A customizable and extensible Tutorial Framework for improved Usability in ExplorViz

Master's Thesis

Helge Müller

June 22, 2019

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Christian Zirkelbach, 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, 22. Juni 2019

Abstract

Software development is rapidly increasing in importance and size. Open-source development is more important than ever. More and more open-source software is used in companies. This causes more people to use the software. Feature the users are not aware about are not used, therefore software is only as good as the introduction of its features. Providing instructions for features is therefore an essential part of every software.

ExplorViz is an open-source software which provides visualizations for live trace data. We develop an extensible and customizable tutorial framework for *ExplorViz*. The framework is developed for the current version of *ExplorViz* which is still under heavy development. The framework is an extension to *ExplorViz* which allows for the tutorials to be executed in the same context as the visualizations are used.

The focus of development for this framework is to provide a structure which will by itself provide basic tutorial functionality. Considering extensibility has a large influence in our decisions during the development process. We describe the overall structure of *ExplorViz* and how the different components are relevant for the development of the tutorial framework.

A tutorial framework which is customizable to suit automatic instructions during experiments is valuable, since *ExplorViz* is also frequently used in research. These features should however not intervene with the instruction of other users. We design the tutorial framework as an extension for *ExplorViz*, therefore either an extension to the framework or a customized variant could be created to exactly match the requirements for the experiments. These extension capabilities are not limited to the use in experiments, other customizations are possible.

The usability of the framework is evaluated by questionnaires answered by participants after executing an example tutorial. This determines if the tutorials usability is as high as expected. The tutorial editor is evaluated similarly with the same participants.

The work provides an overview about development of extensions with *ExplorViz* and highlights the potential problems, solutions and principles to follow when developing extensions for *ExplorViz*.

Contents

Chapter 1

Introduction

1.1 Motivation

ExplorViz is a monitoring and software visualization tool. To efficiently use the tool one needs to be familiar with the functionalities which it provides. When first using the tool, users have to familiarize themselves with the application. Depending on the method by which they do this, different levels of knowledge will be achieved. It is desirable to have users with good knowledge about the tool. That allows the users to use the software efficiently for the intended purpose. Educated users are able to provide more constructive feedback. This is especially helpful when the software is developed as an open-source project, since feedback regarding misunderstood features is minimized. It will also allow new developers on the project to get a overview of features provided by the software.

Achieving a high level of knowledge is possible by providing documentation. Depending on what kind of documentation is provided it is time consuming to produce and consume. We therefore propose to use tutorials to educate the users and developers. These tutorials will be provided by a tutorial framework which was developed for *ExplorViz*. *ExplorViz* does support extensions, and these extensions themselves have additional features, which might require tutorials to be explained. Providing tutorials for new extensions is only feasible if the provided tutorial framework is extensible and customizable. Due to the large variation of extension which could possibly be developed and therefore features which need to be demonstrated, a tutorial framework would not be useful without the possibility to customize and extend it. A customizable framework however allows for the developers of extensions to provide tutorials which explain its features.

It is possible to customize the framework to work in a virtual environment as described by [Zirkelbach et al. 2019a]. Since the tutorial framework will allow to implement specific tutorials for functionality in *ExplorViz*, a tutorial for the *Application Discovery* mechanism [Krause et al. 2018] could be implemented. Furthermore *ExplorViz* is used to visualize monitored software [Krause et al. 2018; Zirkelbach et al. 2015], we provide a framework to create introductions for these visualizations.

The tutorial framework is an optional extension for *ExplorViz* and provides a system to manage tutorials. The framework is developed for the latest version of *ExplorViz* which ensures that the framework is compatible with new versions of *ExplorViz*. We are not aiming to provide a suite of tutorials, we rather are providing the tools to construct and use tutorials. In this work we provide some insight to what is required to construct tutorials. The general structure of and pitfalls to avoid when developing extensions for *ExplorViz* are explained. We discuss our approach and in what cases we had to adjust it. A working framework is developed and its usability evaluated using an example tutorial, which is created in the developed tutorial editor. The usability of the tools developed for creation of tutorials is also evaluated. The processes necessary to develop the framework and the tutorial editor are outlined in this work.

1.2 Goals

In this section we describe the goals are set to be achieved in this work.

1.2.1 G1: Literature / Tool Research

An overview of literature concerning *ExplorViz* and tools used by *ExplorViz* is assembled and analyzed. This serves as a foundation to what is implemented.

1.2.2 G2: Develop a Data Model for Tutorials

We first have to identify the relevant features to create a data structure for the tutorials. This enables us to determine the necessary structures to ensure needed information can be persisted using the data structure. Additional features might be added later but a minimal set of features will help to decide on the right implementation for the data structure.

1.2.3 G3: Implement a Modular Tutorial Extension

The features identified in Section 1.2.2 will be implemented by creating two extensions that will provide the tutorial framework. There will be a backend and a frontend extension. The backend will provide the tutorial data and the frontend will display it to the user. The frontend will also register when the user is executing the actions to proceed in the tutorial.

A data structure will be created that can be used to export and import tutorials from and into a database. This will ensure that Tutorials can be transferred between *ExplorViz* Instances and developed independently. This would also enable a selection of tutorials to be provided for general use, for example as downloadable files on a website. It also allows for tutorials to be managed via *Source Code Management*.

1.2.4 G4: Implement a Tutorial-Management-Tool / Editor

A tutorial editor will be provided that allows the user to create and edit tutorials. The actions required to proceed in the tutorial will be defined using the editor. Instruction text and titles can also be edited.

Different user roles exist in *ExplorViz* and the usable features differ between them, therefore we have to supply the possibility to enable and disable, the relevant features. This is also relevant if some extension will be later uninstalled, with the tutorials remaining in the database. Therefore we will implement a tutorial-management-tool, integrated into *ExplorViz* to enable and disable parts of the tutorial, and configure if a tutorial is required for a given role.

1.2.5 G5: Evaluation

There will be a questionnaire to evaluate the usability of the implementation. Subjects will test the example tutorial before answering the questionnaire. We aim to answer the question if the tutorial is helpful and find parts where it could be improved. The results of the questionnaire will be analyzed and suggested improvements might be implemented.

1.3 Document Structure

In Section 1.2 we listed the goals of this thesis. Then in Chapter 2 we introduce the technologies used when implementing the tutorial framework. We discuss different aspects of tutorials in Chapter 3.

ExplorViz recently switched from a monolithic structure to a service oriented approach. The new structure and general information about *ExplorViz* can be found in Chapter 4. We follow this new approach and implement a tutorial framework, which consists of a backend and a frontend extension, we describe them in Section 6.2 and Section 6.3 respectively. These extensions communicate using an *REST API* based on *JSON:API*, which is further explained in Chapter 2.

Technical limitation and information about the implementation can be found in Section 6.2 for the backend and Section 6.3 for the frontend.

The evaluation is explained in Chapter 7, here the example tutorial along with the execution of the experiment and it results are described. Related work is discussed in Chapter 8. Afterwards conclusions are drawn and future work is discussed in Chapter 9.

Chapter 2

Foundations and Technology

2.1 JavaScript and JQuery

JavaScript[1] is a well-known programming language that is not only used for client side scripts in browsers. It is also used as a programming language for servers and several frameworks are implemented in *JavaScript*. *JQuery*[2] is a library for *JavaScript*. It extends the functionality and simplifies commonly required tasks. Most often *JQuery* is used as a client side script in browsers. There it provides a method to select and interact with elements in the *DOM* via *CSS* like rules, which also supports *CSS Selectors*. This allows for easier adding and removing of *CSS* classes to achieve changes on a web page without reloading it. Event-handling is also simplified, and helper function to execute *AJAX* calls are provided.

JavaScript and *JQuery* are heavily used by *Ember* which itself is used to program the frontend of *ExplorViz*.

2.2 Ember

Ember.js[3] (from hereon referred to as *Ember*) is a framework based on *JavaScript* and *JQuery* for providing web applications. The *commandline* tool of *Ember* can be used to generate a file structure containing a structure to develop the application. Components of the application can be generated during development. This makes adding functionality to the application simple since parts can be added by a single command.

The developed application is then deployed on a server and delivers the scripts to the client. The rendering of the page and execution of the application will therefore happen on the clients machine. This means that *Ember* is a client-side framework. Not the entire application is send to the client at once, this would make it hard to provide security and implement authentication, since the pages would already be delivered to the client. Instead the application is using so called *client-side rendering* which means that the pages are constructed on the client and only authentication and data is send between server and client.

[1]https://www.javascript.com
[2]https://jquery.com
[3]https://www.emberjs.com

This allows *Ember* to be scalable since the workload is shifted to the clients and only the required communications are send to the server. This also allows for subsequent page calls to be handled entirely on the client side if the data is already present on the client.

Ember itself structures the application in *Routes*, *Templates*, *Models*, *Components*, and *Services*. There are more specific components in *Ember* that can be used to customize how the application behaves, these are however the basic building blocks of an *Ember* application. Routes map URLs to Controllers and are responsible to load a model, which is an object that contains parts of the information shown on the page. *Templates* are the components that generate the HTML presented to the user. *Models* provide data, which is then used by the *controllers* to determine how the page will be rendered. The *controllers* therefore provide the information needed by the *templates* to render the pages. *Services* are used to store information without having to pass it between controllers. *Services* can be used in controllers and if a service is defined every controller will use the same instance.

Ember is used in *ExplorViz* to provide the frontend. All user interactions are handled through this application.

2.2.1 Ember Data

Ember Data enables the developer to define data structures, called *models*, which can then be loaded via *JSON:API* and are automatically stored in the *Ember Store*. *Ember* therefore is able to reuse the data without reloading it from the backend multiple times. *Ember Data* also tracks if the model properties were changed and therefore would have to be saved.

Models are defined by extending the *Ember Model Object* and defining which properties are contained in the extended model. These properties are typed. *Ember Data* is able to connect to different APIs to exchange data with the backend. By default *Ember* is compatible with *JSON:API*, for more information see Section 2.7. It is possible to customize *Ember Data* to work with a very wide range of *API*s. This is done by implementing an adapter which specifies the endpoints of the *API*. Additionally a *serializer* can be implemented to affect the data before it is exported from or imported into the application.

2.2.2 Ember Templates

Ember provides templates which define what is shown on any given page. These templates could contain simple *HTML*. If the application is supposed to show data which is provided by *Ember Data*, it can use *handlebars* syntax to include values from the controller in the template.

Most applications will also reused certain components of a page. A prime example is a navigation on a web page. The navigation is mostly the same on every page, since its purpose is to provide an overview of possible pages. *Ember* does provide components to implement sections of pages which are reused. Components can be included in templates and will cause the component to execute their code and provide the data required to render the component.

Components also provide data for the view (which is created from templates), however they do not automatically associate with a route and therefore need to be called by the template. This appears as if it breaks the *Model-View-Controller* design pattern, since the view would be calling the controller. However the component itself has a template. This means the controller is not called by the template, the component is simply a subtemplate, which will execute its controller to ensure that the needed data is provided.

In our example of a navigation component, we want to highlight the current page. This can be achieved by binding a parameter to the component. Binding a variable to a component will cause the component to refresh when the variable is changed. This only occurs when the variable is changed via the setters provided by *Ember*, these should however always be used for setting values. With the bound variable we can now set which navigation component should be highlighted corresponding to the controller that was executed.

Another option to achieve the same functionality would utilize a *service*. Since the currently active page is an application wide state, we could implement a *service* that stores which page is active. This would allow our *component* to check this variable to highlight the corresponding navigation entry. This would not cause the *component* to update if the variable is changed. This is desirable if the variable should not be configurable, since the *component* itself needs to define which variable in which *service* is causing the effect. When the variable is bound it is simple to just bind another variable while adding the *component* to the *template*.

2.2.3 Ember Addons

Ember can be extended by *Ember Addons*. Addons are *Ember Applications* which are used in conjunction with other applications, they are included in the dependency section of the `package.json`. This will cause the addon to be installed and executed with the application. For development the addon can be linked to an application. This causes the application to run the addon from the source code, therefore changes are available to the application without restarting it.

Developing an *Ember Addon* follows mostly the same principles as developing a standalone app. A folder which is called addon contains the source code. A folder called app exist and contains files which are accessible to the application. To avoid that the entire source code of the addon is accessible by the application, it is contained in the addon folder, and the classes are then exported in the file contained in the app folder.

2.3 Java

The main programming language used in the *ExplorViz* backend is *Java*. The backend needs to provide data to the frontend. This is achieved by creating *Restful HTTP Endpoints* which accept requests for data and provide data in the *JSON:API* format. *JSON:API* is further explained in Section 2.7. These endpoints and the conversion from java classes to *JSON:API* responses are provided by the *Jersey framework*. *Jersey* allow us to use annotations to specify identifier and relationships inside the classes. This enables us to return a *Java Object* of the class and *Jersey* will convert the object into the *JSON:API* response.

Where *Jersey* is used for converting java objects into *JSON:API*, *Morphia*[4] is used to store the *JSON:API objects* into a *Mongo DB*[5]. More information about *Mongo DB* can be found in Section 2.4 and information about *Morphia* Section 2.5. We provide services that are used to persist the data in the database.

Another principle that is used in the services is *dependency injection*. Services and configuration files are defined in the `DependencyInjectionBinder`. The *dependency injection* binds classes, mostly implementations of interfaces, onto the interfaces. This enables using the interface as field in classes, without either specifying the implementation as type, nor the need to provide a constructor which contains the field. It also allows to easily exchange the implementation without the need to adjust the class. The only necessary step would be to exchange the implementation in the binding class. The implementation is then injected into the interface inside the classes. The injected fields are specified using the *Inject*-annotation. This annotation ensures that the field will be considered when the injection is executed. It is also possible to always inject the same instance of a class as a singleton. We utilize this in the backend services among other things to only instantiate one class to establish a database connection.

2.4 Mongo DB

Mongo DB is a highly compatible database system. It uses a *JSON*-like structure to persist the data as strings. Since it is text based and has a simple structure, it can be converted into constructs in most programming languages. Therefore it is often used in *APIs* to provide program language independent interfaces. *JSON:API* can be used to transfer data via text based *APIs*, it can however also be used to save data in a human readable and editable format.

2.5 Morphia

We utilize *Morphia* to persist the objects in the *Mongo DB*. *Morphia* allows to annotate relationships between classes and the id of a class. This is very similar to the annotations mentioned in Section 2.3. These classes are then used to insert the objects into the database. *Morphia* is using a *datastore* which is a singleton that contains functions to persist different types of objects. The *datastore* is injected into the *database service* which is used by the resource to persist the objects.

[4]https://github.com/MorphiaOrg/morphia
[5]https://www.mongodb.com/

Morphia does allow us to use annotations for other purposes. The *transient*-annotation can be used to specify fields that are supposed to be ignored when persisting the object. This is useful if the *JSON:API* object contains data that *Jersey* should parse but which should not be persisted into the database.

2.6 JSON

JavaScript Object Notation (*JSON*)[6] is a text based format to store *JavaScript Objects*. Since *JSON* is a string base dataformat it is also highly compatible with most programming languages. It is also widely used in web applications since, as the name implies, it works well with *JavaScript*.

2.7 JSON:API

JSON:API is a specification which was designed to simplify communication between servers. It is compatible with *JSON* since it uses the format, it however defines stricter rules on how data should be structured, among other it requires objects to have an id and a type. Another requirement is to split the type and id from relationships and attributes. The objects referenced via relationships can be included inside of the request.

[6] https://www.json.org

Chapter 3

Tutorials

There exist several methods to familiarize a user with software. One method would be a manual. The manual would contain all of the information needed to use the software. This would however cause a large overhead since the user would have to read the manual to receive the information before using the software. Even if the user just skips through the manual to get basic information, he then has to use the read information to find the right actions. Another problem would be to familiarize the user with the user interface, this would require the manual to have pictures of the software. Depending on the distribution of the manual, many pictures cause higher printing costs or larger file sizes. These pictures would also need to be updated when changes are made to the software.

Explanation by Peer

Another possible method for instructing a new user would be an explanation by a co-worker, this has similar problems as the manual. The overhead before the software could be used would be high, if the co-worker does not explain it while using the software. It would also cause two users to be involved in the process, this would also use the time of an experienced user in addition to the time of the inexperienced user. Not only is the time requirement higher than the manual, the quality of explanations could also vary depending on the explaining user. The knowledge of the software differs between users. This method would even increase the difference if users with different knowledge about the software are instructing new users.

