INSTITUT FÜR INFORMATIK

On the Modularization of ExplorViz towards Collaborative Open Source Development

Christian Zirkelbach,
Alexander Krause,
and Wilhelm Hasselbring

Bericht Nr. 1902
April 2019
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

On the Modularization of ExplorViz towards Collaborative Open Source Development

Christian Zirkelbach,
Alexander Krause,
and Wilhelm Hasselbring

Bericht Nr. 1902
April 2019
ISSN 2192-6247

e-mail: {czi, akr, wha}@informatik.uni-kiel.de

Technical Report

On the Modularization of ExplorViz towards Collaborative Open Source Development

Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring

Software Engineering Group
Kiel University, 24118 Kiel, Germany
{czi,akr,wha}@informatik.uni-kiel.de

Abstract. Software systems evolve over their lifetime. Changing conditions such as requirements or customer requests make it inevitable for developers to perform adjustments to the underlying code base. Especially in the context of open source software where everybody can contribute, demands can change over time and new user groups may be addressed. In particular, research software is often not structured with a maintainable and extensible architecture. In combination with obsolescent technologies, this is a challenging task for developers, especially, when students are involved.
In this paper, we report on the modularization process and architecture of our open source research project *ExplorViz* towards a microservice architecture, which facilitates a collaborative development process for both researchers and students. We describe the modularization measures and present how we solved occurring issues and enhanced our development process. Afterwards, we illustrate our modularization approach with our modernized, extensible software system architecture and highlight the improved collaborative development process. Finally, we present a proof-of-concept implementation featuring several developed extensions in terms of architecture and extensibility.

Keywords: collaborative software engineering · architectural modernization · software visualization

1 Introduction

Software systems are continuously evolving over their lifetime. Changing contexts, legal, or requirements changes such as customer requests make it inevitable for developers to perform modifications of existing software systems. Open source software is based on the open source model, which addresses a decentralized and collaborative software development. Open research software [12] is available to the public and enables anyone to copy, modify, and redistribute the source code without costs and sometimes with only a few restrictions. In this context, where anyone can contribute code or feature requests, requirements can change over time and new user groups can appear. Although this development model features a lot of collaboration and freedom, the resulting software does not necessarily constitute a maintainable and extensible underlying architecture. Additionally, employed technologies and frameworks can become obsolescent or are not updated anymore. In particular, research software is often not structured with a maintainable and extensible architecture [19]. This causes a challenging task for developers during the development, especially when inexperienced collaborators like students are involved.

Based on several drivers, like technical issues or occurring organization problems, many research and industrial projects need to move their applications to other programming languages, frameworks, or even architectures. Currently, a tremendous movement in research and industry constitutes a migration or even modernization towards a microservice architecture, caused by promised benefits like scalability, agility and reliability [14]. Unfortunately, the process of moving towards a microservice-based architecture is difficult, because there a several challenges to address from both technical and organizational perspectives [11].

In this paper, we report on the modularization process of our open source research project *ExplorViz* towards a more collaborative-oriented development on the basis of a microservice architecture. We later call the old version *ExplorViz Legacy*, and the new version just *ExplorViz*.

Summarized, our main contributions in this paper are:

- Identification of technical and organizational problems in our monolithic open source research software system
- A modularization process focusing on the collaborative development moving towards a microservice architecture
- A proof-of-concept implementation, followed by an evaluation based on several developed extensions
- An ongoing modularization process to achieve a profound, separately deployable, microservice architecture

The remainder of this paper is organized as follows. In Section 2, we illustrate our problems and drivers for a modularization and architectural modernization. Afterwards, we illustrate our software system and underlying architecture of *ExplorViz Legacy* in Section 3. The following modularization and modernization process and target architecture of *ExplorViz* is depicted in Section 4. Section 5 describes our proof of concept in detail, including an evaluation based on several developed extensions. Our ongoing work in terms of achieving a profound microservice architecture is presented in Section 6. Section 7 discusses related work on modularization and modernization towards microservice architectures. Finally, the conclusions are drawn and an outlook is given.

2 Problem Statement

The open source research project *ExplorViz* started in 2012 as part of a PhD thesis and is further developed and maintained until today. *ExplorViz* enables a live monitoring and visualization of large software landscapes [9, 8]. In particular, the tool offers two types of visualizations – a landscape-level and an application-level perspective. The first perspective provides an overview of a monitored software landscape consisting of several servers, applications, and communication in-between. The second perspective visualizes a single application within the software landscape and reveals it's underlying architecture, e.g., the package hierarchy in Java, and shows classes and related communication. The tool has the objective to aid the process of system and program comprehension for developers and operators. We already successfully employed the software in several collaboration projects [15, 16] and experiments [7, 5].

We are developing the project from the beginning on GitHub with a small set of core developers and a large number of collaborators over the time. Several extensions have been implemented since the first version, which enhanced the tool's features. Unfortunately, this led to an unstructured architecture due to an unsuitable collaboration and integration process. In combination with technical debt and issues of our employed framework and underlying architecture, we had to perform a technical and process-oriented modularization.

Since 2012, several researchers, student assistants, and a total of 25 student theses as well as multiple projects contributed to *ExplorViz*. We initially chose a web framework that simplified the development for our students. The Google Web Toolkit (GWT), a Java-based framework for web applications, seemed to be a good fit in 2012, since Java is the most used language in our lectures. GWT provides different wrappers for HTML and compiles a set of Java classes to JavaScript (JS) to enable the execution of applications in web browsers. Employing GWT in our project resulted in a monolithic application (hereinafter referred to as *ExplorViz Legacy*), which introduced certain problems over the course of time.

