th>URI Path</th>
<th>HTTP Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentResource</td>
<td>/agent</td>
<td>GET, PATCH</td>
<td></td>
</tr>
<tr>
<td>ProcezzResource</td>
<td>/procezz</td>
<td>PATCH</td>
<td></td>
</tr>
</tbody>
</table>

We can see that the AgentResource allows two HTTP methods. GET is used to obtain the Agent object along with the contained list of Procezz instances. The PATCH request is used to modify resources. Therefore, it must contain a representation of the agent resource in the HTTP body. The PATCH response returns a final representation to circumvent data loss, since restarting an application process will result in a new PID. The same HTTP method is used to modify processes in the ProcezzResource.

7.3.2 Application Discovery

In this section, we introduce the implementation of the application discovery feature. It is based on the design in Section 4.1. Therefore, the Discovery Agent repetitively executes the following implemented steps.

Data Gathering & Process Lifecycle Management  For this thesis, we use the PG-method and the command-line tool pwdx to obtain details of Java processes from the operating system (Chapter 3). The consequent implementation is based on an interface called ProcezzManagementType. Implemented process management types are classes, which provide methods to read and manage processes of the encompassing OS. They essentially represent the extensible application discovery functionality.

Figure 7.2 depicts a UML class diagram that shows this extensible feature. The JavaCLIManagementType realizes the design of our PG-method. For the sake of reusability, we outsourced its implementation into the CLIAbstraction class. This contains methods to start, kill, and read processes based on command-line tools. We use the Java ProcessBuilder to create the necessary OS processes for these tools. Any process management type must further insert its unique identifier into every discovered process. We use this identifier to obtain the correct management type for a process, for example if the agent is instructed to restart the process. Therefore, implemented process management types are accessible by
7.3. Discovery Agent

invoking the ProcezzManagementTypeFactory. This class holds a mapping between identifiers and associated management types. New management types must be simply added to this source class and the application discovery feature will automatically execute them.

![Diagram of ProcezzManagementTypeFactory and its interfaces]

---

**Figure 7.2.** Extensible process management type implementation
7. Implementation

Recognition Strategies The next step of the discovery feature is the recognition of applications based on the found processes.

![UML class diagram](image)

**Figure 7.3. Extensible recognition feature implementation**

Figure 7.3 shows a UML class diagram that presents the implementation of our design for the recognition feature (see Section 4.1.1). As with the process management type feature, we use an interface called `DiscoveryStrategy` to facilitate the implementation of new strategies. These must be inserted in the `DiscoveryStrategyFactory`, so that the Discovery Agent can automatically employ them. In this thesis, we implemented two recognition strategies, namely `KiekerSampleAppStrategy` and `TomcatStrategy`, which are used to recognize the same-named applications based on the execution command. Both strategies set the name attribute of the process, if the recognition succeeds. The `KiekerSampleAppStrategy` additionally proposes a suggestion for a new execution command, if the initial command is not sufficient for restarting.

7.3.3 Monitoring Management

Monitoring with Kieker is achieved by inserting the `javaagent` command-line option to a process’ execution command (see Section 4.2). If the agent is instructed to enable the monitoring for a process, i.e., a process’ monitoredFlag attribute is set to true, it will first kill the running process instance. Then, the agent modifies the execution command of the associated process data model, and starts a new process with this modified command. The insertion of the `javaagent` is automatically done by means of String operations. Since we might not want to use the default monitoring settings of Kieker, the agent hosts the `kieker.monitoring.properties` and `aop.xml` for each found application. The system paths to these files are also inserted in the execution command, so that Kieker uses these specific files. Finally, if Kieker successfully instrumented an application process, it will send the gathered monitoring data to the ExplorViz monitoring interface (see Figure 6.2).
7.4 ExplorViz Backend Extension

7.3.4 Data Coherence
The application discovery and monitoring management features are periodically executed. Therefore, we need to compare processes from successive iterations to recognize new processes or losses. The Discovery Agent reads current processes every 30 seconds and compares them to the old list of processes. For that reason, our implementation uses the pair of PID and execution command to recognize, if a process is still running, lost, or new. If a process will be restarted, the responsible process management type inserts an identifier into the execution command, so that it can be recovered from a new process list. Then, the old PID and execution command of the process are updated.

7.4 ExplorViz Backend Extension
The backend extension acts as gateway between Discovery Agent and frontend extension. In this section, we introduce its implementation.

7.4.1 Root Resource Classes
As with the Discovery Agent, the backend extension contains RRCs, which the frontend can obtain.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>URI Path</th>
<th>HTTP Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentResource</td>
<td>/agent</td>
<td>POST, PATCH</td>
</tr>
<tr>
<td></td>
<td>/agents</td>
<td>GET</td>
</tr>
<tr>
<td>ProcezzResource</td>
<td>/procezz</td>
<td>POST, PATCH</td>
</tr>
</tbody>
</table>

Table 7.2 presents the contained RRCs. Once more, HTTP PATCH requests are used to modify representations of the resources. The AgentResource allows a GET request to obtain the list of all registered agents and their included processes. The POST requests are used by the Discovery Agent to obtain unique identifiers for its single Agent and Procezz objects. As previously mentioned, these are required to form valid JSON API documents (see Section 2.2).

7.4.2 Implementing Client Requests
In comparison to the Discovery Agent, the RRCs of the backend extension do not return internally saved data models. Instead, we “forward” requests to respective agents. In our terminology, forwarding means that an incoming request is submitted as new request from
7. Implementation

the backend to the respective Discovery Agent (see Section 6.1.2). Therefore, the backend acts as client. We use this approach, since Java EE allows forwarding requests only on the same server.

```java
final String uri = "http://127.0.0.1:8080/agent";
final Client client = ClientBuilder.newClient();
final WebTarget target = client.target(uri);
final Builder httpReqBuilder = target.request("text/plain");
final Response serverResponse = httpReqBuilder.get();
final Agent agent = serverResponse.readEntity(Agent.class);
```

Listing 7.3. Exemplary JAX-RS client HTTP GET request

Listing 7.3 shows how to use the JAX-RS client package to submit a HTTP GET request. The ClientBuilder provides a method for obtaining a Client object (line 3). The WebTarget object holds the resource target information, i.e., the URI (line 4). The request method is used to define the accepted response media types (line 4). Furthermore, it returns the Invocation.Builder object, which provides actual methods to submit the request (line 7). The response body can be read and converted to an object (line 8).
7.5 ExplorViz Frontend Extension

The core of ExplorViz uses the JS web framework Ember.js to visualize its data (see Section 2.4 for a broad overview). We already discussed how it is possible to extend this component by means of an Ember addon in Chapter 5. In this section, we introduce the implementation of our ExplorViz frontend extension that belongs to the ExplorViz ADAMMS.

