On the Modernization of ExplorViz towards a Microservice Architecture

Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring
Software Engineering Group
Kiel University, Germany
{czi,akr,wha}@informatik.uni-kiel.de

Abstract

Software systems evolve during their lifetime and therefore face several challenges. Changing requirements or upcoming feature requests make modifications or extensions inevitable. Especially long-living software systems have often been built as monolithic applications and are based on obsolescent architectures and technologies. This circumstance makes it difficult for developers to maintain or extend software. In this paper, we report on the modernization process of our open source research project ExplorViz – moving from a monolithic towards a microservice architecture. We describe our previous version within the project and present how we solved the modernization and handled occurring problems. Afterwards, we illustrate our modernized software system and point out the obtained benefits. Finally, we delineate open questions for the ongoing development.

1 Introduction

Software systems evolve over time and encounter difficulties during their life cycle. Often these systems are modified or extended, induced by new requirements or upcoming requests from customers. In the context of long-living software systems, these systems have often been built in form of monolithic applications and are comprised of obsolescent architectures and technologies. A key problem of monolithic applications is that all components are developed on a single codebase among several developers. This basically means, that if a developer wants to modify code or add a new feature, he needs to be certain that the remaining code and provided services are still working after his changes [10]. A solution to this problem can be employing a different architectural style, namely a microservice architecture, composed of several self-contained systems [9]. This style offers more flexibility and scalability on the one hand and replaceability of single components on the other hand [12]. Since 2012, we develop the open source research project ExplorViz,[1] a web-based monitoring and visualization tool for large software landscapes [11]. ExplorViz features two different visualizations – an abstract software landscape and a detailed application level visualization, which are built-upon collected monitoring information. Since the first version, we have continually developed and improved our software, e.g., by replacing single components or adding new features. Thus, it was inevitable to make changes to the code and even architectural amendments. This circumstance made it more and more difficult over time, to maintain and extend our software, especially for external developers, e.g., computer science students. Overcoming these problems was our initial incentive for the modernization. Similar decision triggers for developers in other projects are depicted in [1]. We performed an architectural modernization of our open source project ExplorViz and provided a simple way to enhance our software via extensions.

The remainder of this paper is organized as follows. In Section 2, we describe the monolithic architecture of our open source project, referred to as *ExplorViz Legacy*, and point out problems during development. Afterwards, we present our approach to address the presented problems in Section 3. In Section 4, we discuss related work regarding our approach. Finally, the conclusions are drawn and open questions are delineated.

2 ExplorViz Legacy

The idea behind ExplorViz was initially conceptualized in 2012 and first published a year later [6]. Since then, the project has evolved greatly in feature count, source lines of code, and research interests. For instance, we investigated alternative visualization approaches with cutting-edge input and output devices in the context of program comprehension [7, 8]. These have been developed in terms of extensions, e.g., a Virtual Reality mode. Most of these extensions are the result of student's theses or seminar papers. As of today we count a growing number of twenty student's theses and more than ten *Git* branches in the context of the visualization functionality, i.e., landscape and application renderings.

[1] `https://www.explorviz.net`

Java and its Remote Procedure Calls (RPC) are taught early in lectures, therefore we utilized the Google Web Toolkit (GWT) as primary web framework for our project. GWT enables writing Java code for both server- (backend) and client-logic (frontend) in a single project. It compiles client-related Java- to respective JavaScript-code (JS), thus enabling the execution in web browsers. Additionally, the toolkit introduces GWT RPC (GRPC) for triggering actions on the server or exchanging data over HTTP. Therefore, client-server communication is easily usable for non-professional developers and does not require manual parsing of Java objects to obtain a common transport format, e.g., JavaScript Object Notation (JSON). This particular technology of network communication eases the development, especially for our students.