2.1 Extensibility & Integrability

ExplorViz Legacy's concerns are divided in core logic (core), e.g., predefined software visualizations, and extensions. When *ExplorViz Legacy* was developed, students created new git branches to implement their given task, e.g., a new extending feature. However, there was no extension mechanism that allowed the integration of features without rupturing the core's code base. Therefore, most students created different, but necessary features in varying classes for the same functionality. Furthermore, completely new technologies were utilized, which introduced new, sometimes even unnecessary (due to the lack of knowledge), dependencies. Eventually, most of the developed features could not be easily integrated and thus remained isolated in their branch.

2.2 Code Quality & Accessibility

We chose GWT as framework for our project, since its programming language Java has great prominence among our students. After a short period of time, modern JS web frameworks became increasingly mature. Therefore, we started to use GWT's JavaScript Native Interface (JSNI) to embed JS functionality in client-related Java methods. For example, this approach allowed us to introduce a more accessible JS-based rendering engine. Unfortunately, JSNI was overused and the result was a partitioning of the code base. Developers were now starting to write Java source code, only to access JS, HTML, and CSS. This partitioning reduced the accessibility for new developers. Furthermore, the integration of modern JS libraries in order to improve the user experience in the frontend was problematic. Additionally, Google announced that JSNI would be removed with the upcoming release of Version 3, which required the migration of a majority of client-related code. Google also released a new web development programming language, named *DART*, which seemed to be the unofficial successor of GWT. Thus, we identified a potential risk, if we would perform a version update.

Computer science students of our university know and use supporting software for code quality, e.g., static analysis tools such as Checkstyle or PMD, since we teach them and expect their usage in mandatory lectures. However, we did not define a common code style supported by these tools in *ExplorViz Legacy*. As a result, some of the most common Java conventions were ignored and bugs occurred in our software. Therefore, a vast amount of extensions required a lot of refactoring, especially when we planned to integrate a feature into the core.

2.3 Software Configuration & Delivery

In *ExplorViz Legacy*, integrated features were deeply coupled with the core and could not be easily taken out. Often, users did not need all new features, but only a certain subset of the overall functionality. Therefore, we introduced new branches with different configurations for several use cases, e.g., a live demo. Afterwards, users could download resulting artifacts, but the maintenance of related branches was cumbersome.

Summarized, the mentioned problems worsened the extensibility, maintainability, and comprehension for developers of our software. Therefore, we were in need of modularizing and modernizing *ExplorViz*.

3 *ExplorViz Legacy*

In [28] the authors gave a very brief description on the modernization of *ExplorViz* towards a microservice architecture. In order to understand the modularization process, we provide more detailed and technical information about our old architecture (*ExplorViz Legacy*) in the following. The overall architecture and the employed software stack of *ExplorViz Legacy* is shown in Figure 1.

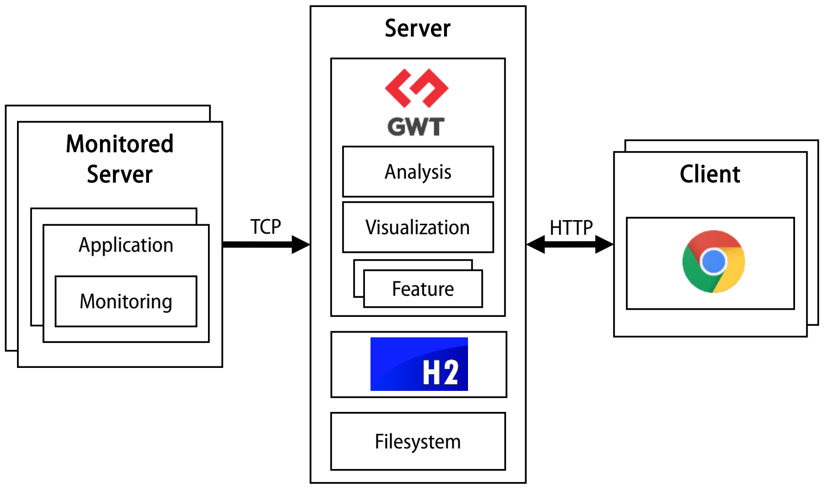

Fig. 1. Architectural overview and software stack of *ExplorViz Legacy*

We are instrumenting (Java) applications, regardless whether they are native applications or deployed artifacts in an application server like Apache Tomcat. The instrumentation is realized by our monitoring component, which employs in the case of Java *AspectJ*, an aspect-oriented programming extension for Java. *AspectJ* allows us to intercept an application by bytecode-weaving in order to gather necessary monitoring information for analysis and visualization purposes. Subsequently, the monitoring information is transported via TCP towards a server, which contains our GWT application. This part represents the two major components of our architecture, namely *analysis* and *visualization*.

The *analysis* component receives the monitoring information, reconstructs traces, and creates software landscapes consisting of monitored applications and communication in-between. These landscapes are stored in the file system. Our user-management employs a *H2 database* to store related data. The software landscape *visualization* is provided via HTTP and is accessible by clients with a web browser.

GWT is an open source framework, which allows to develop JS front-end applications in Java. It facilitates the usage of Java code for server (backend) and client (frontend) logic in a single web project. Client-related components are compiled to respective JS code. The communication between frontend and backend is handled through asynchronous remote procedure calls (ARPC) based on HTTP. The usage of ARPC allows non-professional developers, in our case computer science students, to easily extend our existing open source research project. ARPC enables a simple exchange of Java objects between client and server.

In *ExplorViz Legacy*, the described advantages of GWT proved to be a drawback, because every evolutionary change affects the whole project due to its single code base. Hence, every new developed feature was hard-wired into the software system and could not be easily maintained, extended, or replaced by another component.

This situation was a leading motivation for us to look for an up-to-date framework replacement. We intended to take advantage of this situation and modularize our software system in order to move from a monolithic, to a distributed (web) application divided into separately maintainable and deployable backend and frontend components.