Figure 7.4. Overview of the implemented ExplorViz frontend extension

Figure 7.4 shows the main JS classes of the implemented ExplorViz frontend extension. Each class has a specific purpose. The colors show the affiliation of a class.

7.5.1 Data Model

Green-colored classes in Figure 7.4 are used for the internal data storage. The model class Agent and Procezz are derived from Figure 7.1 and converted into Ember models. Therefore, they contain the same attributes like the Java models. The Agent Repository service is a globally accessible singleton object, i.e., an Ember service, that is used to hold a list of the latest Agent records. By using this object, we can circumvent querying and subsequently filtering the Ember Store for the latest Agent records (see Section 2.4). Instead, we simply obtain the respective list from the Agent Repository service. The mentioned Ember Store represents an Ember service that contains all loaded records from the backend. This service further provides methods which essentially simplify submitting network requests (see Section 2.4).
7. Implementation

7.5.2 Graphical User Interface

Blue-colored classes in Figure 7.4 are responsible for any visible aspect of the frontend extension. Therefore, these classes have separate Handlebars template files, except for the Discovery Controller.

Discovery Route & Discovery Controller  The Discovery Route is the entry point of our extension. Thus, it is invoked by Ember if we navigate to http://ip:port/baseURI/discovery. This can be done by manually inserting the URI into a web browser. We additionally used the provided ExplorViz frontend core service called page-setup that enables us to add a button to the frontend’s navigation bar (see Section 5.2). Therefore, we can also simply click on this button to invoke the Discovery Route.

```
{{#if procezzForDetailView}}
  {{procezz-details errorHandling=(action errorHandling) procezz=procezzForDetailView}}
{{else if agentForDetailView}}
  {{agent-details errorHandling=(action errorHandling) agent=agentForDetailView}}
{{else}}
  {{discovery-overview showDetails=(action showDetailsComponent)}}
{{/if}}
```

Listing 7.4. Handlebars template file of the Discovery Route

Listing 7.4 presents the Handlebars template file of the Discovery Route. We can see that the route invokes different Ember components based on conditions rather than to show raw HTML code. We therefore use Handlebar’s built-in if block helper. On first startup, the route initializes its controller and then invokes the Discovery Overview component. The latter is achieved by line 6. We also pass a closure action to this component. Therefore, if we call the showDetails function in the Discovery Overview component, we actually invoke the showDetailsComponent function of the Discovery Controller. This functionality is commonly used in Ember applications and enables developers to recognize changes or user interactions inside of a (usually self-contained) component. The invoked showDetailsComponent function received a parameter from the Discovery Overview component. This parameter is used by the controller to set either the procezzForDetailView or agentForDetailView attribute of the Discovery Route. Eventually, Ember’s built-in binding feature detects that an attribute’s value which is used in a template file changed and automatically re-renders the route. Therefore, the related template is once again invoked and line 2 or 4 are executed.

---

3https://handlebarsjs.com/builtin.helpers.html(last retrieval March 26, 2018)
4https://guides.emberjs.com/v3.0.0/object-model/bindings(last retrieval March 26, 2018)
7.5. ExplorViz Frontend Extension

**Discovery Overview**  On initial startup, the **Discovery Route** will show the **Discovery Overview** component. Based on the retrieved data from the **Agent Repository**, the component will notify the user that no data is available or visualize the obtained **Agent** records.

![ExplorViz Visualization](image)

**Figure 7.5.** The graph-based visualization of the ExplorViz frontend extension

Figure 7.5 shows the visualization that is provided by the **Discovery Overview** component. Thus, it is shown if we navigate to the **Discovery Routes’ URI** or click on the **Discovery banner**. The depiction is an implementation of our introduced graph design (see Section 4.4). It is based on the Cytoscape.js graph theory library.\(^5\) A built-in breadth-first algorithm is utilized to depict model records by means of a tree data structure. Within the graph, we further use arrows to indicate the underlying agent and application topology. The tree root represents the frontend extension (see ⁹). It is supposed to provide a fixpoint, so that users do not get lost in large software landscapes. ⁸ represents a registered and running ExplorViz Discovery Agent. The label above the graph node shows the ip address of the encompassing host machine and the port of the agent. Alternatively, it shows a user-defined name if declared. ⁷ illustrates a running application that was discovered by ⁹. This kind of graph node is labeled with the PID of an underlying process or with a name. The latter was either declared by a user or inserted by a recognition strategy. Failed applications are highlighted by using red-colored graph nodes and edges (see ⁹).

---

\(^5\)http://js.cytoscape.org (last retrieval March 26, 2018)

47
7. Implementation

the failure is first recognized, the frontend temporarily displays a pop-up that contains a problem description and a potential root cause (see ➀). Users are able to stop applications, i.e., killing the underlying process. Graph nodes and edges of stopped applications are yellow-colored (➁).

The introduced graph visualization must be updated accordingly, if the frontend extension requests and obtains resources from the backend. Otherwise, we would lose information. Therefore, the template file of the Discovery Overview uses a value binding similar to the one used in the Discovery Route. The bound value is the list of Agent records that is held by the Agent Repository. If the list is updated, Ember triggers a re-rendering of the Discovery Overview component. However, in contrast to the Discovery Route, we need additional logic to handle the rendering. Cytoscape.js requires a single HTML div and subsequently draws the graph on a dynamically created HTML canvas element. We must programmatically define the underlying graph JS object and potential graph node listeners, i.e., callback functions that are triggered if a node was clicked, in source code. This cannot be achieved from within the related Handlebars template. Therefore, we exploit the provided lifecycle hooks ➆ of the encompassing Ember component to invoke the necessary code. The didRender hook is automatically called on initial and re-rendering. We override this hook to call setup code, i.e., initializing the graph object and graph node listeners, as well as the update and re-rendering function for the graph and includes nodes, respectively. Eventually, a boolean variable is used to ensure that the setup code is executed only once. We additionally use the willDestroyElement hook to remove bound listeners. This hook is automatically called if the component will be destroyed due to an action, e.g., another component will be invoked. The introduced graph-based visualization provides an overview of the current ExplorViz Discovery Agent and application topology. Precise information of an application process or agent are shown if the related graph node is clicked.

Procezz Details & Agent Details  The Procezz Details and Agent Details components are used to present in-depth information of the same-named models, e.g., process ID, execution command or ip address. We classify three types of information, i.e., general, execution, and monitoring information. Each classification is depicted by using a self-contained dialog. Therefore, the Procezz Details and Agent Details components act as parents for further nested Ember components, similar to the Discovery Route which acts as parent for the Discover Overview component. We can recognize this in Figure 7.4. By using nested Ember components, we ensure extensibility. To see that, we assume that the Procezz model receives new attributes, which constitute a new information type. We do not want to embed these attributes in the existing dialogs, since in terms of semantics they might not belong to any type. Therefore, we would create a new Ember component and simply invoke it from the parent’s template. Data modifications, e.g., changing the value of an attribute, must then be handled by the component.

