Enabling Dynamic Analysis and Software Visualization in Continuous Integration Platforms

Bachelor's Thesis

Jan Erik Petersen

April 10, 2020

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Alexander Krause, M.Sc.

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, 10. April 2020

Abstract

Software development is moving more and more towards a continuous process, where fast and frequent deployments are a requirement. This is often realized with continuous integration. At the same time, the complexity of applications is rising. Extensive testing of applications by hand is expensive and not feasible, especially when using continuous deployment. Hence the need for automated software analysis arises. Static analysis is well-proven but being based on the source code alone it can only gain superficial knowledge. To get desired insights into application runtime behavior, dynamic analysis must be performed, but that is complex to implement in a fully automated build process, where no user is available to interact with the system. In this thesis we conceptualize and implement an approach that offers dynamic analysis for applications from continuous integration builds. Specifically, it allows developers to visualize each build of the software as a 3D model by utilizing the ExplorViz live trace visualization software.

Contents

Chapter 1

Introduction

Static software analysis (SSA), an established method to find common mistakes in software and estimate software quality, is based solely on statically analyzing the source code of the software. As such, it is fast and easy to use and therefore often utilized in automated Continuous Integration (CI), which is the process of automatically building and testing individual changes in software, to detect likely bugs. However, with increasingly large software systems, it becomes important to understand the composition and interaction of separate software components. Static analysis cannot make any statements about runtime behavior.

By monitoring a running software that is performing real work, information about the software's runtime behavior can be obtained. Kieker is a dynamic analysis tool that offers such functionality for Java applications by monitoring application behavior at the JVM-level [van Hoorn et al. 2012]. The collected information is stored in records, which can be saved for Kieker-based analysis methods or sent to another applications.

The live trace visualization tool ExplorViz[1] acts as a consumer of these records to visualize the software system and its internal communication [Fittkau et al. 2017]. ExplorViz is designed to be run as a standalone application alongside the monitored software system.

1.1 Motivation

Each time a developer wants to perform dynamic analysis on an application they have to start ExplorViz, configure Kieker to submit records to that ExplorViz instance, run the target application manually with Kieker and generate load by some means. Oftentimes a developer wants to isolate the cause of a performance regression, meaning they have to go through most steps again for a multitude of versions of the application. This is a tedious process, hindering widespread adoption of dynamic analysis methods. It also means ExplorViz cannot be used in automated software builds.

In this thesis, we develop and evaluate ExplorViz as a Service(EaaS), a Software as a Service-platform (SaaS), which is a distribution model where software is continuously made available as a hosted service instead of running it on-premise. SaaS applications have the advantage that multiple customers can be served from a single, shared service [Sengupta and Roychoudhury 2011]. EaaS collects build artifacts from automated software

[1] `https://www.explorviz.net`, accessed 2020-02-22

builds. Users can navigate through these builds and start ExplorViz instances for specific builds they want to visualize at any time.

This makes dynamic analysis more feasible and allows for quick insights into runtime behavior of an application, without having to manually load-test the application every time analysis is desired.

1.2 Goals

The primary objective of this thesis is to enable developers to make use of dynamic analysis in continuous integration builds, specifically to visualize the software using ExplorViz. This means having to overcome the problem that ExplorViz is a live inspection tool that can only visualize applications running at the same time. This objective is fulfilled by developing a server software that fits the requirements defined hereafter.

G1: Service to Launch ExplorViz On-Demand

G1.1: Collect Build Artifacts

Automated builds must be able to submit build artifacts to the server where they are stored so users can analyze them at any time. The submissions must include a means to generate load on the software so interesting analysis can be performed. The interface to submit builds must be designed in a way that it is possible to use it from a most commonly used build environments. Build artifacts must be stored permanently in a way that uses disk space efficiently.

G1.2: Start ExplorViz Instances from Web Interface

The server must offer a web-based user interface to browse through submitted build artifacts. Users can choose to dynamically analyze any build from the list at any time. Upon starting a build the server should automatically launch an instance of ExplorViz, then run the application and load generation and finally provide the user with an address to access the ExplorViz live visualization on.

G2: Submit Builds from Continuous Integration

G2.1: GitHub Actions

GitHub Actions is a workflow automation service recently launched on the code hosting platform GitHub. It should be possible to send build artifacts to an ExplorViz as a Service server from this service in a streamlined way. Therefore we need to develop a tool that

automates packaging the build artifact and uploading it to the server while only needing minimal configuration from the developer.

G2.2: Script for Other CI Services

The aforementioned tool should be reusable for a wide range of CI services. This includes self-hosted CI servers and public continuous integration services, which are of high importance for many projects who might be unable to afford a build server or lack the necessary time or knowledge to maintain it. The solution should specifically work with Travis CI, a public continuous integration service with widespread use in open source projects.

G3: Evaluation

A sample application and build procedure showcasing the developed software must be put together, which includes load generation for the application and submitting builds to the server. We present a general usage scenario for the software and test its resource consumption. We will evaluate how issue-tolerant the system is and how well errors can be diagnosed by deliberately injecting faults into the process in a number of different scenarios.

1.3 Document Structure

After this introduction, we begin by introducing the technology and software used in this thesis in Chapter 2 and explaining their purpose and use. Thereafter Chapter 3 covers the approach how the aforementioned goals will be accomplished. In Chapter 4 we cover the technical implementation of the approach in depth. We then evaluate this implementation and discuss the results in Chapter 5. After we look at related work in Chapter 6 we draw a conclusion and discuss possible future work in Chapter 7.

Chapter 2

Foundations and Technologies

2.1 Application Framework Spring Boot

Spring Boot[1] is an application framework for Java, built on top of the Spring framework. Spring Boot simplifies Spring configuration by letting developers use Java annotations to configure the framework instead of separate XML files. Spring is one of the biggest frameworks for application development in the Java ecosystem.

Spring makes use of the Inversion-of-Control (IoC) pattern to streamline application development. Through individually selectable modules many commonly needed components for application development can be used.

We use Spring Boot as framework when developing the ExplorViz as a Service application server. We use the dependency injection (DI) built into Spring, a technique to supply all dependencies to a class that it needs to perform its task. Furthermore we make use of Spring Data JPA, a module that builds on top of Jakarta Persistence (formerly Java Persistence API, JPA) to provide automated implementations of so-called repositories that allow to query for relational data as Java objects. The data that we store includes users, projects and metadata of builds.

JPA by itself is only a set of interface definitions. Hibernate[2] is an object-relational mapping (ORM) framework implementing JPA that Spring Data JPA uses by default. Hibernate maps between Java objects and tables in relational databases, by automatically generating the SQL code to query and update entries in the database.

For web applications specifically, Spring offers the Web model-view-controller (MVC) framework[3]. It allows for writing RESTful APIs by mapping URLs to methods in a classes marked as controller. Such controllers are used to implement the build submission API. Furthermore Spring Security is used as it provides necessary authentication and other security-related functionality like password hashing.

[1] `https://spring.io/projects/spring-boot`, accessed 2020-02-22

[2] `https://hibernate.org/`, accessed 2020-03-05

[3] `https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html`, accessed 2020-03-05

2.2 Relational Database H2

H2[4] is a relational database written entirely in Java. Like many databases, it uses the well-known SQL language for queries. In-memory databases like H2 can be embedded into applications and profit from having no I/O overhead. This has lead to a widespread use of such database systems [Soares and Preguiça 2018].

We use H2 to persist our application data, by using it as the storage for Spring Data JPA. Therefore we don't need to run a separate database server, simplifying deployment. Benchmarks[5] carried out by H2 developers conclude that it faster than a separated SQL-server like MySQL, making it a good choice to save resources.

2.3 Web Application Framework Vaadin

Vaadin[6] is a UI framework for building web interfaces in Java. Vaadin applications are single-page applications (SPA), a modern approach to building web applications where the page isn't reloaded completely every time the user navigates to another page, but instead only the changed contents are transmitted between the server and the browser, in order to decrease loading times and give the application a more native feel.

We use Vaadin to create the web interface of the server, where users can manage their projects, look at the list of collected build artifacts and start visualizations.

As shown in Figure 2.1, UI views are composed entirely from Java code, in a similar manner to Javas GUI framework Swing. Each component, like a button or a text-field, is a Java object. Likewise, event listeners for frontend interactions are written as server-side lambdas with Vaadin handling all of the synchronization between frontend and backend.

Vaadin simplifies development by only having to work on a code base when introducing new features or making changes. That also means validation logic doesn't have to be duplicated in both frontend and backend code, leading to less bugs. This makes Vaadin an ideal choice for a bachelor's thesis, removing the need to adopt a frontend framework in addition to the backend. Vaadin also integrates well with the dependency injection and request routing of Spring by processing all paths that aren't registered to any Spring MVC controller.

On the frontend side, Vaadin components translate into functional, reusable web components. This removes the need to write any frontend code, with the exception of stylesheets to adjust sizing and layout of the components.

[4] `https://www.h2database.com/html/main.html`, accessed 2020-03-05

[5] `https://www.h2database.com/html/performance.html`, accessed 2020-03-05

[6] `https://vaadin.com/`, accessed 2020-02-22

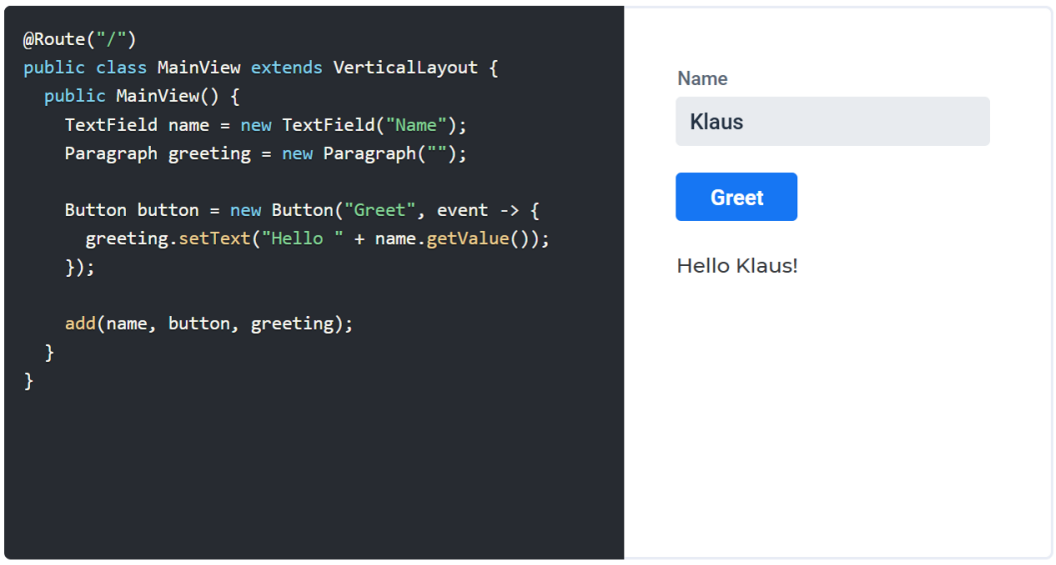

Figure 2.1. Building views from individual components in Java code

2.4 Monitoring Framework Kieker

Kieker[7] is a tool to monitor the runtime behavior of running software. It uses *monitoring probes* instrumenting the software to generate monitoring records. These records can be persisted or transferred to external applications, which in turn can use the contained information to perform dynamic analysis on the monitored application. Applications include performance evaluation, problem localization and reverse engineering [van Hoorn et al. 2012].

Kieker installs monitoring probes into the application using aspect-oriented programming frameworks. Which probes are to be installed and which classes are covered is configured by the developer in a file called `aop.xml`. Additionally, Kieker reads a key-value configuration file `kieker.monitoring.properties`, containing metadata and the writer configuration. The writer is responsible for processing the generated monitoring probes, e.g. by writing them into a file or sending them to another program over the network [Kieker Project 2013].

Kieker is used to generate monitoring records of the application in question and send them to ExplorViz.

[7] `http://kieker-monitoring.net/`, accessed 2020-02-22

2.5 ExplorViz

ExplorViz[8] is an application to create software landscapes visualizations from Kieker monitoring records. It uses a layered approach, allowing a user to observe the software system from an overview level (Figure 2.2) down to the application level (Figure 2.3). It consumes the monitoring records generated by Kieker and generates a visualization of the software, remotely resembling a city [Fittkau et al. 2017].