Our open source research project is publicly accessible since the beginning on GitHub and is licensed under the Apache License, version 2.0. Our development process facilitated the maintainability and extensibility of our software by means of so-called feature branches. Every code change, e.g., a new feature or bugfix, whether is was developed by a researcher, a student assistant, or a student during a thesis or project, had to be implemented in a separated feature branch based on the master branch. After performing a validation on the viability and quality of the newly written source code, the branch needed to be merged into the master project and thus permanently into the project. This fact often led to an intricate and time-consuming integration process, since all developers worked on a single code base. For that reason, we had to improve our development process to perform a modularization and technical modernization.

4 Modularization Process and Architecture of *ExplorViz*

The previously mentioned drawbacks in *ExplorViz Legacy* and recent experience reports in literature about successful applications of alternative technologies, e.g., RESTful APIs [25, 27], were triggers for a modularization and modernization.

4.1 Requirement Analysis and Goals

We no longer perceived advantages of preferring GWT over other web frameworks. During the modularization planning phase, we started with a requirement analysis for our modernized software system and identified technical and development process related impediments in the project. We kept in mind that our focus was to provide a collaborative development process, which encourages developers like students to participate in our research project. Furthermore, developers, especially inexperienced ones, tend to have potential biases during the development of software, e.g., they make decisions on their existing knowledge instead of exploring unknown solutions [24]. As a result, we intended to provide plug-in mechanisms for the extension of the backend and frontend with well-defined interfaces. We intended to encourage developers to try out new libraries and technologies, without rupturing existing code.

According to [22], the organization of a software system implementation is not an adequate representation of a system's architecture. Thus, architectural changes towards the implementation of a software system have to be documented before or at least shortly after the realization. If this is not addressed, the architecture model has a least to be updated based on the implementation in a timely manner. Thus, we took this into account in order to enhance our development process.

Architectural decay in long-living software systems is also an important aspect. Over time, architectural smells manifest themselves into a system's implementation, whether they were introduced into the system from the beginning or later during development [23]. For the modularization process of our software system it was necessary to look for such smells in order to eliminate them in the new system. In the end, we identified the following goals for our modularization and modernization process:

- The software system needs to be stripped down to it's core, anything else is a form of extension.
- We need to focus on the main purpose of our software system – the visualization of software landscapes and architectures. Thus, we need to look for a monitoring alternative.
- The backend and frontend should be separately deployable and technologically independent. The latter goal allows us to exchange them with little effort. Additionally, they store their own data and use no central storage or database.
- Code skeletons are provided for the development of extensions.
- We stick to the encapsulation principle and provide well-defined interfaces.
- The overall development process needs to be enhanced, e.g,, by using continuous integration (CI) and quality assurance, like code quality checks.

Detailed decision triggers and the decision making process are published in [28]. As a result of this process, we agreed on building upon an architecture based on microservices. This architectural style offers the ability do divide monolithic applications into small, lightweight, and independent services, which are also separately deployable [3, 1]. Adopting the above mentioned goals lead us finally to the microservice-based architecture shown in Figure 2.

4.2 Extensibility & Integrability

In a first step, we replaced our custom-made monitoring component by the monitoring framework *Kieker* [18]. This framework provides an extensible approach for monitoring and analyzing the runtime behavior of concurrent or distributed software systems. Monitored information is sent via TCP to our backend. Kieker employs a similar monitoring data structure, which fits our replacement requirements perfectly.

We modularized our GWT project into two separated projects, i.e., backend and frontend, which are now two self-contained microservices. Thus, they can be developed and deployed separately on different server nodes. In detail, we employ distinct technology stacks with independent data storage. This allows us

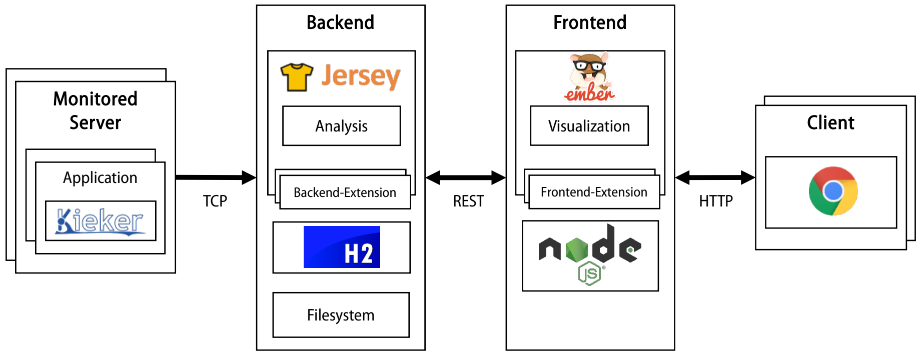

Fig. 2. Architectural overview and software stack of the modularized *ExplorViz*

to exchange the microservices, as long as we take our well-defined interfaces into account. The backend is implemented as a Java-based web service providing a RESTful API for clients. The web server frontend is implemented in JS. The client is a standard web browser.

The backend provides its API for frontend instances and employs the Jersey framework.[1] *Jersey* implements the Servlet 3.0 specification, which offers javax.servlet.annotations to define servlet declarations and mappings. We assume that this procedure eases the development process in the backend, especially for students. The backend employs the filesystem and H2 for storage.

The frontend uses the JS framework Ember.js,[2] which enables us to offer visualizations of software landscapes to clients with a web browser. Since *Ember* is based on the model-view-viewmodel architectural pattern, developers do not need to manually access the Document Object Model and thus need to write less source code. *Ember* uses *Node.js*[3] as execution environment and emphasizes the use of components in web sites, i.e., self-contained, reusable, and exchangeable user interface fragments. We build upon these components to encapsulate distinct visualization modes, especially for extensions. Communication, like a request of a software landscape from the backend, is abstracted by so-called *Ember* adapters. These adapters make it easy to request or send data by using the convention-over-configuration pattern. The introduced microservices, namely backend and frontend, represent the core of *ExplorViz*. As for future extensions, we implemented well-defined extension interfaces for both microservices, that allow their integration into the core.