[6](https://guides.emberjs.com/v3.0.0/components/the-component-lifecycle)
7.5. ExplorViz Frontend Extension

Figure 7.6. General information dialog for a single application process

Figure 7.7. Execution information dialog for a single application process
7. Implementation

Figure 7.6 shows the implemented general information dialog for a single application process. The depicted attributes are explained in detail in todoRef. Modifications that are performed in any dialog of the GUI are saved on the responsible agent by clicking the Save button. The Restart button furthermore saves and subsequently restarts an application process. Therefore, the PID will change accordingly and if enabled, the monitoring will be started. Finally, the Stop button will kill the application process. Any button interaction is followed up with a notification pop-up that states if the clicked operation was successful (similar to the pop-up in Figure 7.5).

Figure 7.7 presents the execution information dialog. This contains any information regarding the application process execution. The initial execution command is depicted, so that users can see if they need to define a custom user execution command. The responsible HTML input might indicate the recommendation that was proposed by an applied recognition strategy. We can manually define another user execution command or use this recommendation by clicking the button below. The final command that the ExplorViz Discovery Agent uses to start a new application process is shown in the Agent Execution Command row. This command might include the Kieker monitoring command-line options as well as the indicator that is used to recognize the newly started process.

Figure 7.8. Monitoring information dialog for a single application process
7.5. ExplorViz Frontend Extension

Figure 7.8 shows the monitoring information dialog. This is used to enable or disable an application monitoring and to define the content of the Kieker aop.xml. During a save or application restart, the ExplorViz Discovery Agent will update the affiliated aop.xml with the content of this HTML textarea, therefore apply the user monitoring settings.

Figure 7.9 shows the general information dialog for an Agent record. It is provided by the Agent Details component. As Figure 7.4 shows, this component contains one instead of three nested components. This is due to the fact that the implementation does not contain further information types for the Agent model at this time. In the presented general information dialog, we can name this instance of an ExplorViz Discovery Agent and hide it in the graph visualization. The hiding is an additional feature that the GUI provides for single applications or the agent along with all its discovered applications. We can show hidden entities and subsequently revoke this operation by using a button in the frontend extension configuration dialog (see Section 5.2). It is accessible by clicking the cogwheel symbol in the navigation bar.

7.5.3 Networking

To communicate with the backend extension, for example to save a modified application process, we use the orange-colored classes in Figure 7.4.
7. Implementation

**Agent Reload** The Agent Reload class is an Ember service that is used to periodically request the list of ExplorViz Discovery Agents with their discovered applications. It is an extension of the Data Reload service from the ExplorViz core’s frontend. The recurrent execution is handled by using the Ember `run.later` method.² Thereby, it is ensured that the Ember application will request new data every 10 seconds.

```
1 this.get("store").findAll("agent").then(success, failure).catch(error);
```

Listing 7.5. Instruction that results in a HTTP request to obtain all ExplorViz Discovery Agents and their data

Listing 7.5 shows the instruction that the Agent Reload service uses to request new data. The used `findAll` method of the Ember Store requests all agent resources from the backend and returns a JS promise object. Depending on the promise’s state, the program proceeds and for example saves the data within the execution of the success callback function. For that reason, the Agent Reload service overrides the attribute of the Agent Repository with this new data. Looking back at the graph visualization in Section 7.5.2, we can see that this single set instruction would now trigger the re-rendering of the graph.

**Adapters & Serializer** Networking in Ember is based on Adapter and Serializer classes. Developers can create these two files for any model to modify the model’s conventional networking aspects such as the backend URI. Adapter classes convert any store request, e.g., `findAll` method, into HTTP requests. They use a Serializer that transforms records into a data-interchange format such as JSON and vice versa. Ember provides a built-in `JSONAPISerializer` that automatically transforms records into JSON API documents during requests. Our backend does not follow the Ember-defined conventions, because we use singularized model names in our URIs. Therefore, we use a pair of Adapter and Serializer classes for each model (see Figure 7.4).

²https://guides.emberjs.com/v3.0.0/applications/run-loop/last retrieval March 27, 2018
Chapter 8

Evaluation

The introduced implementation of an application discovery and monitoring management system in ExplorViz is evaluated by means of two experiments. A feasibility experiment is used to examine, if the system behaves and operates correctly. Subsequently, we conduct a pilot study with one proband to indicate the usability of the system. This chapter presents the related details of the evaluation process.

8.1 Goals

Our ExplorViz ADAMMS is supposed to provide an alternative and more usable procedure to setup and configure the monitoring of Java applications. As a result, the previous manual monitoring procedure might be obsolete.

The first goal of the evaluation is to determine the feasibility of our implementation and to verify if it correctly operates. This includes the operation in new environments.

The second goal of the evaluation is to examine the usability of our system. ExplorViz is mainly directed to software researchers, developers, and operators. Users from these groups come with general knowledge about usability and comprehension of software. Nevertheless, any GUI-based software should maximize its usability. The GUI of the ExplorViz ADAMMS is designed with respect to the five usability characteristics (see Section 4.4). As we preliminary mentioned, the usability evaluation is conducted by means of a pilot study. Therefore, we are interested in perceived usability rather than comparable results.

8.2 Methodology

This section describes the methodology that we use for the evaluation of our implementation. Feasibility experiments are commonly used in research for software, which addresses issues with new approaches [Shaw 2003]. Therefore, we conduct an feasibility experiment to show, if our developed system operates correctly in an exemplary software environment. In a pilot study, we examine the usability of our implementation. We employ a usability test with tasks that our proband has to solve. In a following structural interview we gather feedback from our evaluation participant, regarding perceived usability and potential enhancements.
8. Evaluation

8.3 Feasibility Experiment

The first goal of the evaluation is to show that the implementation is feasible. Our ExplorViz ADAMMS provides several small features, which essentially belong to two major functionalities, i.e., application discovery and monitoring management. We verify, if these two functionalities correctly operate in an exemplary multi-server environment with different applications.

Table 8.1. Mechanics that are examined in the feasibility experiment

<table>
<thead>
<tr>
<th>Mechanics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
</tr>
<tr>
<td>$M_2$</td>
</tr>
<tr>
<td>$M_3$</td>
</tr>
<tr>
<td>Application discovery</td>
</tr>
<tr>
<td>Monitoring management</td>
</tr>
<tr>
<td>Fault tolerance</td>
</tr>
</tbody>
</table>

Table 8.1 shows a reduced overview of ExplorViz ADAMMS’ functionalities and characteristics. $M_1$ represents the correct detection and recognition of application details as well as the correct visualization. $M_2$ addresses the lifecycle management of application processes and the handling of monitoring aspects, such as enabling and disabling the monitoring for a single application. In summary, $M_1$ and $M_2$ cover all feature-related functionalities of the ExplorViz ADAMMS. The monitoring management includes lifecycle operations on processes, such as stopping a process. Problems can occur while our system is supposed to execute these operations, e.g., an invalid execution command will result in a process loss. Therefore, we implemented several aspects to provide fault tolerance and subsequently notify the operator. In our experiment, we examine if these mechanisms work as supposed ($M_3$).