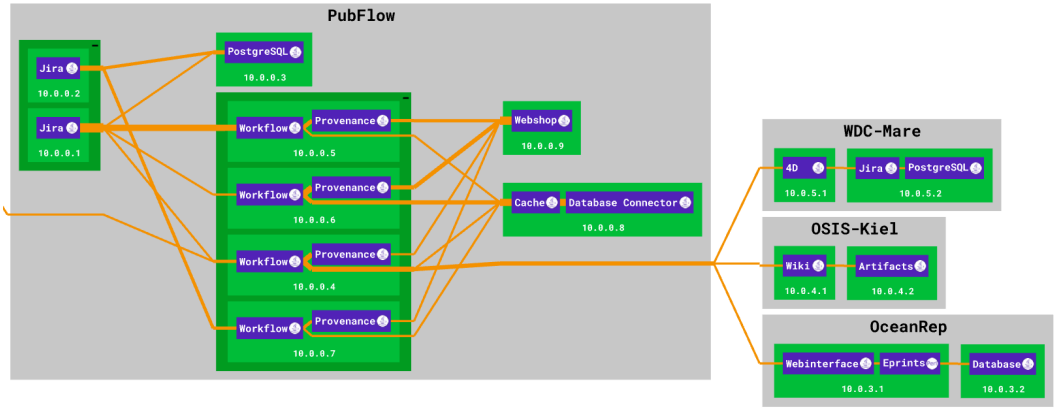

Figure 2.2. Landscape view in ExplorViz

Since its inception, ExplorViz has gone through significant changes and, as a research project, has been used and advanced by many researchers. To better support this development style, ExplorViz has been rewritten in a microservice-based approach (see Figure 2.4). This approach simplifies extending the software and allows for higher flexibility in adapting to different usage scenarios [Zirkelbach et al. 2019].

To compose these microservice into a full ExplorViz instance, Docker Compose[9] is used, running each microservice in its own Docker container. ExplorViz extensions are separated microservices and can be enabled by adding them to the `docker-compose.yml` file. ExplorViz visualization are presented in a web interface that can be accessed through a regular web browser, but require the presence of a 3D graphics adapter on the client.

ExplorViz is used to visualize the applications and provide the dynamic analysis capabilities.

[8] `https://www.explorviz.net/`, accessed 2020-02-22
[9] `https://docs.docker.com/compose/`, accessed 2020-02-22

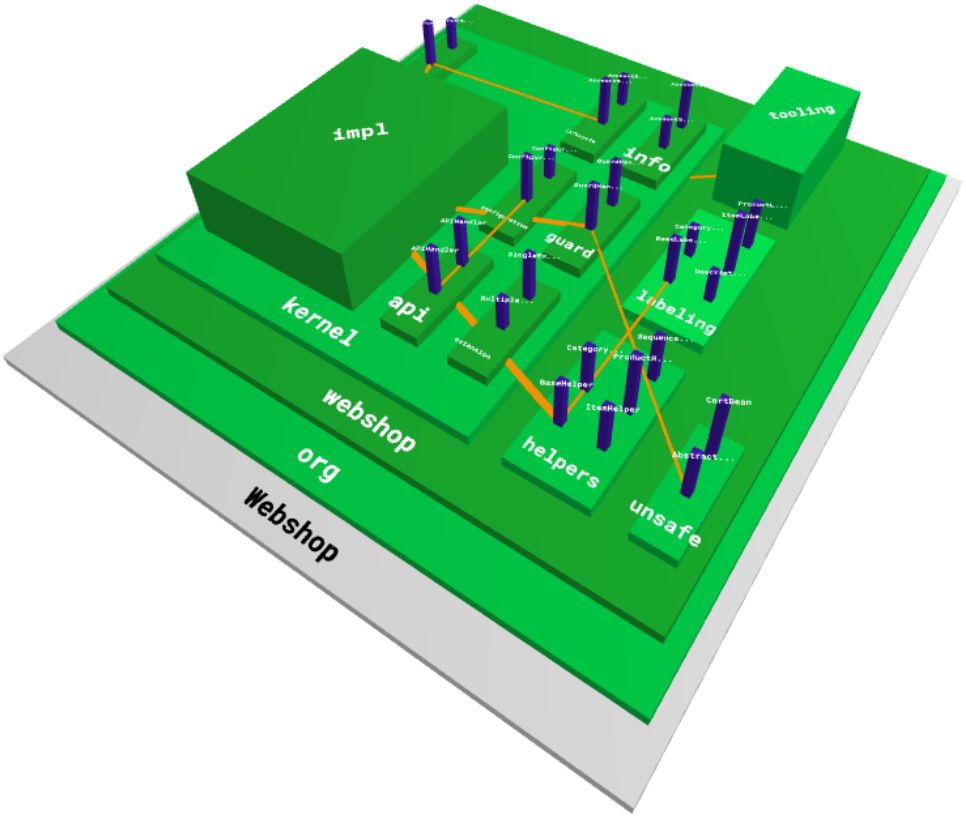

Figure 2.3. Application view in ExplorViz

2.6 Container Platform Docker

Docker[10] is a tool to simplify application deployment. Applications are executed in containers, isolated from the rest of the system, bringing their own execution environment [Rad et al. 2017], which is a stripped-down version of an operating system, including only the parts necessary for the system to run in a container and for the application to function properly. By controlling the entire environment of the application, the developer can make sure the application works the same for every user and doesn't have to invest as much time in testing different system configurations.

Docker provides an API to control all container-related tasks like loading images, starting new containers In theory, the use of Docker also opens up the future possibility of using remote machines to deploy both ExplorViz and the tested applications.

[10] `https://www.docker.com/`, accessed 2020-02-22

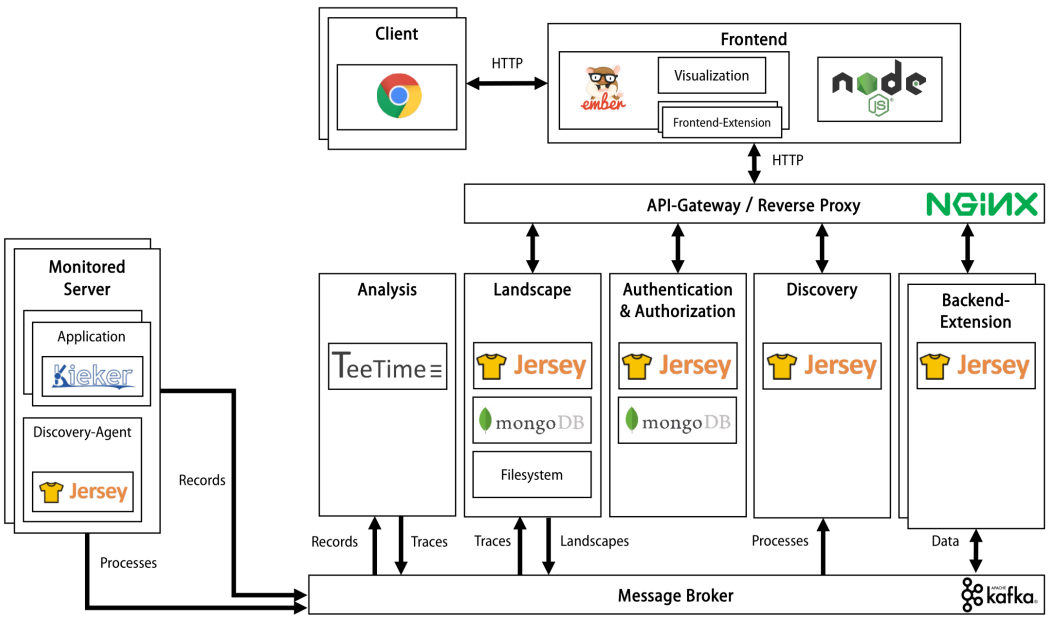

Figure 2.4. Microservice-based architecture of ExplorViz

ExplorViz instances are run inside Docker containers and Docker images are also be used to package the applications builds for visualization. Furthermore, the ExplorViz as a Service server itself also runs in a Docker container to simplify its installation, improve security and make sure it runs correctly on all operating systems.

2.6.1 API client docker-java

To control all of the used Docker containers from within Java, the docker-java[11] library is used, allowing the application server to send commands to the API of the Docker daemon through a streamlined Java interface.

2.6.2 Docker Compose

Docker Compose[12] simplifies management of applications that are composed from multiple containers. It allows the developer to write a single `docker-compose.yml` file defining all containers their application is made of. Then, the `docker-compose` command-line program can be used to configure and start all of these containers at once using only a single command. It does this by communicating with the Docker API to first set up shared

[11] `https://github.com/docker-java/docker-java`, accessed 2020-02-22
[12] `https://www.docker.com/`, accessed 2020-02-22

resources for the entire compose file, like networking, then creating and starting each container individually, just like you can do by hand using the `docker` command line program.

This approach is most useful for applications with a microservice architecture like ExplorViz, as they are made up of many containers that run one microservice each. Furthermore `docker-compose.yml` files are helpful to system operators by acting as a permanent storage for the configuration for each container, as Docker by itself only stores container configuration for as long as the container runs. There is no way to modify the configuration of a container after it has been started - it needs to be stopped and than started again providing all options.

Official ExplorViz versions are released by publishing such `docker-compose.yml` definition files. These are also be used by the ExplorViz as a Service server to run ExplorViz instances in the officially supported way. Furthermore Docker Compose is also used to run the ExplorViz as a Service server itself, as Docker Compose files can act as a way of storing configuration.

2.7 Continuous Integration

Continuous Integration (CI) is the process of continuously building new versions of a software whenever a change is made available to the source control system. Oftentimes, this includes running automated tests to verify the application is working correctly in its current state. Use of CI in projects helps them release more often and be more confident about changes not breaking the build. Of the most popular projects on GitHub, 70% make use of CI and further growth is expected [Hilton et al. 2016].

CI enables developers to immediately identify code changes that break previously working functionality, called regressions. Making sure the application can be built and is working at all times is critically important to be able to release a new version whenever needed. For example, when a security vulnerability is found, a new version of the application needs to be released as soon as possible to minimize potential damage caused by the vulnerability.

Furthermore, it is advisable to analyze the source code and artifacts in continuous integration builds. Static analysis is already widely used through various tools to track code quality metrics over time and ultimately improve software quality [Bolduc 2016]. Examples include CheckStyle, which verifies readability of the source code, or SpotBugs which attempts to identify common programming issues in the code. It is common to generate metrics from the number of issues found, as seen in Figure 2.5. Dynamic analysis is a major point of interest to obtain more detailed metrics by instrumenting the applications runtime behavior, but is harder to utilize in continuous integration.

ExplorViz as a Service should be usable from a variety of different CI services, of which two have been selected specifically based on their popularity and the fact they allow the use of Docker in builds. In the evaluation of our implementation, we are testing with these

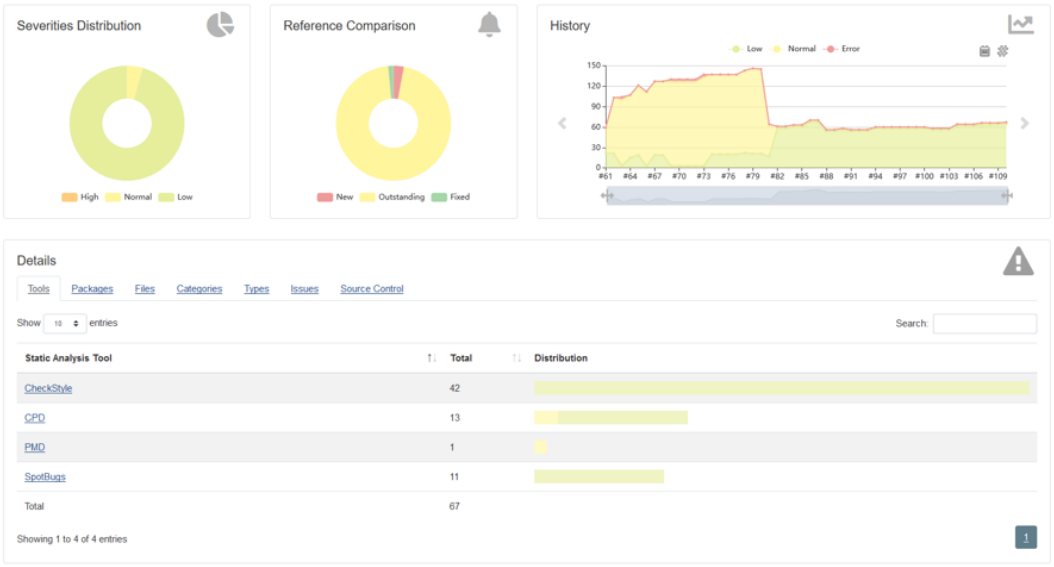

Figure 2.5. Graphs showing static analysis results over time

services. All of the tools we develop also use continuous integration with static analysis throughout the development phase to keep track of the software quality as development progresses.

2.7.1 GitHub Actions

Github Actions[13] is a recent addition to GitHubs offerings for open-source projects and business users. It is a workflow automation service, allowing maintainers to automatically run tasks in a virtual machine whenever an event happens in the repository. Events include updated code pushed by a developer, a comment written on an issue or a new pull request being handed it.

While GitHub Actions can be utilized for many purposes, we only look at it as a CI service in our evaluation.