4.3 Code Quality & Accessibility

New project developers, e.g., students, do not have to understand the complete project from the beginning. They can now extend the core by implementing new

[1] `https://jersey.github.com`
[2] `https://www.emberjs.com`
[3] `https://nodejs.org`

mechanics on the basis of a plug-in extension. Extensions can access the core functionality only by a well-defined read-only API, which is implemented by the backend, respectively frontend. This high level of encapsulation and modularization allows us to improve the project, while not breaking extension support. Additionally, we did no longer have a conglomeration between backend and frontend source code, especially the mix of Java and JS, in single components. This eased the development process and thus reduced the number of bugs, which occurred in our old software. Another simplification was the use of *JSON:API*[4] as data exchange format between backend and frontend. *JSON:API* is a specification for building APIs based on JSON. It allows the definition of attributes or relations for a data object. This minimizes the amount of data and round trips needed when making API calls. Due to its well-defined structure and relationship handling, developers are greatly supported when exchanging data.

4.4 Software Configuration & Delivery

One of our goals was the ability to easily exchange the microservices. We fulfill this task by employing frameworks, which are exchangeable with respect to their language domain, i.e., Java and JS. We anticipate that substituting these frameworks could be done with reasonable effort, if necessary. Furthermore, we offer pre-configured artifacts of our software for several use cases by employing Docker images. Thus, we are able to provide containers for the backend and frontend or special purposes, e.g., a fully functional live demo. Additionally, we implemented the capability to plug-in developed extensions into the backend, by providing a package-scanning mechanism. The mechanism scans a specific folder for compiled extensions and integrates them at runtime.

5 Proof-of-Concept Implementation

5.1 Implementation

We realized a proof-of-concept implementation and split our project as planned into into two separate projects – a backend project based on *Jersey*, and a frontend project employing the JS framework *Ember*. Both frameworks have a large and active community and offer sufficient documentation and tutorials, which are important for new developers. As shown in Figure 2, we strive for an easily maintainable, extensible, and plug-in-oriented microservice architecture.

5.2 Developed extensions

Since the end of our modularization and modernization process in early 2018, we were able to successfully develop several extensions both for the backend and the frontend.

[4] `https://jsonapi.org`

Application Discovery Although we employ a monitoring framework, it lacks a user-friendly setup configuration due to its framework characteristic. Thus, users of our live trace visualization tool *ExplorViz*, experienced problems with instrumenting their applications for monitoring. In [21], we reported on our application discovery and monitoring management system to circumvent this drawback. The key concept is to utilize a software agent that simplifies the discovery of running applications within operating systems. An example visualization of the extension's user-interface is shown in Figure 3. The figure shows three discovered applications on a monitored server. Furthermore, this extension properly configures and manages the monitoring framework. Finally, we were able to conduct a first pilot study to evaluate the usability of our approach with respect to an easy-to-use application monitoring. The improvement regarding the usability of the monitoring procedure of this extension was such a great success, such that we integrated this extension after a few months into the core.

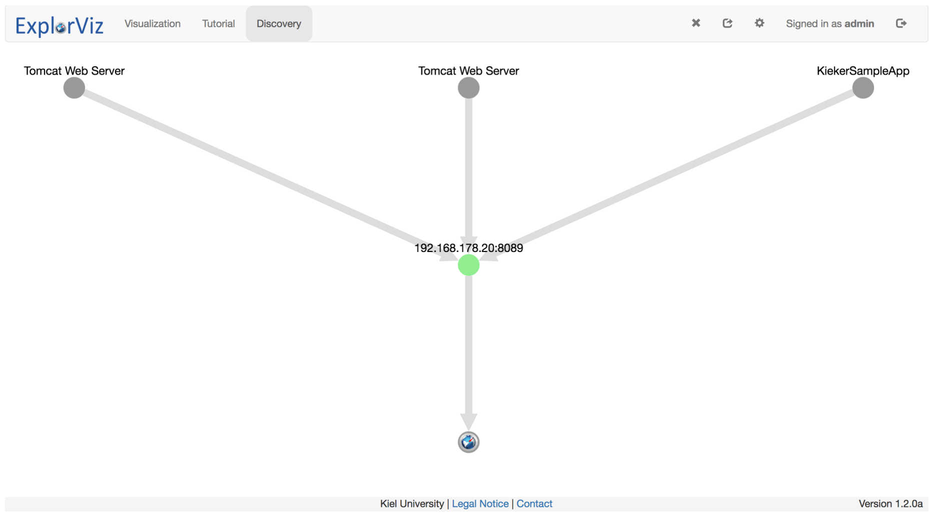

Fig. 3. Screenshot of the application discovery extension of *ExplorViz*

Virtual Reality Support An established way to understand the complexity of a software system is to employ immersive visualizations of software landscapes [6]. However, with the help of visualization alone, exploring unknown software is still a potentially challenging and time-consuming task. For these extensions, two students followed a new approach using virtual reality (VR) for exploring software landscapes collaboratively. They employed head mounted displays (HTC Vive and Oculus Rift) to allow the collaborative exploration of software in VR. A screenshot of the VR extension featuring the application-perspective and both VR controllers is shown in Figure 4. They built upon our microservice architecture and employed WebSocket connections to exchange data to achieve modular extensibility and high performance for this real-time user environment. As a proof of concept, they conducted a first usability evaluation with 22 subjects. The results of this evaluation revealed a good usability and thus constituted a valuable extension to our project.