8.3.1 Experimental Setup

This section presents the soft- and hardware setup of the conducted experiment.

**Hardware** Our experiment is conducted in a multi-server environment. Therefore, we are going to employ different machines that act as hosts for applications. Table 8.2 shows the specification of the used machines. node3 and node4 are cloud servers provided by the Software Engineering Group of the Kiel University. lp1 is the laptop that we used to interact with the ExplorViz ADAMMS GUI. All machines are located in the same local area network.
8.3. Feasibility Experiment

Table 8.2. Used environment setup for the evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Quantity</th>
<th>Specification</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>node3, node4</td>
<td>2</td>
<td>CPU: 2x Intel Xeon E5-2650</td>
<td>Discovery Agent, KiekerSampleApp, JPetStore 6,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAM: 128 GB</td>
<td>(Explorviz)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OS: Debian 64 Bit</td>
<td></td>
</tr>
<tr>
<td>lp1</td>
<td>1</td>
<td>CPU: Intel Core i5-4278U</td>
<td>Discovery Agent, KiekerSampleApp, JPetStore 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAM: 16 GB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OS: macOS High Sierra 64 Bit</td>
<td></td>
</tr>
</tbody>
</table>

Software The ExplorViz ADAMMS is supposed to find running Java applications in the OS and handle their monitoring management. To examine these features in our evaluation, we require a set of applications. We employ an Apache Tomcat web server with an included JPetStore 6.¹ The latter provides a web application that includes an online shop. JPetStore is a common application for testing in research. Additionally, we use a simple Java program that independently runs in the OS. It is called KiekerSampleApp and simply calculates Fibonacci numbers and executes SQL queries.²

In our feasibility experiment, the machines from Table 8.2 host the introduced applications. However, the cloud servers are simultaneously used by other researchers. We provided a Docker image³ that includes the two Java applications to avoid an unwanted killing of foreign processes. The image further contains another Tomcat web server with the ExplorViz Discovery Agent. All three machines will execute a resulting Docker container of this image. Therefore, the overall software landscape of the evaluation consists of six Java applications deployed on these three hosts. A second Docker image provides the extended ExplorViz backend and frontend. The resulting container is also running on node3. Thus, all ExplorViz Discovery Agents and the potentially inserted monitoring components send their data to node3.

8.3.2 Execution of the Experiment

During the implementation, we defined two recognition strategies. These were used in this experiment to verify the correct functionality of the recognition feature of the encompassing discovery feature. These strategies are supposed to recognize and label the two sample

¹https://github.com/mybatis/jpetstore-6 (last retrieval March 18, 2018)
²https://github.com/czirkelbach/kiekerSampleApplication (last retrieval March 18, 2018)
³https://www.docker.com/what-docker (last retrieval March 18, 2018)
8. Evaluation

applications. One of the strategies should also propose a user execution command for the KiekerSampleApp application. They are both annexed in the appendix of this thesis to facilitate reproducibility of the experiment. Additionally, we tested the monitoring of both applications on a separate device to find out an include directive for the aop.xml that results in a functional monitoring. We used net.explorviz..* for the KiekerSampleApp application and org.mybatis..* for the Tomcat web server in the experiment. After the setup on the introduced machines, we begun the feasibility experiment. We used all features that are introduced by Table 8.1 multiple times in random order. Furthermore, we intentionally forced process failure with invalid user execution commands to verify if the system is fault tolerant. Finally, we tested if suddenly offline ExplorViz Discovery Agents are recognized by the system and are able to re-register during runtime.

8.3.3 Results

We now introduce the results of the feasibility experiment.

**M1** The ExplorViz ADAMMS’ GUI visualized three agents and nine applications. Three of these applications were labeled as kiekerSampleApp. The others were labeled as Tomcat Web Server. The execution dialog of the applications presented the correct process execution commands. Furthermore, one recognition strategy proposed an alternative user execution command for the kiekerSampleApp applications.

**M2** Running applications could be stopped and the visualization subsequently yellow-colored the related graph nodes. If the (user) execution command was valid, we were able to restart the process. The system then updated the PID and grey-colored the related graph nodes. We were able to modify the aop.xml of every found application. If monitoring was enabled and we restarted an application, the ExplorViz frontend visualized their runtime data after a short time. Furthermore, our visualization green-colored monitored applications. This was only achievable if we used the previously found out include directives.

**M3** We inserted invalid user execution commands for every application and subsequently restarted them. The system notified us about the process loss of these applications and red-colored the related graph nodes. We were able to rectify the user execution commands and restart once again. All processes correctly started and the graph nodes were grey-colored. In terms of fault tolerance regarding monitoring, we additionally inserted different include directives into the aop.xml files. Depending on these directives, the system recognized a process loss that was rectifiable or notified us about a functional monitoring and green-colored the related graph nodes. We intentionally stopped the Docker container running on lp1. The system recognized the agent loss, red-colored the related graph node, and notified us about this problem. We restarted the Docker container and the agent registered at the
8.3. Feasibility Experiment

backend extension. Finally, the graph node was updated and we could proceed to monitor applications on this server.

8.3.4 Discussion

This section discusses the previously presented results.

M1 The system found nine instead of six applications. This is due to the fact that the three ExplorViz Discovery Agents found their encompassing Tomcat web servers. We did not expect this behavior, since it did not occur in local tests. Besides this, the application discovery found all three agents and the six test applications. The included recognition feature correctly labeled all applications and proposed the alternative user execution command for the kiekerSampleApp applications. Regardless of this result, we recognized shortcomings of the employed strategies. The kiekerSampleApp strategy assumed relative paths inside of the execution command. The Docker container started the kiekerSampleApp application from within the associated directory. Therefore, the execution command of the process contained a relative path. The strategy would have proposed a wrong user execution command, if we initially started the application with an execution command containing an absolute path.

M2 The process lifecycle management, in combination with disabling and enabling the monitoring with the correct include directives, worked for every application. However, we perceived non-deterministic system behavior, if we used a directive that results in a non-functional monitoring. This is due to the fact that Kieker endlessly holds the process execution. Therefore, our system assumes that the process is running with a functional monitoring. At the time of writing this thesis, there is no other way to verify if the monitoring works correctly, since we cannot obtain this information from the ExplorViz core. During the experiment, we could resolve the problem by restarting the application with another include directive.