2.7.2 Travis CI Build Service

Travis CI[14] is a well-known public build service. It is free to use for open source projects. Travis builds are configured by build scripts placed in the same repository as the code.

[13] `https://github.com/features/actions`, accessed 2020-02-22
[14] `https://travis-ci.org/`, accessed 2020-02-22

Build scripts are written in a declarative and procedural way, listing the steps, which can be thought of as commands, to build and test the software.

Being used by the ExplorViz project itself, it is familiar to ExplorViz developers, making it a good choice as a reference for a public continuous integration service.

2.8 Spring PetClinic

Spring PetClinic is a sample Spring Boot application implementing a simple website for an exemplary pet clinic [Spring Project 2020]. It is developed by the Spring project and is intended to showcase some of the features of the Spring Boot framework. The application is complex enough to produce interesting visualization in ExplorViz.

We use Spring PetClinic by utilizing it as the application to be visualized in our evaluation of EaaS. It is an ideal candidate for such a task because it is fully open-source, so we can modify it for various tests. As a long-standing sample application for Spring it is well-tested, so we can focus on the evaluation of our own implementation, avoiding problems that could occur in an untested self-made application.

Chapter 3

Approach

3.1 Deployment

Before we discuss how we build the ExplorViz as a Service platform, we want to clarify our approach for its deployment. Understanding the way EaaS is set up is critical to be able to follow development choices. Our overall goal is to implement a platform that continually provides the service of collecting build artifacts and offering visualizations for builds on-demand.

Therefore, our approach is to use EaaS in a deployment as shown in Figure 3.1, where EaaS is installed on a server, permanently running to accept build artifacts from continuous integration services and visualization requests from users.

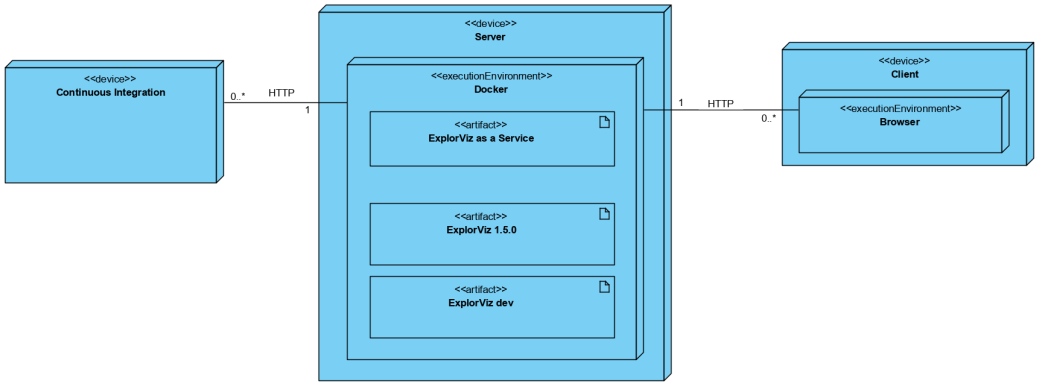

Figure 3.1. Deployment diagram for ExplorViz as a Service

The server software is written in Java as a Spring Boot application and utilizes a number of technologies that are explained in-depth in the following chapters. The software is deployed on the server inside a Docker container. This has the advantage that we can be sure our server will run everywhere Docker is available, without the need to install dependencies or handle operating system compatibility. We also use a Docker Compose file that our server can be started from.

3.2 Build Packaging

One of our main goals is to open the possibility of performing dynamic analysis on a build artifact that was build in a continuous integration environment at any point in time, even long after they were built and even if the application has undergone significant changes in the meantime. To visualize the build artifacts, they need to be run alongside an instance of ExplorViz. This presents a problem, because applications have dependencies and may require that their runtime environment is set up in a specific way to function properly. Recreating such an environment manually whenever we want to visualize a build is tedious and error-prone, as we cannot cover all parameters of the environment.

3.2.1 Docker Images

Our solution is to package build artifacts in Docker images, which include the entire operating system environment. This means that in the CI build, after we have produced the artifact in question, we build a Docker image from a developer-supplied `Dockerfile`. This file must be kept updated by the developer as the applications environment requirements change. As a result, for each build there will always be a compatible environment to run the application in.

Kieker Monitoring To visualize an application in ExplorViz, it is instrumented with the Kieker monitoring tool to produce monitoring records and write them to a target location. Therefore we need a compatible version of Kieker when running visualizations. The weaver configuration `aop.xml` to specify the classes that should be instrumented and which probes will be installed must be provided by the application developer as it cannot be inferred automatically. Furthermore Kieker reads a `kieker.monitoring.properties` configuration file where some metadata like the applications name and the record writer is configured.

Handling all of this complexity just-in-time when we are about to start a visualization is not a feasible approach as it would require that we modify or extend images for each run. We instead solve these requirements by including a copy of Kieker and the configuration files inside the image. Most importantly, Kiekers writer must point to the ExplorViz monitoring ingress that will be present during the visualization. The correct writer configuration results from the way we run ExplorViz instances and is discussed in Section 3.4.

Load Generation In dynamic analysis we look at the runtime behavior of an application. If the application is idling, no interesting analysis can be performed. Therefore, the application must be made to perform some work while it is instrumented. For example, the developer could choose to add a special mode where it keeps itself busy. The more general approach, which applies to web servers like Spring applications, is to include a load generation script inside the same image that will continually perform requests to keep the application busy.

While running both the application and load generation processes in the same container is bad practice in Docker, where the rule of thumb is to use one container per application, it is acceptable in this case — the containers are only run temporarily for the purpose of analysis and do not need to meet the standards of a production deployment. Allowing applications to consist of multiple Docker images would drastically increase the complexity of our approach.

3.2.2 Base Images

On the server, we permanently keep the images of all builds, so we can use them later. The images have to contain all of the dependencies necessary for the application. Therefore they need to include an entire operating system and a Java runtime environment and as a result become much larger than the build artifact alone.

We remedy this issue by utilizing Dockers layer caching. Each `Dockerfile` command, like `COPY`, creates a new layer that only contains the changes compared to the previous layer. Layers that contain exactly the same files get the same ID. This way they can be reused among multiple images. Only once the files start to differ do the layers diverge into separate paths. This can be seen in Figure 3.2. The main build artifact, called *application.jar* is different between two builds, but all previous layers, including base images loaded with the `FROM` command, are exactly the same and do not take up additional space for the second image. If this form of deduplication is used correctly, significant storage savings can be achieved [Zhao et al. 2019].

To make sure all builds share as many layers as possible and in turn minimize the storage requirement each build image adds, we introduce a common base image that already incorporates most of the tools and steps that build images need. Specifically, the images are based on OpenJDK images to provide a Java runtime environment and they contain a copy of Kieker and the `kieker.monitoring.properties` configuration file.

This way, the individual build images can share almost all of the files in common layers. Only the applications own files remain in separate layers, which lowers the effective size down to the build artifact itself again, eliminating the overhead our Docker packaging approach has. Another advantage of the base images is that it becomes easier for developers to build their images, because they don't need to set up Kieker themselves.

3.2.3 Build Submission

To run the build images on our EaaS server later we need to be have them available on the Docker host we run visualizations from. We accomplish this by saving the images in the local Docker image library. In our approach we utilize Dockers *save* and *load* commands to transmit the Docker image built in the CI environment to the EaaS server. Once an image is built, Docker keeps it in its local image library. The *save* command allows exporting an image with all of its layers into a single archive file, while *load* imports the image from such an archive.

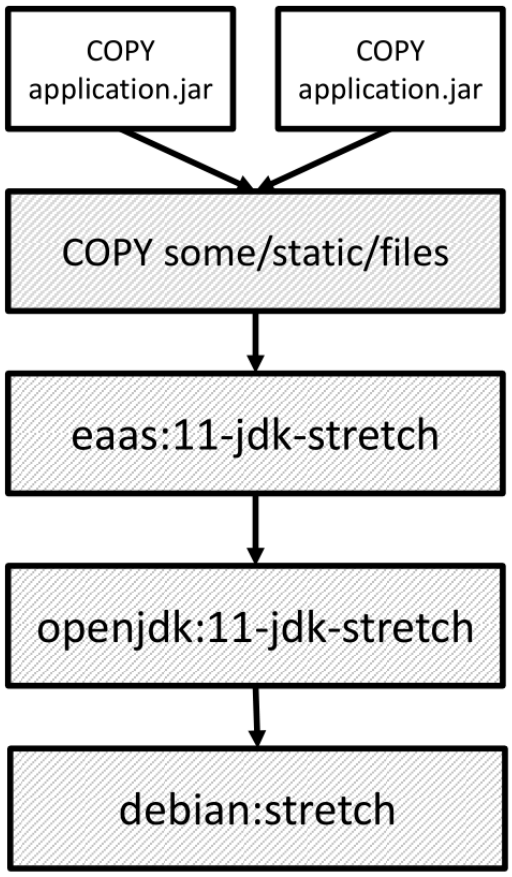

Figure 3.2. Schematic showcasing how Docker images make use of layers.

Submission API Our server should offer a simple API[1] over the HTTP protocol to receive new builds. HTTP is widespread and therefore tools to send data over this protocol are readily available. From the CI environment we need to be able to communicate with the Internet, so we can send the exported archive file to the servers API. The server then saves it into its Docker image library and adds a build entry to its database.

Along with the archive the API should accept some metadata. Because projects might begin utilizing ExplorViz as a Service only after some time into their development, build numbers won't be synchronized if EaaS assigns numbers on its own. This leads to a confusing disconnect between the CI service and the EaaS interface. It is solved by letting the client set a name when submitting builds, which EaaS will use when displaying the

[1] Application programming interface

build in its interface.

Access Control To prevent attackers from uploading their own images to the server, the API implements a form of access control with tokens. All requests to the API must contain a secret string. If it is missing or incorrect, the server discards the request and won't add the new build.

Users should be able to generate these secret tokens in the EaaS interface. They must be available in the CI build process so they can be specified when sending the upload request to EaaS. To not expose the tokens to the public, especially when dealing with open source projects, where a secret value cannot be hidden inside the public source code repository, the CI services GitHub Actions and Travis CI offer to store *Secrets*, which are variables that are only made available to the commands running in the build.

3.3 User Interface

To let users access the list of collected builds, we integrate a web interface into our application based on the Vaadin web framework. This allows us to create a single-page application from reusable components, without having to implement the frontend ourselves. Instead, it is automatically generated by Vaadin during the build process of the server and included inside the servers executable Jar file.

Structure The web interface will provide basic functionality for a multi-project platform. Multiple users can use the platform and create their own projects. There is a page to display all projects that are hosted on the server.

When builds are submitted to the server they are added to a specific project. That way, completely independent projects can be hosted on the same EaaS instance. Each project has its own list of builds. Each build entry in turn features a button to start visualizing it in ExplorViz.

Access Control While some information, like the list of the saved builds including their names and upload date, will be publicly readable by anyone who opens the web interface, most actions require being logged in with a user account. Upon first start, the server creates a user with known default credentials that the administrator can use to gain full access to the web interface.

To allow multiple users to access the system without all sharing the same account with administrator rights, we implement a rudimentary permission system. Administrators can create new users and determine which permissions they have. For example, one such permission is *Can create new projects*. With the *Can manage users* permission, users become administrators and can themselves create new users. Users can change their password

when they're logged in and additionally administrators can reset the passwords of other users.

The owner of a project can create, display and delete secrets for the build submission API. He also has the option to hide the project from the public altogether, meaning only himself and users with the *Can see hidden projects* permission can access the project. Users with the *Can manage all projects* can modify these settings for all project, not only for the ones they own.

3.4 Running Visualizations

The central task of the EaaS server is to run visualizations with ExplorViz for the builds it collected whenever a user requests to do so. That means we need a running instance of ExplorViz and then start the Docker image corresponding to the requested build to let it send monitoring records to ExplorViz. No manual setup of ExplorViz should be necessary. The server should handle starting ExplorViz automatically without user involvement. Our approach how we deal with ExplorViz instances is lined out in the following chapters.

3.4.1 Docker Integration

Both to run ExplorViz and to store build images as described in Section 3.2.3, the EaaS server needs administrative access to Docker. Each Docker daemon provides an API, which we call an endpoint, that programs can use to perform Docker commands. We will utilize this API from our server with the `docker-java` library. This library can be configured to contact a specific endpoint. However, to keep our approach simple for now, we let the library fall back to obtaining the configuration for a single endpoint from environment variables, most notably `DOCKER_HOST`, which defaults to the local Docker daemon communication socket `/var/run/docker.sock`. This socket is also made available inside the servers container by our Docker Compose file, where server operators can also adjust the variables in case they want to use another endpoint.