Video Instructions

An instruction video could be used to avoid a difference in instruction quality. The video option would also address most of the problems explained earlier. An experienced user would only have to produce the video once, which would eliminate the need to spend more time, every time a new user is introduced. Since the software would be shown in the video, the need for additional pictures does not exists. The size of the file is however higher than the manuals, this could be a problem if disk space is limited. It is possible to stream video files, however this would require an internet connection or further infrastructure. If the video is streamed from a central server this would also allow for the instructions to be updated and therefore all new users would automatically receive the updated information.

The production of a video might be time consuming and if the quality is below average, it might not help the users in understanding the software.

Another problem with video as a medium is that updating parts of the video, if for example one functionality was changed, is complicated. This is due to the screen being visible at all time. If the *UI* of components change which might be visible even when another functionality is explained. This could cause confusion. Therefore the section of the video for the other functionality would have to be redone as well. This causes overhead in production and updating which is not desirable.

Learning by Example

All these methods are missing one essential part. [Charney et al. 1988] found that giving the user opportunity to practice the learned material will have an improved learning effect. Therefore it is more effective to learn while using the software, since the user is practicing what he is taught immediately. This allows to learn how the software behaves and while learning the features the user can get used to the functions commonly used. These functions include navigation and authenticating. It also trains to use functionalities that might not be intuitive. Learning while using the application can be achieved with an interactive tutorial.

Integration into Software

An interactive tutorial could be implemented separate of the actual software, this would however not allow for the tutorial to use features directly from the application. The navigation of the software would not be learned since the tutorial would be executed in another software.

Instead the tutorial can be integrated into the software, which does allow the tutorial to use and reference features of the application. It would also eliminate problems that might occur when installing or logging into the tutorial software, since the user is authenticating with the actual software.

Exploration of the software would also be possible when the tutorial is integrated into the software and explain feature by using popups. This could however hinder experienced users because of popups and notices regarding features they already use or know how to use. Allowing experienced users to disable the tutorials could avoid this problem.

The tutorial could also be a more separate feature, having it's own menu entry and therefore requiring action from the user to be activated. This would allow new users to specifically request to execute the tutorial. The possibility to chose not only to execute the tutorials but chose which tutorial will be run, enables a focused retrieval of information. This means that even experienced users could chose which tutorial might contain information they are not aware of, without forcing them to consume the already known information via popups, notices or otherwise forced tutorials.

The opportunity to force users to complete a tutorial is useful in a more corporate setting where it is required for all users to have at least basic knowledge of the program.

This feature would be even more useful in research applications at university since it ensures that the participants all execute the tutorial. This would ensure that the participant are not missing the tutorial or accidentally closing or skipping it. This is important because the result would otherwise be skewed by the fact that participant did not know how to use program. If the participant chooses to not continue the tutorial it will be visible when determining what was achieved during the experiment.

The application needs to track more data when forcing users to complete the tutorial. Since it would need to keep track of how far the user completed the tutorial, this progress would need to be persisted. At least the status if the user did complete the tutorial would need to be saved.

3.1 Features

There are several possible features for the tutorial presentation, which can enable the creation of effective instructions and explanations. We discuss these features and possible benefits and problems.

A basic feature of tutorials is text which contains the information that should be conveyed to the user. This text could include some of the other features described later. The text is used to deliver information and instructions to the users. The length of the text should not be limited, however if the instructions are too long they might not be readable. The implementation of other discussed features might require to save more data than is visible, this could cause the text to truncate if the text length is limited.

Formatting options for the text are useful to highlight and emphasize the important sections. Highlighting parts of text does support the providing structure in text. It allows the user to identify the information necessary to complete the task as also mentioned by [van der Meij 2008]. This could be done in several ways. One option would be line breaks. This would enable the writer to structure the text into paragraphs and highlight sentences by separating them from the other text.

It should be considered that special characters could also be used to mark and emphasis words. A more sophisticated variation would be to enable bold, italic and underlined text. This would allow highlighting of words inside sentences. It could also be used to highlight important parts or mark specific words more effectively than with special characters. This could however introduce more complexity, since a syntax would have to be used to define these text variations.

Another possible feature would be to enable linking of elements, this feature is further explained in Chapter 9.

Apart from the features regarding the text, the framework should be able to detect if the user has executed the expected action and then proceed to the next instruction. Detecting a wrong actions could also be possible, however the feedback would have to be situational. Since general feedback would not be as helpful as situational feedback. There are also more possible incorrect actions than correct actions since the user could even execute actions that never have an effect. These would have to be detected to show the notice, it might not even be possible to detect certain actions. It would be possible to disable all action except for the correct action. This might prevent the user from executing wrong actions, it could however also cause the user to simply randomly executing arbitrary actions in the hope of executing the required action. This could cause the user to execute the expected action and without what he did to achieve the result.

3.2 Minimal Featureset

We focus on the development of a framework, it's extensibility and integration into the existing software, and therefore we are only including a minimal feature set. The minimal features we deem necessary to manage and run tutorials for *ExplorViz*. A structure which contains instructions is required to implement the presentation of tutorials. Since multiple instructions should be provided, multiple of those items are required. A relationship between the instructions and the tutorial is also required. A transition between those instructions would need to be triggered when an instruction is completed. This could be triggered by a button. However this would not provide the means to demonstrate a feature, it would instead just be segmented text, which is more similar to a manual. When the user misunderstands such instructions, since they are not verified, he might not notice that the instructions have an other intention, therefore the user might continue with the tutorial without understanding the information presented. A system should try to verify that the user did understand the information presented to them.

3.2.1 Detecting User Input

A system that is trying to verify that some presented information was understood, will have to rely on the data which it has access to. There are different methods on how to obtain this data. A simple approach would be to assume that a user only clicks a button if he read and understood all information presented. Therefore the system would assume that the user has all knowledge that was presented earlier. This would be a very optimistic assumption.

Another option would be a question and answer system, which is posing questions that require knowledge which was presented during the tutorial. If the questions are answered correctly the system assumes that the knowledge was obtained. This would not guarantee an understanding since the answers, or positions of them, could be remembered without considering the content of the questions. If we assume there is no way to perfectly determine if the knowledge was obtained, we have to ask ourselves what we are able to verify.

A system that can verify that an action was performed, could assume that the user understood the instruction. Therefore if the instruction are worded in a way that the information has to be understood to understand the instruction, we can assume that the information was extracted and understood if the instruction is executed. This would give us verification that the information was processed, which is more than assuming it was. A system which recognizes actions performed is needed to use this assumption.

We also need a method to specify these actions and targets, to acknowledge the execution of the action on the target specified in the instruction. These references need to be able to be persisted. In a 3D environment they also need to be independent of the camera angle, since it would not be possible for a user to judge the degree of camera rotation by eye.

3.2.2 Tutorial Editor

We could assume that an editor is not required since tutorials could be programmed or scripted. This could however be more time consuming than creating a tutorial from an editor. The tutorial framework would need a compiler or other way to verify correctness of the code. Additionally not every user would be able to create a tutorial since programming skill is required to program a tutorial. This would mean that not every users with specific knowledge of the software would be able to provide tutorials for other users. Therefore an editor is a basic feature for the tutorial framework. The functionalities of the editor should however be simple and intuitive as to enable more users to provide tutorials.

Chapter 4

ExplorViz and Structure

We develop the tutorial framework for *ExplorViz*[1]. *ExplorViz* is a software landscape visualization and monitoring tool, that is developed by the Software Engineering Group at the Christian-Albrechts-University of Kiel. It uses live trace data generated by *Kieker* [2] and visualizations created by *Kieler*[3] to visualize the software landscape, instances and the interactions between them.

There are two different visualizations, the landscape perspective and the application perspective. The landscape perspective shows a 2D representation of the entire software landscape. This view includes all systems, node groups, nodes and applications running inside the environment, this can be seen in Figure 4.1. The grey boxes represent systems,

[1]https://www.explorviz.net
[2]http://kieker-monitoring.net/
[3]https://www.rtsys.informatik.uni-kiel.de/en/research/kieler/welcome-to-the-kieler-project

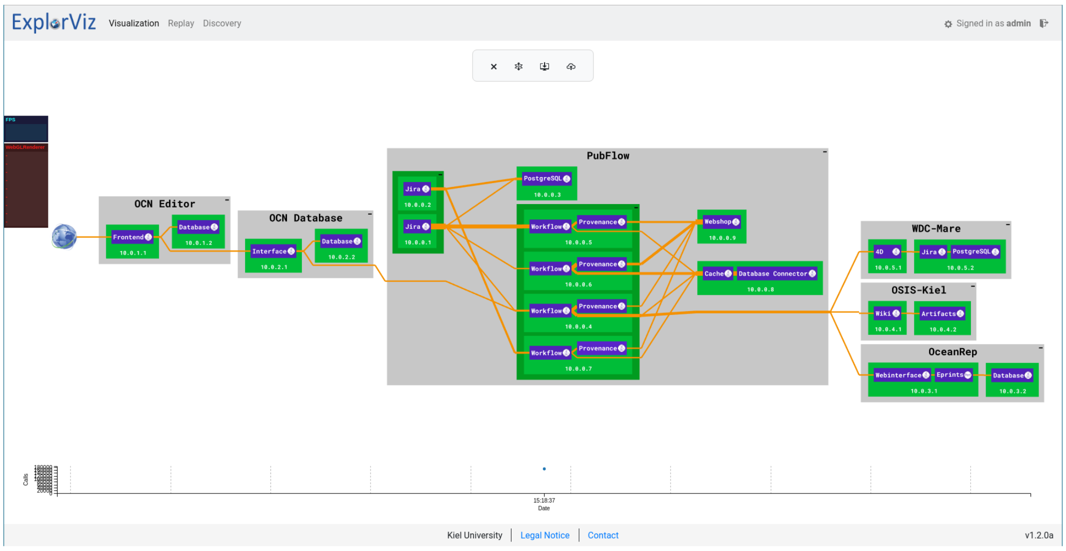

Figure 4.1. 2D landscape view

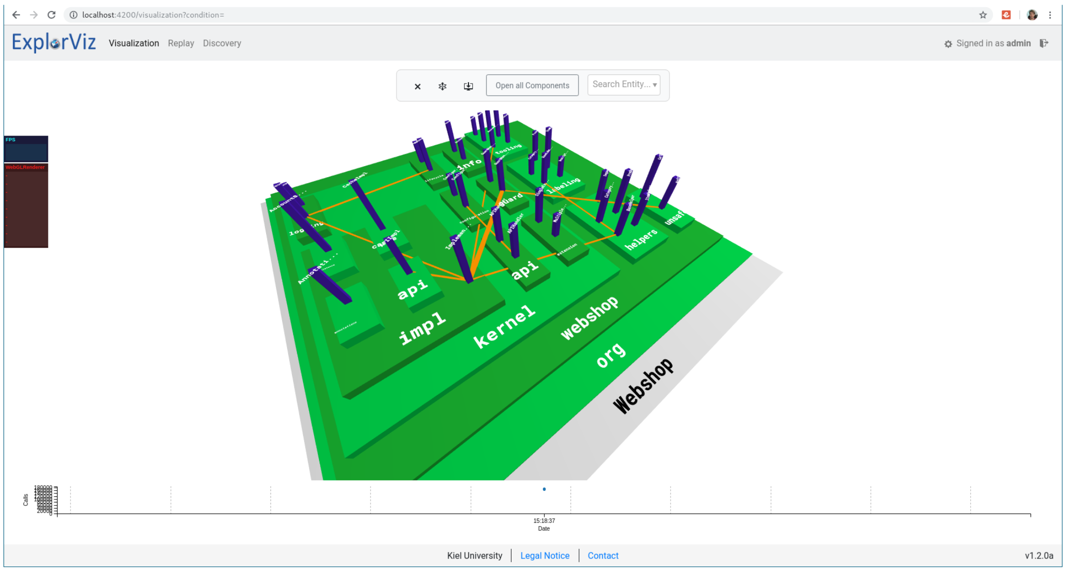

Figure 4.2. 3D view of communications

nodes groups are visualized in a dark green. Blue boxes represent the applications and the primary programming language is represented as a small icon next to the application name. Orange lines represent connections between applications. Systems can be toggled, which will cause all contained node groups, nodes and applications to be hidden.

In the application perspective which can be seen in Figure 4.2 only the selected application is shown, however in greater detail. This perspective shows a 3D representation of the software following the [Shneiderman 2003] mantra of 'Overview first, zoom and filter, then details-on-demand'. The application is shown as a hierarchical structure based on the modules of the programming language. In the case of a Java application these modules are packages. These packages, represented as green boxes, can be opened to show contained packages. When classes, which are represented by blue boxes, are contained in the packages they are visualized, their height determined by the number of calls inside them. When a package is closed it still retains the height of the highest class contained in the package. Classes are connected by orange trace lines which represent calls between classes. These traces are also shown when the packages containing the classes are closed. Packages, classes and trace lines can be selected which cause non connected classes to become transparent. The trace lines also have indicator for the direction of the call.

4.1 Extensions for ExplorViz

ExplorViz consists of a frontend and a backend. It supports extensions on the server and client side. The frontend extensions can be build into the application and provide additional functionalities. The backend consists of mostly independent services. Therefore the backend is extended by developing a new service which provides additional functionalities. The *Core* functionalities of *ExplorViz* is provided by the Ember Application and a suite of services in the backend, we call these the *Core* of *ExplorViz*.

Extensions are programs independent of each other which extend the functionality of *ExplorViz*. They are optional, therefore they might be used but are not required. They might not be able to run independently of the *Core* software. However they should be able to be combined with each other provided the *Core* application is available.

4.2 Structure and Services

The frontend is responsible for rendering the pages and providing methods for user interaction. It is an Ember application. The backend is responsible for persisting, collecting and providing data to the frontend. As further explained in [Zirkelbach et al. 2018] the *ExplorViz Backend* structure was recently changed from a monolithic software Figure 4.3 to a service based structure Figure 4.4. This service structure was extended during the development of the tutorial framework. Several new systems where introduced into the structure, which are providing a good inter-service connectivity, while also ensuring a good separation of dependencies.

Even though the services are separated, a *Core* suite of services is required for the basic functionality of *ExplorViz*. We call these services *Core Services*. The *Core* is combined in a *git repository* which contains the repositories of the services as *submodules*. This allows for checking out all or just the a subset of services for development. The *Core* contains the following services:

- ▻ *Analysis*
- ▻ *Broadcast*
- ▻ *History*
- ▻ *Authentication*
- ▻ *User*
- ▻ *User settings*

In the case of some services missing functionalities that require these services will not be available in the frontend. Without the authentication and user service it would not be possible to login into the application. When the services providing landscape data

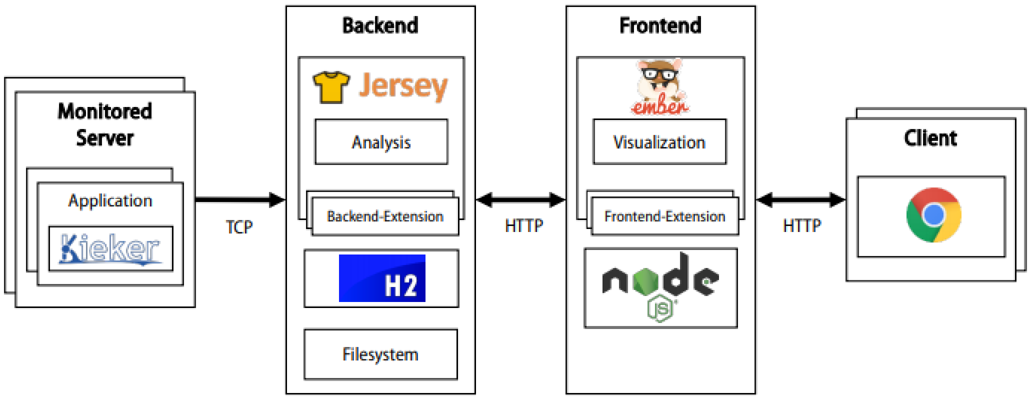

Figure 4.3. Monolithic Structure of *ExplorViz* taken from [Zirkelbach et al. 2019b]

are not running, nothing can be visualized. If a user is already authenticated and the authentication service is disabled, the visualization would not be affected until the user needs to be reauthenticated.

This also means that if all *Core Services* are running the *Core* functionality will be available. This also implies that if any extensions are missing, the *Core* functions are still available. Keeping the *Core* free of dependencies will therefore be necessary, especially regarding dependencies towards extensions. It cannot be guaranteed that the extensions are available or that their backend services are running.

The trace data is collected by *Kieker*[4] and then send to the *analysis-service*. We list and explain the functions of the *Core Services*.

The *analysis-service* receives the trace data from *Kieker*, this is done by a class called `KiekerAdapter` which receives the data and translates it into data usable by other *ExplorViz* services. It then transfers the data to the *landscape-service*.