M3 If a sudden agent loss was recognized, the system always notified us by showing a pop-up in the GUI. This pop-up contained a problem description and a potential root cause. The results further show that it was always possible to rectify issues and proceed with the operation of the system.

8.3.5 Threats to Validity

The implemented recognition strategies may be too tightly coupled to the two test applications and their initial execution commands. Nevertheless, we verified that the recognition system essentially functions, but requires further tests with new generic strategies that are applicable regardless of the initial setup. Therefore, we suggest to conduct another
8. Evaluation

experiment to examine this feature. The employed strategies are attached in the appendix and can therefore be used as starting point for new implementations.

The experiment was essentially conducted on two systems, since the cloud servers resemble each other. The results may not be representative for all kinds of available systems. For this reason, we suggest to conduct further experiments with multiple different systems.

8.4 Usability Evaluation

For our pilot study, we are going to examine the usability of our system with an emphasis on the graphical user interface. The GUI was designed and implemented with respect to the five usability characteristics (see Section 4.4). We therefore conduct an experiment to observe the realization of these characteristics, so that we can derive indications for the overall usability [Holzinger 2005]. We asked a researcher of the Software Engineering Group of the Kiel University to test our system. The proband holds a master’s degree in computer science and utilizes ExplorViz in his daily work. For this experiment, we designed a set of tasks that covers the main features of our system.

<table>
<thead>
<tr>
<th>Table 8.3. Task for the probands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
</tr>
<tr>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_2$</td>
</tr>
<tr>
<td>$T_3$</td>
</tr>
<tr>
<td>$T_4$</td>
</tr>
<tr>
<td>$T_5$</td>
</tr>
<tr>
<td>$T_6$</td>
</tr>
<tr>
<td>$T_7$</td>
</tr>
</tbody>
</table>

Table 8.3 shows the set of tasks that the proband has to solve. $T_1$ and $T_2$ are supposed to show how rapidly the proband is able to start his work (learnability). The general deployment requires only copying the provided build artifacts into instances of web servers. This is a common procedure and does not differ from the general ExplorViz setup. Therefore, we exclude this step. The remaining setup step is a mandatory modification of the ExplorViz Discovery Agent configuration file. This file contains the ip address of the ExplorViz backend. $T_2$ represents the main purpose of our system, i.e., enabling monitoring through the new GUI. $T_3$ is supposed to verify if obtained knowledge can be applied to unknown tasks that resemble each other (memorability). $T_4$ is used to examine if the proband is able to use his obtained knowledge in repetitive tasks (efficiency). $T_5$ should show the memorability of the graphical user interface. $T_6$ and $T_7$ are used to investigate how the user acts with the system while an error occurred and if he is able to rectify this error (low error rate). The satisfaction while using the system is discussed with the proband.
8.4. Usability Evaluation

after the tasks have been solved. On that account, we conduct a structural interview to
gather feedback from the proband.

8.4.1 Experimental Setup

This section presents soft-and hardware setup of the conducted usability experiment.

**Hardware** We use the same hardware as shown in Section 8.3.1. Additionally, we employ
another laptop that the proband will use. The specification of this device are presented in
Table 8.4. As with the other machines, this device is located in the same local area network.

<table>
<thead>
<tr>
<th>Name</th>
<th>Quantity</th>
<th>Setup</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>lp2</td>
<td>1</td>
<td>CPU: Intel Core i7-6700HQ</td>
<td>Discovery Agent, Discovery Agent,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ram: 32 GB</td>
<td>KiekerSampleApp, JPetStore 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OS: Windows 10 64 Bit</td>
<td></td>
</tr>
</tbody>
</table>

**Software** We also use the same software setup as shown in Section 8.3.1. In this evaluation,
lp2 runs another Docker container with the two test applications and the ExplorViz
Discovery Agent.

8.4.2 Execution of the Experiment

The experiment and the subsequent interview were conducted at the proband’s place of
work. Before the experiment started, we introduced the ExplorViz ADAMMS to the proband.
In this process, we explained the Discovery Agent’s configuration file and all interactions
within the GUI. Furthermore, we explained that TI only requires the modification of
the distributed (node3, node4, lp1, and lp2) configuration files. Therefore, the proband
was able to connect to any device with Secure Shell (ssh). He additionally received the
include directives of the feasibility experiment (see Section 8.3.2) to eventually achieve an
application monitoring. Furthermore, we showed him that he can detect the Apache Tomcat
web servers of the ExplorViz Discovery Agent by examining the execution command (see
Section 8.3.4). During the overall introduction, the proband was allowed to ask questions.
The experiment itself had no time limit and since we are interested in a first-time usability
experiment, we did not measure any time. The tasks of the experiment were communicated
orally. The proband was allowed to ask questions at any time. We wrote down issues
and our perceived usability during the experiment. The proband used lp2 to interact with
the graphical user interface. After the tasks were solved, we ended the experiment with
the structural interview. This included an informal type of a pluralistic walkthrough, i.e.,
8. Evaluation

a method for usability inspection, to gather possible enhancements from the proband’s perspective [Nielsen 1994; Holzinger 2005].

8.4.3 Results

In this section, we present the results of the usability experiment and subsequent interview.

T1 The proband used the Linux command-line tool ifconfig to read the ip address of all four devices. He entered the ip address of the ExplorViz backend into the four configuration files. He seemed to have no problems and did not ask any questions during this task.

T2 The proband was able to monitor the KiekerSampleApp application on node3. He asked one question regarding the aop.xml content. We notated that he traversed multiple times through the application dialogs, before he eventually started to modify the user execution command.

T3 The task was rapidly solved by the proband. He did not ask any questions.

T4 The proband navigated straight to the execution dialog of the correct application. He asked a question about the display user execution command. He was able to monitor the application.

T5 The proband successively opened the application detail dialogs and clicked on the stop button to kill the application. He did not ask any questions.

T6 The proband modified the user execution command of the Tomcat web server with included JPetStore on node4. He restarted the application and the system notified him about process loss due to a potential wrong user execution command. He did not ask any questions.

T7 The proband removed the invalid user execution command and restarted the application. He did not ask any questions.

Structural Interview At the begin of the interview, we asked the proband about his perceived satisfaction while using the system. He stated that he liked to work with the system, because of the simplicity of the graph-based visualization and the general design of the GUI. Furthermore, he thinks that the system’s feedback on process loss is a great benefit for users. We proceeded with an informal pluralistic walkthrough and hereby asked the proband for enhancements regarding the graph-based overview and the application detail dialogs. He mentioned that the graph does not seem to scale well with a high number of
8.4. Usability Evaluation

agents and applications. He thinks that it might be hard to quickly overview large software landscape. In terms of the application detail dialogs, he stated that they should be easier to understand, since he was for example uncertain about what he had to do in T2.