Furthermore, to run ExplorViz from the Docker Compose files the project releases, we need an implementation of Docker Compose. Unfortunately no such implementation exists as a Java library, so we resort to including the `docker-compose` command line utility in the Docker image of our server and using it as an external process. As the environment is passed through to child processes, this also works with our approach to configure the endpoint through environment variables.

3.4.2 Docker Compose Definition

In the upstream Docker Compose files, ExplorViz exposes a number of ports to the host system. The most important ones that are required is the port to access the web frontend

on and the port where monitoring records from the instrumented application are accepted that are used to generate the visualization.

Our approach is to start a new, separate instance of ExplorViz for every single visualization started by the user. ExplorViz adopted a microservice architecture, where the software is split into many smaller application, so-called microservices [Zirkelbach et al. 2019]. Several of them expose some ports to accept data from the outside world. If we started multiple visualizations from the same Docker Compose file, the used ports would collide and the instance would fail to start. Hence we start with the `docker-compose.yml` files provided by ExplorViz but modify them according to our needs. Additionally, some options are replaced with placeholder strings that are replaced on-the-fly when starting new visualizations. That way, a single `docker-compose.yml` file effectively acts as a template we can run any number of instances from.

- Remove the discovery-service that does not apply for our use-case.
- Remove all volumes because we do not want to keep persistent data.
- Make the ExplorViz frontend port a variable that we can choose dynamically for each instance.
- Do not expose any other ports to avoid collisions.
- Add a unique container label to all services. This is required for the *Traefik*[2] reverse proxy utilized by ExplorViz — without it, *Traefik* is unable to differentiate different instances and the frontend won't be accessible

Docker Compose assigns each instance a project name. Usually, this is the name of the current working directory the user is running the `docker-compose` command from. However, as our Docker Compose files are generated on-the-fly and never saved to disk, we generate a unique project name ourself to avoid name collisions.

Running the Build Image Lastly, the build image of the application we wish to visualize has to be run as well. The difficulty herein is to configure the Kieker monitoring tool to send records to the right ExplorViz instance. Our solution is to not expose the ExplorViz analysis port to the host at all and instead run the build image from the same Docker Compose file. To do so we just add another service to the `docker-compose.yml` file and specify the image as a variable, which will be filled with the image ID of the build the users wants to visualize.

From within the Docker Compose environment the address of the ExplorViz analysis port is always the same. The hostname is the name of the ExplorViz analysis microservice, `analysis-service`, and the port is the default of `10133`. Therefore we can set the Kieker writer configuration to this address in our base images and do not need to adjust it when running the visualization.

[2] `https://docs.traefik.io/`, accessed 2020-04-09

Frontend Port Selection For each instance, we need to select a unique, unoccupied port where the user can access the ExplorViz frontend corresponding to the build he chose to run. We implement this by letting the server operator configure a range of ports that the EaaS server may assign for this purpose and use the individual ports in a round-robin fashion.

ExplorViz Versions Due to our approach, we can trivially add another feature to our server. Since Docker Compose files are generated on-the-fly, we can let the user choose the version of ExplorViz they want to run. This is accomplished by keeping several different template files in our severs resources. This way older ExplorViz versions can be kept to support visualizing older builds, even if newer ExplorViz versions become incompatible to the Kieker version used in older images. This approach also provides us with the option to extend the selection of dynamic analysis programs in the future.

3.4.3 Container Log Output

To debug errors it can be helpful to see the log output of the containers involved. Most importantly, if a visualization shows unexpected results the user should be able to obtain a log output of their application to diagnose the issue. Logs from containers run through Docker Compose are obtained in real-time with the `docker-compose logs -f` command.

In our interface, the user can choose to receive either the output of his application only, or the combined log of all containers belonging to the visualization instance. Upon starting the log, the server will start the `logs` command and forward all output. This log output should appear in the users browser in real-time as it is produced by the application. The WebSocket protocol is widely supported in browsers and provides bidirectional communication, making it a good choice for real-time communication [Qigang and Sun 2012].

Chapter 4

Implementation

In this chapter, we present selected key parts of our implementation. The primary component is the ExplorViz as a Service server, written in Java using Spring Boot and Vaadin. As part of our implementation, smaller side projects were developed and together, they form the ExplorViz as a Service platform. Everything we developed is included in the provided data package [Petersen 2020].

4.1 Web Interface

The interface is built on top of Vaadin. This means all views are composed from Java objects on the server side, each representing a component or container.

We realize that with each additional tool incorporated into the build workflow, the complexity of the CI process rises and leads to the problem that relevant information for developers is scattered across a multitude of systems [Brandtner et al. 2014]. Therefore we concentrate on implementing a lightweight and simple web interface presenting the core functionality necessary in a clear way.

4.1.1 RichList Component

The *RichList* is one of the core components of the web interface and is reused for many purposes. It can be seen in Figure 4.1. A *RichList* only contains objects a specific class. In the constructor, a converter function has to be given to convert between these objects and UI components displayed in each list entry. Then, the view can add and remove objects to the list with the `addEntry` and `removeEntry` method, even dynamically.

4.1.2 Layout

Upon opening the interface and logging in, the user is presented with the view seen in Figure 4.2. To the left is a sidebar menu that can be toggled with the drawer button in top navigation bar, which was introduced to save space on smaller screens. From the sidebar new projects can be created. It also lists all project the currently logged in user owns.

When opening a project, the sidebar is changed to a project-specific one looking like Figure 4.3, where all pages of the project are listed. The *Secrets* and *Settings* menu entries are only listed if the user is permitted to manage the project.

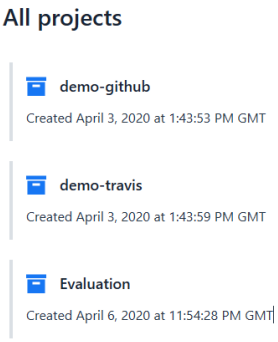

Figure 4.1. A RichList component on the Explore page listing the available projects on the server.

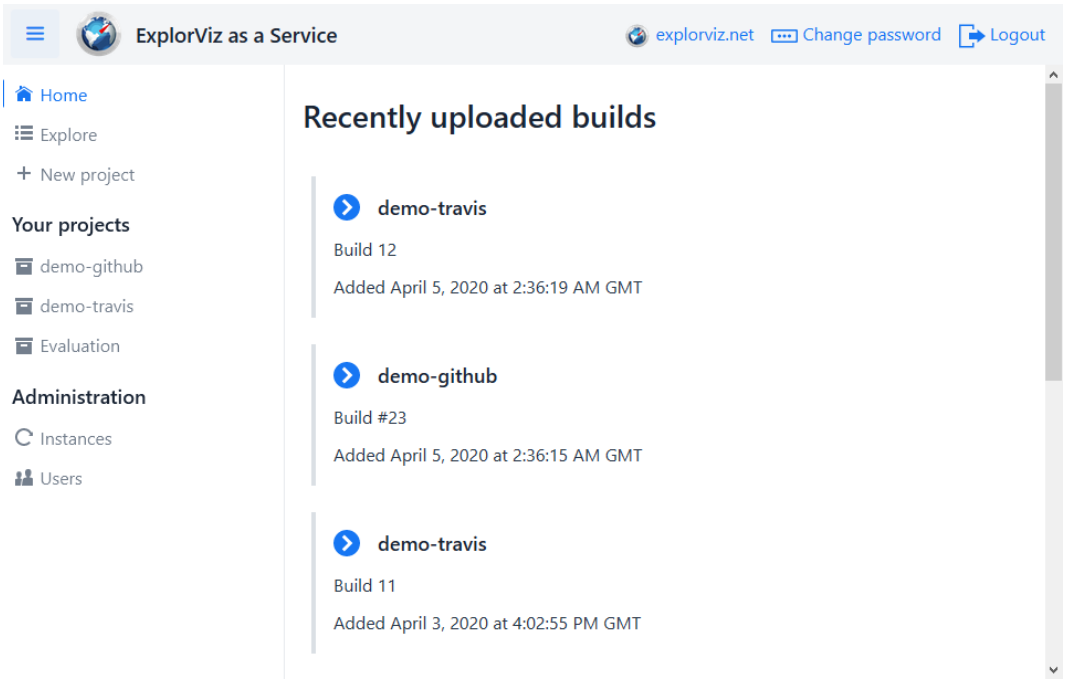

Figure 4.2. Screenshot of the EaaS web interface. To the left is a sidebar menu where all views can be reached from.

On the *Builds* page, all collected builds are listed and new visualizations can be started. This procedure is explained in-depth in Section 4.3. The *Instances* page is like the *Builds* page, but only lists the builds that have an instance running at the moment.

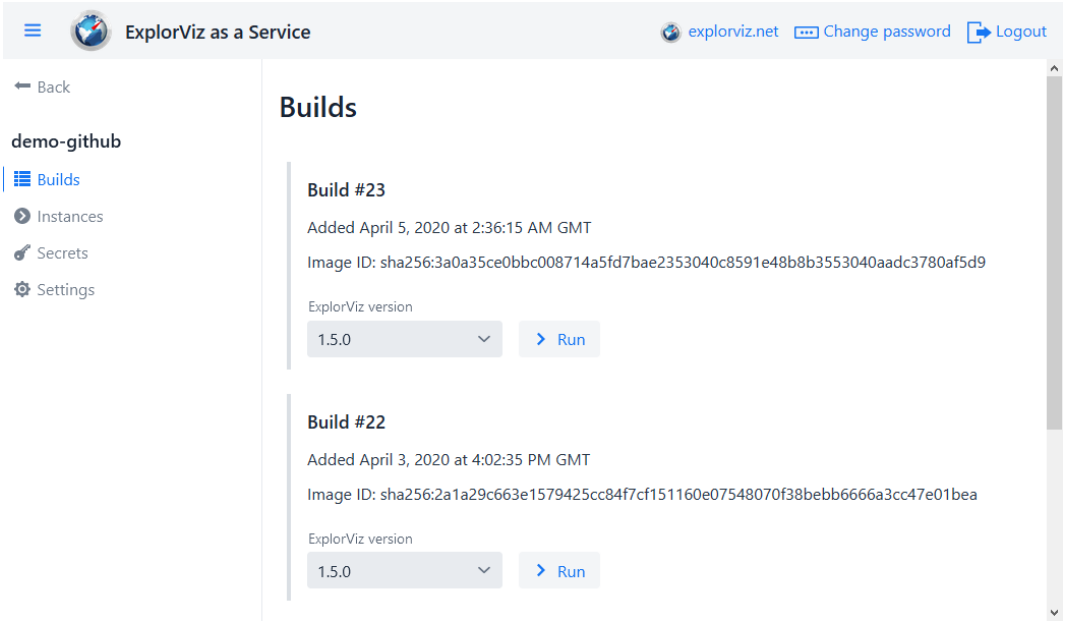

Figure 4.3. Different layout for project views.

4.1.3 Authentication

Due to the nature of Vaadin making single-page applications, page navigations do not trigger a regular HTTP request to a different path for each view, but are instead accomplished with internal requests that bypass the filtering mechanism of Spring Security. Therefore, the authentication checks have to be performed in the views. For this purpose, views implement the *BeforeEnterObserver* interface. When a user tries to enter a view, a *BeforeEnterEvent* is fired before the view is renderer and we can check the permissions of the user to make sure he is allowed to enter the view. Otherwise we redirect him to the login page or display an *Access Denied* error page.

Additionally, for some more coarse-grained permission checks we make use of the *Secured* annotation on some views provided by Spring Security. These annotations by themself have no effect on Vaadin views. We implement a *UIEnterAuthenticator* that is registered to every new Vaadin UI that is created on the server. By doing this, we also get *BeforeEnterEvents*, but in a central location for every view. Then we check if the view about to be entered has the *Secured* annotation and make the user is allowed in.

Login When users want to run visualization or create a project, they need to login. The login procedure looks like Figure 4.4. To avoid having to configure e-mailing, a self-service password reset functionality has not been implemented. Instead, administrators

can generate new passwords for each user.

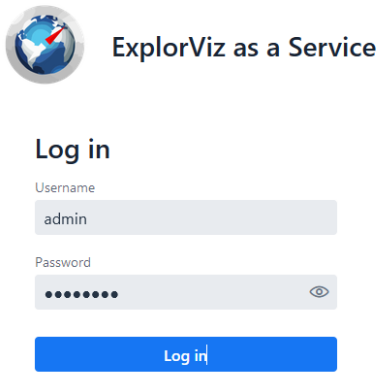

Figure 4.4. Screenshot of the login view.

Users can change their password from the *Change password* page as seen in Figure 4.5. This view has numerous checks to make sure the old password is correct, the new and repeated passwords match and the new password is long enough as per the configuration. The default minimum password length is 8 and can be raised if the administrator deems it necessary.