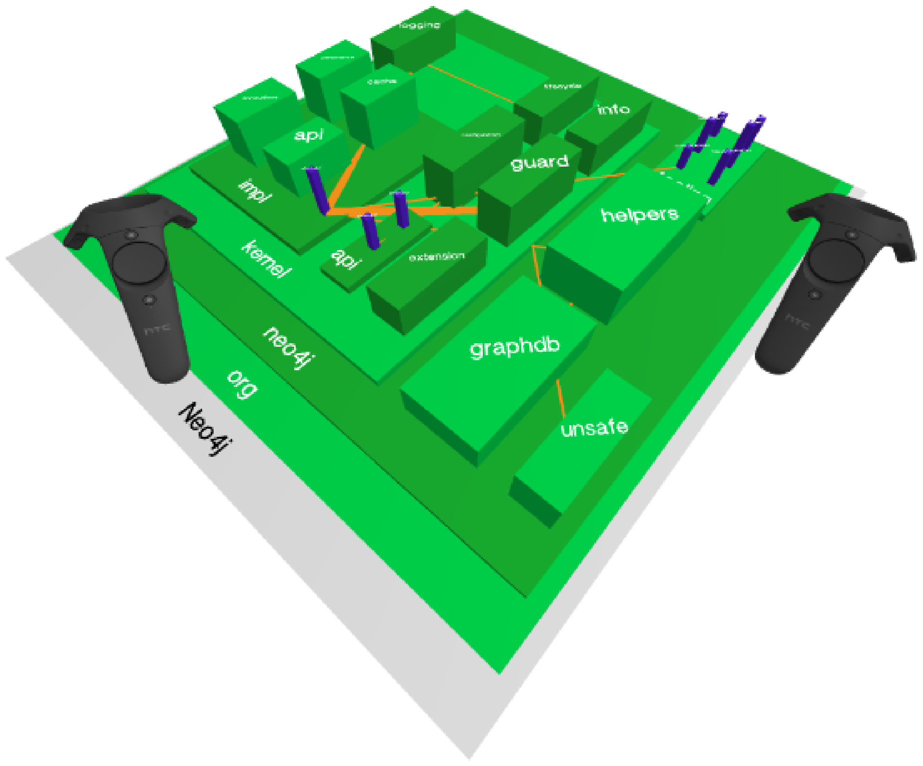

Fig. 4. Screenshot of the VR extension of *ExplorViz* showing the application perspective and both VR controllers

Architecture Conformance Checking Software landscapes evolve over time, and consequently, architecture erosion occurs. This erosion causes high maintenance and operation costs, thus performing architecture conformance checking (ACC) is an important task [10]. ACC allows faster functionality changes and eases the adaptation to new challenges or requirements. Additionally, software architects can use ACC to verify a developed version against a previous modeled version. This can be used to verify whether the current architecture complies with the specified architecture and can detect constraint violations. An example architecture conformance visualization of a monitored software landscape against a modeled one is shown in Figure 5. The visualization illustrates missing, additional, or modified nodes and applications and related communication in-between for a software landscape. In this extension, a student developed an approach to perform an ACC between a modeled software landscape consisting of applications using an editor and a monitored software landscape. Thus, enabling a visual comparison between both versions on an architectural level. In order to evaluate the extension, the student conducted a usability study with five participants, applying the model editor for a desired software landscape and performing an ACC of a modeled software landscape against a monitored one. The results indicated a good user experience of the approach, although the usability of the editor could be improved.

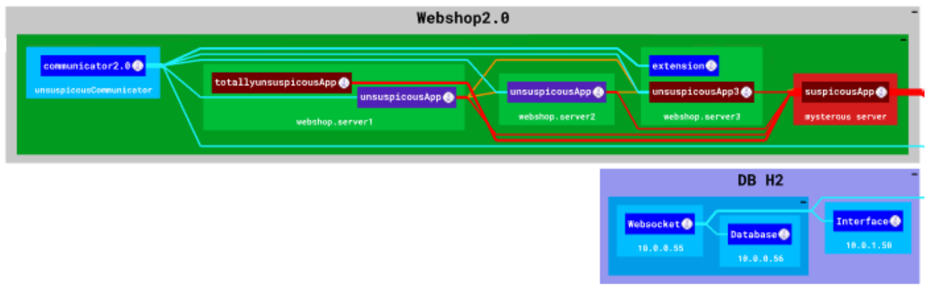

Fig. 5. Screenshot of the ACC extension of *ExplorViz* user interface showing the architecture conformance visualization

Visualizing Architecture Comparison Identifying architectural changes between two visualizations of a complex software application is a challenging task, which can be supported by appropriate tooling. Although *ExplorViz* visualizes the behavior and thus the runtime architecture of a software system, it is not possible to compare two versions. In this extension one student developed an approach to perform a visual software architecture comparison of two monitored applications (e.g., indicating a removed or changed component). This facilitates a developer to see at a glance which parts of the architecture have been added, deleted, modified, or remained unchanged between the two versions. Finally, an evaluation based on a qualitative usability study with an industrial partner was conducted. Five professional software engineers participated in the study and solved comparison tasks. The evaluation showed that the extension is applicable in the task of solving architecture comprehension tasks with different versions.

6 Restructured Architecture and new Process

Our modularization approach started by dividing the old monolith into separated frontend and backend projects [28]. Since then, we further decomposed our backend into several microservices to address the problems stated in Section 2. The resulting, restructured architecture and the new development process are introduced in this section.

6.1 Extensibility & Integrability

As previously stated, frontend extensions are based on *Ember's* addon mechanism. This approach works quite well for us as shown in Section 5. The backend, however, used the package scanning feature of the Jersey framework to include extensions. The result of this procedure was again an unhandy configuration of a monolithic application with high coupling of its modules. Therefore, we once again restructured the approach for our backend plug-in extensions. In Figure 6 we can see that the extensions are now decoupled and represent separated microservices. As a result, each extension is responsible for its own data persistence and error handling.