8.4.4 Discussion

This section discusses the previously presented results.

T1 The task was supposed to show how rapidly the proband is able to start his work. We did not measure the time, but perceived no problems, while the proband quickly solved the task.

T2 We noted that the proband traversed multiple times through the application detail dialogs during this task. Since this was the first time that the proband used the ExplorViz ADAMMS, we think that this may be the main reason for the initial uncertainty. Nevertheless, a detailed written tutorial might have achieved another result for the learnability.

T3 The proband had no problem during this task. We think that he memorized the interaction for invoking the application detail dialogs and applied this knowledge to the agent’s graph node.

T4 T4 is used to examine the efficiency of a trained user. Compared with T2, we perceived that the proband generally knew what he had to do in order to solve the task. It seemed that he purposely invoked the execution detail dialog to verify the execution command. This is a valid approach. However, he got distracted by a placeholder inside of the user execution command that was ironically supposed to help him. The rest of the required changes, i.e., modifying the aop.xml, were quickly executed. The GUI does not provide many features, but we think that its design facilitates the proband’s efficiency. However, future work should refactor the placeholder and verify our results. A further experiment should also compare the efficiency of probands using the ExplorViz ADAMMS and probands using the manual monitoring procedure.

T5 At this point in the experiment, the proband had seen every detail of the GUI. Therefore, T5 was supposed to show if the proband can memorize a single detail. We think that he intentionally used the green-colored graph nodes to open the currently monitored applications. He seemed to know where the stop button was located. In summary, we think that the GUI design enables the memorization of single features. Nevertheless, a further experiment should evaluate if users generally achieve the same result.

T6 The proband had no problems during this task. He seemed to carefully read the notification pop-ups regarding the resulting process loss.
8. Evaluation

T7  In terms of low-error rate, our system does not actively prevent issues before they happen. Therefore, users might achieve a process loss, if they insert an invalid user execution command. Nevertheless, these issues are recognized and highlighted by means of a pop-up. We think that the system’s pop-up helped the proband to recognize the problem and solve the task. This observation matches with the related statement from the interview. However, future research should examine the proband’s behavior without the necessity of T6, since this might influence the task.

Structural Interview  The structural interview revealed that the proband liked the graph-based visualization of our implementation. However, he criticized the underlying graph layout. Figure 8.1 shows a screenshot that we took during the experiment. We can see that the graph is already hard to read with only four Discovery Agents and twelve applications. Users are able to freely zoom, however an alternative layout might facilitate the readability and therefore could achieve another result. If future work resolves this problem, we think that this visualization is even more pleasant to use. In the interview, the proband also mentioned that he had problems with the application detail dialogs. This matches the behavior that we observed in T2. We think that the current visualization of the user execution command is the root cause for this problem.

Figure 8.1. Screenshot of the proband’s perspective showing the horizontal graph layout
8.4.5 Threats to Validity

An usability experiment with one proband does not provide reasonable results. However, we preliminary mentioned that we sought indications for the overall usability and the realization of the characteristics rather than comparable results. Nevertheless, the latter are required. We suggest that a new experiment with at least 30 probands [Nielsen and Landauer 1993] should be conducted.

The employed proband had already much knowledge about internals and functionalities of ExplorViz and the used Kieker monitoring framework. Therefore, he might be biased and the results might be distorted. Since we conducted this pilot study to also indicate the necessity of further research, we think that this fact can be neglected. We suggest that further experiments consider probands with diverse knowledge on the mentioned tools.

The size and structure of the graph-based visualization depends on the used software landscape. The employed landscape might have been too small to reveal further usability problems. Furthermore, a diverse distribution of applications, i.e., different amounts, might also achieve other results. We therefore suggest to conduct an usability experiment including a larger number of servers and applications.

8.5 Summary

The feasibility experiment shows that our implementation operates correctly. The exemplary applications have been recognized and we could achieve a monitoring through our system. However, the employed recognition strategies were too tightly coupled to the initial setup of the applications. As a result, they are not versatile enough to be used in different environments. Thus, the strategies require more generic approaches. We think that our implementation provides necessary scaffold.

We further conducted a pilot study to briefly examine the usability of our implementation. This included a task-oriented experiment and a structural interview with one proband. The proband was able to solve all tasks and provided feedback on potential enhancements for the GUI. We examined that the GUI seems to implement the usability characteristics of the initial design (see Section 4.4). In summary, both experiments reveal reasonable indications, so that the development of the ExplorViz ADAMS should be continued in the future.
In this chapter, we present related work for the topic of this thesis. To the best of our knowledge, there exist no comparable approaches in the research domain at this time. However, we found several commercial tools with similar application discovery and monitoring management mechanics. These tools are not used to research software visualization, but to show performance metrics of running applications and encompassing systems.

Dynatrace is an application performance monitoring (APM) tool that enables users to gather and analyze performance properties of various kinds of applications. It is developed by the same-named company. An artificial intelligence engine (AI) is supposed to facilitate the detection of performance problems and the localization of root causes. In terms of application discovery, Dynatrace provides an agent component that auto-discovers running processes on a single server node. The related web site states that the setup of the so-called Dynatrace OneAgent requires only the installation of the program on all desired servers. No further configuration is necessary. Based on the agent’s found results, it automatically enables the instrumentation and monitoring for processes. Additionally, new applications in the environment are automatically discovered and processed by the agent. Compared to our approach, the tool supports more types of applications and is therefore not restricted to Java processes. Furthermore, the Dynatrace OneAgent automatically enables the monitoring of found processes, whereas our approach is based on user interaction.

The company ManageEngine distributes another APM tool called Applications Manager. It provides a dashboard that visualizes performance metrics of monitored applications. The software includes an Application Discovery and Dependency Mapping (ADDM) feature that discovers these technologies throughout the IT environment. This feature is essentially realized by using a port scanner. As a result, operators can define ip address ranges and start the scanning. Alternatively, it is possible to schedule a periodic execution of the ADDM. Discovered applications can be instrumented and monitored by using pre-defined default monitoring templates. Compared to our approach, the Applications Manager uses ports to discover running applications. It supports different types of applications and provides pre-defined templates that operators can use as starting point. To monitor Java applications, Applications Manager requires users to manually add Java runtime options to execution command, whereas our system undertakes this task.