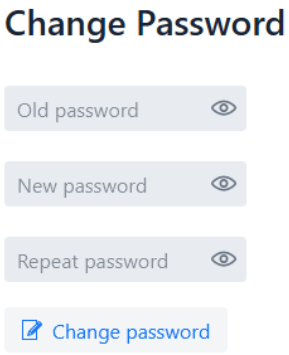

Figure 4.5. Screenshot of the change password view.

4.1.4 User Management

The user management can be reached from the *Users* tab in the sidebar. This menu entry is only visible if the user has the necessary permission to manage users. The user management looks like Figure 4.6. Notably, the currently logged admin is unable to delete or disable his own account. This makes sure that the user management is never left in a state where no administrator exists anymore.

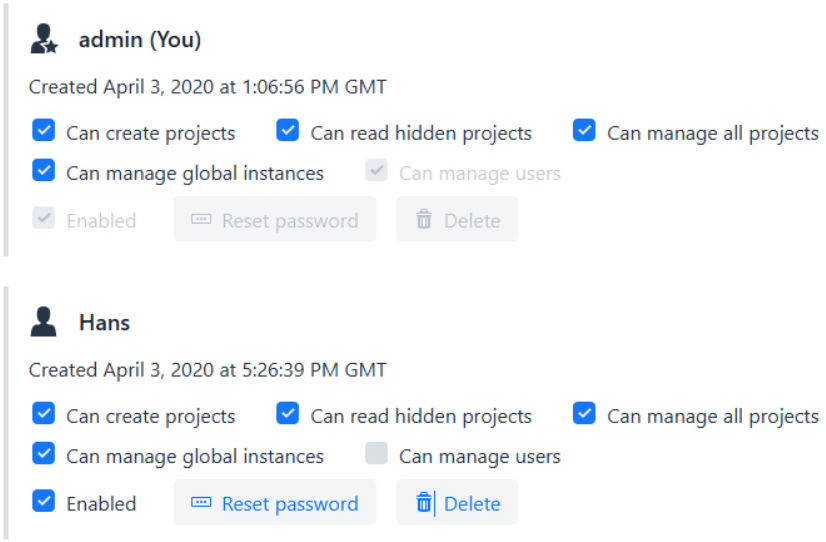

Figure 4.6. Screenshot of the user management view.

4.2 Build Submission API

To accept build submissions, we implement a HTTP API in our application by using the Spring *RestController* annotation. In classes with this annotation, we can use the *RequestMapping* annotation. Java methods annotated with this will be called for incoming HTTP requests matching the path and method specified in the annotation. The signature for our submission API implementation is seen in Listing 4.1. We decided to use form data for input and we output a single plain text string instead of JSON, in order to keep the API as simple as possible.

Listing 4.1. Signature of the HTTP method to submit new builds.

```
1    @RequestMapping(path = "/{project}/builds", method = RequestMethod.POST,
         produces = "text/plain")
```

```
2     public String postProjectBuild(@RequestHeader(value = SECRET_HEADER, required
          = false) String secret,
3                                    @PathVariable("project") long projectId,
4                                    @RequestParam("name") String name,
5                                    @RequestParam("imageID") String imageID,
6                                    @RequestParam("image") MultipartFile image) {
```

4.2.1 Secrets

To manage the secrets accepted by the build submission API, we introduce a *Secrets* view to the project layout. As seen in Figure 4.7, here the owner of the project can add new secrets and view or delete existing ones. One notable feature is the display of the date when the secret was last used. This way the owner can decide whether the secret is still needed.

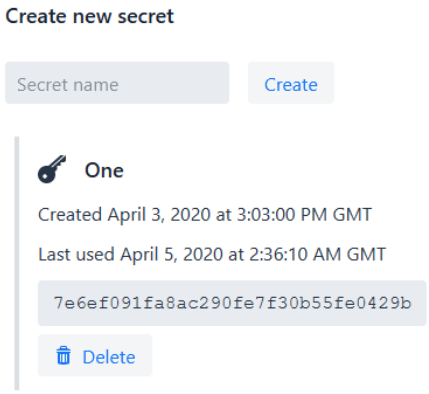

Figure 4.7. Screenshot of the Secrets view for a project.

4.2.2 Submission Script

To simplify the use of EaaS from a CI build, we developed a script, `submission/submit-eaas.sh` that can be copied to the application repository and run during the build workflow. The script is included in Listing 4.2. We have chosen to implement it in shell code rather than a more sophisticated script language, because a shell is available in every Linux environment and dealing with external processes is a task shell scripts perform very well.

Listing 4.2. Shell script to simplify submission of Build images to EaaS.

```
1 #!/bin/sh
2 set -eu
```

```
3
4  hash "docker" 2>/dev/null || { echo "Command docker missing" >&2; exit 1; }
5  hash "curl" 2>/dev/null || { echo "Command curl missing" >&2; exit 1; }
6
7  echo "Building image" >&2
8  IMAGE_ID="$(docker build -q -f "${IMAGE_DOCKERFILE:-Dockerfile}" "${IMAGE_CONTEXT
       :-.}")"
9  echo "Built image: $IMAGE_ID" >&2
10
11 echo "Uploading to EaaS with name: $BUILD_NAME" >&2
12 BUILD_ID="$(docker image save "$IMAGE_ID" | curl -fsS -X POST -H "X-EaaS-Secret:
       $EAAS_SECRET" \
13       -F "name=$BUILD_NAME" -F "imageID=$IMAGE_ID" -F "image=@-" "$EAAS_URL/api/v1
             /projects/$EAAS_PROJECT/builds")"
14 echo "This build has ID #$BUILD_ID" >&2
15
16 # Optional; if your EaaS instance runs on the same machine as you CI builds then
       you MUST comment this out
17 echo "Removing image locally" >&2
18 docker image rm "$IMAGE_ID"
```

Line 4-5 test for the presence of required commands. If they're missing, the script aborts with an error message. In line 8 we build the Docker image for EaaS. The user is expected to specify where the `Dockerfile` is located through the `IMAGE_DOCKERFILE` variable. By using the `-q` option of Docker, we do not get any informational output but only the image ID instead.

This image ID is then used in line 12 to export the image and submit it to the server with `curl`. The headers and parameters we submit match the definition of the controller method in our server. The required environment variables can be specified in the build file corresponding to the CI service in question or through secrets stored by the CI service. This is further looked into in our evaluation in Section 5.3.3.

In line 18 we remove the image from the local Docker image library, to not permanently use up storage space unnecessarily, in case the build is not running in an ephemeral environment that is created for each build. If the build and EaaS run on the same server however, this will delete the only copy of the image and therefore the line must be removed manually in this case.

4.3 Running Visualizations

When the user presses the *Run* button on a build, a new visualization is started. Great care has been taken in our implementation to separate all different aspects of this process into

respective services. In Figure 4.8 we see a sequence diagram showing the communication between the components that we developed for this purpose.

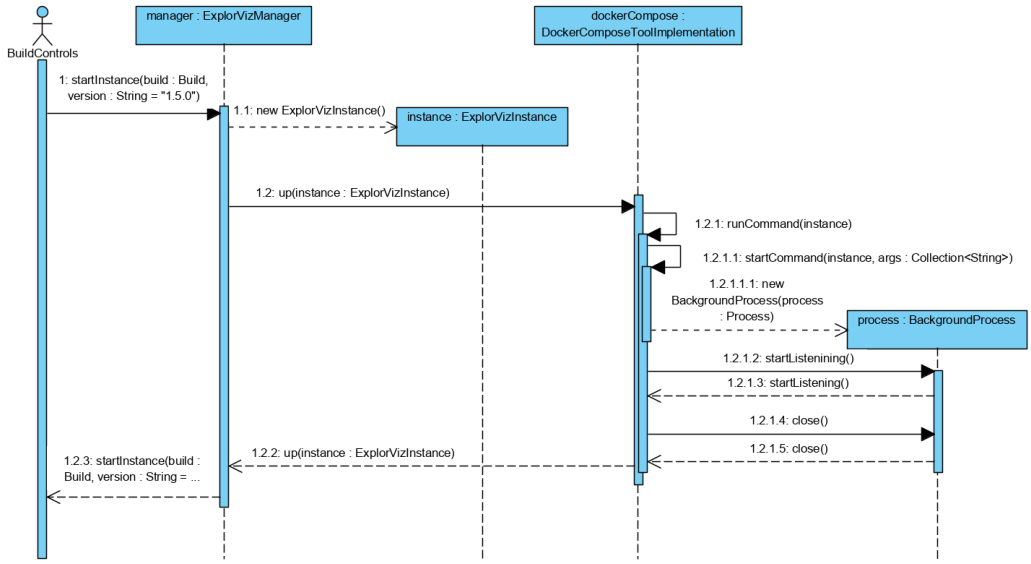

Figure 4.8. Sequence diagram of the execution flow when a new visualization is started by a user.

ExplorVizManager keeps track of all running ExplorViz instances and is also used to start and stop instances. Each instance of ExplorViz gets a corresponding *ExplorVizInstance* object that tracks some metadata, like the port number that was assigned to it and a unique name that is used as the Docker Compose project name and the container label for *Traefik*. The instance class also reads the `docker-compose.yml` template file for the requested ExplorViz version and replaces the placeholder variables we introduced in our approach.

DockerComposeToolImplementation is a stateless adapter to execute Docker Compose commands. *ExplorVizInstance* is a child class of *DockerComposeDefinition*, which provides a method to get the generated `docker-compose.yml` contents. Instead of saving this file to the disk, we let `docker-compose` read the file from standard input and write the compose definition to the processes standard input directly from Java code.

4.3.1 Running Docker-Compose

Docker Compose commands are performed through the `docker-compose` command line utility by running it in a background thread. This functionality is facilitated by a class *BackgroundProcess*. It offers a method to forcefully stop the process — for example after an operation took to long and is believed to have crashed. The main functionality is

the *startListening* method, which takes a *ProcessListener* interface. Implementation of this interface receive real-time notifications about console output printed by the process and are informed when the process exits.

For the `up` and `down` commands used to start and stop ExplorViz instances respectively, this interface is utilized to capture potential error messages. The class is also used to abort an operation if it took to long.

4.3.2 Running the Build Image

As explained in our approach, the build image is run from the same Docker Compose definition simply by adding it to our template. This is achieved by adding a new service definition for the application as shown in Listing 4.3.

Listing 4.3. Addition to ExplorViz docker-compise.yml file.

```
1  application:
2    image: "%APPLICATION_IMAGE%"
3    depends_on:
4      - analysis-service
```

The placeholder `%APPLICATION_IMAGE%` variable is replaced with the Docker image ID of the corresponding build. The `depends_on` directive makes sure the analysis-service of ExplorViz is started before the visualized application runs. This is necessary because the Kieker writer expects that the port is available as soon as it starts, crashing if it is not yet available.

4.3.3 Frontend Access

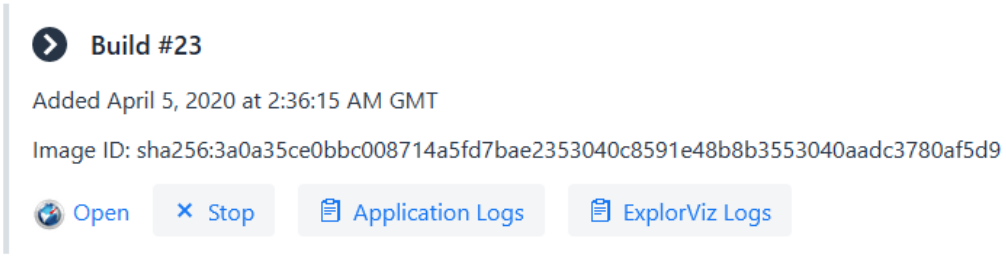

Figure 4.9. Build entry when an instance of the build is running.

When the visualization started successfully, users see a different set of controls for the build entry as seen in Figure 4.9. Upon clicking *Open*, the address as generated from the `accessUrlTemplate` in the servers configuration and the frontend port assigned to this

instance is opened in the users browser. The built-in user authentication of ExplorViz is still in place. The default credentials of ExplorViz must be used to log in.

4.3.4 Logs

To fulfill the task of presenting a live output of the logs in the clients browser, a functionality of Vaadin is leveraged called the *Push* annotation. Annotating a view with this lets Vaadin automatically build a WebSocket connection between the frontend and the server that stays connected even after the view is built and the corresponding JVM thread dies. Other background threads can then modify the users view by accessing the corresponding UI object.

For this task *BackgroundProcess* is used as well. We start the `docker-compose logs -f` command in the background to obtain log output from the container. Furthermore we introduce a component *LogDialog* that implements *ProcessListener* and can therefore listen to output from the *BackgroundThread*. The log dialog is opened when the user presses one of the *Logs* buttons in the *InstanceControls* component as seen in Figure 4.9 and looks like Figure 4.10. When it receives a notification about new console output from the *BackgroundProcess*, an HTML `<pre>` element is appended to the dialog contents with the new text.