The *landscape-service* accumulates this data and constructs a snapshot every ten seconds. We call this snapshot a *landscape*. Generating *landscapes* every ten seconds ensures that enough data has accumulated and the resulting *landscape* does contains meaningful data. The classes contained in the data for each *landscape* are grouped into their respective packages and classes. Afterwards objects are created to represent theses classes and packages as well as systems, machines and applications. The *landscape* is the send to the *broadcast-* and *history-service* . This is done via *TeeTime*[5].

The *broadcast-service* is receiving the *landscape* and delivers it to the frontend where it is displayed. This only happens if the live *landscape visualization* is active. This visualization will then always show the most recent *landscape*. This is achieved by providing a resource which can be used by the frontend to register as an observer. The request causes the

[4]http://kieker-monitoring.net
[5]http://teetime-framework.github.io

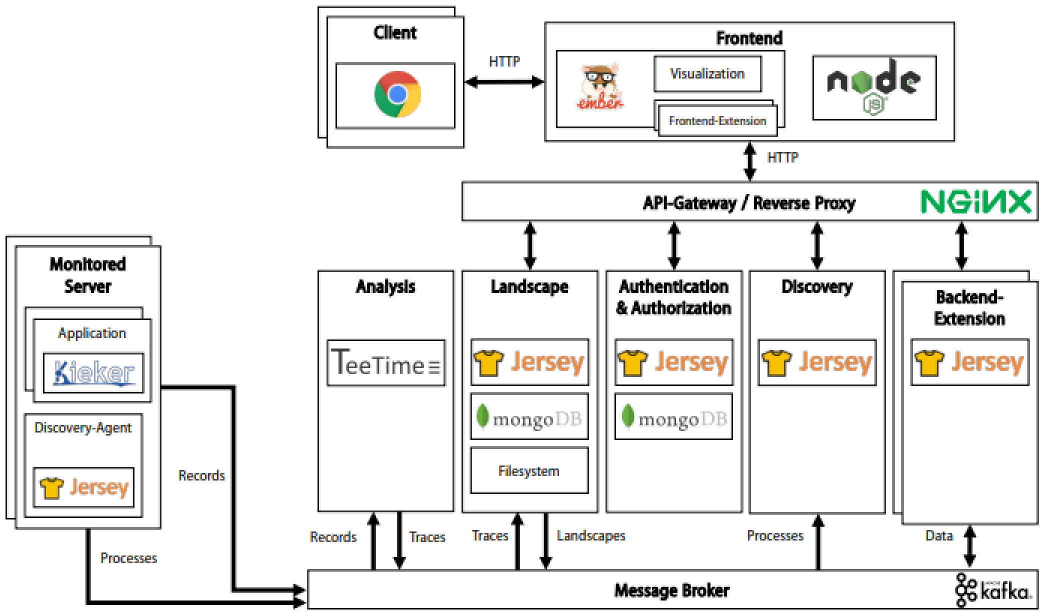

Figure 4.4. Service Oriented Structure of *ExplorViz* taken from [Zirkelbach et al. 2019b]

broadcast service to establish a connection to the frontend and provide new *landscapes* directly to the frontend connected to it.

The *landscape* is also transmitted to the *history-service* which will persist all received *landscapes* and allows to retrieve them by providing either an id or a timestamp. This service is used to provide *landscapes* that are requested. Another service is the *authentication-service* which is used to authenticate a user and check his roles. The users and roles are also saved in a *Mongo DB*. The *authentication-service* also provides the ability to login and produces a token which can then be used to authenticate and authorize all requests between backend and frontend. The roles and users are managed by the *user-service*, it also creates users and assigns passwords to them. The users can also be assigned user-settings, these are managed by the *user-settings-service*. This service provides settings for *ExplorViz* on a by user basis. This allows *ExplorViz* to provide customized settings for users.

4.3 JSON:API

JSON:API is used to communicate and transfer objects between services. This ensure clean separation between services. It also ensures that there is a clear definition of how the transferred data should be structured. Using *JSON:API* does also allow the services to use *Jersey* to serialize and deserialize the transmitted data directly to java objects. The services are not required to implement all specifications belonging to the standard, since the communication is mostly restricted to services interacting with *ExplorViz*. Therefore it is possible to only implement the interfaces required by other connected services. If for example an object will never be saved it is not necessary to implement the service providing the functionality, even though *JSON:API* specifies the interface. This requires developers to possibly implement the functionalities, if the interface becomes relevant. Since the interfaces are clearly defined if implemented the interface will adhere to the specification.

Relationships of objects can also be transmitted via *JSON:API*. When fewer requests are desired the objects can be included in the response. This requires the backend to provide the referenced objects inside the response. Otherwise the objects are only referenced via their id and not included. When the id is provided it is be possible to request the referenced objects. This allows for fewer objects to be transmitted since they are only loaded on demand, it would however cause more requests to be processed. Either might be desirable depending on the amount of overhead for a request and the objects used by the other service.

Chapter 5

Design Approach

5.1 Approach Overview

This chapter contains the design approach we took for the development of the tutorial framework. As mentioned before *ExplorViz* can be extended by creation of an Ember addon for changes regarding the frontend and it can be extended via development of a service for the backend.

We considered if both where necessary. We found there were several reasons why a frontend and backend extension would be necessary. We now discuss these reasons and also highlight the design approach that was taken and revised during the development of the tutorial framework.

The goal is to develop a tutorial framework which is able to create tutorials and execute them. We are developing a service which will be able to receive tutorials, persists them in a database and deliver them to the frontend. As described in Chapter 4 the services are connected via Kafka. This was however not the case when we started to develop the framework. First the principles of a service oriented structures where applied to the structure of *ExplorViz*. We therefore discuss which dependencies exist and how we are avoiding unwanted dependencies to the *Core* application. We will then go into detail how the services are connected and which parts of the *ExplorViz* infrastructure must be used or reconfigured. We then highlight the inner structure of already existing services, to apply this knowledge for the development of our own tutorial service.

Afterwards we will discuss what should be considered when developing an Ember Addon for *ExplorViz*. We therefore discuss compatibility issues with other extensions, how the extension is connected to the backend and finally we explore options we considered when designing the tutorial execution and tutorial editor.

5.2 Tutorial Framework Backend

Developing a backend service for managing the data required was necessary because the tutorial data needs to be persisted in a database. This data could have been integrated into one of the existing services this would however introduce dependencies between this service and the tutorial mode. Avoiding these dependencies was one of the principles we were following when designing the backend.

Minimizing Dependencies

The *Core* of *ExplorViz* should not have any dependencies towards the tutorial framework. This would cause the base functionalities to be unavailable if there is a problem with the tutorials. We decided to implement a separate service to manage the tutorial data. Dependencies in the *tutorial-service* towards the *Core* are less of a problem since the *Core Services* are required for the basic functions of *ExplorViz*. If they are not available the tutorial framework could not be started and therefore the dependency is not critical.

Dependencies would be introduced if for example the discovery service would use any of the *tutorial-services* functionalities. We therefore aim to introduce as little change as possible to the *Core*. This also solidifies the decision of implementing a separate service, since this does not require the *Core Services* to be modified.

5.2.1 Connecting Services and Deployment

We are aiming to integrate the new services into the environment *ExplorViz* runs in, we will therefore develop *dockerfiles* which will create the service and enable it to run in conjunction with the other services. To achieve a uniform URL as also mentioned by [Zirkelbach et al. 2019b] a reverse proxy is used, we are therefore including the tutorial service into this reverse proxy. The proxies configuration needs to be adjusted and therefore the docker image needs to be rebuild.

The *landscape-service* could be used to persist the landscapes, this would allow the *tutorial-service* to only reference landscapes, which could then be loaded from the *landscape-service*. This would cause the *tutorial-service* to be dependent on the *landscape-service*. Missing landscapes would need to be considered and error handling for this case need to be implemented. The most detrimental impact however would be, that if landscapes are imported to be used by the tutorial, they could not be distinguished from live landscapes. We therefore decide that landscapes should be managed by the tutorial-service.

5.2.2 Service Structure and Persistence

We also decide to use the same technologies already utilized in the development of *ExplorViz*, this allows developers to familiarize themselves with the service. When most services use the same technologies, developers are not restricted to develop on one service. A similar structure allows for faster familiarization with any given service, since the developer already knows which components are responsible for which functions of the service.

This also allows for all services to be configured the same way, which allows for developers to customize their development systems. Adjusting ports and specifying *dockerfiles* become simpler when the structure of all services is similar.

We therefore aim to use the same technologies in a similar structure as the other services. Even though not many services existed when the development started, we investigated the few services that did exist, and determined the technologies and structure of the services.

We therefore implement a backend service which uses *Jersey* to provide a *JSON:API*. It should persist the received objects with *Morphia* and utilize a *Mongo DB* for storage. The service needs to provide *JSON:API*s for all objects the tutorial framework is using. It should also persist the relationships between objects, which is possible when using *Morphia*. Editing and deletion of persisted objects should also be possible.

The ports of the backend should be configurable. This is necessary as mentioned earlier to avoid collisions on ports, and to enable developers and users to customize the installation of *ExplorViz*.

5.3 Tutorial Framework Frontend

Developing the frontend requires another approach since the method by which the frontend of *ExplorViz* can be extended is vastly different. The principle of minimizing dependencies is still valid, however since the *Ember Addons* are installed into the *ExplorViz* frontend, it is not possible to decouple the frontend from the extension. Therefore dependencies with the *Core* are less problematic than in the backend. In practice the principle however does still apply, it is for example possible to override functions in the frontend, which might introduce dependencies to the extensions backend. The developer should still consider that the overridden components might not be available if overridden.

5.3.1 Frontend Modularity

Considering that backend services might not be available and that overriding functionalities in that case might disable base functions, we decided to integrate the tutorial into *ExplorViz*. We decide against integrating the tutorial into the existing visualization, since it would disable the *Core* functionality. Another factor in this decision is that it would be difficult to convey to the user if the visualization is used or a tutorial is executed. A clear separation of features is preferred. Overriding code in *Core* components would also require to reimplement changes in the visualization if they were overridden.

Additionally problems might occur when multiple extensions inject and override code in the *Core* application, we therefore decided the extension should provide a separated section for it's functionalities.

If extensions are not injecting their code into the default visualization, the *Core* visualization will remain available.

5.3.2 Models and Connection to Backend

The frontend extension should be able to connect to the backend and exchange data with it. We rely on the *Ember Data* to serialize and deserialize the frontend models. It uses the *JSON:API* provided by the backend since *Ember* is compatible with *JSON:API*.

The *Ember Model* should therefore be identical to the java classes in the backend, this will allow *Ember Data* to serialize and deserialize the *JSON:API* requests and use the models when presenting the data to the user.

5.3.3 User Interactions

The frontend does not only exchange data it also provides the user with information and allows the user to interact with the tutorial. We therefore implement a feature to execute the tutorials. Additionally we implement an editor to create and edit tutorials.

Designing the visuals for tutorial execution, we consider a popup similar to what was implemented in the earlier version implemented by [Finke 2014]. This design however was able to obstruct the view onto the visualization. We therefore decide to create a *sidebar*. This causes the visualization to render in a smaller area, however the information cannot obstruct the view onto the visualization.

We determine which problems might occur when creating tutorials. We find if the editor vastly differs from the visualization, the creator might not be able to determine how the text would be displayed in the *sidebar*. A preview window would constrict the room for the visualization even more. Therefore we use a design the resembled the structure shown when executing tutorials.

To visualize the look of the final application we construct a picture from existing components (see Figure 5.1).

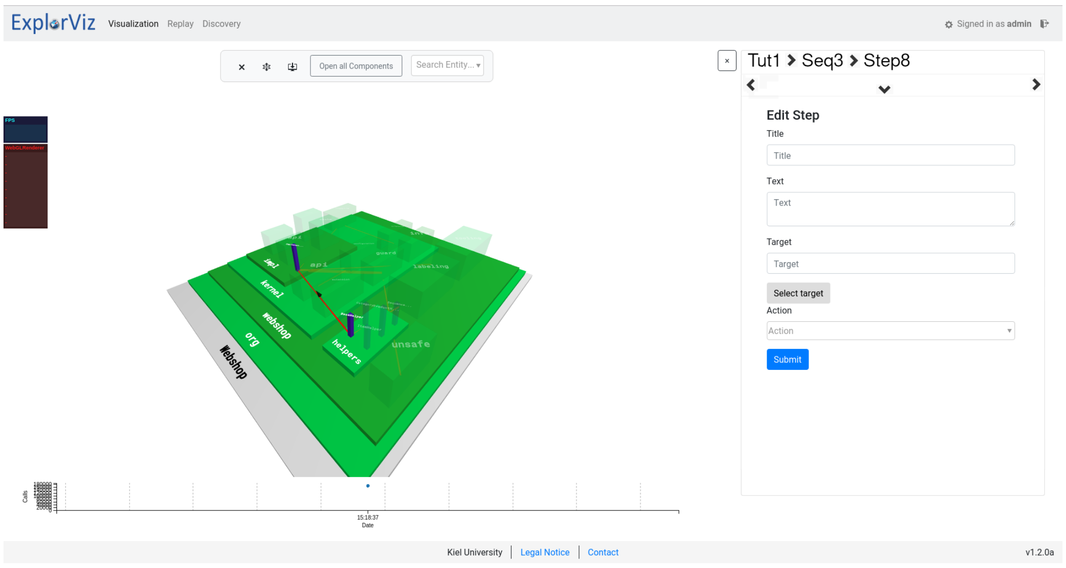

Figure 5.1. Constructed picture of tutorial editor UI.

Chapter 6

Implementation

This chapter features descriptions of the features we implement as well as problems which cause us to adjust our design during development. We implement two extensions, the backend which is managing the data required for the tutorials, and the frontend which handles user interactions and presentation.

6.1 Implementation Overview

We discussed our approach towards developing the tutorial framework in Chapter 5, we now describe the specific problems and how we resolve them. We follow the same order as in the approach. Therefore we first discuss how the Service is connected to the frontend and how we avoid problems with other services. We focus especially on the configuration of the development system and how it is affected by the components of *ExplorViz*. We then describe the classes which store the tutorial data. Problems regarding *Jersey* and how the *landscapes* are persisted are described. This leads us to highlight which options were available to manage communication between services. The last part regarding the backend is dedicated to a problem that occurred regarding identifiers.

We then follow by discussing the development of the frontend. We specify how the models correspond to the classes in the backend, this includes adapters and serializers. We discuss problems regarding *landscape* serialization in the frontend before explaining which options for mapping routes to functionality where can be considered. We then give an overview of how variables can be bound to templates. Then interaction components are explained and we explain how these can be utilized to select targets inside of the visualizations. A structure to transmit information between interaction components and form components is described. We also explain how services can be used to avoid binding variables and why we cannot utilize this for the visualizations. At last we describe a problem with serialization in the frontend.

6.2 Tutorial Framework Backend

The tutorial backend is providing data for the frontend. We mentioned our design approach in Chapter 5 and will now report how we implement the extensions. We also highlight which problems cause us to adjust our approach.

6.2.1 Connecting Services and Deployment

The *tutorial-service* will be deployed using *Docker*[1], to ensure that the service is embedded into the environment we need to create the infrastructure which allows us to connect the *tutorial-service* to the frontend.

Docker and Reverse Proxy

As mentioned before *Docker* is used to run the services. The started service needs to be able to communicate with the other services and the frontend. Communication between services was not possible at the time development started, since *Kafka* was not yet integrated into *ExplorViz*.

The frontend sends requests to either the URL specified by *JSON:API* unless another URL is specified in the *adapter* for the *model*. *Adapters* will be discussed more in Section 6.3. As mentioned in Chapter 5 a *reverse-proxy* maps the path of a request to the corresponding server or port and then redirects the request to the determined location. Therefore in the development environment the *reverse-proxy* is mapping the request URL to the port where the service is running. We edited the configuration for the *reverse-proxy* to map a URL to the *tutorial-service*. The path chose the path `tutorials`.

Even though *Docker* is supposed to enable simple deployment on different architectures there still is a discrepancy between *Docker* running under *Linux* and *Docker* running under *Windows*. This means there exist two different *docker-compose files*. The configuration files of the *reverse-proxy* are also different. This is mostly due to a difference in how networking is managed by the two operating systems. The *Linux docker-compose file* has to specifically define that the host network structure should be used.

Since we adjust the dockerfiles in our repository, which is not part of the *Core* repositories, we have to adjust the ports dedicated to the tutorial-service when new services are introduced in the *Core*. This means also updating those ports in the configuration of the reverse proxy.