1 https://www.dynatrace.com (last retrieval March 24, 2018)
2 https://www.manageengine.com/products/applications_manager (last retrieval March 24, 2018)
9. Related Work

*Instana* is an APM tool developed by same-named company.\(^3\) It is powered by an AI that uses machine learning to monitor applications and predict health for components and encompassing systems. The so-called *Stan AI* (or agent) is comparable to *Dynatrace’s* application discovery approach. Therefore, it also provides automatic discovery of processes and technologies which *Instana* can monitor. *Stan* automatically provides sensors for detected components that know which metrics to collect and which instrumentation is needed. Unfortunately, the related web site\(^3\) does not state how the AI knows which metric and instrumentation are required. Based on the previously mentioned machine learning feature, *Stan* also suggests optimizations and root causes of problems. In comparison, our approach does not employ machine learning to discover running applications. Furthermore, our system does not provide pre-defined monitoring options for found applications and is restricted to Java processes.

\(^3\)https://www.instana.com/application-management (last retrieval March 24, 2018)
Chapter 10

Conclusion and Future Work

In this chapter, we conclude the thesis and give an outlook for potential future work.

10.1 Conclusion

In this thesis, we developed an approach for an application discovery and monitoring management system. We conceptualized a design and subsequently implemented it by means of an extension for the ExplorViz research tool. The resulting ExplorViz ADAMMS is composed of three components, i.e., the backend and frontend extensions as well as the ExplorViz Discovery Agent. The latter can be deployed on multiple servers. It is able to read details of running Java processes inside of the operating system and uses a recognition feature to propose application details, e.g., the name of the application. Furthermore, it can handle process lifecycle and monitoring aspects of found applications. The developed discovery feature is extensible, so that other developers are able to support new kinds of applications and operating systems. The obtained application details are visualized and accessible in a new GUI provided by the ExplorViz frontend extension. The related backend extension acts as central gateway for the communication between frontend and agents.

We evaluated the implementation in terms of feasibility and usability. A feasibility experiment showed the correct functionality of our system. However, we recognized a lack of generic strategies for the recognition feature. The following pilot study included a task-oriented experiment and a structural interview with one proband. The results are not sufficient or comparable, but reveal indications for the usability. As a result, we detected small issues in our implementation, e.g., graph scaling in large software landscapes. Nevertheless, the pilot study indicates a good usability of the overall system. Therefore, the ExplorViz ADAMMS is worthwhile to be further developed and researched. Surprisingly, we could not find any comparable approach in the research domain, while several commercial APM tools already provide similar automatic application discovery and monitoring management systems. This condition also speaks for the necessity of future research in the given context.
10. Conclusion and Future Work

10.2 Future Work

In this section, we discuss suggestions and ideas for future work on the ExplorViz ADAMMS.

**Graphical User Interface** The evaluation participant mentioned his concerns about the scalability of the graph regarding large software landscapes. We agree that this is a problem. One idea to resolve this shortcoming might be to show graph nodes for applications only on demand. An option might be, that users click on an agent node to reveal and hide contained applications. Another approach to address this problem might be to employ different designs for the graph. For example, the Discovery Agents might be visualized as rectangles, which include their discovered applications. We think that later versions of the ExplorViz ADAMMS should consider to implement alternative graph layouts and designs.

The usability experiment additionally revealed that the application detail dialogs require small adjustments. The proband particularly mentioned he was confused about the user execution command inside of the execution dialog. We agree that the current GUI of the user execution command is confusing to new users. Therefore, we suggest to consider an alternative way to depict the attribute.

**Application Discovery** Oracle overhauled the Process API in the Java JDK 9 [Oracle 2017]. Therefore, Java itself now provides methods to read process details of running applications.\(^1\)\(^2\) As a result, the new Process API has the potential to provide a cross-compatible and native mechanism that can be used for application discovery and monitoring management. Thus, external tools such as ps might be obsolete for the ExplorViz ADAMMS. It should be examined, if the new Process API is suitable for our system in terms of applicability and availability on target systems.

One downside of the current implementation is that instances of web servers such as the Apache Tomcat are treated as single applications. Thus, users are unaware about the true applications that the web server hosts. These applications however directly affect the content of the monitoring (aop.xml), since users might want to monitor only a single deployed application. Therefore, we suggest to find a way to detect deployed applications in web servers.

**Monitoring** Because ExplorViz employs the Kieker framework for monitoring, the monitoring configuration for a single application is based on two files. The employed recognition strategies of the evaluation already propose some of these options, but lack a generic applicability. Furthermore, they did not propose monitoring templates, i.e., what should be monitored according to the recognition strategy, for discovered applications. Later versions could propose include directives for the aop.xml of recognized applications.

\(^1\)https://docs.oracle.com/javase/9/docs/api/java/lang/Process.html (last retrieval March 26, 2018)
\(^2\)https://docs.oracle.com/javase/9/docs/api/java/lang/ProcessHandle.html (last retrieval March 26, 2018)
Usability The current implementation of the ExplorViz ADAMMS recognizes problems and suggests potential root causes. We think that small adjustments can actively prevent issues, therefore contribute the low error-rate usability characteristic. For example, the system might compare an initial execution command with the passed user execution command. Depending on the deviation, the system could ask the user if the data is correct, before the operation is eventually executed.
Bibliography


Bibliography


Appendix A

Dockerfiles

A.1 ExplorViz Discovery Agent & Test Applications

FROM openjdk:8-jdk
MAINTAINER Alexander-Krause <akr@informatik.uni-kiel.de>

# Debian Stretch

# 1. Download the explorviz.properties file from the WEB-INF folder
# of the Discovery Agent
# 2. Insert the IP addresses of the hosting server and ExplorViz backend
# 3. Provide the files that are indicated by the COPY instructions below
# 4. docker build -t alexanderkrause/explorviz-discovery-agent
#    -f Dockerfile _Agent .
# 5. docker run -t -d --name target1 -p 8088:8080 -p 8089:8081
#    alexanderkrause/explorviz-discovery-agent
# 6. docker exec target1 /bin/sh
# 7. docker run -it --name target1
#    alexanderkrause/explorviz-discovery-agent /bin/sh

   .5.29/bin/apache-tomcat-8.5.29.tar.gz

RUN apt-get update && apt-get install -y wget dos2unix vim less && rm -rf /var/lib/apt/lists/*
RUN wget -O tomcat.tar.gz "$TOMCAT_TGZ_URL" \