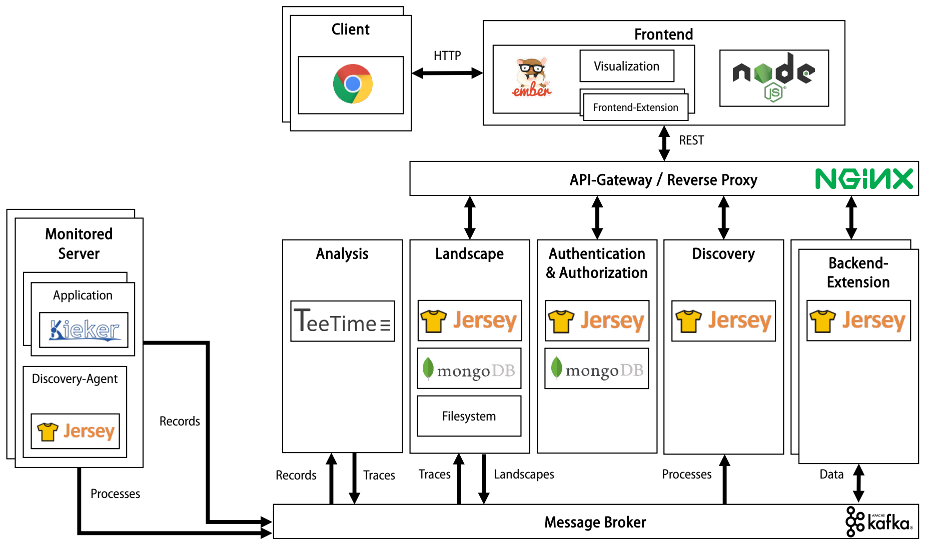

Fig. 6. Overview and software stack of our restructured architecture

Due to the decomposition of the backend, we are left with multiple URIs. Furthermore, new extensions will introduce additional endpoints, therefore more URIs again. To simplify the data exchange handling based on those endpoints, we now employ a common approach for microservice-based backends. As shown in Figure 6, the *Ember*-based frontend communicates with an API gateway instead of several single servers, thus only a single URI. This gateway, an *Nginx* reverse proxy,[5] passes requests based on their URI to the respective proxied mi-

[5] `https://www.nginx.com`

croservice, e.g., the landscape service. Extension developers who require a backend component extend the gateway's configuration file, such that their frontend extension can access their complement.

Some extensions must read data from different services. In the past, we used HTTP requests to periodically obtain this data. Each request was processed by the providing service, therefore introducing unnecessary load. As shown in Figure 6, inter-service communication is now realized with the help of Apache Kafka.[6] *Kafka* is a distributed streaming platform with fault-tolerance for loosely coupled systems. We use *Kafka* for events that might be interesting for upcoming microservices. For example, the landscape service consumes traces from the respective *Kafka* topic and produces a new landscape every tenth second for another topic. Microservices can consume the topic, obtain, and process the data in their custom way. As a result, the producing service does not have to process unnecessary HTTP requests, but simply fires its data and forgets it. Simple CRUD operations on resources, e.g., users and their management, are provided by means of RESTful APIs by the respective microservices. The decomposition into several independent microservices and the new inter-service communication approach both facilitate low coupling in our system.

6.2 Code Quality & Comprehensibility

The improvements for code quality and comprehensibility, which were introduced in our first modularization approach, showed a perceptible impact on contributor's work. For example, recurring students approved the easier access to *ExplorViz* and especially the obligatory exchange format *JSON:API*. However, we still lacked a common code style in terms of conventions and best practices. To achieve this and therefore facilitate maintainability, we defined compulsory rule sets for the quality assurance tools Checkstyle[7] and PMD.[8] In addition with SpotBugs,[9] the successor of FindBugs, we impose their usage on contributors. These tools can be used for Java. For JS, we currently employ ESLint,[10] i.e., a static analysis linter, with an *Ember* community-driven rule set. The latter contains best practices for *Ember* applications and rules to prevent programming flaws. In the future, we are going to enhance this rule set with our custom guidelines.

All of these tools are integrated into our continuous integration builds. We employ TravisCI[11] for *ExplorViz*'s core and any extension to build, test, and examine the code. Therefore, if a threshold of quality assurance problems is exceeded, the respective TravisCI build will fail and the contributor is notified by mail. A similar build is started for each pull request that we receive on GitHub

[6] `https://kafka.apache.org`
[7] `http://checkstyle.sourceforge.net`
[8] `https://pmd.github.io`
[9] `https://spotbugs.github.io`
[10] `https://eslint.org`
[11] `https://travis-ci.org`

for the now protected master branch. Therefore, contributors must now create a new branch or fork *ExplorViz* to implement their enhancement or bug fix and eventually submit a pull request.

6.3 Software Configurations and Delivery

One major problem of *ExplorViz Legacy* was the necessary provision of software configurations for different use cases. The first iteration of modularization did not entirely solve this problem. The backend introduced a first approach for an integration of extensions, but their delivery was cumbersome. Due to the tight coupling at source code level we had to provide the compiled Java files of all extensions for download. Users had to copy these files to a specific folder in their already deployed *ExplorViz* backend. Therefore, configuration alterations were troublesome.

With the architecture depicted in Figure 6 we can now provide a jar file for each service with an embedded web server. This modern approach for Java web applications facilitates delivery and configuration of *ExplorViz*'s backend components. In the future, we are going to ship ready-to-use Docker images for each part of our software. The build of these images will be integrated into the continuous integration pipeline. Users are then able to employ docker-compose files to achieve their custom *ExplorViz* configuration or use a provided docker-compose file that fits their needs. As a result, we can provide an alternative, easy to use, and exchangeable configuration approach that only requires a single command line instruction.

The frontend requires another approach, since (to the best of our knowledge) it is not possible to install an *Ember* addon inside of a deployed *Ember* application. We are currently developing a build service for users that ships ready-to-use, pre-built configurations of our frontend. Users can download and deploy these packages. Alternatively, these configurations will also be usable as Docker containers.