Usually the image of the *reverse-proxy* corresponding to the operating system is specified in the *docker file*. This would eliminate the need to edit the configuration file for the *reverse-proxy*, for *Core* services, however since the *tutorial-service* is not part of the *Core*, there is no configuration for this service in the prebuilt images.

We therefore need to build the image on our machine after editing the proxies configuration file to include the service. This is done via docker-compose, the repository for the *reverse-proxy* is checked out and the configuration is adapted. We are then using docker-compose to build the docker image, this requires us to specify which docker file should be used. We have to specify the path of the checked out *reverse-proxy* repository to build the image.

This causes the tutorial-service to be included in the built image, however since the changes are not committed to the *reverse-proxy* repository, the *tutorial-service* will not be

[1]https://www.docker.com

included by default. In the future there will probably be a git branch of the proxy that includes the *tutorial-service*. Even if this is not the case and the tutorial is never included in the *reverse-proxy*, we have to assume that the production environment will be adjusted if additional extensions are loaded. If this would not be the case, other extensions could not be referenced and they would also not work. These settings are highly dependent on the structure and configuration in which *ExplorViz* is deployed. There might not even be the need for a *reverse-proxy* since every service might be running on a single server. In that case the URLs would not have to be matched to a port since the URL already uniquely identifies the server and therefore the service running on the server. The service provides a configuration file which can be used to customize ports and server locations to enable the service to work with different configurations.

Avoiding the 'next free port' collision

Since we are deploying the *ExplorViz* services on the same host, we have to manage the ports for the different services. This has to also be done in the configuration of the *reverse-proxy* and in the configuration of the specific service. It has to be assured that no two services are running on the same port. If two services would run on the same port the service started first would block the port and the later service would not be able to start properly. The *Core* services use different ports. When new *Core* services are added to the environment the next 'free port' is used, which is also the port the *tutorial-service* uses. Therefore the port would need to be adjusted. This causes us to change the port if new services are introduced. We use a port which is not subsequent to the highest used port, to avoid having to change the port in the *Nginx*[2] and service configuration every time a service is added to the *Core*.

Gradle

During development *Gradle* is used to build the *Java* applications of the services, this allows for a more seamless integration into the developers *IDE*. The *classpath* is set and libraries which are necessary are automatically downloaded and included. It also serves to include the *Core* repositories if classes are required. *Gradle* works by reading the *Gradle configuration file*, which specifies the dependencies. This causes *Gradle* to retrieve the dependencies and to configure the *IDE*. The dependencies can contain a version, which allows to use a specific version of the dependency.

6.2.2 Service Structure and Persistence

Adhering to the general structure of *ExplorViz* consisting of a *REST API* provided by *Jersey* which is connected to a *Mongo DB* we also utilize *Morphia* to persist objects with their relationships into the *Mongo DB*. The *JSON:API* provided by *Jersey* is used by the frontend to communicate with the backend and makes persisting objects possible. The

[2]https://www.nginx.com

services implement resources which are listening for *HTTPS* requests. These *JSON:API* conform requests contain the objects that should be persisted. They are then parsed and the corresponding objects are created by *Jersey*. These objects can then be used to either load the requested objects from the database or persist the changes the received objects contain. This method can also be used to create new objects. The endpoints which are listening for the requests are called resources and they are configured using annotations. The annotations are defining the path, the type of requests and whether objects are consumed or produced by the resource.

The classes introduced by *tutorial-service* are the following:

- **Tutorial**
  `Tutorials` are the main object they contain the references to `sequences` and a `TutorialLandscape`.
- **Sequence**
  `Sequences` can be used to structure `tutorials`, they reference an optional `TutorialLandscape`.
- **Step**
  `Steps` contain the instructions, `target` and action to be performed on the target.
- **TutorialLandscape**
  `TutorialLandscape` contains the id, `timestamp` reference and a serialized version of a `landscape`.
- **TutorialTimestamp**
  `TutorialTimestamps` extend `timestamps` and are used to be referenced by `TutorialLandscapes`.

The frontend uses *Ember Data Models* corresponding to the backend classes, those are explained in further detail in Section 6.3. The classes contain fields which are annotated to enable *Jersey* and *Morphia* to identify the fields used to create identifiers and relationships. Listing 6.1 shows the source-code of the `tutorial` class and how the annotations are used to specify ids and relationships. Also seen in the figure Listing 6.1, the `tutorial` class has a relationship to multiple `sequences`.

The `tutorial` class is used to specify the `landscape` which will be displayed when executing the `tutorial`. It also contains a title for easier identification by users. Most importantly however the `tutorial` contains a list of `sequences`. `Sequences` are used to further subdivide a `tutorial` into sections. `Sequences` therefore also contain a title which can be used to specify what the `sequence` will teach. `Sequences` can also refer to a `landscape`, this allows for `sequences` inside the `tutorial` to use a different landscape than specified in the `tutorial`. This allows for `sequences` of the `tutorial` to visualize landscapes and explain how they can give different information.

`Sequences` contain a list of `steps`. `Steps` are used to deliver instructions to the user, they contain a title and instruction text. They also contain the target for the expected action. The target is composed of three fields, a `targetType`, a `targetId` and an `actionType`. The

Listing 6.1. Annotations and fields in Tutorial class

```java
@Type("tutorial")
@Entity("tutorial")
public class Tutorial implements Serializable {
   @Id
   @com.github.jasminb.jsonapi.annotations.Id
   private String id;

   private String title;

   private String landscapeTimestamp;

   @Reference
   @Relationship("sequences")
   private List<Sequence> sequences = new ArrayList<>();
```

`targetType` is required since ids are not guaranteed to be unique across types. The class diagram of the `tutorial` class in relation to `sequences` and `steps` can be seen in Figure 6.1.

The other two classes, `TutorialTimestamp` and `TutorialLandscape`, are classes that represent classes which exist in the *Core* of *ExplorViz*, due to technical limitations described in Section 6.2.3 it is however not desirable to use these classes. The `TutorialLandscape` contains an id, a reference to a `TutorialTimestamp` and a string containing the `landscape`. `Landscapes` and the objects contained in them are not instantiated, instead they are stored as a *JSON:API* string. `TutorialTimestamps` and `TutorialLandscapes` use the same ids as the corresponding `timestamps` and `landscapes`, this allows to retrieve the `landscape` by id without the need for a parser to extract the id from the string while inserting it. The referenced `TutorialTimestamp` id referenced in the `landscape` will still be a valid reference since the `TutorialTimestamp` will also be inserted with the same id as the `timestamp`. This works due to reusing the already generated ids. `Tutorials`, `sequences` and `steps` are inserted with newly generated ids since they do not have corresponding counterparts in the *Core*.

We generate ids by using the id generator, which is provided by the *shared repository*. The backend is providing the functionality to load `tutorials` with all `sequences` and `steps` included in one response. It also allows to load `sequences` and `steps` via their ids. However since the inverse of the relationship between `sequences` and `step`, and `tutorials` and `sequences` is not modeled, this would not result in the parent objects to be included. An extensive list of which endpoints are provided by the *tutorial-service* and used by the fronend can be found in Figure 6.2

Object creation does require a complete object which does not have an id set. The object is then inserted into the database. *Ember* stores all objects in its *Store*, therefore when the object is inserted into the database and an id is generated, *Ember* needs to assign the new

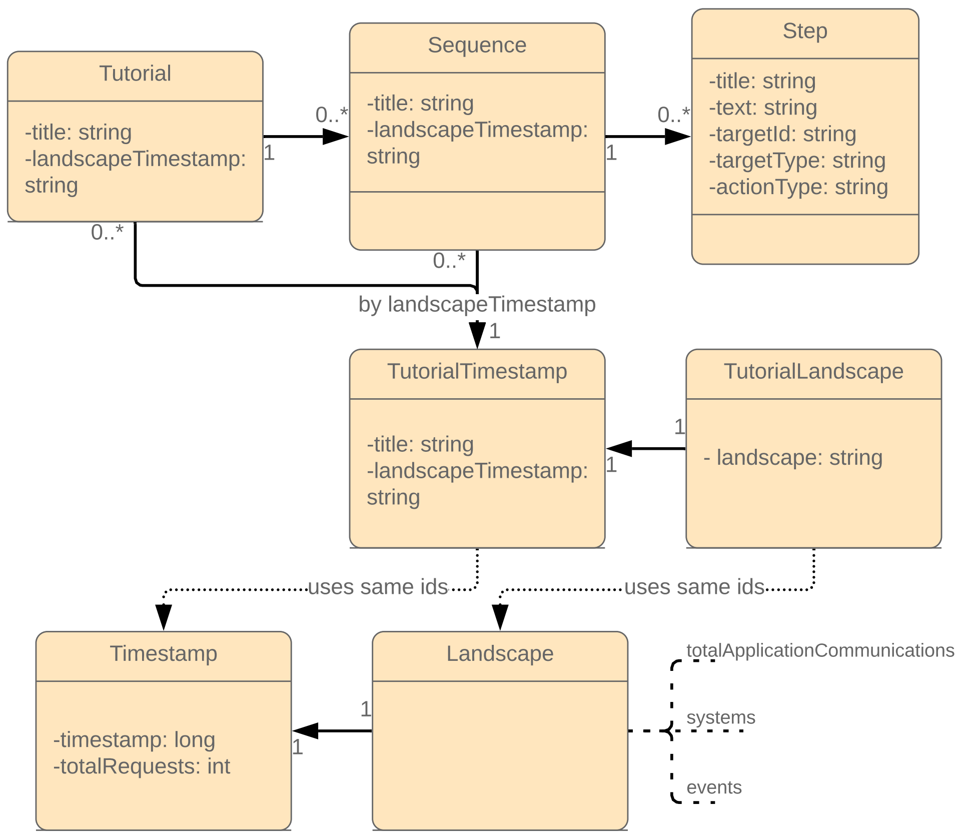

Figure 6.1. Class diagram of `tutorial`

Request type	Endpoint	Landscapes	Timestamps	Tutorials	Sequences	Steps
POST	/	n/a	n/a	used	used	used
GET	/<id>	n/a	used	used	used	used
DELETE	/<id>	n/a	used	used	used	used
PATCH	/<id>	n/a	n/a	used	used	used
GET	/	n/a	n/a	used	used	used
GET	/by-timestamp	used	n/a	n/a	n/a	n/a
POST	/import	used	used	n/a	n/a	n/a

Figure 6.2. Table of provided and used endpoints by *tutorial-service*

id to the object in the *Store*. This is done in such a way that the service will respond to the insert request with the object, including the id just generated.

A request containing the object which the changes were performed on is send to the service to save these changes. The object with the corresponding id is loaded the changes are made to the object and the object is then saved to the database again.

A simple way to request an object is to call the service with a *GET-HTTPS-request* containing the id of the desired object. In the *tutorial-service* the id is included in the URL. Usually the URL pattern is similar to `/v1/<servicename or objecttype>/<id>`. The object with the id is then loaded from database and returned to the frontend.

Another way to request objects from the backend is to filter by a property. The properties are usually also contained in the URL, however they are included as *GET-parameters*. From the properties given in the filter a database request is constructed which will then fetch the required object, and deliver it to the frontend. We use the filer method to allow requesting `TutorialLandscape` by `timestamp`.

6.2.3 Technical Limitations and Considerations

`Landscapes` contain `events` and `systems` and a list of information about application communications and a reference to a `timestamp`. The `events`, `systems` and communication data are hierarchical objects that can contain a lot of data. Landscapes are transferred to the *tutorial-service* therefore they need to be serialize and deserialized. Since a `landscape` can contain a large amount of data we need to be careful in how they are retrieved considering which information the backend is required to process.

Saving Serialized Landscapes

The `landscapes` contain a lot of data represented in multiple objects which are referencing each other. If the `landscape` is instantiated as an object all hierarchical objects contained in the `landscape` would also need to be instantiated. Furthermore the objects inside the `landscape` are not changed or rearranged by the backend, and therefore the `landscapes` can be handled as a single object. We therefore were decide to store the serialized string in the database, this allows for all included objects to always be present in the stored string, since it is not changed.

We are not able to receive a `landscape` as a request and store it directly into the database. *Jersey* expects a serialized object. This means that if the `landscape` is send to the service *Jersey* will try to instantiate the objects transmitted. However since we do not want to instantiate the objects that are contained inside the `landscape`, we need to circumvent the automatic deserialization by *Jersey*.

The only information necessary for storing and retrieving of the `landscapes`, that are contained in the string are the id and the `timestamp`. This means that the serialized string can be saved in the database and then retrieved by either requesting it via `timestamp` or id. Even if *Jersey* is circumvented and the raw string could be persisted into the database, the id and `timestamp` would still be unavailable without parsing it from the string. We therefore introduce a class that contains the id and a reference to the `timestamp` of the `landscape` as well as the entire serialized string. This class can be saved into the database using *Morphia*, since the object persisted does have and id and `timestamp` to query for. We created a corresponding class called `TutorialTimestamp`. When the `TutorialTimestamp` is received the id and `timestamp` reference are already separated from the string, making a parsing of this information unnecessary. The `landscape` does not contain the `timestamp` but a reference to an object which contains the `timestamp`. These classes are also saved into the database and linked to the `TutorialLandscapes`.

This allows us to not instantiate all objects in the `landscapes` and still allows to query for ids and `timestamps` in the database. Instantiating those objects would not only require a lot of memory, it would also require the objects to be persisted into the database as separate objects. By doing this we would have to ensure that the relationships are preserved. This would be prone to errors because the deleting of one object that is referenced could cause the `landscape` to not load. A separate persisting of the objects would also require to use the given ids for the objects, it could however occur that two different objects are transmitted using the same id. This would cause similar problems to the ones explained in Section 6.2.3. Additionally we do not want to implement new classes for these. We then need to import a repository which includes a lot of classes. The *ExplorViz* project has a *shared repository* for exactly this purpose. In this case some of the classes were missing from the repository, which could have been fixed by including them. This would however mean that the *Core* needed to be changed. Since at the time it was discussed which classes should be included in the *shared repository*, we decide to not use the shared classes. Including all classes that could be used by other services is generally a good idea, there is however the problem of including all classes from every service in the *Core* causing the repository to create a dependency.

Shared Repository

Integrating classes from the *shared repository* also poses a problem since the technologies used in the service are not identical. For example the *broadcast-service* is not using *Morphia*, it instead is creating *Mongo DB* records without using a datastore. The *tutorial-service* however uses *Morphia* to save objects to the *Mongo DB*. This means that a datastore factory is used which creates a datastore for each object. This would include the `landscapes`, which would be imported from the *shared repository*. *Morphia* then expects the given class to determine which field is used as the primary key, and creates the datastore which then can be used to store and load objects of this type. The problem is that if the `landscape` is imported from the *shared repository* only the annotation that specifies the ids for *Jersey* are present in the file. This means that *Morphia* cannot save the object since no key, according to *Morphias* annotations is found.

An approach to resolve that is to include the annotation for *Morphia* into the *shared repository*, this however causes problems since in other services *Morphia* might not be used.

Due to *Gradle* (see Gradle paragraph in 6.2.1) being used to build the services this would cause *Morphia* to be included in every service that uses any of the classes in the *shared repository*. The inclusion of *Morphia* by itself is not desirable because it inflates the size of the *shared repository* and version conflicts could be caused if different services use different versions of *Morphia*.

Since including *Morphia* into the *shared repository* is not feasible, another approach is considered. We extend the `landscape` class which is included in the shared library. This would allow to import *Morphia* only in the subclass, and therefore in the extension instead of the *shared repository*. The id field would be overridden and the annotations would be placed at the appropriate positions inside the new file. This however does not work since *Morphia* does not support inheritance when checking for annotations. Therefore it only checks for annotations in the subclass, which means it will not find the other fields in the superclass. This would require us to override all fields present in the `landscape`, which would in turn defy the purpose of the class being imported from the *shared repository*. This in accordance with the problems regarding Jersey instantiating the object described earlier leads us to the approach which is implemented. We do not instantiate the class in the backend, but instead use the data as a string, and only include the data necessary to retrieve the desired `landscapes` later. The persisted object, called `TutorialLandscape`, is therefore a class that does not extend the `landscape` class however contains the serialized `landscape` as string.

Interservice Communication

We consider which channels can be used to transfer the `landscapes`. An option would be for services to communicate with each other.

At the time *ExplorViz* did not yet utilize Kafka for communication between services, therefore the implementation of another communication channel would have been required. Many questions needed to be answered regarding how communication could be implemented. Among others the question of how the backend services would authenticate each other was not answered.