A. Dockerfiles

```bash
&& mkdir explorviz-discovery-agent \\
&& mkdir apache-tomcat \\
&& tar -xzf tomcat.tar.gz -C apache-tomcat --strip-components=1 \\
&& tar -xzf tomcat.tar.gz -C explorviz-discovery-agent --strip-components=1 \\
&& rm apache-tomcat/bin/*.bat \\
&& rm explorviz-discovery-agent/bin/*.bat \\
&& rm tomcat.tar.gz*

RUN wget "$AGENT_TGZ_URL" \\
&& mkdir /explorviz-discovery-agent/webapps/explorviz-discovery-agent \\
&& unzip explorviz-discovery-agent.war -d /explorviz-discovery-agent/webapps/ \\
explorviz-discovery-agent/ \\
&& rm explorviz-discovery-agent.war \\
&& wget "$JPETSTORE_WAR_URL" \\
&& mkdir /apache-tomcat/webapps/jpetstore \\
&& unzip jpetstore.war -d /apache-tomcat/webapps/jpetstore/ \\
&& rm jpetstore.war \\
&& wget "$SAMPLE_APP_TGZ_URL" \\
&& mkdir kiekerSampleApp \\
&& tar -xzf sampleApplication.tar.gz -C kiekerSampleApp \\
&& rm sampleApplication.tar.gz

COPY agent/explorviz.properties /explorviz-discovery-agent/webapps/explorviz-discovery-agent/WEB-INF/classes/explorviz.properties
COPY tomcat/server-agent.xml /explorviz-discovery-agent/conf/server.xml
COPY tomcat/server-tomcat.xml /apache-tomcat/conf/server.xml
COPY agent.sh /agent.sh

RUN chmod +x /agent.sh

RUN dos2unix /explorviz-discovery-agent/webapps/explorviz-discovery-agent/WEB-INF/classes/explorviz.properties \\
&& dos2unix /explorviz-discovery-agent/conf/server.xml \\
&& dos2unix /apache-tomcat/conf/server.xml \\
&& dos2unix /agent.sh

EXPOSE 8080
EXPOSE 8081

CMD ["/agent.sh", ""]
```

Listing A.1. Dockerfile used in the evaluation to run agent and test applications
# A.2 ExplorViz Frontend & Backend with Extensions

FROM tomcat:8.5-alpine

MAINTAINER Alexander-Krause <akr@informatik.uni-kiel.de>

# 1. Provide the files that are indicated by the COPY instructions below
# 2. docker build -t alexanderkrause/explorviz -f Dockerfile_ExplorViz .
# 3. docker run -t -d --name explorviz -p 8090:8080 -p 10133:10133 alexanderkrause/explorviz
# 4. docker exec explorviz /bin/sh
# 5. docker run -it --name explorviz alexanderkrause/target-system /bin/sh

ENV TOMCAT_TGZ_URL=http://ftp.halifax.rwth-aachen.de/apache/tomcat/tomcat-8/v8.5.29/bin/apache-tomcat-8.5.29.tar.gz


RUN apk --no-cache add wget dos2unix vim less

WORKDIR webapps

RUN wget "$BACKEND_WAR_URL" \
&& mkdir explorviz-backend && unzip explorviz-backend.war -d explorviz-backend/ \
&& rm explorviz-backend.war \
&& wget "$FRONTEND_TGZ_URL" \
&& mkdir explorviz-backend \
&& tar -xzf explorviz-backend.tar.gz -C explorviz-backend \
&& rm explorviz-backend.tar.gz

COPY backend/explorviz.properties explorviz-backend/WEB-INF/classes/explorviz.properties
COPY tomcat/server-backend.xml /usr/local/tomcat/conf/server.xml

RUN dos2unix explorviz-backend/WEB-INF/classes/explorviz.properties \
&& dos2unix /usr/local/tomcat/conf/server.xml

EXPOSE 8080
EXPOSE 10133

Listing A.2. Dockerfile used in the evaluation to run the extended ExplorViz frontend and backend
Appendix B

Recognition Strategies

B.1 Tomcat Strategy

```java
public class TomcatStrategy implements DiscoveryStrategy {

    @Override
    public boolean isDesiredApplication(final Procezz newProcezz) {
        boolean doesContainTomcatName = false;

        if (newProcezz.getOsExecutionCommand() != null) {
            doesContainTomcatName = newProcezz.getOsExecutionCommand().toLowerCase(Locale.ENGLISH).contains("tomcat");
        }

        return doesContainTomcatName;
    }

    @Override
    public void detectAndSetName(final Procezz newProcezz) {
        if (isDesiredApplication(newProcezz)) {
            newProcezz.setName("Tomcat Web Server");
        }
    }

    @Override
    public void detectAndSetProposedExecCMD(final Procezz newProcezz) {
        if (isDesiredApplication(newProcezz)) {
            newProcezz.setProposedExecutionCommand(DiscoveryStrategyFactory.USE_OS_FLAG);
        }
    }
}
```
B. Recognition Strategies

```java
@override
public void detectAndSetProperties(final Procezz newProcezz) {
    // TODO
}
```

Listing B.1. Strategy used in the evaluation to recognize Tomcat web servers

### B.2 KiekerSampleApp Strategy

```java
public class KiekerSampleAppStrategy implements DiscoveryStrategy {
    private static final String EXPLORVIZ_MODEL_ID_FLAG = "-Dexplorviz.agent.model.id=";

    @Override
    public boolean isDesiredApplication(final Procezz newProcezz) {
        boolean doesContainSampleAppJar = false;
        if (newProcezz.getOsExecutionCommand() != null) {
            doesContainSampleAppJar = newProcezz.getOsExecutionCommand().toLowerCase(Locale.ENGLISH).contains("sampleapplication");
        }
        return doesContainSampleAppJar;
    }

    @Override
    public boolean applyEntireStrategy(final Procezz newProcezz) {
        final boolean isDesiredApplication = isDesiredApplication(newProcezz);
        if (isDesiredApplication) {
            detectAndSetName(newProcezz);
            detectAndSetProposedExecCMD(newProcezz);
        }
        return isDesiredApplication;
    }
}
```
Listing B.2. Strategy used in the evaluation to recognize the KiekerSampleApp

```java
@override
deflectAndSetName(final Procezz newProcezz) {
    if (isDesiredApplication(newProcezz)) {
        newProcezz.setName("KiekerSampleApp");
    }
}

@override
deflectAndSetProposedExecCMD(final Procezz newProcezz) {
    if (!isDesiredApplication(newProcezz)) {
        return;
    }

    final String osExecCmd = newProcezz.getOsExecutionCommand();
    final String workingDir = newProcezz.getWorkingDirectory();

    if (osExecCmd.contains(EXPLORVIZ_MODEL_ID_FLAG)) {
        // was already restarted by agent, probably correct os exec path
        newProcezz.setProposedExecutionCommand(DiscoveryStrategyFactory.USE_OS_FLAG);
    } else if (osExecCmd != null && workingDir != null) {
        final String delimiter = "-jar ";

        final String[] splittedAtJarFlag = osExecCmd.split(delimiter, 2);

        final String proposedExecCMD = splittedAtJarFlag[0] + delimiter
            + workingDir.trim() + "/" + splittedAtJarFlag[1].trim();

        newProcezz.setProposedExecutionCommand(proposedExecCMD);
    }
}

@override
deflectAndSetProperties(final Procezz newProcezz) {
    // nothing to do
}
```