Figure 4.10. Application log output dialog updated live for a running instance.

Chapter 5

Evaluation

The last goal we have defined in Section 1.2 is to evaluate our implementation. This is done by setting up CI pipelines with a sample application in which we try to make use of ExplorViz as a Service. First, we lay down the goals we want to achieve. Then, we describe how we execute the evaluation and specify our environment. We conduct a feasibility experiment and present the results afterwards.

5.1 Goals

Our goal is to evaluate the functionality and serviceability of our implementation. Primarily, we want to verify if our solution works with the CI services we originally set to support in Section 1.2. We want to find out how ExplorViz as a Service can be utilized from these services and how to configure the build scripts for them. Furthermore we measure the resource usage of the visualizations and discuss the robustness of the implementation in regards to the debuggability of error conditions.

5.2 Methodology

We assess ExplorViz as a Service by running a feasibility experiment. This involves running an instance of it and observing its behavior in a number of scenarios. These scenarios have been selected to reflect (a) normal usage scenarios and (b) scenarios where we artificially introduce fault conditions that are likely to occur in real-world applications at some point. Based on the outcome of the scenarios we discuss how easily errors can be identified and corrected and in turn make conclusions about the serviceability of the software.

5.3 Experiment

For the experiment we run an instance of EaaS, and then in each scenario attempt to submit a build artifact to it from a CI build and visualize the application in ExplorViz. Because we need an application to test with, we fork[1] the *Spring PetClinic* sample application [Spring

[1] Copying a source code repository to modify it independently

Project 2020] as *EaaS-demo-application* [Petersen 2020] and modify it to our needs. For the duration of this experiment, this project is hosted in a public GitHub repository. This is necessary so we can use it from the public CI services we want to test with.

In Section 5.3.1 we specify the environment in which we run the server. Afterwards we describe the scenarios in Section 5.3.2 and then go into detail how they were carried out in Section 5.3.3.

5.3.1 Experimental Setup

To evaluate our implementation in an environment as close to real-world usage as possible, we deploy EaaS on a dedicated server hosted in a data center. The server is equipped as shown in Table 5.1. Additionally, no swap space is configured. That way, when we run out of memory, the operating system has no choice but to forcefully kill one of the running processes, instead of slowing down while moving memory pages into the swap. These specifications are important as a reference point when we discuss the resource usage of the software in Section 5.4. The versions of the software used can be obtained from Table 5.2.

Table 5.1. Components of the server used to run ExplorViz as a Service in the evaluation.

Component	Specification
Virtualization	KVM (fully virtualized)
Processor	8 vCores, 2.3 GHz
Memory	16 GB DDR4 ECC
Storage	160 GB SSD (RAID10)
Network	1 Gbit/s

Table 5.2. Software installed on the server for the evaluation.

Software	Version
Debian GNU/Linux	10 (buster), x86_64
git	2.20.1
Docker	19.03.7
Docker Compose	1.25.0

ExplorViz as a Service is run in a Docker container. The image is built from the `Dockerfile` included in the *EaaS-server* project directory [Petersen 2020] and the container is started with docker-compose using the included `docker-compose.yml`.

The same docker daemon running the EaaS container is used to store build images and run ExplorViz instances. This is the default configuration specified in the `docker-compose.yml`. Since we use a remote server that we access over the Internet we configure `eaas.explorviz.accessUrlTemplate` to point to the server. To test the amount of ExplorViz visualization we can run in parallel we set `eaas.explorviz.maxInstances` to 100.

5.3.2 Scenarios

We run six different scenarios. The first three represent normal usage to evaluate the server when functioning as intended, the other three model various error conditions that we expect to occur during prolonged use of the software.

Scenario 1: Use from GitHub Actions

In this scenario, we build our sample application on the GitHub Actions service. We use the submission script we developed as part of *EaaS-server* [Petersen 2020] to package and submit the build artifact to our server. We then verify if the build artifact is visible from the ExplorViz as a Service web interface. This requires writing a workflow script for GitHub Actions.

Scenario 2: Use from Travis CI

We repeat the previous scenario, except we try to run the build from the Travis CI service instead. Trying with different CI services is beneficial, because the environments where builds are executed differ for each service. Travis CI also uses its own syntax for build scripts.

Scenario 3: Running Multiple Visualizations

Our implementation allows us run any number of visualization concurrently, bound only by the configured limit and the capability of the host system. We want to determine how many visualizations of our sample application the server can handle on the same system. From the results we derive the resource requirements per ExplorViz instance.

Scenario 4: Visualized Application Fails to Start

It is possible that the docker image submitted to the server is inherently broken and cannot even start the application. This could happen for a number of reasons, the most likely being a faulty `Dockerfile` or entry point script due to human failure. Within this scenario we deliberately create a non-functional entry point script and observe how our implementation handles it.

Scenario 5: Visualized Application Crashes

Applications can have bugs that make them crash completely, even after they started correctly and a visualization is already displayed in the ExplorViz interface. Images that come with a load generation script continuously keep the application busy. A bug could crash the application at any time. In this scenario we simulate such a bug by forcefully

killing the application some time after the visualization started. We want to assess if and how this issue can be diagnosed.

Scenario 6: Port Assigned for ExplorViz Already Used

ExplorViz as a Service automatically assigns unique port numbers for different ExplorViz instances. The starting port can be specified in the configuration and it is the responsibility of the server operator to make sure no port collisions occur, as the server cannot reserve the ports to itself. We want to know how our implementation handles the case when ExplorViz fails to start because the port is already in use by another application.

5.3.3 Execution of the Experiment

First, we setup an EaaS instance on the server. We do this by cloning the *EaaS-server* repository, changing to the directory that was just created and then running `docker build -t eaas-server:latest -f docker/Dockerfile .`. We now have an application image ready to run ExplorViz as a Service. Before we start an instance, we configure some options by modifying `docker/docker-compose.yml` as described in Section 5.3.1. We finally start the instance with `docker-compose -f docker/docker-compose.yml up -d` and leave it running in the background throughout the evaluation. We open the server web interface in the browser on port `8080` and log in with the default credentials.

Next, we need to write a `Dockerfile` that will wrap the build artifact of our sample application together with a load generator into a docker image, which can then be submitted to ExplorViz as a Service. We use one of the *EaaS-base-images* as parent image. Because our application is a *Spring Boot* application that is packaged as a jar-of-jars[2] it uses a custom *ClassLoader*. This causes problems with Kiekers monitoring probe injecting. We solve this problem by unpacking the jar and therefore not using the custom *ClassLoader*. We make use of the `set-kieker-property` command to set a directive in the `kieker.monitoring.properties` that the base image handles for us automatically. The kieker properties are explained in the kieker user guide [Kieker Project 2013]. In the end, our Dockerfile looks like Listing 5.1.

Listing 5.1. Dockerfile to build the EaaS image of our sample application.

```
1 FROM explorviz/eaas-base:11-jre-alpine
2
3 # We need curl to make HTTP request to put load on the application, unzip to
      unpack the jar
4 RUN apk add --no-cache unzip curl
5
6 # Set a nice name to display in ExplorViz
```

[2] A jar is an archive of multiple files. Jar-of-jars bundle all of the applications dependencies as jar inside a bigger jar file.

```
7  RUN set-kieker-property kieker.monitoring.applicationName "Spring PetClinic"
8
9  COPY eaas-image /opt/app/
10 COPY target/spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar /opt/app/
11
12 # Spring Boot uses a jar-of-jars that we need to unpack, otherwise monitoring
       probes cannot be installed
13 RUN unzip spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar \
14  && rm spring-petclinic-2.2.0.BUILD-SNAPSHOT.jar
15
16 USER runner
17 CMD ["/opt/app/run.sh"]
```

Furthermore, we write a script called `create-load.sh` that uses *curl* to continuously send some HTTP requests to our sample applications. This way the application will have some load which results in a more interesting ExplorViz visualization. A `run.sh`, shown in Listing 5.2 is the main command for our container. It first starts the application and then the load script.

Listing 5.2. Shell script that acts as the main command for our image.

```
1  #!/bin/sh
2  set -eu
3
4  # Allow analysis-service some time to start
5  sleep 10
6
7  java-with-kieker -cp ".:BOOT-INF/classes/:BOOT-INF/lib/*" org.springframework.
       samples.petclinic.PetClinicApplication &
8
9  # Now create some load on the server so we will have an interesting visualization
10 ./create-load.sh
```

In the following we describe how we perform each scenario in detail.

Scenario 1: Use from GitHub Actions

GitHub Actions workflows are configured by placing a YAML file in the directory `.github/workflows/` into the repository. We write a workflow that triggers on the `push` event and runs the necessary commands to compile the application, build the docker image and upload it to our instance. The latter steps are carried out by utilizing the submission shell script we developed. To make it available in the GitHub Actions environment, we upload it into the repository.

Because our `Dockerfile` makes use of the *EaaS-base-images*, they need to be available in the GitHub Actions environment. As they are not available from the public DockerHub yet, we simply clone the repository and build them before building our image. The finished workflow looks like Listing 5.3.

Listing 5.3. Workflow file to build our sample application on GitHub Actions.

```yaml
name: Build
on: [push]

jobs:
  build:
    runs-on: ubuntu-18.04
    name: Build
    steps:
    - name: Checkout
      uses: actions/checkout@v2
    - name: Setup Java
      uses: actions/setup-java@v1
      with:
        java-version: 8
        architecture: x64

    - name: Maven Build
      run: mvn -B package

    # Building EaaS-base-image because it is not on DockerHub yet
    - name: Checkout EaaS-base-image
      uses: actions/checkout@v2
      with:
        repository: "ExplorViz/EaaS-base-image"
        path: "EaaS-base-image"
    - name: Build EaaS-base-image
      run: ./build-all.sh
      working-directory: EaaS-base-image

    - name: ExplorViz as a Service
      run: ./submit-eaas.sh
      env:
        IMAGE_CONTEXT: .
        IMAGE_DOCKERFILE: eaas-image/Dockerfile
        BUILD_NAME: "Build #${{ github.run_number }}"
        EAAS_URL: ${{ secrets.EAAS_URL }}
        EAAS_PROJECT: ${{ secrets.EAAS_PROJECT }}
        EAAS_SECRET: ${{ secrets.EAAS_SECRET }}
```

Next, we create a new project in the EaaS web interface. In the GitHub repository settings we create three secrets.

1. **EAAS_URL**: We set this to the URL we access the web interface on.
2. **EAAS_PROJECT**: This ID is copied from the Settings page of the EaaS project.
3. **EAAS_SECRET**: This key is copied from the Secrets page of the EaaS project.

Finally, we create a commit with all of the files. GitHub immediately runs the workflow after we push the commit to the repository.

Scenario 2: Use from Travis-CI

For this scenario we create another new project in the EaaS web interface. The Travis build is configured by placing a `.travis.yml` file in the root directory of the repository, similar to the GitHub workflow. We perform the same build steps as in the GitHub workflow. Travis requires requesting Docker explicitly by specifying it as a service. Our final file looks like Listing 5.4. The remaining three environment variables that specify how to access our EaaS instance are created as secrets in the repository configuration on Travis CI, with details from the newly created project.

Listing 5.4. Workflow file to build our sample application on GitHub Actions.

```
language: java
dist: bionic # Ubuntu 18.04
jdk:
  - openjdk8
services:
  - docker

after_script:
  - git clone https://github.com/ExplorViz/EaaS-base-image.git
  - cd EaaS-base-image
  - ./build-all.sh
  - cd ..
  - ./submit-eaas.sh

env:
  global:
    - IMAGE_CONTEXT="."
    - IMAGE_DOCKERFILE="eaas-image/Dockerfile"
    - BUILD_NAME="Build $TRAVIS_BUILD_NUMBER"
```

After we push this into the repository and log in on Travis CI we can see the build is already running.

Scenario 3: Running Multiple Visualizations

For this scenario, we keep the setup from Scenario 1 and individually push several more changes to the repository. This way, GitHub builds and in turn pushes a build artifact to EaaS for each commit. The changes we make are not relevant for this scenario, but they have to be in the sample applications code in order to produce unique build artifacts. If two build artifacts were exactly the same, their images might have the same ID and EaaS would automatically deduplicate them such that no new build is saved to the EaaS project. We keep pushing changes until we have twenty builds available in EaaS.

Then we start running build visualizations from the web interface. We start them one after another, each with version `1.5.0` and open the ExplorViz in our local browser. We login, wait for the landscape to appear, open the application view and close the tab again. After the memory usage on the server settles, we note down the operating systems total memory used.

Scenario 4: Visualized Application Fails to Start

We start with the working setup from Scenario 1. This time we push a change to the entrypoint script of our image that will make the sample application fail to start. This is done by introducing a misspelling in the name of the main class in line 7 of our `run.sh` script, see Listing 5.5.