Since the development of a backend communication framework which later could be used by all services, is outside of the scope of this work, we decide to take another approach. A system to authenticate frontend connections to the backend was already exists, so we decide to use the frontend for this communication.

Identifiers

Using the already generated ids for the `TutorialLandscapes` causes a problem if the ids are not unique. At first we assume that this is not be likely to happen. The hypothesis is that only when a `landscape` is exported and imported into another instance of *ExplorViz* an id collision could occur. This could then however been avoided, by reassigning an new id if there was a collision. A direct insertion into the database is not possible since the id is the primary key. Therefore only an import function can cause a collision.

`Landscapes` are generated in the *landscape-service*. The `landscapes` are generated by adding the systems, applications classes and traces into a `landscape` object which is stored in memory. When the collection time elapses the timestamp is created and the `landscape` is send to the *broadcast-service*. This however causes the `landscape` object not to update its id. Instead the id stays the same.

This problem is fixed in newer versions of *ExplorViz* by not only updating the elements in the `landscape` but also creating a new id.

6.3 Tutorial Framework Frontend

The *ExplorViz* frontend provides a login page when the user is not authenticated. After logging in, as long as no other page was requested by the entered URL, the user is redirected to the visualization page. There is also a discovery entry in the menu, it is used for the monitoring setup. As mentioned before the visualization is the main feature of *ExplorViz*, and therefore most of the features are centered around the visualization. When no application is selected a `landscape` of the observed applications and machines is shown.

In the visualization we have to implement a system that recognizes user inputs and matches it to the specified action. When the specified action is executed on the target the `step` is completed and the next `step` in the `sequence` is loaded. When a `sequence` is completed, by completing the last `step` in the `sequence`, the first `step` of the next `sequence`. When a `tutorial` is completed the next `tutorial` will not automatically be loaded, instead a message is shown that the last `step` was completed.

Detecting User Input

Since most of the features of *ExplorViz* are based on the visualization, the referencing of elements inside the visualization needs to be implemented. The detection of scrolling and moving inside the visualization does however require a more in depth analysis of how the movements correlates to camera movements. The detection of simple movements like zoom in and zoom out could be achieved by implementing a listeners for these events. However more complex examples, like scrolling to specified part of the application would be considerably more complex. It would also require a system for referencing specific parameters for these actions, for example the amount of zoom required or the distance panned inside the `tutorial`. There also might be complications if the visualization is already zoomed in and the required zooming is not possible without zooming out first. Since the only benefit at the moment for implementing a simple detection would be one `tutorial` which tracks if the user moves the camera. This detection would only check if zoom or panning was executed and not track the amount zoomed. We decide this feature does not provide enough benefit and therefore we are not including this functionality in this version.

6.3.1 Models and Connection to Backend

The `tutorial` model, similar to the backend classes, consists of three hierarchically ordered objects. The topmost object is also called `tutorial`. A `tutorial` is connected to a `landscape` which will be loaded if no other `landscape` is defined. It also contains a `hasMany` relationship to `sequences`, a `landscape` can be connected to each `sequence`. If a `landscape` is referenced it will be loaded upon starting the `sequence`. Sequences contain a `hasMany` relationship to `steps`. A `step` consists of a text for instructions and a target composed of the id and type of the target and an action (e.g. double click).

Appart from the `tutorial`, `sequences` and `steps` there are two other models defined for the frontend. These are the `TutorialLandscape` and `TutorialTimestamp`, they are the corresponding models to the classes with the same name in the backend. However the model of `TutorialLandscape` does not contain a serialized landscape, instead it contains the deserialized version of the landscape. This allows ember do manage the `TutorialLandscape` as if it is a`landscape`. The model objects are managed by *Ember Data*.

Already existing ids, from the corresponding `landscape`, are used for the `TutorialLandscape`. This means that the `TutorialLandscape` which reference `landscapes` can be referred to by the id of the included `landscape`. We also include a reference to the `TutorialTimestamp` in the `TutorialLandscape`, the id will also be the same as the `timestamp`. This allows for the `TutorialLandscape` to be requested via `timestamp`.

Adapters

Ember provides *adapters* for the *Ember Data Framework* to be compatible with arbitrary *APIs*. An *adapter* can be used to modify the URLs for requests made by *Ember Data*. This is used to set the URL to which requests are send to. There are different methods on how to load data, if the data was requested via the `queryAll` or `findAll` command the `urlForFindAll` or `urlForQueryAll` method is used to determine the URL. There are also methods for updates and creating objects. There are *adapters* defined for most of the objects in *ExplorViz*. The objects which have an explicitly defined *adapter* can be loaded in *Ember* and are instantiated from data provided by the backend. We therefore are providing *adapters* for the `tutorial`, `sequence`, `step`, `TutorialTimstamp` and `TutorialLandscape` model. Objects without an *adapter* would be loaded via the default *JSON:API* URL. These URLs might not be part of the *reverse-proxy* and therefore the service would not be reached and the object cannot loaded.

The *adapters* are also used to define how the frontend will authenticate with the backend. The authentication method is configured through the authorize method. In this case the authentication method sets a `Bearer`-header which contains the authentication token to which can be validated by the backend.

Serializer

The only model that does not use the default *adapter* is the `TutorialLandscape`. The `TutorialLandscape` is special since we include a string which contains the *JSON:API* string.

This means that the *adapter* is serializing the objects contained in the `landscape`.

Serializers are used to serialize and deserialize the data received by *Ember Data*. This also enables *Ember* to be used with a variety of *APIs*. There are several methods in the *serializer* that can be overridden to enable customizing of the *API*. The `normalizeResponse` and other `normalize` methods are called after the response for a request is received. They are used to modify the received data to match what *Ember Data* is expecting. The default *serializer* uses the *JSON:API* specification for well formed data, therefore these methods need to convert the received data to match this specification. Usually this includes renaming of properties and class names, to ensure that the properties are names correctly and the classes are loaded. To achieve this a *JSON* object is parsed and passed to the method, along with the *Ember Store*, the `primaryModelClass`, the requested id and the `requestType`. This allows to make changes the *JSON* object and load objects to the *Ember Store*. We however utilize the *serializer* to serialize the `landscape` and embed the serialized `landscape` into the `TutorialLandscape`. Since the *JSON* object is passed to the function, we can make changes to the object and then pass it to the `normalizeResponse` function of *Ember Data*, and therefore calling the `super` function with the modified *JSON* object.

For data send from the frontend to the backend we are overriding the `serialize` method, this method is called for all outgoing data of this model. The `serialize` functions parameters are a `snapshot` of the object to be serialized and an option parameter. The option parameter can used to include an id in the serialized object or omit it. When the `serialize` function is called the passed `snapshot` is serialized, if the object does not represent the expected object an error is thrown. No constructor is available for `snapshots` of an object. Therefore we cannot change the type of object since the `snapshot` would not match the new type and since there is no constructor we cannot create a matching `snapshot`. We first serialize the object by calling the `super serialize` function. This function returns the *JSON* object which will then be send in a request. We can however edit this *JSON* object before returning it in our overridden function. This would cause the request to include our changes.

To save a `landscape` we could call the `serialize` function and send the result to the backend, however *Ember* does not include child objects in these requests. This does make sense for the normal mode of operation where *Ember* loads only object or group of the same type per request. In this case it sends another request for further objects that are referenced via relationships. This behavior can be changed either by marking the relationship as synchronous (`async:false`), which means that the objects which would otherwise be loaded via the relationship are included in the request. This causes *Ember* to load all included objects it does not however cause *Ember* to save all relationships.

To mitigate this a mixin called `ember-data-save-relationships` can be used. This mixin can find and serialize objects which are connected via relationships and include them in a single request. This is especially useful if objects are supposed to be created and referenced using a single request. To achieve this the mixin does collect the referenced objects. The mixin then checks if the object is missing an id, it is therefore not yet inserted into the database. It will then create a temporary id, which has to be returned by the backend, and will then be used by the mixin to insert the returned id into the corresponding object.

This seems like it would be a solution: All objects referenced via relationship are included in the request which means that we can simply serialize the request and insert it via the tutoriallandscape into the database as string. The mixin however only serializes the first layer of these relationships.

This means if we only have a parent and a child model, and the child class is referenced in a relationship from the parent model, we could serialize the parent which would then include the child. However in this case there are more than two layers, therefore we would serialize the `landscape`, the `systems` would be referenced, however no elements referenced by the `system` would be included.

This means that the mixin does not solve our problem and another solution has to be found. We therefore write a method that iterates through all relationships and collects all objects referenced. These objects are then serialized and stored in an array in the `TutorialLandscape` *JSON* object. This method however also poses some difficulties. When we implement a recursive function which is able to collect all referenced Objects. This function has to check for two types of relationships, `belongsTo` and `hasMany`. This is necessary since a `hasMany`-relationship contains many objects and therefore a loop is needed, a `belongsTo`-relationship does not need this loop.

There is however a problem with the `threeJSModel` object contained in the `landscape`, it contains a circular reference therefore creating an infinite loop. When removing the `threeJSModel` when serializing errors appear when the loaded `landscape` is visualized. This defect could not be resolved, therefore persisting `landscapes` is currently not possible. Another possibility why this error occurs could be that the system implemented to check if a type of object is already serialized is preventing some objects to be serialized. If this system is not in place however the same error occurs. It is also possible that the hierarchical depth of the `landscapes` is exhausting the stack space of *JavaScript* in the browser.

Since we want the `TutorialLandscapes` to be deserialized with every request we override the `normalizeResponse` and `serialize` methods. We only need to collect the objects when serializing. When the *JSON* is loaded back into the *Ember Store* any referenced and included objects are loaded by *Ember Data*. This means that all previously loaded elements will be included and therefore loaded and correctly referenced by *Ember*.

6.3.2 Frontend Modularity

As mentioned in Section 2.2 an *Ember Application* is constructed from *routes*, *components* and *templates*. To ensure better integration with other *Ember Addons* we register a main route called `tutorial`. All other *routes* we introduce are subroutes of the `tutorial` route. This causes our *routes* to be longer than necessary, it ensures however that we do not have collisions with other addons using a similar approach. Otherwise there might be multiple *routes* that are registered to the same URLs, for example `add` or `edit`. These collisions could cause that the application might not work as intended.

First we introduce a subroute for each of our defined models, which are supposed to have subroutes for each action possible on the given type.

This however leads to very unintuitive URLs. The URL for editing a `tutorial` with the id `tutorial-1` would be `/tutorial/tutorial/edit/tutorial-1`. The doubling of `tutorial` is caused by the main route. The possible actions are `edit`, `create`, `list` and `delete`. The `list` route requires an additional parameter for `sequences` and `steps`, to reference which parent object had to be loaded. This causes an unintuitive URL for lists. To list the `sequences` of `tutorial-1` either the URL references the type to be loaded, causing it to be `/tutorial/tutorial/list/tutorial-1` or the type to be listed which causes the URL to be `/tutorial/sequence/list/tutorial-1`. To avoid this another keyword is considered, however `/tutorial/list/sequences/for/tutorial/tutorial-1` is to long and it suggests that other possible function could be available instead of `for`. This could cause confusion. Additionally the `tutorial` and `sequence` have a `selectTarget` action and the tutorial has the `run` action. This means that fifteen routes would have to be implemented. The purpose of these routes is to utilize *Ember Data* to load the data for the given ids and to provide the opportunity to share links as references to specific objects. The editing URL of the previous example would allow users to directly access and edit `tutorial-1`. It also allows for links to be saved as bookmarks. This is however only necessary in some cases.

To reduce the complexity of implementing fifteen routes, we consider which routes need the ability to be shared between users or used as bookmarks. Providing a link that specifically created or deletes parts of a `tutorial` could even be considered a security risk. A disguised link could then be used to delete or alter `tutorials` unwillingly. Therefore we decide to only implement the `edit` routes, the `run` route for the `tutorial` and one `list` route which is modified to show all types in a hierarchical table. All other routes are unnecessary to provide direct links to. This decision reduces the number of routes from fifteen to five. This also allowes us to simplify the naming convention of the routes, since every type only has one route, it is sufficient to reference the type and id. We can also drop one occurrence of `tutorial` for the routes regarding the `tutorial` type. The new `list` route does not require an parameter since all `tutorials` are loaded. These changes cause the previous example `/tutorial/tutorial/edit/tutorial-1` to compress into `tutorial/tutorial-1`. `/tutorial/list/sequences/for/tutorial/tutorial-1` was compressed to `/tutorial/list`. Having only one route per type means that the loading of types have to be implemented less frequently. This reduces redundancies and simplifies the development. This resultes in implementing the following routes:

▻ **/tutorial**
redirects to **/tutorial/list**

▻ **/tutorial/list**
lists all tutorials

▻ **/tutorial/<tutorial-id>**
editing page for tutorial with id <tutorial-id>

```
Router.map(function() {
    this.route("tutorial", function(){
        this.route("list", { path: '/list' });
        this.route('tutorial', { path: '/:tutorial_id' });
        this.route('sequence', { path: '/sequence/:sequence_id' });
        this.route('step', { path: '/step/:step_id' });
        this.route('run', { path: '/run/:tutorial_id' });
    });
});
```

Figure 6.3. Injection of *routes* into the router of *ExplorViz*

▷ **/tutorial/sequence/<sequence-id>**
editing page for sequence with id <sequence-id>

▷ **/tutorial/step/<step-id>**
editing page for step with id <step-id>

▷ **/tutorial/run/<tutorial-id>**
executes tutorial with id <tutorial-id>

The code for injecting these *routes* into the router of *ExplorViz* can be seen in Figure 6.3.

The editing, creation and selection of targets is now contained in these *routes* and implemented via actions that are passing the value as parameter and applying it to the *model*. This means the *routes* are fetching the *model*, which are retrieved from the backend or stored in memory via the *Ember Store*.

With these routes defined the *controllers* are implemented next. The *controllers* contain the logic which is used to control the actions and provide the functions which are called via *interaction* with *templates*. The objects retrieved by the *routes* are passed to the *controller* which will modify the object, managing variables that are determining the state of the application and executes actions called by the templates. The `edit` *routes* need to save changes made to the model which requires an action.

Controllers are linked with the *route* that has the same name as them. This means the `/tutorial/run`-route will execute the `tutorial/run`-controller. This ensures that there can only be one *controller* per route, and each *controller* can also only be associated with one *route*. It is technically possible to call a *controller* from another, but this is not done by *Ember* per default.

Binding Variables in Templates

Next we tried to avoid duplicate code for our *templates*. Embedding *components* into *templates* does not only allow to minimize duplicate code, *Ember* also automatically registers which *components* need to be executed and does so. When a *component* is used in a *template*, parameters can be defined. These parameters are bound to the given values. If the value is a variable, both the variable from the used *component* and the passed variable are linked. Values of bound variables are synchronized as long as the *Ember* getters and setters are used. If we define a *template* `callerTemplate` with the variable `foo` which is set to `bar` and a component `sampleComponent` with the variable `bindMe`, we can use the *component* and bind the variable by using the following code:

```
{{sampleComponent bindMe=foo}}
```

This would cause `bindMe` and `foo` to bind, therefore `bindMe` will have the value `bar`. This binding is bi-directional, therefore if we change `bindMe` to `affectedFoo` via the setter (`this.set('bindMe','affectedFoo')`) the change would propagate to `foo` in `sampleComponent`. This can be used to propagate changes from one *component* into all other *components* that use the bound variable.

We consider using only one *component* to implement most functions, this causes the *component* to contain a lot of actions and therefore code. It also causes the *template* to be divided into sections which were displayed when specific conditions apply, e.g. the tutorial edit page when the tutorial model is loaded. If a `tutorial` is supposed to be edited the *component* would detect which type the model is. To view the corresponding fields, a variable is used to signal to the template that a tutorial is loaded and therefore the tutorial form should be shown. When changes are made to the input fields the *component* reads these changed properties and compares them to the object, if changes are present the properties are transferred into the object and persisted. This is however not necessary if the changes are directly executed into the properties of the model. We therefore create the forms in such a fashion that they include the model and changes are directly made to the model.

Therefore changes in the form automatically propagate into the model, which then only have to be persisted. The different models however require different fields inside the form. Therefore it is not possible to use the same *component* for all types. Therefore a form *component* is created for every type (`tutorial-form`, `sequence-form` and `step-form`).

Interaction Components

We introduce multiple other components, the *landscape-visualization*, *landscape-interaction* and the timeline component inherit the functionality of the corresponding rendering *components* (*landscape-rendering* and *application-rendering*) from the *ExplorViz*-frontend. This enables us to adjust their templates by overriding them. It also allows to extend the *components* to implement additional features. In this extension of the *components* we are also able to insert *services* which are needed to access the data we are trying to visualize.