7 Related Work

In the area of software engineering, there are many papers that perform a software modernization in other contexts (e.g., software maintenance or reverse engineering). Thus, we restrict our related work to approaches, which focus on the modernization of monolithic applications towards a microservice architecture.

In [17], the authors conducted industrial case studies concerning the evolution of long-living software systems. The addressed legacy software systems were scarcely documented, which made an modernization challenging. The case study involved an architecture recovery and planning and execution of several evolutionary iterations. Compared to our approach, we did not reconstruct the underlying software architecture, since it was not our goal to keep the obsolete monolithic architecture provided by GWT. Furthermore, we did not need to apply multiple refactoring iterations to modernize our software system.

[26] evaluates monolithic and microservice architectures regarding the development and cloud deployment of enterprise applications. Their approach addresses similar elements to our modernization process. They employed modern technologies for separating microservices, e.g., Java in the backend and JS in the frontend, like we did. Contrary to their results, we did not face any of the mentioned problems during the migration, like failures or timeouts.

In [4] an approach regarding the challenges of the modernization of legacy J2EE applications was presented. They employ static code analysis to reconstruct architectural diagrams, which then can be used as a starting point during a modernization process. In contrast to our approach there was need for us to reconstruct the software architecture, because we wanted to modernize it from the beginning due to previously mentioned drawbacks. Thus, we split our application based on our knowledge into several microservices and developed a communication concept based on a message broker.

[20] presented a migration process to decompose an existing software system into several microservices. Additionally, they reported from their gained experiences towards applying their presented approach in a legacy modernization project. Although their modernization drivers and goals are similar to our procedure, their approach features a more abstract point of view on the modernization process. Furthermore, they focus on programming language modernization and transaction systems. Our focus is to facilitate the collaborative development of open source software and also addresses the development process.

[2] presented a survey containing architectural smells during a modernization towards a microservice architecture. They identified nine common pitfalls in terms of bad smells and provided potential solutions for them. *ExplorViz Legacy* was also covered by the survey and categorized by the "Single DevOps toolchain" pitfall. This pitfall concerns the usage of a single toolchain for all microservices. Fortunately, we addressed this pitfall since their observation during their survey by employing independent toolchains by means of pipelines within our continuous integration system for the backend and frontend microservices. We are further planning to develop our pipeline towards continuous delivery for all microservices mentioned in Section 6 to minimize the release cycles and offer development snapshots.

8 Conclusions

In this paper, we report on our modularization and modernization process of the open source research software *ExplorViz*, moving from a monolithic architecture towards a microservice architecture with the primary goal to ease the collaborative development, especially with students. We described technical and development process related drawbacks of our initial project state until 2016 in *ExplorViz Legacy* and illustrated our modularization process and architecture. The process included not only a decomposition of our web-based application into several components, but also technical modernization of applied frameworks and libraries. Driven by the objective to easily extend our project in the future and facilitate a contribution by inexperienced collaborators, we offer a plug-in extension mechanism for our core project, both for the backend and the frontend.

We realized our modularization process and architecture in terms of a proof-of-concept implementation and evaluated it afterwards by the development of several extensions of *ExplorViz*. Each of these extensions was developed by students and evaluated afterwards, in each case by at least a usability study. The results showed an overall good usability of each extension, and in the case of our application discovery extension, we integrated it into our core project.

However, the modularization process is not fully completed, as yet. We are still improving the project, as described in Section 6, in order to achieve a fully decoupled microservice architecture, consisting of a set of self-contained systems and well-defined interfaces in-between. The inter-service communication is handled via the message broker *Kafka* and the requests from the frontend towards the backend are passed trough our reverse-proxy in form of *Nginx*.

In the future, we are planning to evaluate our finalized project, especially in terms of developer collaboration. Additionally, we plan to move from our continuous-integration pipeline towards a continuous-delivery environment. Thus, we expect to decrease the interval between two releases and allow users to try out new versions, even development snapshots, as soon as possible. Furthermore, we plan to use architecture recovery tools like [13] for refactoring or documentation purposes in upcoming versions of *ExplorViz*.

References

1. Alshuqayran, N., Ali, N., Evans, R.: A Systematic Mapping Study in Microservice Architecture. In: Proceedings of the 9th International Conference on Service-Oriented Computing and Applications (SOCA). pp. 44–51 (Nov 2016). https://doi.org/10.1109/SOCA.2016.15
2. Carrasco, A., Bladel, B.v., Demeyer, S.: Migrating Towards Microservices: Migration and Architecture Smells. In: Proceedings of the 2nd International Workshop on Refactoring. pp. 1–6. IWoR 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3242163.3242164
3. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, Today, and Tomorrow, pp. 195–216. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-67425-4_12
4. Escobar, D., Cárdenas, D., Amarillo, R., Castro, E., Garcés, K., Parra, C., Casallas, R.: Towards the understanding and evolution of monolithic applications as microservices. In: Proceedings of the XLII Latin American Computing Conference (CLEI). pp. 1–11 (Oct 2016). https://doi.org/10.1109/CLEI.2016.7833410
5. Fittkau, F., Finke, S., Hasselbring, W., Waller, J.: Comparing Trace Visualizations for Program Comprehension through Controlled Experiments. In: Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015). pp. 266–276 (Mai 2015). https://doi.org/10.1109/ICPC.2015.37
6. Fittkau, F., Krause, A., Hasselbring, W.: Exploring software cities in virtual reality. In: Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015). pp. 130–134. IEEE (Sep 2015). https://doi.org/10.1109/VISSOFT.2015.7332423