Listing 5.5. Changes to the Dockerfile of our EaaS-demo-application.

```
java -with-kieker -cp ".:BOOT-INF/classes/:BOOT-INF/lib/*" org.
    springframework.samples.petclinic.PetClynycApplication &
```

We push this change to the repository, which is then built automatically by our GitHub Actions workflow. Afterwards we start the corresponding build in the EaaS web interface.

Scenario 5: Visualized Application Crashes

Again, we start with the working setup from Scenario 1. After we run a working build in the EaaS interface, we open ExplorViz and log in. When we see the application in the landscape, we open it and leave it open like that. No, to simulate the application crashing we purposefully kill it on the server by executing the `kill -KILL <PID>` command, which will forcefully exit the application without going through a regular shutdown.

Scenario 6: Port Assigned for ExplorViz Already Used

For this scenario, we can reuse any existing, working build and do not need to make a change to the repository. Before we start the build however, we start the command `nc -l -p 8800` a network listener on port 8800 by running the command `nc -l -6 -p 8800` on the server to occupy one of the ports in the range that EaaS assigns for ExplorViz instances.

Then we start and stop a build visualization until EaaS tries to use the already occupied port number.

5.4 Results and Discussion

Now we present the results of the scenarios we just executed.

5.4.1 Scenarios

Scenario 1: Use from GitHub Actions

In the Actions tab on our GitHub repository, we can see the build that was triggered by our commit. Each build step defined in the workflow file has its log output saved. Listing 5.6 is the log output of the EaaS step, where we can see an unwanted behavior: Several characters have been replaced by asterisks. This is caused by GitHub automatically censoring secrets in the log output. Since we have stored the EaaS project ID, which in this case was 2, as a secret, GitHub removed every occurrence of that digit. This does not affect the functionality however and is a purely visual issue. It could be fixed by saving the project ID in the workflow file, instead of using a secret.

Listing 5.6. Log output of the ExplorViz as a Service build step in the GitHub workflow.

```
 1 Run ./submit-eaas.sh
13 Building image
14 Built image: sha***56:33d***730c6bed6ca***35b7***6
    efe9a900561d10aab88b5b5163fc0a8fe7e58***d340
15 Uploading to EaaS with name: Build #1
16 This build has ID #***
17 Removing image locally
```

In the EaaS web interface we can see the build listed with the same image ID, name and the time it was uploaded, as shown in Figure 5.1

Scenario 2: Use from Travis-CI

In the Travis CI interface we can see the build triggered by the commit we pushed. The job log contains all commands run during the build, including those we specified in our Travis build script. The output of the submission script is shown in Listing 5.7. We can find the corresponding build in the EaaS web interface. On the home page of EaaS, we can now also see the list of recently uploaded builds, sorted by date, as shown in Figure 5.2.

Listing 5.7. Log output of the ExplorViz as a Service build step in Travis CI.

```
5495 $ ./submit-eaas.sh
5496 Building image
```

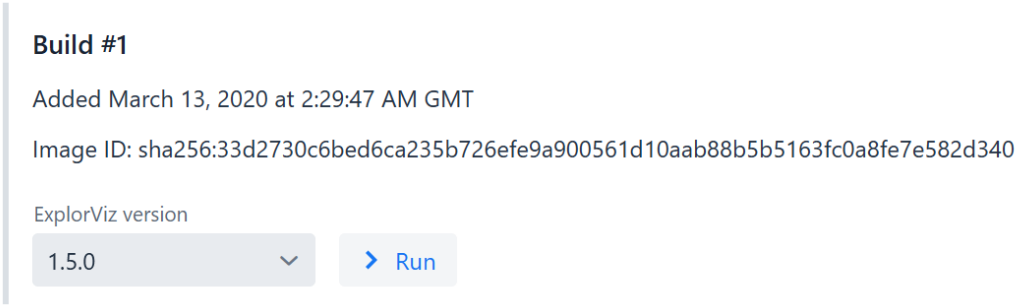

Figure 5.1. A build entry displayed in the EaaS web interface.

```
5497 Built image: sha256:73
    d9dd12e85d397cc3187f1e9e36c539277a2197929c2fee62cd28b51fdeceea
5498 Uploading to EaaS with name: Build 6
5499 This build has ID #41
5500 Removing image locally
```

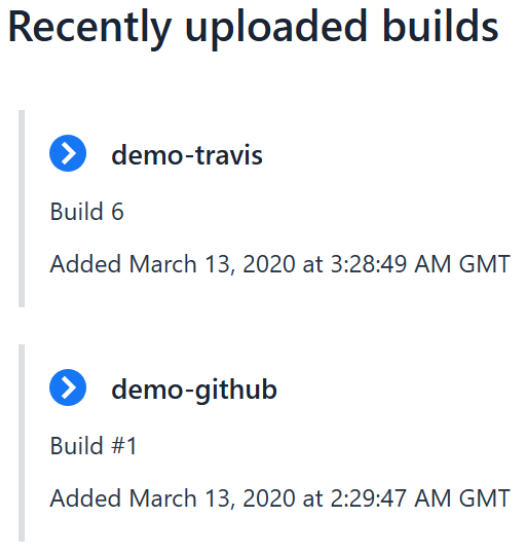

Figure 5.2. Home page of ExplorViz as a Service listing the recently uploaded builds across all projects.

When writing the build script for Travis CI, we had to explicitly request the availability of Docker in our build. This reveals a problematic requirement for EaaS image creation:

Full access to a Docker endpoint is required to build the Docker image for EaaS submission. Docker might not be available on all CI services. Certain types of continuous integration run builds inside containers. Letting applications inside the container access the Docker daemon takes away the security advantage of container-based build environments. This problem however cannot be fixed with our approach to build packaging.

Scenario 3: Running Multiple Visualizations

Starting with only Docker, EaaS server and standard operating system processes running on the system, we measure $1185MiB$ used memory on the system with the `free -m` command before executing the scenario.

With each new ExplorViz instance the total memory usage on the system rises. After starting the visualization, opening it and logging in once, we wait for the memory usage to stop rising before taking the measurement. As we can see in Figure 5.3, the total memory usage increases roughly linearly with the number of instances. The demo applications each use memory in the range of $524MiB$ to $807MiB$, on average $721MiB$. The reason for this large variance is unknown, but we suspect it is a result of the Java garbage collector making different decisions about running a collection or letting the heap size grow. The load scripts that are running parallel to the sample application can be neglected, because their memory usage is around $1MiB$ each. The memory usage of the `EaaS-server` instance is not changing throughout the scenario, which is in line with our expectations. EaaS only keeps a minimum amount of state about which visualization are running and no actual data, it merely runs `docker-compose` commands in the background.

Immediately after starting the seventh visualization, the server becomes unresponsive. Actions in the EaaS interface time out, and the server console doesn't respond to commands anymore. After several minutes, the server starts responding again, and we can see in the output of the `dmesg` command shown in Listing 5.8 that the Linux kernel killed one of the applications because it ran out of memory.

Listing 5.8. Kernel log buffer output showing that the system ran out of memory and killed a java process to free some up.

```
[7726882.861255] Out of memory: Kill process 6844 (java) score 46 or
    sacrifice child
[7726882.861582] Killed process 6844 (java) total-vm:8962960kB, anon-rss
    :759116kB, file-rss:0kB, shmem-rss:0kB
[7726883.134747] oom_reaper: reaped process 6844 (java), now anon-rss:0kB
    , file-rss:0kB, shmem-rss:0kB
```

We conclude that six instances is the maximum we can run on this server. Total memory used was measured at $14649MiB$ for six instances. Subtracting the baseline system memory usage, we find that all our visualizations use a combined $14649MiB - 1185MiB = 13464MiB$ of memory, or $13464MiB/6 = 2244MiB$ on average. Subtracting the memory used by our

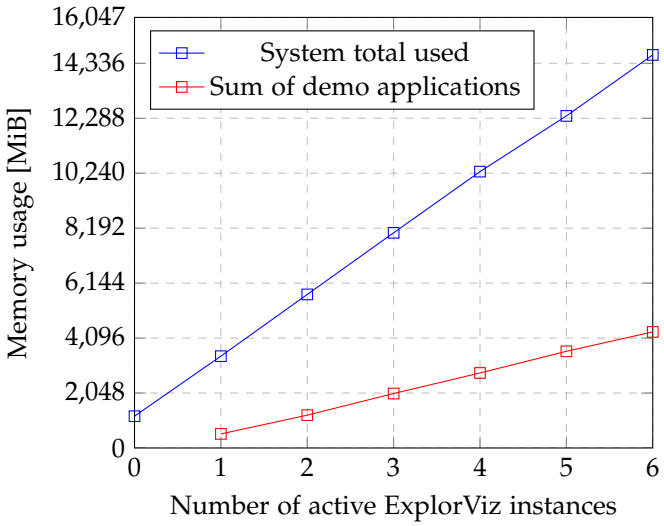

Figure 5.3. Memory usage of ExplorViz visualizations

sample application, we calculate that all ExplorViz instances use $13464MiB - 4326MiB = 9138MiB$ in total or $9138MiB/6 = 1523MiB$ on average.

Because our sample application isn't particularly complex and the visualization that ExplorViz generates is fairly simple, we assume that we approach the baseline memory consumption of an actively used ExplorViz instance very closely. Therefore we postulate a minimum requirement of $1.5GiB$ memory per active ExplorViz instance, plus the memory used by the visualized application.

Scenario 4: Visualized Application Fails to Start

When we run the corresponding build in the EaaS interface, the visualization starts without any errors. However, when we log in to the ExplorViz instance we do not receive a landscape even after several minutes of waiting. This means ExplorViz is not receiving any monitoring records from the application. EaaS offers a way to diagnose application issues: We can read the log output of the applications Docker container by pressing the *Application Logs* on the build entry. We find that we can indeed identify the cause of this problem, because an explanatory error message was printed to the output, as shown in Listing 5.9.

Listing 5.9. Application log output.

```
Attaching to eaas-13-66_application_1
application_1          | Error: Could not find or load main class org.
    springframework.samples.petclinic.PetClynycApplication
```

```
application_1          | Caused by: java.lang.ClassNotFoundException: org.
    springframework.samples.petclinic.PetClynycApplication
eaas-13-66_application_1 exited with code 7


End of output
```

Application issues can only be identified this way if they print an error message to the log. If they don't, the log output might only show the `eaas-XX-YY_application_1 exited with code Z` message, which would require more profound diagnosis and is dependent on the specific command used for the Docker image.

Scenario 5: Visualized Application Crashes

Shortly after killing the sample application, we can see in ExplorViz that all operations stopped, as the visualization becomes flat. The timeline at the bottom also drops to 0 requests. Other than that, there is no indication that the application is not running anymore. If we suspect that the application is dead, we can again check with the *Application Logs* function, where in this case we see the output shown in Listing 5.10 at the very bottom.

Listing 5.10. Last lines of the application log output

```
eaas-14-40_application_1 exited with code 7


End of output
```

There is no explanation however why the application crashed. This is expected because we forcefully killed it, not allowing it to print any output. In case of a real crash, an *Exception* stack trace is likely printed to the output. In this case we can see that the container running our application exited. If the application would hang instead of crashing, this would not be evident from the log.

We conclude that some types of application crashes can be diagnosed easily from EaaS, but simulating a crash by killing the application simplifies this scenario too much and may not represent how most crashes occur in the real world.

Scenario 6: Port Assigned for ExplorViz Already Used

When we try to run the build in the EaaS interface, we receive the following error message: `Error starting instance: Operation failed (error 1). See log for more information` and the build won't show up as a running instance. Therefore, EaaS correctly recognized that our visualization failed to start and does not present a non-functional link to the user. Notably, the error message EaaS presents us with does not provide any details why the launch failed and instead points to the log. This seems adequate however, because administrative server access is required to fix the problem anyway.

When we look into the log output of the docker container running EaaS, we find the excerpt Listing 5.11, which provides information about the error. It clearly states that the listen address `0.0.0.0:8800` is already in use. Therefore we conclude that the error output of EaaS is detailed enough to find the cause of the failure.