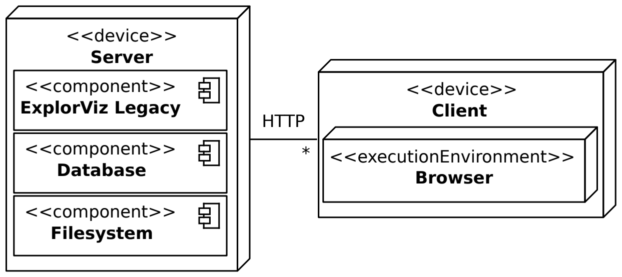

Figure 1: Architectural overview of *ExplorViz Legacy*

In Figure 1 the deployment and simplified software stack of our GWT-based *ExplorViz Legacy* is shown. It can be deployed on a single server node. On startup, *ExplorViz Legacy* automatically creates a database for user management. The filesystem is facilitated to store serialized landscape objects, i.e., the underlying data models, that are retrieved from monitoring data and used for visualization.

The presented project setup was used since 2012 and published in 2013 on Github.[2] During subsequent development we frequently migrated client code from Java to JS using GWT's JavaScript Native Interface (JSNI), i.e., embedded JS code in Java methods. The reason behind this alteration of GWT's intended workflow was the utilization of modern JS libraries to simplify the usability for users. The result was a fragmentation of ExplorViz's codebase in JSNI- and Java-methods. This was further deteriorated when we substituted GWT's WebGL implementation with the JS-based library *three.js*. *three.js* provides a high level of abstraction for 3D rendering and thus offers a better maintainability and extensibility for new developers.

In 2016 we stopped the development of new features in ExplorViz. GWT seemed to disappear in a variety of other projects. Additionally, there was no major update of the toolkit for at least a year. Meanwhile Google released a new programming language which can also be used for web development, called *Dart*. This language is used by Google itself to build many applications as noted on the related website.[3] Since it was announced that JSNI will be removed with the release of GWT 3, we were in need of migrating code once again. At this time *ExplorViz Legacy* contained a great amount of JS code. Therefore, we decided to drop GWT as scaffold and modernize the monolithic project with new technologies and less dependencies to modules of the underlying web framework.

3 Modernization Procedure

Two major communication technologies emerged from practical realization when implementing web services. Research shows that SOAP-based services are in fact less performant and do not support mobile devices as good as their RESTful counterparts [5]. Furthermore, the latter eases the development and influences the characteristics of a system, e.g., scalability and flexibility [3, 5].

In [13] the authors present how the German e-commerce provider *Otto* modernized the underlying software system of their online shop. Instead of refactoring the old monolithic system, they completely re-implemented the functionality, using a microservice architecture. The developed microservices communicate only by accessing REST APIs. This redesign resulted in a highly scalable and fault tolerant software system.

The previously mentioned issues in *ExplorViz Legacy* (see Section 2) and the experience reports about successful utilization of alternative technologies, e.g., RESTful APIs, were triggers for a modernization of the ExplorViz project. We no longer saw advantages of preferring GWT over other web frameworks. Therefore, we decided to split the codebase into two separated projects, i.e., backend and frontend. The backend is implemented as a Java-based web service providing a RESTful API for clients. Since client-side code is mostly written in JS nowadays, we choose this programming language for the frontend.

Figure 2 depicts the new architecture and simplified software stack of ExplorViz. Backend and frontend are now two self-contained microservices. Thus, they can be deployed on different server nodes. In detail, we employ distinct technology stacks with integrated storage. This allows us to exchange a single or both microservices, as long as we take our specified interfaces into account.

The backend provides a RESTful API for frontend instances and is based on the Jersey framework,[4] which implements the Servlet 3.0 specification. This is utilized to implement a web service without the need to state a *web.xml* file, i.e., the servlet configuration file. Instead, we use javax.servlet.annotations to