7. Fittkau, F., Krause, A., Hasselbring, W.: Hierarchical software landscape visualization for system comprehension: A controlled experiment. In: Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015). pp. 36–45. IEEE (Sep 2015). https://doi.org/10.1109/VISSOFT.2015.7332413
8. Fittkau, F., Krause, A., Hasselbring, W.: Software landscape and application visualization for system comprehension with ExplorViz. Information and Software Technology **87**, 259–277 (Juli 2017). https://doi.org/10.1016/j.infsof.2016.07.004
9. Fittkau, F., Roth, S., Hasselbring, W.: ExplorViz: Visual runtime behavior analysis of enterprise application landscapes. In: 23rd European Conference on Information Systems (ECIS 2015 Completed Research Papers). pp. 1–13. AIS Electronic Library (May 2015). https://doi.org/10.18151/7217313
10. Fittkau, F., Stelzer, P., Hasselbring, W.: Live visualization of large software landscapes for ensuring architecture conformance. In: Proceedings of the 2014 European Conference on Software Architecture Workshops (ECSAW 2014). pp. 28:1–28:4. ACM (2014). https://doi.org/10.1145/2642803.2642831
11. Francesco, P.D., Lago, P., Malavolta, I.: Migrating Towards Microservice Architectures: An Industrial Survey. In: Proceedings of the IEEE International Conference on Software Architecture (ICSA). pp. 29–2909 (April 2018). https://doi.org/10.1109/ICSA.2018.00012
12. Goble, C.: Better Software, Better Research. IEEE Internet Computing **18**(5), 4–8 (Sept 2014). https://doi.org/10.1109/MIC.2014.88
13. Granchelli, G., Cardarelli, M., Francesco, P.D., Malavolta, I., Iovino, L., Salle, A.D.: MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems. In: Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 298–302 (April 2017). https://doi.org/10.1109/ICSAW.2017.9
14. Hasselbring, W., Steinacker, G.: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 243–246 (April 2017). https://doi.org/10.1109/ICSAW.2017.11
15. Heinrich, R., Zirkelbach, C., Jung, R.: Architectural Runtime Modeling and Visualization for Quality-Aware DevOps in Cloud Applications. In: Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 199–201 (April 2017). https://doi.org/10.1109/ICSAW.2017.33
16. Heinrich, R., Jung, R., Zirkelbach, C., Hasselbring, W., Reussner, R.: An architectural model-based approach to quality-aware DevOps in cloud applications. In: Mistrik, I., Bahsoon, R., Ali, N., Heisel, M., Maxim, B. (eds.) Software Architecture for Big Data and the Cloud, pp. 69–89. Elsevier, Cambridge (Jun 2017). https://doi.org/10.1016/B978-0-12-805467-3.00005-3
17. van Hoorn, A., Frey, S., Goerigk, W., Hasselbring, W., Knoche, H., Köster, S., Krause, H., Porembski, M., Stahl, T., Steinkamp, M., Wittmüss, N.: Dynamod project: Dynamic analysis for model-driven software modernization. In: Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011). vol. 708, pp. 12–13. CEUR (2011)
18. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In: Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012). pp. 247–248. ACM (April 2012). https://doi.org/10.1145/2188286.2188326

19. Johanson, A., Hasselbring, W.: Software engineering for computational science: Past, present, future. Computing in Science & Engineering **20**(2), 90–109 (Mar 2018). https://doi.org/10.1109/MCSE.2018.021651343
20. Knoche, H., Hasselbring, W.: Using Microservices for Legacy Software Modernization. IEEE Software **35**(3), 44–49 (May 2018). https://doi.org/10.1109/MS.2018.2141035
21. Krause, A., Zirkelbach, C., Hasselbring, W.: Simplifying Software System Monitoring through Application Discovery with ExplorViz. In: Proceedings of the Symposium on Software Performance 2018: Joint Developer and Community Meeting of Descartes/Kieker/Palladio (November 2018), `http://eprints.uni-kiel.de/44502/`
22. Le, D.M., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A., Medvidovic, N.: An Empirical Study of Architectural Change in Open-Source Software Systems. In: Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR). pp. 235–245 (May 2015). https://doi.org/10.1109/MSR.2015.29
23. Le, D.M., Link, D., Shahbazian, A., Medvidovic, N.: An Empirical Study of Architectural Decay in Open-Source Software. In: Proceedings of the IEEE International Conference on Software Architecture (ICSA). pp. 176–17609 (April 2018). https://doi.org/10.1109/ICSA.2018.00027
24. Tang, A., Razavian, M., Paech, B., Hesse, T.: Human Aspects in Software Architecture Decision Making: A Literature Review. In: Proceedings of the IEEE International Conference on Software Architecture (ICSA). pp. 107–116 (April 2017). https://doi.org/10.1109/ICSA.2017.15
25. Upadhyaya, B., Zou, Y., Xiao, H., Ng, J., Lau, A.: Migration of SOAP-based services to RESTful services. In: Proceedings of the 13th IEEE International Symposium on Web Systems Evolution (WSE). pp. 105–114 (Sept 2011). https://doi.org/10.1109/WSE.2011.6081828
26. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: Proceedings of the 10th Computing Colombian Conference (10CCC). pp. 583–590 (Sept 2015). https://doi.org/10.1109/ColumbianCC.2015.7333476
27. Vinoski, S.: RESTful Web Services Development Checklist. IEEE Internet Computing **12**(6), 96–95 (Nov 2008). https://doi.org/10.1109/MIC.2008.130, `http://dx.doi.org/10.1109/MIC.2008.130`
28. Zirkelbach, C., Krause, A., Hasselbring, W.: On the Modernization of ExplorViz towards a Microservice Architecture. In: Combined Proceedings of the Workshops of the German Software Engineering Conference 2018. vol. Online Proceedings for Scientific Conferences and Workshops. CEUR Workshop Proceedings, Ulm, Germany (Februar 2018), `http://eprints.uni-kiel.de/42119/`