Listing 5.11. Last lines of the application log output

```
2020-03-29 06:20:44.942  INFO 1 --- [io-8080-exec-16] n.e.e.s.explorviz.
    ExplorVizManager        : Starting instance eaas-0-40 (#0) on port 8800
2020-03-29 06:20:44.943  INFO 1 --- [io-8080-exec-16] .e.s.d.c.
    DockerComposeToolImplementation : Running command: docker-compose --no
    -ansi -p eaas-0-40 -f /dev/stdin up -d
Creating network "eaas-0-40_default" with the default driver
[...]
Creating eaas-0-40_frontend_1           ... error

ERROR: for eaas-0-40_frontend_1  Cannot start service frontend: driver
    failed programming external connectivity on endpoint eaas-0-40
    _frontend_1 (
    c9deb8229af680ce628d957cc55a52d440d15842e774ea7a1ae30722760ef50f):
    Error starting userland proxy: listen tcp 0.0.0.0:8800: bind: address
    already in use
Encountered errors while bringing up the project.
2020-03-29 06:20:52.460 ERROR 1 --- [io-8080-exec-16] .e.s.d.c.
    DockerComposeToolImplementation : docker-compose exited with error
    code 1
2020-03-29 06:20:52.469 ERROR 1 --- [io-8080-exec-16] n.e.e.s.explorviz.
    ExplorVizManager        : Error starting ExplorViz instance
```

Looking at the list of running Docker container, we notice a mismatch between Docker and the view EaaS has on the situation. EaaS assumes that the instance isn't running at all because the `docker-compose up` command returned an error. However, all containers except the frontend container — which failed to start because it binds to the address that is already in use — are running and cannot be stopped from the EaaS interface. A server operator has to manually stop the containers, but this isn't trivially possible. Because the Docker Compose files are automatically generated by EaaS and never saved to disk, we cannot use `docker-compose` to stop the instance. However, we can manually stop and remove all Docker containers belonging to this instance by executing the command `docker ps -filter name=eaas-0-40-* -aq | xargs docker rm -f`. This is not a perfect solution, because other Docker resources like the network remain. Further improvements might be necessary to handle partial instance startup failures better and not let dysfunctional Docker resources stay up permanently.

5.4.2 Threats to Validity

Several aspects might threaten the validity of our results.

Virtualized Environment Our scenarios were executed on a KVM server. In such a virtualized environment, where the operating system doesn't have dedicated access to the hardware, our system shares some resources with other systems running on the same virtualization host. That means performance measurements are impacted by the load other systems are doing simultaneously. Since memory is a reserved resource in KVM, we believe that our memory usage measurements in Scenario 3 are correct, but to increase the validity of our results, the tests should be repeated in several, different environments, including dedicated hardware.

Inadequate Selection of Scenarios The scenarios designed to test fault-tolerance have been hand-picked with prior knowledge of implementation details. This might cause a bias in our selection towards issues that we know our implementation handles gracefully, instead of issues that are most likely to occur. Therefore, our scenarios might be insufficient to represent real problems. Furthermore, we only tested for a small amount of error conditions. These are certainly not enough to cover all aspects of a real-world deployment of ExplorViz as a Service.

Small-Scale Application Our sample application is a very simple application without external dependencies and can be run without needing any configuration. This makes it trivial to package in a Docker image for EaaS submission. Bigger applications may require access to external services like a database server, which we haven't considered in our evaluation, or face additional challenges when packaging them into a Docker image. Additionally, the applications amount of code is small, resulting in visualization of only a few components in ExplorViz. While the size of the visualized application is not relevant to the CI scenarios, it might affect the memory usage of ExplorViz and change our results of scenario 3. Validity of our results can be increased by trying to setup EaaS for bigger applications.

Chapter 6

Related Work

Dynamic analysis is a topic of high interest. In this section we present related work that covers the topics of software analysis and continuous integration and compare them to our approach. As far as we are aware, there are no scientific publications covering software platforms that keep a history of executable build images with the intent to let developers analyze every build visually.

[Kupsch et al. 2017] introduce the concept of continuous assurance, whose goal is to extend the continuous integration process with automated code analysis. For this purpose they created the Software Assurance Marketplace (SWAMP) platform. Users can upload a package of their softwares source code and SWAMP will automatically run a variety of static analysis tools to find issues. SWAMP integrates with both integrated development environments such as Eclipse and CI tools and can be used as a hosted cloud service or downloaded to be run locally. This work is very similar to ours, also following the approach to upload a package to a hosted service, but focuses on running static analysis tools and combining their results instead of performing dynamic analysis. The authors found that a key to the successful use of code analysis is to reduce obstacles for the programmers, which is something that we tried to do with our approach as well.

[Pina and Cadar 2015] research an approach to dynamically analyze software in their production deployment. This is different from our approach as we visualize builds packaged in a way specifically made for the purpose of visualization. To do so, they use existing debugging tools like Valgrind and address sanitizers. These tools incur a large performance penalty and are therefore not usually applicable for production environments. The authors propose a solution to reduce the performance overhead by doing partial dynamic analysis checking, where only part of the execution is analyzed. Their results show that this multi-version execution approach adds only little overhead and can be feasible for deployments.

[Costin et al. 2016] presents an automated framework that uses dynamic analysis on firmware images to find security vulnerabilities. This is done by fully emulating the system to execute the firmware and then leveraging existing static and dynamic analysis tools to discover vulnerabilities. Their framework is targeted at finding as many issues as possible across a large number of projects, whereas our approach concentrates more closely on different versions of the same project. Their results show that dynamic analysis can be successfully employed to find issues in software.

Apart from publications, there are some notable tools that are relevant to the topic of dynamic analysis.

SonarQube[1] is a platform for continuous inspection of the source code. It tracks software quality metrics over time and is highly related to our work as it similarly integrates into the continuous integration process, however it uses static analysis tools to detect issues and determine the software quality.

A notable mention is InspectIT[2] as a direct competitor of ExplorViz. Like ExplorViz, it is run alongside the application and offers a number of instrumentations like inspecting SQL Statements and HTTP Requests. Their equivalent to the Kieker monitoring agent is IspectIT Ocelot, which is used in the same way and fulfills the same fundamental task of instrumenting the application.

[1] https://www.sonarqube.org/, accessed 2020-04-10
[2] https://inspectit.rocks/, accessed 2020-04-10

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we presented the ExplorViz as a Service platform to run ExplorViz visualization for build artifacts and in turn make it possible to use dynamic analysis and software visualization for continuous integration. Docker makes up the foundation of our approach, allowing us to store and run build artifacts long after they were built, without any environmental constraints. Proceeding to the implementation, the Vaadin framework enabled us to rapidly develop a rich user interface. To evaluate this implementation, we conducted a feasibility experiment in which we tested the system in a number of scenarios. Our results indicate that EaaS as a whole works as we envisioned it, successfully enabling dynamic analysis in continuous integration builds, but there are shortcomings. The requirement of full access to a Docker daemon during the build excludes some CI environments. Furthermore, our evaluation is unsuitable to determine the practicability of the system in a production setting over a longer period of time.

7.2 Future Work

In the future, ExplorViz as a Service could become one of the primary ways to run ExplorViz visualizations. More so, it might make dynamic analysis more popular in continuous integration. The long-term goal is to make ExplorViz more accessible for a larger number of projects. We discuss several approaches to make the use of EaaS more viable.

Scale Using Multiple Docker Endpoints The EaaS server does not take control of the Docker endpoint configuration for `docker-java` and `docker-compose`. Instead, a single endpoint can be configured through environment variables. Therefore, all build images are stored on a single host system. Because ExplorViz and the build image are run from the same `docker-compose` file, all ExplorViz instances also have to run on the same host. As we have seen in our evaluation, each ExplorViz visualization requires a considerable amount of resources. The available memory on the host will most likely be the limiting factor for running more visualizations.

To achieve a higher number of visualizations running in parallel, the server needs to handle the endpoint configuration and allow server operators to configure multiple Docker endpoints. We can then distribute ExplorViz instances across multiple hosts to spread the resource usage evenly and increase the number of concurrent visualizations. One challenge to this are the build images, which have to be stored on the same host system they are running from. Several solutions are possible, each with their own drawbacks. The server could either store new build images on all endpoints or copy build images to the selected endpoint just before it is needed. The former approach would waste a lot of storage space, the latter leads to a significant increase in startup-time for visualizations. A future implementation has to evaluate these approaches and possibly other considerations.

ExplorViz Authentication Backed by EaaS Users Right now, each ExplorViz instance has its own user management. This is because ExplorViz was developed to be used standalone. The ExplorViz versions launched by ExplorViz as a Service are completely unmodified official releases of ExplorViz, which are not aware of EaaS and no integration between the two exist. New ExplorViz instances are started on-the-fly for each visualization and no persistent data is kept. This makes the user management built into ExplorViz pointless, as users are expected to always log in with the default credentials.

By unifying the user management of EaaS and the ExplorViz instances, we could implement authentication for ExplorViz. Then users would need to enter their EaaS credentials when opening a visualization and we could block access if they don't have the rights to access the corresponding project. We could possibly implement this behavior by replacing the `user-service` of ExplorViz. This microservice handles all login- and authentication-related requests in ExplorViz. By writing a custom microservice we can delegate all login requests to the EaaS server, which would be extended with an authentication API.

Robustness and Production Considerations In our evaluation, we hand-picked a few error cases we believe to be probable occurrences in real-world usage. These hand-crafted scenarios are certainly insufficient to fully evaluate the system. More thorough tests must be performed to ensure viability for a larger number of projects. This can be done by setting up EaaS for several Java applications that have a long development history and trying to submit builds of a wide range of revisions. In particular, attention must be paid how often the EaaS setup in the repository must be updated and what other problems occur.

Regarding production environments, our implementation is unable to fulfill certain requirements that will arise in such a setting over longer time. Management of build images is insufficient. Images could be deleted by accident, for example when pruning unused images from the host. Furthermore, there is no way to move build images to a different host, which means all previous builds will be lost when changing the host EaaS is running on.

Bibliography

[Bolduc 2016] C. Bolduc. Lessons learned: using a static analysis tool within a continuous integration system. In: *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2016, pages 37–40. (Cited on page 11)

[Brandtner et al. 2014] M. Brandtner, E. Giger, and H. Gall. Supporting continuous integration by mashing-up software quality information. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 2014, pages 184–193. (Cited on page 23)

[Costin et al. 2016] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: Association for Computing Machinery, 2016, pages 437–448. URL: https://doi.org/10.1145/2897845.2897900. (Cited on page 49)

[Fittkau et al. 2017] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with explorviz. *Information and Software Technology* 87 (2017), pages 259–277. URL: http://www.sciencedirect.com/science/article/pii/S0950584916301185. (Cited on pages 1, 8)

[Hilton et al. 2016] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pages 426–437. URL: https://doi.org/10.1145/2970276.2970358. (Cited on page 11)

[Kieker Project 2013] Kieker Project. *Kieker user guide*. Apr. 2013. URL: http://kieker-monitoring.net/documentation/. (Cited on pages 7, 36)

[Kupsch et al. 2017] J. A. Kupsch, B. P. Miller, V. Basupalli, and J. Burger. From continuous integration to continuous assurance. In: *2017 IEEE 28th Annual Software Technology Conference (STC)*. 2017, pages 1–8. (Cited on page 49)

[Petersen 2020] J. E. Petersen. *Thesis artifacts for: enabling dynamic analysis and software visualization in continuous integration platforms*. Apr. 2020. URL: https://doi.org/10.5281/zenodo.3746968. (Cited on pages 23, 34, 35)

[Pina and Cadar 2015] L. Pina and C. Cadar. Towards deployment-time dynamic analysis of server applications. In: *Proceedings of the 13th International Workshop on Dynamic Analysis*. WODA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pages 35–36. URL: https://doi.org/10.1145/2823363.2823372. (Cited on page 49)

[Qigang and Sun 2012] L. Qigang and X. Sun. Research of web real-time communication based on web socket. *International Journal of Communications, Network and System Sciences* 05 (Jan. 2012), pages 797–801. (Cited on page 22)

[Rad et al. 2017] B. B. Rad, H. J. Bhatti, and M. Ahmadi. An introduction to docker and analysis of its performance. In: *IJCSNS International Journal of Computer Science and Network Security, VOL.17 No.3*. Mar. 2017, pages 228–235. (Cited on page 9)

[Sengupta and Roychoudhury 2011] B. Sengupta and A. Roychoudhury. Engineering multi-tenant software-as-a-service systems. In: *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*. PESOS '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pages 15–21. URL: https://doi.org/10.1145/1985394.1985397. (Cited on page 1)

[Soares and Preguiça 2018] J. Soares and N. Preguiça. Database engines on multicores scale: a practical approach. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: Association for Computing Machinery, 2018, pages 2335–2340. URL: https://doi.org/10.1145/2695664.3200145. (Cited on page 6)

[Spring Project 2020] Spring Project. *Spring petclinic repository*. 2020. URL: https://github.com/spring-projects/spring-petclinic. (Cited on pages 13, 33)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22–25, 2012: ACM, Apr. 2012, pages 247–248. (Cited on pages 1, 7)

[Zhao et al. 2019] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt. Slimmer: weight loss secrets for docker registries. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pages 517–519. (Cited on page 17)

[Zirkelbach et al. 2019] C. Zirkelbach, A. Krause, and W. Hasselbring. Modularization of research software for collaborative open source development. In: *The Ninth International Conference on Advanced Collaborative Networks, Systems and Applications (COLLA 2019)*. June 2019. URL: http://eprints.uni-kiel.de/46777/. (Cited on pages 8, 21)