[2]`https://github.com/ExplorViz/Explorviz`

[3]`https://webdev.dartlang.org`

[4]`https://jersey.github.io`

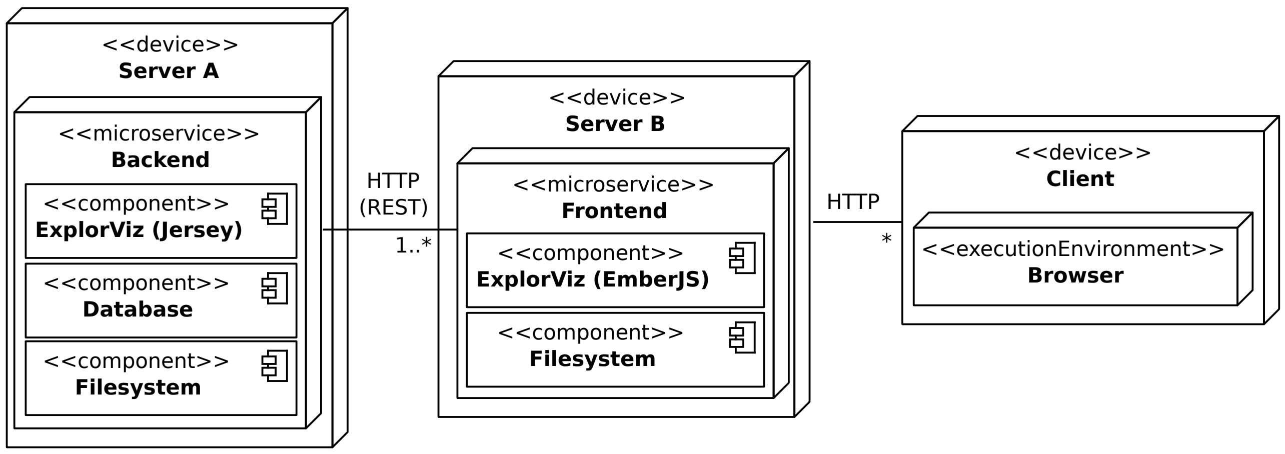

Figure 2: Overview of the modernized ExplorViz architecture

define servlet declarations and mappings. We expect this approach will ease the development, especially for students.

ExplorViz's new frontend uses the client-side JS framework Ember.js (Ember), [5] which allows us to use to provide software visualizations with a WebGL-enabled browser. Ember is based on the Model View ViewModel architectural pattern. As a result, manual Document Object Model accesses are not necessary and developers need less code. Ember allows and emphasizes the use of components in web sites, i.e., self-contained, reusable, and exchangeable user interface fragments. We employ this feature to encapsulate visualization modes. Therefore, they can be included, containing all necessary logic by inserting a single line of code. Network communication, e.g., fetching a landscape from the backend, is abstracted by so-called adapters. These make it easy to send or request data by using convention over configuration, if the backend applies the same rules for URL definitions.

The introduced microservices represent the core of ExplorViz. As for future extensions, we implemented clean and comprehensive interfaces for both components, that allow the registration of extensible functionalities. A student implementing new mechanics will therefore use a template extension as starting point. Those extensions access core mechanics only by a defined read-only API, which is implemented by the backend, respectively frontend. The modularization enables us to improve the backend or frontend, while not breaking extension support.

In summary, both frameworks are exchangeable with respect to their language domain. The backend would primarily need to define new ways to provide data. Since client-side JS frameworks have similar elements and approaches, we think substituting Ember can be done with little effort.

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

4 Related Work

In [2], the authors conduct a case study addressing the evolution of a software system, which has been scarcely documented. The case study involves architecture recovery and planning and execution of several evolution cycles. Compared to our approach, we did not recover the architecture, since we did not want to keep the obsolete monolithic architecture, which was provided by GWT. Furthermore, we did not need to apply a series of refactoring iterations to modernize our software system.

[10] compares the development and cloud deployment of an enterprise application based on a monolithic approach and a microservice architecture. Their approach contains common elements to our applied process. They employ modern technologies for separate microservices, e.g., Java in the backend and JS in the frontend. Contrary to their results, we did not face any of the mentioned problems during the migration, like failures or timeouts.

According to [5], RESTful services can improve system flexibility, scalability, and performance in comparison to SOAP-based web services. Additionally, REST-based services are easier to consume and compose, based on well-defined standards and heterogeneous operations. They provide an approach to migrate SOAP-based to RESTful services. Unfortunately, their approach is not applicable for us, since our project is based on GWT instead of SOAP.