We are using the *interaction* to select elements inside the `landscape`. We extend the *ExplorViz* frontend interaction components, which are used for interaction with the *landscape-visualization* and *application-visualization*. The timeline component is extending the timeline component and overrides the action that is called when a `timestamp` is selected, this causes the timeline to select the clicked `timestamp`. We are however also calling the `super function` which enabled us to execute code which enables us to import the `landscape` to our *tutorial-service*. Calling the `super function` causes the timeline to still retain the basic functionality, to select a `landscape` and request the loading of it, and allows the new code to then import it into the *tutorial-service*.

Another component is the *landscapelist*, this component requests all `TutorialTimestamps` that have available `TutorialLandscapes` in the tutorial-service and provide a list. It displays the names of the `TutorialTimestamps`. It also marks the selected `TutorialTimestamps` as such. When clicking on an unselected `TutorialTimestamps` the component selects the `TutorialTimestamps`, and inserts the `timestamps` value into the currently edited *model*. To mark a `timestamp` as selected, we are comparing the `timestamp` value in the list against the `timestamp` value of the selected `landscape`, if it matches the `timestamp` in the list is marked as selected.

Structuring Components

In Figure 6.4 the structure of the frontend components can be found. Bold names define the part of our application. *Services*, *routes* and *components* each have a different color. Smaller boxes inside these parts are variables. The big green arrows show bound variables, the names in the boxes of these arrow show one of the variables names. Dotted lines inside of a grey box, represent different possible states of the *component*. These different states do cause different *templates* to load. The variable is only bound to one *component* if the arrow points to the specific *component*. It is bound to all *components* if an arrow touches the outside of the grey box. The big orange arrow, represents a variable which is only bound in some constellations. The active `step` is only bound to the model of `step-form` if the application is either in *runmode* or *runmode* is triggered by a missing role.

As mentioned before we have to use different forms for the *models* since the properties are different, most of the structure around the form however is the same for all forms. We want to avoid a *template* that has to rely on variables to determine which kind of *model* is loaded. We do however have a *route* for each type. This means that in the *template* of these *routes* we know which *model* will be loaded. This allows us to include the corresponding form. The structure around the form is the same for all *models*. We therefore introduce a component called `side-form-layout`, which provides the surrounding structure. But since we need to call the right form *component* for the corresponding *model* defined by the *route*, we define a parameter as seen in Figure 6.5 which contains the name of the form are including. Inside `side-form-layout` we now use this name to include the correct *component* for our model. This is visualized in Figure 6.4 by dotted lines between the form *components*, only one of which will be loaded at once.

This code also shows that we are not only passing the *model* as `model` but also as a `runmode` variable, which is set to `false` in this case, since it is the edit routes of the `tutorial`. The `runmode` variable specifies if the `tutorial` is being edited or executed. We

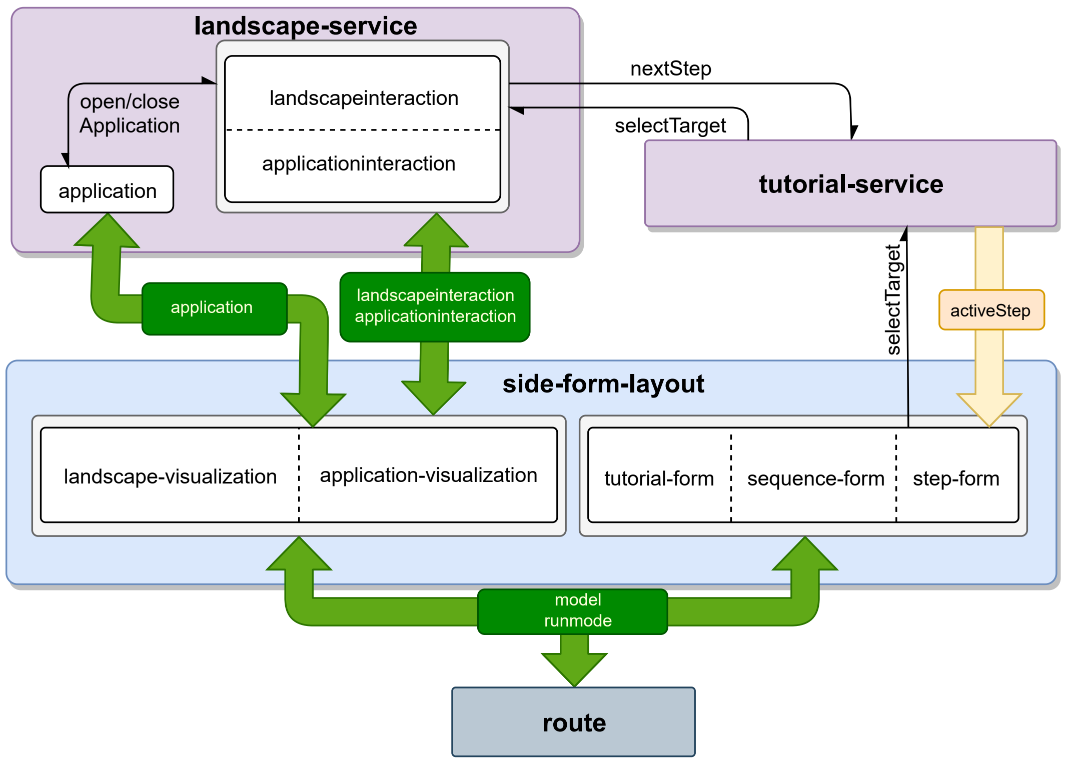

Figure 6.4. Structure of frontend components

```
{{side-form-layout form="tutorial-form" model=model runmode=false}}
```

Figure 6.5. Binding of the form parameter to determine form loaded by `side-form-layout`.

are also able to load the `step-form` in `runmode`, by binding another value to the variable, even though the *model* referenced is a `tutorial`, instead `steps` should be displayed when in `runmode`. Therefore we can use the `side-form-layout` in conjunction with the `tutorial-form`, `sequence-form` and `step-form` to edit and display the properties of the model.

The *component* responsible for displaying the `landscape` is called *landscape-visualization*. The *component* responsible for displaying an application is called *application-visualization*. These use interaction components to manage *interactions* with object which are visualized.

The extended *interaction components* are named *landscape-interaction* and *application-interaction* respectively. These *components* are providing the functions that are used when certain *interactions* are executed by the the user inside of the visualization.

Selecting the target for `steps` is initialized in the `step-form`, therefore we need to pass the selected target and the executed action from the *interaction* to the form. The *interaction* need to pass the selected target back to the form. It would be possible to access the interaction by accessing the *landscape-visualization* and getting the *interaction* from the *component*, this would however cause the *landscape-* and *application-visualization* to be bound to the form.

Services

Ember does provide the means to bind parameters between *components*, however this means that for all variables needed in multiple *components* the parameters would need to be passed to many *components*. If a new *component* would be added which is higher in the *template* hierarchy, a common ancestor would have to be found, to pass the variable from already existing *components* to the new *component*. Instead of binding these variables to all *components* a *service* can be used. This allows for variables to be stored in the *service* and read from every *component* which contains the *service*. Additionally actions can be implemented inside the *service* which allows all *components* to call these functions. It is possible to bind functions to parameters of *components*. This however does cause the context in which it is executed to be different. Therefore if the function uses a *service* which is not present in the *component* the function will not be able to access this *service*. The tutorial framework uses two *services*, one for the functionalities related to `tutorials` and one *service* for the `landscapes`. The *landscape-service* is responsible for loading the `landscape` referenced in the `tutorial` (or `sequence`) and provides the loaded `landscape` for the visualization *components*. It is also responsible for importing `landscapes` into the *tutorial-service*. This also includes loading the `landscape` when selected from the *landscapelist*. The import and load functions are interconnected in such a way that when a `tutorial`, which is not already imported, is loaded by the *landscape-service* the import will automatically occur, in this case the `landscape` will be named `new landscape`.

The *tutorial-service* is responsible for determining the current `step` in a running `tutorial` and contains functions to determine parent objects. The *interactions* for application and `landscape` are instantiated inside of the *landscape-service*, this allows us to reference then an inject them into the visualizations. It is also used to switch the behavior of the *interaction* depending on if a target should be selected. It also enables us to set the *model* variable which is used to determine which *model* is updated when a target is selected.

6.3.3 Technical Limitations and Considerations

We assume that a system to persist `landscapes` already exists, since it seems as if `landscapes` could be requested by the frontend. This assumption however is wrong. The `landscapes` are only saved by in the *landscape-service* and requested from the same service that created the `landscape`. This is due to the structure of the backend. The data is only broadcast once which transmits a `landscape` to the frontend. This `landscape` contains a timestamp, which can be send to the *landscape-service* to retrieve the same `landscape`. Since (previous to the implementation of *Kafka*) the `landscape` is not send to any service, instead the `timestamp` was recorded and used to request the `landscape` from the backend again. This means there is no need to serialize the `landscape` except for the service which is creating the `landscape` and therefore is already storing the objects of the `landscape` in memory.

Usually it is possible to use *Embers* `generate` function to generate files which contain a minimal example. So generating the files will construct files and links the files in the `app` folder of the addon to the `addon` folder. However the *Ember* `generate model` function does not work for addons, instead it will cause the error `blueprint not found`. It will work for a standalone app. This means that the files had to be created by hand. Creating those files per hand is error prone and if the wrong file is referenced by mistake, the addon will still start. It will however give errors like `action not found` since another *component* is imported instead. Debugging these errors is very time consuming and frustrating, especially if the other *component* does have an action with the same name.

Chapter 7

Evaluation

The developed tutorial framework was evaluated by an experiment for which a *example tutorial* was created. The experiment consisted of participants following the tutorial and answering a questionnaire with questions regarding the usability of the tutorial.

7.1 Methodology

We are executing a usability experiment, in order to verify that the implemented solution is usable. We therefore select participants which are following a tutorial and answering a questionnaire. The questionnaire was used to ensure that the participants answer the same questions.

7.2 Experiment

As mentioned before the experiment is executed to verify usability,we now further explain the setup, execution and how the results are obtained. We also discus what can be deducted from the results and list threats to validity of our results.

7.2.1 Experiment Setup

An *example tutorial* was created to give an overview of the visualization of *ExplorViz*. It was also designed to use multiple `sequences` and both visualizations, to ensure that all features of the tutorial framework is being used. The `tutorial` was separated into three `sequences`. The first `sequence` was dedicated to introducing *ExplorViz* and the visualization. The second `sequence` was dedicated to the *landscape-visualization* and the third `sequence` is dedicated to the application perspective.

The provided `landscape` and therefore applications are the same for all participants. It is a landscape containing a `sampleApplication`[1]. Since this application is only providing one application, there are no connections between machines in the landscape perspective. This can be seen in Figure 7.1. The landscape was created by executing the `sampleApplication` and retrieving the `landscape` generated by *ExplorViz*.

[1]https://github.com/czirkelbach/kiekerSampleApplication

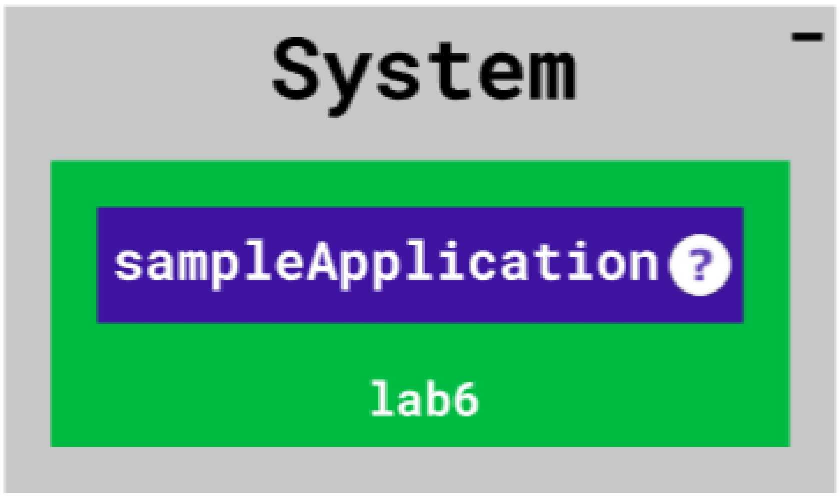

Figure 7.1. The `landscape` provided for the example `tutorial`.

The application included in this `landscape` contains some *Java packages*. The main part of the application is situated inside the `net.explorviz.sampleApplication` package. The `main` class in this package however is only called once and therefore has a low height. This causes the class to not be visible when the package is opened without moving the camera from default position. The application with opened packages can be seen in Figure 7.2.

The introduction only contains one `step` which explains how the navigation bar is used, and explains the camera controls. This `step` does not have a selected target which causes a `next`-button to be displayed. This was done so that the user does not accidentally click on the target when starting the `tutorial`.

The second `sequence` introduces the *landscape-visualization*, at first *landscapes*, *machines* and *applications* are explained. A click on the *machine* `lab-6` is used to verify that the user did understand which elements represent *machines*. This was done even though a single click on a machine does not have an action associated in the visualization. We therfore veryfied that actions that do not yield a result in the visualization could still be used for instructing the user. Then the maximizing and minimizing of *systems* is introduced.

The third `sequence` then introduces the application perspective. The packages and how to open and close them is explained. The user is then prompted to find a package which is not visible without opening other packages. The java package name is provided, so the

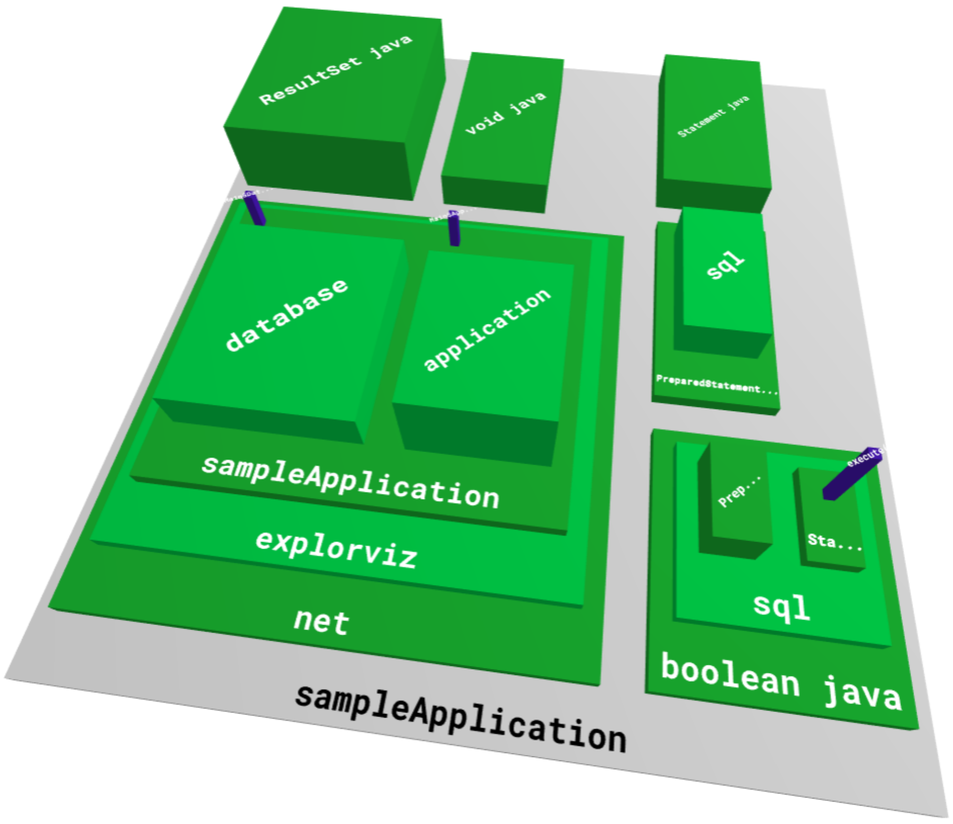

Figure 7.2. The application perspective of provided `landscape` for the *example tutorial*.

user can assume that the `sampleApplication` package in `net.explorviz.sampleApplication` can be found by opening subsequently opening the `net` and `ExplorViz` packages. Then selection of classes is introduced. The class that is supposed to be selected is only visible if the camera is moved, this ensures that the user is required to move the camera and verifies the the user remembered the camera controls from an earlier `step`. Next the class should then be deselected. The last `step` that requires an action from the user, does explain that packages do determine their height by the contained classes. It also explains that the classes are higher if they are called more often. The instruction is to find the larges class inside of the `sampleApplication` package. The instruction requires the user to recognize that another package has to be opened and then the class can be selected. This was therefore 'mixed practice' as introduced by [Chamey and Reder 1986].

7.2.2 Questionnaire

The questionnaire consists of three parts, the first part asked for general information from the participants. The second part was centered around the execution of the *example tutorial*. The final and third part contains questions about the tutorial editor.

General Information Questions

The general information part covers the following questions:

▷ **Name**
The name only served administrative purposes.

▷ **Whats your gender?**
Possible answers where female, male and other.

▷ **How old are you?**
The age in accordance with the other personal data is used to categorize the participant group.

▷ **Are you currently a student at the 'Christian-Albrechts-Universität zu Kiel'?**
This data is used to establish if the participants where students.

▷ **Are you currently studying 'Informatik' at the 'Christian-Albrechts-Universität zu Kiel'?**
This question is used to establish if the participants might have an interest in *ExplorViz*.

▷ **If the previous answer was yes in what semester are you currently studying?**
This question has the options `master` and `bachelor` and is asking for the semester.

▷ **Do you have experience with ExplorViz in general?**
This question has five possible answers: `not experienced`, `somewhat experienced`, `very experienced`, `i'm an expert`.

▷ **Do you have experience with the ExplorViz frontend / UI?**
This question has the same five possible answers as the previous question.

▷ **Are you currently developing ExplorViz or software related to ExplorViz?**
This question is used to distinguish if the participant are familiar with the backend, and therefore might draw non obvious conclusions during the tutorial.

Tutorial Execution Questions

The next part is the tutorial execution part the questions in this part were the following:

▷ **Did the tutorial execute properly?**
This question distinguishes if he tutorial execution did have any technical problems.

▷ **How intuitive was the tutorial UI ?**
This question had five possible answers: `very unintuitive`, `somewhat intuitive`, `average intuitive`, `very intuitive`, `it was perfect`.

▷ **Did the tutorial teach you (or would it have taught you, if not already known) the basic controls of ExplorViz?**
This questions purpose was to determine if the participant felt like anything was missing from the tutorial.

▷ **Was the goal of the tutorial unclear at any point?**
We want to determine if the participant was able to understand the instructions presented by the tutorial.

▷ **Would you have preferred any additional feedback, apart from showing the next instruction?**
This question allowed the user to provide information about which feedback, visual or otherwise, might improve the experience.

▷ **Do you have any other suggestions?**
The participant were able to suggest changes, to determine how the tutorial execution might be improved.

Tutorial Editor Instructions

The instructions for tasks to be executed in the editor:

▷ **Create a new tutorial, name it and select a live landscape.**
This task tests the intuitiveness of the editor in regards to creation and editing of tutorials.

▷ **Create at least 1 sequence with at least 2 steps.**
This task ensures that the tutorial can proceed from the first to the second step in execution mode.

▷ **Select targets for the steps.**
This enables the participant to execute the tutorial, without using the next button only.

▷ **Execute the tutorial.**
This enables the participants to see the created tutorial in the same mode the tutorial was executed, possible errors in the execution of earlier tasks would be visible here.

Tutorial Editor Questions

The last part is the tutorial editor part and the question are the following:

OS	Windows 10 (64 Bit)
CPU	Intel Core i5 6500 @ 3.20GHz
GPU	2047MB NVIDIA GeForce GTX 950
RAM	16,0GB @ 1063MHz
Monitor	24" @ 1920x1200 pixels

Figure 7.3. Specifications of machine used for *example tutorial*

▻ **Were you able to execute the tasks in the editor?**
We want to determine if the participants found all function they needed to execute the tasks.

▻ **How intuitive was the tutorial editor UI ?**
This question has five possible answers: `very unintuitive`, `somewhat intuitive`, `average intuitive`, `very intuitive`, `it was perfect`.

▻ **Would you have preferred any additional feedback from the editor UI?**
This question allows the user to provide information about which feedback, visual or otherwise, might improve the experience.

▻ **Do you have any other suggestions?**
The participant are able to suggest changes, to determine how the tutorial editor might be improved.

7.2.3 Execution of the Experiment

The Experiment was executed with 8 participants over two days in the rooms of the 'Software Engineering Group' on a development computer, specifications as seen in Figure 7.3. The participants where selected either because they where working at the 'Software Engineering Group' as student assistants and therefore where likely to be students. Since this did not result in enough participants some 'Informatik' and a former 'Informatik' student where invited to participate. The goal was to get participants with a mixture of experience with *ExplorViz*, but having good familiarity which software.

After arrival of a participant the general information questions where answered in a text file on the same computer and monitor as the tutorial was executed. After answering the general information questions the `tutorial` was executed without further instructions. The participant was supervised, to prevent the `tutorial` to be accidentally deleted or modified. Questions regarding the execution of the tutorial were not answered instead it was referred to the text in the `tutorial`.

When the last `step` was completed by the participant, they where instructed to answer the tutorial execution part of the questionnaire. The first participant made remarks that the `tutorial` was intuitive but mistakenly marked the `very unintuitive`-option, we recognized that this option might be understood as `very intuitive`. This caused us to notify every

	Student at CAU	'Informatik' student	ExplorViz developer
yes	87.5%	87.5%	37.5%
no	12.5%	12.5%	62.5%

Figure 7.4. Participant educational background and connection to *ExplorViz*

participant that the first option was the not the `very intuitive`-option. This was possible since it was noticed when the first participant was answering, and therefore all participants received the notice.

After answering the tutorial part of the questionnaire the participants where asked to perform the actions noted in the tutorial editor part. The instructions for the tutorial editor where designed to test how intuitive the controls are. Therefore simple tasks where chosen and initially no further instructions where given, the only additional information the participants were given was how to select the landscape on the timeline and to save the tutorial after selecting the landscape. They did not receive instruction on how to achieve the requested tasks we therefore used 'Pure Problem Solving Practice' as introduced by [Chamey and Reder 1986]. Upon completion of the tasks the participant was asked to answer the remaining tutorial editor part in the questionnaire.

7.3 Results

The information gathered in the general information part enables us to categorize our participants, we were aiming for a group that might have an interest in using *ExplorViz* and therefore might be likely to use the tutorial. We also tried to have some variation in experience with *ExplorViz* to ensure that the tutorial will be useful for beginners and experienced users. 37.5% of participants rated their experience with *ExplorViz* as `very experienced` or higher, as can be seen in Figure 7.5. Therefore 62.5% of the participants are not familiar with *ExplorViz*. Experience with the *ExplorViz UI* was even less with 50% not having any experience with it.

37.5% of the participants where currently developing *ExplorViz* or software related to it. Of the currently studying participants 42.86% were master's students, the rest were bachelor's students. The average age of participants was 26.29 years. We did not receive an age of one participant and the current semester was not submitted by two participants, one of these was not currently studying.

It was recognized that the question for the current semester was phrased in such a way that the answers varied. Some master students where answering with the semesters they where studying including their bachelor's degree and some answered excluding it. Therefore the absolute value cannot be used, however it is still possible to divide the participants into groups of master and bachelor students. There were 50% were master students and 37.5% were bachelor students.

In some cases the transition to the application perspective did not trigger the next step,

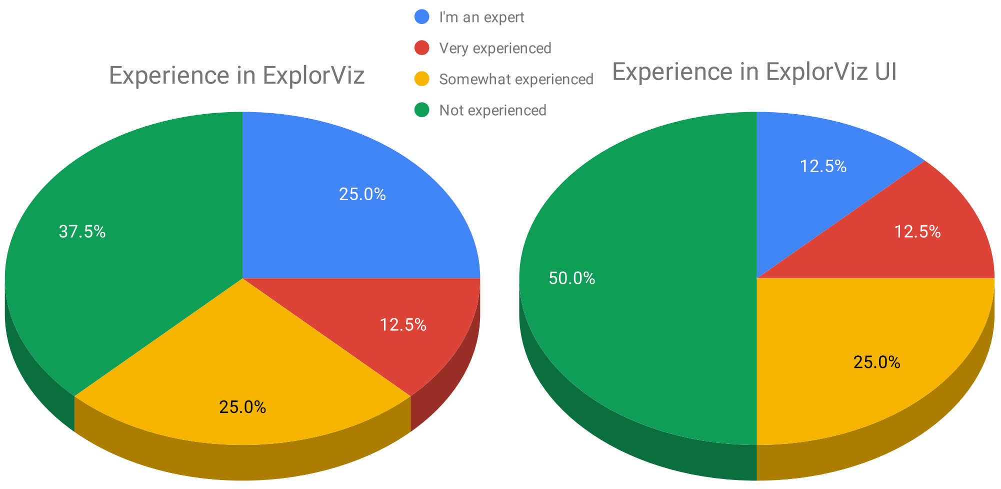

Figure 7.5. Experience of participants in *ExplorViz* and *ExplorViz UI*

the participant was then instructed to restart the tutorial. This was the case with 37.5% of the participants as seen in Figure 7.7. Even though the tutorial got stuck in these cases, after restarting the tutorial it was then completed by the participants. This was the only error preventing the execution of the tutorial. Despite of this error the tutorial was mostly rated as intuitive, as can be seen in Figure 7.6.

The participants did answer the question if they would have preferred more feedback from the `tutorial` and editor. In the case that they would have preferred more feedback they wrote down what exactly they wanted.

Regarding the `tutorial` most (75% of the participants) would not prefer more feedback. 25% of the participants would have preferred some feedback, they suggested that they might miss when the instruction text changes and would prefer a more visual feedback when the correct task was executed.

All the participants currently developing software related to *ExplorViz* found the `tutorial` at least `very intuitive`. The tasks in the `tutorial` were clear to most of the participants, however 25% of them did have problems with the last task.

The editor was perceived as less intuitive than the `tutorial` as seen in Figure 7.6, however both were rates as at least `somewhat intuitive`. The participants were requesting more feedback for the editor, 75% of the participants wanted more feedback.

All participants where able to perform the tasks in the editor and 87.5% of them felt like they learned the basic skills needed to use *ExplorViz* as also shown in Figure 7.7.

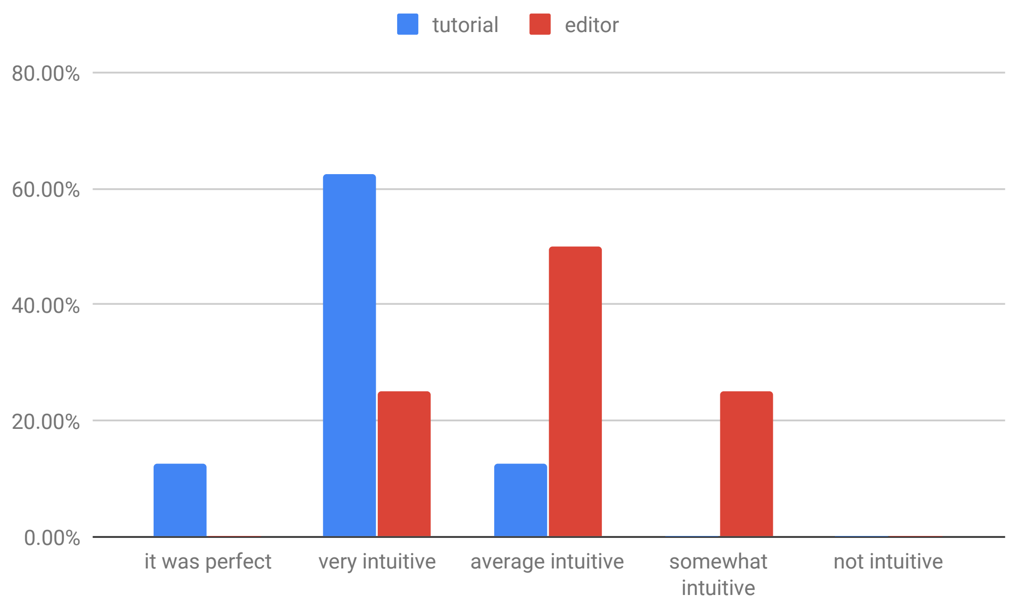

Figure 7.6. Feedback to Editor and Tutorial UI

7.4 Discussion

There were also some remarks about the visualization in general, the class size was criticized. Even though it is possible to change the visualization only for the tutorial framework, this would introduce a difference between the default visualization and `tutorial`. A better solution to solve a problem like that would be to allow customization for the visualizations.

The last step was phrased without a specific instruction, since the goal was to teach the participants that packages can convey information about the classes inside of them even when closed. Therefore it was expected that the last step might not be clear to everyone. The clarity of the step might have also been misjudged since the last step was the only step that required thought and did not provide a direct instruction on what to do. This might have caught the participants by surprise. They where not told in advance what was expected, instead the instruction was only to run the tutorial.

If it would have been mentioned in the introduction that the task might not always be clear and some combination of knowledge might be require, it might have caused the participants to consider even the last task as clear. The task did specify what the participant had to achieve it was however not stated which steps were required to achieve the task.

Regarding the editor the most requested additional feedback was to show the selected target in the step editor. This would not only make it easier for experienced users to identify the target selected, it would also make the editor more intuitive which allows new users to learn creating tutorials faster. It is possible that this request was made so often since there was note shown which said 'no target selected' if no target was selected. This might have caused the participants to assume that the information about the target would be shown in the same spot. The expectation in combination with the unexpected disappearing of the information might have caused additional feedback. Especially in the last steps before the part of the questionnaire regarding the tutorial editor might have caused more participants to make that remark.

Another request was to automatically open the hierarchy in the list, when adding a step or sequence. This was probably due to the confusion caused by the buttons, which are used to expand the view for sequences and steps, being shown even when no objects were contained. This confusion has probably also an impact on the perceived intuitiveness of the editor.

We also received the feedback that the possibility to navigate back to already completed `steps` would be helpful. A participant executed a `step` before completely reading the instruction. This could cause information to be missed.

It was also requested to add a button to to directly add a step or sequence from the list without opening the menu. This remark only occurred once and even though it would make the interface contain more buttons and therefore make it more confusing, there could be a problem if a user creates multiple `tutorials`. This user would have to click twice for each object created, this could get tiring when creating a large amount of `tutorials`. This could be resolved by adding a customization option for these extra buttons.

It was observed that the participants tried to add an element by clicking on the plus sign which are used to expand the hierarchy. This could be prevented by either hiding the button if there are no sub elements, or changing the symbol from a plus to an arrow.

The higher percentage of requests and suggestions for the editor, are indicative of the editor being less refined than the tutorial. We however also determined several factors which might have negative influence on the perception of the editor. As mentioned before the timing of some problems discovered might have had an influence on the requested improvements. Additionally the tutorial editor does also have more features therefore it might have been easier for the participants to decide on a feature they wanted to improve.

The instruction on how to select a landscape was given since the `tutorial` backend was not serializing the landscapes and therefore the list of available landscapes where empty, which does make the process convoluted. Another reason was that a bug was discovered and later fixed that caused the editor to get stuck when the timeline was clicked next to a dot representing a landscape.

7.5 Threats to Validity

The evaluation for the editor did contain less instructions than the `tutorial` itself, it is therefore possible that this had an effect on how intuitive the editor was perceived.

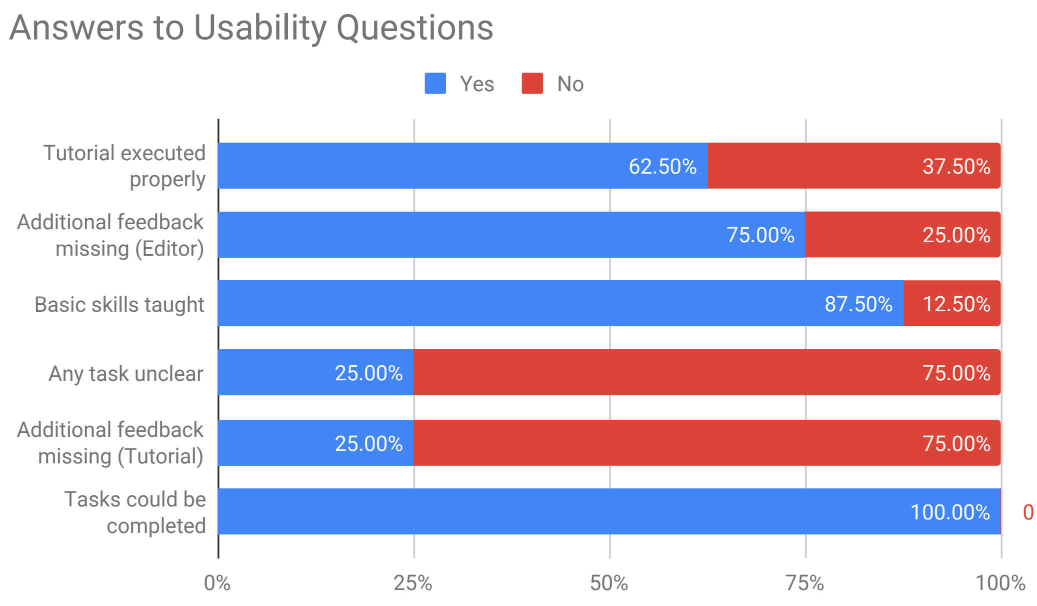

Figure 7.7. Answers to Usability Questions

The `tutorial` was designed to be an instruction to the basics of *ExplorViz*, more advanced `tutorials` might yield other results, since the users would be more experienced. The tools provided in the editor might not be suited for more complex `tutorials`. Since the `tutorial` framework is designed to be extended the main purpose for the framework would be to introduce the basic functionality. Therefore the questions asked and features introduced are representative of an actual `tutorial`.