In [4], the authors present a survey of various approaches to move from legacy systems towards a Service-Oriented-Architecture (SOA) environment. Basically, they distinguish between four different categories – *replacement*, *wrapping*, *redevelopment*, and *migration*. While *Replacement* is self-explanatory, *wrapping* employs a new interface for existing components to make them accessible in form of services. *Redevelopment* employs reverse and reengineering approaches to add necessary functionality to the legacy system. Finally, a *migration* moves a legacy system to the more flexible environment. Thus, the original system's data and functionality can be preserved. Based on the type of legacy system, tool support, and further criteria, a different technique or strategies can be employed. Unfortunately, their present migration and redevelopment strategies are not adaptable for our process, since these focus on SOA environments instead of microservices.

5 Conclusions

In this paper, we report on our modernization process of ExplorViz from a monolithic towards a microservice architecture. We pointed out encountered problems during our development since 2012, especially those related to the architecture underneath our software. Consequently, we described *ExplorViz Legacy*, the previous version of our open source research project, and presented solutions to address the existing problems. Afterwards, we revealed our modernized software system and emphasized the obtained benefits. Even though our modernization process is still in progress, we were already able to employ a microservice architecture in order to ease maintainability on one hand, and extensibility on the other hand. Finally, we would like to delineate some of our open questions:

- How can we derive best practice guidelines from our migrations for other projects?
- Does the rapid evolution of JS frontend frameworks influence the ongoing evolution of ExplorViz?
- How can we reposition ExplorViz as an open source visualization framework for diverse data, as exemplarily shown in [14]?

References

[1] J. Koskinen et al. “Software Modernization Decision Criteria: An Empirical Study”. In: *Proceedings of the 9th European Conference on Software Maintenance and Reengineering.* Mar. 2005, pp. 324–331.

[2] F. Cuadrado et al. “A Case Study on Software Evolution towards Service-Oriented Architecture”. In: *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications.* Mar. 2008, pp. 1399–1404.

[3] S. Vinoski. “RESTful Web Services Development Checklist”. In: *IEEE Internet Computing* 12.6 (Nov. 2008), pp. 96–95.

[4] A. A. Almonaies, J. R. Cordy, and T. R. Dean. “Legacy system evolution towards service-oriented architecture”. In: *Proceedings of the International Workshop on SOA Migration and Evolution.* 2010, pp. 53–62.

[5] B. Upadhyaya et al. “Migration of SOAP-based services to RESTful services”. In: *Proceedings of the 13th IEEE International Symposium on Web Systems Evolution (WSE).* Sept. 2011, pp. 105–114.

[6] F. Fittkau et al. “Live trace visualization for comprehending large software landscapes: The ExplorViz approach”. In: *Proceedings of the First IEEE Working Conference on Software Visualization (VISSOFT).* Sept. 2013, pp. 1–4.

[7] F. Fittkau, A. Krause, and W. Hasselbring. “Exploring software cities in virtual reality”. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT).* Sept. 2015, pp. 130–134.

[8] F. Fittkau, E. Koppenhagen, and W. Hasselbring. “Research Perspective on Supporting Software Engineering via Physical 3D Models”. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT).* IEEE, Sept. 2015, pp. 125–129.

[9] S. Newman. *Building microservices: designing fine-grained systems.* O'Reilly, 2015.

[10] M. Villamizar et al. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *Proceedings of the 10th Computing Colombian Conference (10CCC).* Sept. 2015, pp. 583–590.

[11] F. Fittkau, A. Krause, and W. Hasselbring. “Software landscape and application visualization for system comprehension with ExplorViz”. In: *Information and Software Technology* (2016). http://dx.doi.org/10.1016/j.infsof.2016.07.004.

[12] T. Salah et al. “The evolution of distributed systems towards microservices architecture”. In: *Proceedings of the 11th International Conference for Internet Technology and Secured Transactions (ICITST).* Dec. 2016, pp. 318–325.

[13] W. Hasselbring and G. Steinacker. “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce”. In: *Proceedings of the International Conference on Software Architecture Workshops (ICSAW).* Apr. 2017, pp. 243–246.

[14] C. Zirkelbach. “Juggling with Data: On the Lack of Database Monitoring in Long-Living Software Systems”. In: *Proceedings of the 4th Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems (EMLS).* Softwaretechnik-Trends 2. 2017, pp. 62–65.