The used `landscape` was very limited, both on regards of *machines* and *systems* in the landscape perspective as well as in regards to packages and classes. This might have an effect on the simplicity of the `tutorial`. We however did not evaluate the effectiveness of the `tutorial` itself, but instead the usability of the framework. We therefore recommend more research on how effective constructed `tutorials` can be .

The participants did not have a reason to learn how to use *ExplorViz*. The participants that were not familiar with *ExplorViz*, which were 37.5% of the participants, might not have any other reason than the evaluation to do the `tutorial`. Therefore they might not have been invested into learning the information presented. `Tutorials` should however also be useful if the user is less interested, and since 87.5% of the participants felt educated about the basic skills, and all where able to complete the tasks in the editor it can be assumed that the motivation through the participation in the evaluation was representative of users interested in *ExplorViz*.

Having 8 participants does qualify for a small project [Nielsen and Landauer 1993] since we are however evaluating the usability and further development will be needed for the `tutorials`, we determined that the number of participants is sufficient.

The instructions did not require the participants to create sensible steps for the tutorial, we therefore evaluated if the editor is sensible, not if it is optimized for professional use. More research is required to determine if the editor would be feasible for a large amount of tutorial or in a professional setting.

Chapter 8

Related Work

In this chapter we presents related work and explains the differences between them and this thesis. We are working with *ExplorViz* which was recently modernized redefining its structure, which was discussed by [Zirkelbach et al. 2018]. They also mention *ExplorViz Legacy*, which is no longer supported for the development of extensions, differences in structure are highlighted. This information is valuable, especially when trying to understand the principles relevant for development on *ExplorViz* and therefore also for development on extensions of *ExplorViz*. A general overview of strategies used in the development of *ExplorViz* and other currently developed software can be found in [Hasselbring 2018].

[Eichhorst 2017] did research the usability of the visualizations, this was however focused around understanding the software which was visualized, not the usability of *ExplorViz*. The insights can however be useful when developing new tutorials. Features required to achieve this also need to be considered when developing a tutorial framework.

[Krause et al. 2018] present the *Application Discovery* of *ExplorViz*, since this is an essential part of the application, it is also essential to be presented to potential users. Especially since the setup process of *ExplorViz* has to be done before it can be used. Also inexperienced users are more likely to use the feature without prior knowledge of the application. We therefore need to consider the information provided by [Krause et al. 2018].

Legacy Tutorial

In *ExplorViz Legacy* a tutorial system did already exist. It was however not suitable to be converted into the new structure. As [Finke 2014] mentions a modular system is preferred when developing extensions for a software which is in active development. The goal of [Finke 2014] however was to provide a tutorial mainly used in experiments. These functions where considered when designing the tutorial framework. Not implementing the specific functionality developed by [Finke 2014] allowed us to provide a more extensible framework. Time measuring might be useful for experiments it would however provide unnecessary workload for a tutorial outside of experiments.

Converting an extension to the new structure was done by [Häsemeyer 2017], some parts of the virtual reality implementation could be converted. We also considered this, however the technologies used to implement the old tutorial mode where not used in the new frontend. This was different for the virtual reality extension.

In the area of tracking user actions the work [Kosche 2013] does provide some insight.

Logging user actions could be used to track if an action was performed, however this would need and integration of the logging software. The frontend would need to process the logs, this would eliminate the independence of logging and user actions, and therefore not processing user action and instead only comparing the action to the expected value was chosen.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this chapter we are drawing conclusions about our work and if the goals we defined where reached, we also discuss possible future work.

9.1.1 Goals

Our first goal `G1` as mentioned in Section 1.2.1 was a Literature research and tool overview. An overview over tools used in *ExplorViz* can be found in Chapter 2. An overview over literature regarding *ExplorViz* can be found in Chapter 8. Most notable here are the references to the infrastructure papers about *ExplorViz*, which are elaborating upon the changes in structure of *ExplorViz*.

The second goal `G2` which we specified in Section 1.2.2 was to determine the needed features and develop a data structure which can support those features. The most prevalent feature of the tutorial is the detection of executed actions. For the detection a target needs to be determined and saved inside the structure. To avoid problems with types a string based solution was chosen. The target is a combination of and id, a type and the action to be performed. These three values, each stored as a string, can than be used to identify the target. Listeners for the expected action can then be used, upon detection the next instruction is shown. Instructions and the targeting values are stored in a structure we call a step. Multiple steps can be combined in a sequence. Since the `tutorials` are referencing objects in the `landscapes`, a landscape needs to be referenced. This is achieved by storing the timestamp, identifying the `landscape`, in the `tutorial`. Since a `tutorial` might contain steps referencing different `landscapes`, we decided to include the same kind of references in the sequence structure. This allows for a more diverse construction of `tutorials` without the need to create multiple tutorials for the same topic to change the landscape.

The next goal `G3` which is outlined in Section 1.2.3, was to develop a `tutorial` extension which includes the features determined in the earlier goals. This goal mentioned that `tutorials` will be able to exported and imported from and into the database. Even though `tutorials` can technically be exported from the *Mongo DB*, no feature was implemented to allow this from the frontend. This was decided with respect to the difficulties which prevented us from implementing the saving of landscapes. Not being able to persist the `landscapes` in combination with the tutorials would not allow for the landscape to be exported along with the `tutorial`. This would also cause all references to targets to be valid only if the landscape could be obtained in another way. Even though this feature was not implemented, the other parts of this goal where met. Two extensions where created which are using the service and addon structure of *ExplorViz*. The frontend is providing the user interface which is detecting the actions of the user and triggering the next step when the action was performed.

The next goal `G4` which is defined in Section 1.2.4 was reached by implementing the `tutorial` editor. When the tutorial editor is used by a user without the `admin` role, editing the `tutorial` is not possible instead it can only be started. Since the role system is currently very limited even in the development version of *ExplorViz*, we could not implement a more sophisticated integration with roles. This was also due to no relation existing between users and tutorials, which could be implemented into another service.

The goal `G5` mentioned in Section 1.2.5 was achieved as described in Chapter 7, the usability of the tutorial framework including the tutorial execution and editor was evaluated. The usability was tested by a usability experiment, where an example T was executed and a questionnaire regarding the steps executed was answered by the participants. The answers and additional feedback was collected via the questionnaire and later analyzed. Most of the participants rated the `tutorial` execution as 'very intuitive'.

9.1.2 Extensibility

Enabling other developers to extend the software is similarly important to the implemented features, we therefore provide a framework does fit into the general structure of *ExplorViz* and does not require the inclusion of technologies which are not already used in *ExplorViz*.

A tutorial framework inherently is connected to all of the features of an application. This is caused by the framework being able introduce all features. Providing an extensible framework therefore requires to consider the features of the tutorial framework separately from the features of its application. This allows to notice necessary features. For these features possible extensions have to be considered. Only a distinction can be made for how extensible a framework is. We then concentrated on developing a tutorial execution for the main feature of *ExplorViz*. Still considering how other tutorials would be implemented. This resulted in a framework that provides a solid foundation for providing many different customized and extended versions, which are all using the base functionality of the framework. Therefore we took essential steps in developing a rich tutorial environment for *ExplorViz*. This further enriches the extensions which already exist, are currently developed or will be developed in the future. For most extensions the implemented tutorial editor can be used, this allows for a central and uniform interface to manage tutorials for *ExplorViz*.

Even more complex extensions could profit from the tutorial execution and editor, since extending the backend and frontend does not render the extended framework incompatible to the basic framework. Therefore tutorials designed for the visualization could still be used even using an extended framework. The framework also supports the development version of *ExplorViz* which implies that it does not have to be reimplemented when

ExplorViz evolves. Instead it can be adjusted and improve by taking advantage of the newest improvements to *ExplorViz*.

Our evaluation determined the framework to be useful in executing and creating tutorials, which provides the basis for more tutorials being created by and for users of *ExplorViz*. New tutorials being created will create incentive for developers to support the framework with their extension. This in turn will enable users to learn what new extensions have to offer.

Even though the tutorial framework provides a solid foundation there are some aspects that could be improved upon, these are discussed in the next.

9.2 Future Work

Future improvements to the tutorial service are described in this chapter. These include visual improvements on features as well as structural improvements that could improve and extend the tutorial framework.

9.2.1 Improving the Data Structure

Improve JSON:API between Backend and Frontend

An improvement to the framework would be to improve the encapsulation of the interface between the frontend and backend. We are serializing the `landscape` in the frontend and sending the serialized string to the backend. This introduces a dependency since every backend would need to extract the objects from the string. A better variation would be to send a valid *JSON:API* object which could then be interpreted as by every service without further logic. This would also remove the need for the complicated serializing function since all elements could be loaded and saved via the interface and automatic serialization of *Ember Data* could be used. It might also be possible to achieve this by utilizing *Kafka*.

Referencing Landscapes by Id

Another aspect that could be improved upon is that the `tutorials` are referencing the `landscape` by `timestamp` instead of by id. Changing this would enable the frontend to directly reference the `TutorialLandscape`. This would however require some effort since the frontend also references landscapes by timestamp and not via id. Changing the references to id in the frontend would not eliminate the need for `timestamps`, however it would be easier to detect duplicate entries.

By referencing the `landscape` via `timestamp`, multiple `timestamp` entries with different ids and the same value can exist in a database. This causes all of the entries to be retrieved when the `timestamp` is used for the request to load `landscape`. Using ids as reference would eliminate this possibility. It would still be possible to insert multiple `timestamps` with the same `timestamp` value, however due to different ids only one could be referenced by `landscapes`.

Inverse References for Sequence and Tutorial

Another improvement could be done by including inverse references in `sequences` and `steps`, this would allow the frontend to load the parent object on demand. This could retire a function of the *tutorial-service* which determines the parent object via searching. This will improve the performance and would significantly reduce the amount of `tutorial` objects that are loaded when executing a `tutorial`.

9.2.2 Navigation in Steps

A feature that would be desirable and was also mentioned in the feedback of the evaluation was the ability to navigate in the steps after they are completed. This gives the user the ability to revise information already consumed. The feature could be implemented by showing a button that causes the previous steps to be selected. This would already be possible, however after reverting to the previous step the action would have to be executed again. Since this might cause confusion a better implementation would be to track the steps already executed by the user and allow skipping of already executed steps. This could be implemented in correlation with the tracking of user progress in a `tutorial`.

9.2.3 Referencing Other Elements

Another possible extension of the framework would be to select elements which are not part of the visualization. This would allow for the `tutorial` to reference buttons that affect the visualization without being part of it. This effect could be achieved by adding a new type to the target parameter. Additionally the id could be replaced by the id of a another element, e.g. DOM object. This could cause the frontend to instead of passing the parameters to the visualization, select the given id and register a *JavaScript* listener for the given action. The listener would then trigger the next step to be activated. This would also allow for the id to contain a *CSS* selector for the selection of multiple elements. This would cause the next step to be triggered if any of the selected elements would be executed. This feature would immensely expand the capabilities of the framework to identify targets, and therefore better tutorial could be written. It is important to consider what would happen if another entry in the navigation was targeted, since the tutorial might not be able to continue when the transition to the new page is completed.

9.2.4 Linking Elements

The linking of elements in the text of a `step` would allow editors of tutorials to reference elements visible to the user. This could be done by highlighting the element when the reference in the text is hovered or clicked. One possible variant of visualization would be a simple coloring of the element. This could be done by calling a function for the visualization, passing an id and type to the visualization. The visualization could then find the element based on the passed information and edit the element in such a way the it is highlighted. There are however aspects that need to be considered e.g. zoom , camera position and unopened classes.

Outside of the visualization elements could be referenced. This would enable to add a highlighting *CSS*-class to the element, and therefore highlighting it. If the element would not be visible it could also be made visible via *JavaScript*. It has to be considered what happens if the element does not exist in the current perspective, this would cause no element to be highlighted, messages could be shown if the selected element is not available in the current perspective.

Bibliography

[Chamey and Reder 1986] D. Chamey and L. Reder. Designing interactive tutorials for computer users. *Human-Computer Interaction* 2 (1986), pages 287–317. (Cited on pages 51, 55)

[Charney et al. 1988] D. H. Charney, L. M. Reder, and G. W. Wells. “Studies of elaboration in instructional texts”. In: *Effective documentation: What we have learned from research*. Volume 1. The MIT Press, 1988, pages 47–72. (Cited on page 12)

[Eichhorst 2017] F. Eichhorst. Analyse der microservices eines digitalen marktplatzes mittels explorviz. Master thesis. Kiel University, Oct. 2017. URL: `http://eprints.uni-kiel.de/39982/`. (Cited on page 61)

[Finke 2014] S. Finke. Automatische anleitung einer versuchsperson während eines kontrollierten experiments in explorviz. Masterarbeit. Kiel University, Sept. 2014. URL: `http://eprints.uni-kiel.de/25632`. (Cited on pages 26, 61)

[Häsemeyer 2017] T. Häsemeyer. Kollaboratives erkunden von software mithilfe virtueller realität in explorviz. Bachelor thesis. Kiel University, Sept. 2017. URL: `http://eprints.uni-kiel.de/39670/`. (Cited on page 61)

[Hasselbring 2018] W. Hasselbring. “Software architecture: past, present, future”. In: *The Essence of Software Engineering*. Edited by V. Gruhn and R. Striemer. Cham: Springer International Publishing, June 2018, pages 169–184. URL: `http://eprints.uni-kiel.de/43455/`. (Cited on page 61)

[Kosche 2013] M. Kosche. Tracking user actions for the web-based front end of explorviz. PhD thesis. Kiel University, 2013. (Cited on page 61)

[Krause et al. 2018] A. Krause, C. Zirkelbach, and W. Hasselbring. Simplifying software system monitoring through application discovery with explorviz. In: *Symposium on Software Performance 2018: Joint Developer and Community Meeting of Descartes/Kieker/Palladio*. Nov. 2018. URL: `http://eprints.uni-kiel.de/44502`. (Cited on pages 1, 61)

[Nielsen and Landauer 1993] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In: *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*. CHI ’93. Amsterdam, The Netherlands: ACM, 1993, pages 206–213. URL: `http://doi.acm.org/10.1145/169059.169166`. (Cited on page 60)

[Shneiderman 2003] B. Shneiderman. “The eyes have it: a task by data type taxonomy for information visualizations”. In: *The craft of information visualization*. Elsevier, 2003, pages 364–371. (Cited on page 18)

[Van der Meij 2008] H. van der Meij. Designing for user cognition and affect in software instructions. *Learning and Instruction* 18.1 (2008), pages 18–29. URL: http://www.sciencedirect.com/science/article/pii/S0959475206000776. (Cited on page 13)

[Zirkelbach et al. 2015] C. Zirkelbach, W. Hasselbring, and L. Carr. Combining kieker with gephi for performance analysis and interactive trace visualization. In: *Symposium on Software Performance 2015: Joint Developer and Community Meeting of Descartes/Kieker/Palladio*. Volume 35. 3. Softwaretechnik-Trends, 2015, pages 26–28. URL: http://eprints.uni-kiel.de/30101/. (Cited on page 1)

[Zirkelbach et al. 2018] C. Zirkelbach, A. Krause, and W. Hasselbring. On the modernization of explorviz towards a microservice architecture. In: *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*. Volume Online Proceedings for Scientific Conferences and Workshops. Ulm, Germany: CEUR Workshop Proceedings, Feb. 2018. URL: http://eprints.uni-kiel.de/42119/. (Cited on pages 19, 61)

[Zirkelbach et al. 2019a] C. Zirkelbach, A. Krause, and W. Hasselbring. *Hands-on: experiencing software architecture in virtual reality*. Research Report. Kiel University, Jan. 2019. URL: http://eprints.uni-kiel.de/45728/. (Cited on page 1)

[Zirkelbach et al. 2019b] 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 20, 21, 24)