Integration of Live Trace Visualization into Software Development

Bachelor´s Thesis

Lennart Ideler

Bachelor's Thesis
March 30, 2023

Software Engineering Group
Department of Computer Science
Kiel University

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

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Program comprehension is a crucial task in software development that developers traditionally do by understanding the source code of an application. Software visualizations are used as a complementary program comprehension tool for software development. We can see a gap between software development and software visualization where the status quo was to integrate an editor or source code viewer into a visualization tool. This approach limits the development process because the developer has to rely on the software development toolkit integrated into a visualization tool.

In this thesis, we integrate software visualization into software development while reducing context switches between visualization and development tools and preserving the IDE as a toolkit with all inherent benefits. We integrate ExplorViz, a live trace visualization tool, into an extension for Visual Studio Code that communicates with the ExplorViz visualization to perform user interactions in Visual Studio Code that are automatically translated to events in the ExplorViz visualization and vice versa. We evaluated the operability of our implementation in an example scenario. Results show that the extension successfully provides visual feedback in the IDE, representing the ExplorViz runtime behavior and bidirectional interactions to open a landscape component within the IDE and the respective source code while interacting with a landscape component.

Contents

Chapter 1

Introduction

1.1 Motivation

In software development, it is crucial to comprehend software to implement additional features and maintain an existing project. Program comprehension is traditionally done by understanding the source code of an application, and software visualization is used as a complementary software comprehension tool. Due to software evolution, developers are required to repetitively comprehend their software. An omission of this task can lead to gaps in the expected and actual runtime behaviors. The status quo for closing the gap between software development and software visualization was to integrate an editor or source code viewer into a visualization tool. This approach limits the usability of software development because the development tools have to be newly implemented in the visualization and often lack the capabilities and tools a traditional IDE provides. Integrating software visualization for runtime behaviors into software development has the potential to close the gap and reduce the time spent on software comprehension tasks that take developers about half their time during development [16]. ExplorViz is a tool that monitors, analyzes, and visualizes runtime behaviors as a software landscape in a standalone application. In this thesis, we will integrate the ExplorViz visualization into an IDE by implementing an extension that shows an ExplorViz visualization and supports interactions between the codebase and ExplorViz, respectively.

1.2 Document Structure

We will define the goals in Chapter 2 that describe what has to be done to integrate ExplorViz in an IDE. In Chapter 3, we present the foundations and technologies used or relevant for this thesis. The approach in Chapter 4 describes how we plan to achieve the implementations we make in Chapter 5. We will set up and evaluate the implementations in Chapter 6, outline some related work in Chapter 7, and present the conclusion and future work in Chapter 8.

Chapter 2

Goals

In this chapter, we will describe the goals resulting in a software product that integrates ExplorViz into software development.

G0: Define the Backend to Integrate ExplorViz Into an IDE

To integrate ExplorViz into software development, we implement a backend that can be used to propagate data between the ExplorViz frontend and an IDE. We define a communication scheme and needed data types for the backend to propagate. In the implementation, we will create a backend web application using events to handle connections from the IDE and ExplorViz.

G1: Extend the ExplorViz Frontend to support IDE Integration

Implement features in the ExplorViz frontend to connect with an event-driven backend web application to propagate the ExplorViz landscape state and provide support for interactions with an IDE. Interactions are, for example, code highlighting in the IDE based on the current ExplorViz landscape and interacting with the frontend to invoke interactions in the IDE.

G2: Implementation of a Visual Studio Code Extension that Integrates ExplorViz

We choose Visual Studio Code(VS Code) as the IDE to integrate the ExplorViz runtime behavior visualization. It is a free and lightweight editor highly customizable to the developer's needs due to custom extensions extending VS Code for development in various scenarios[13].

We implement an extension for VS Code that connects with a specific ExplorViz frontend instance of the program in question to visualize it in the VS Code extension. The extension will show the ExplorViz frontend in a split window for the developers to interact with. We will also implement interactions invoked by the ExplorViz frontend and interactions where the VS Code extension invokes interactions in the frontend. Furthermore, we will implement visual feedback about the connected ExplorViz frontend in VS Code to highlight relevant packages, classes, and methods.

G3: Evaluate the Integration of ExplorViz into a VS Code Extension

Lastly, we will show how to set up a sample project workspace to demonstrate our additions to ExplorViz, the implementation of the Visual Studio Code extension, and the backend. We demonstrate our implementations on different ExplorViz frontend sample projects and highlight limitations encountered during development and evaluation.

Chapter 3

Foundations and Technologies

Several foundations and technologies subject to this thesis are used to implement the desired software product which we explain in the following sections.

3.1 ExplorViz

ExplorViz is an open-source tool to support software comprehension and software visualization that is developed with web technologies to ensure platform-independent interoperability [12]. It uses a multi-level visualization of the software landscapes via a 3D visualization that is also available for virtual reality and augmented reality. ExplorViz analyzes and visualizes software system parts like architecture, runtime behavior, structure, and execution [6]. An additional aspect of ExplorViz is the collaboration to enable program comprehension, and software visualization for multiple users and not a single-user approach like in related tools [7].

We will focus on the visualization of the runtime behavior, which ExplorViz achieves as a hierarchical software landscape visualization using the city metapher [3]. The architectural concept of ExplorViz is shown in Figure 3.1 where we will focus on the visualization area that we will extend to integrate the ExplorViz frontend into an IDE. The visualization is built with components that are stacked upon one another to display their hierarchy as illustrated in Figure 3.2 in the *petclinic-costumer-service* foundation [4]. Components are, for example, Java packages (green/blue) and classes as a (gray) pillar inside a package. The foundation (gray) is a rectangular component and is the application visualized in the ExplorViz frontend [11]. We also have communication lines(communication links) that represent at least one executed method call between two classes but can also include multiple to reduce visual clutter [12]. In Figure 3.2 illustrated are the two different types of communication links, namely cross foundation communication links(violet) and foundation internal communication links(orange).

This thesis subject is to immerse the developer deeper into software development with an extension that integrates ExplorViz into an IDE. To do this, we will modify the ExplorViz frontend to support data exchange of the ExplorViz software landscape and implement interactions with an IDE.

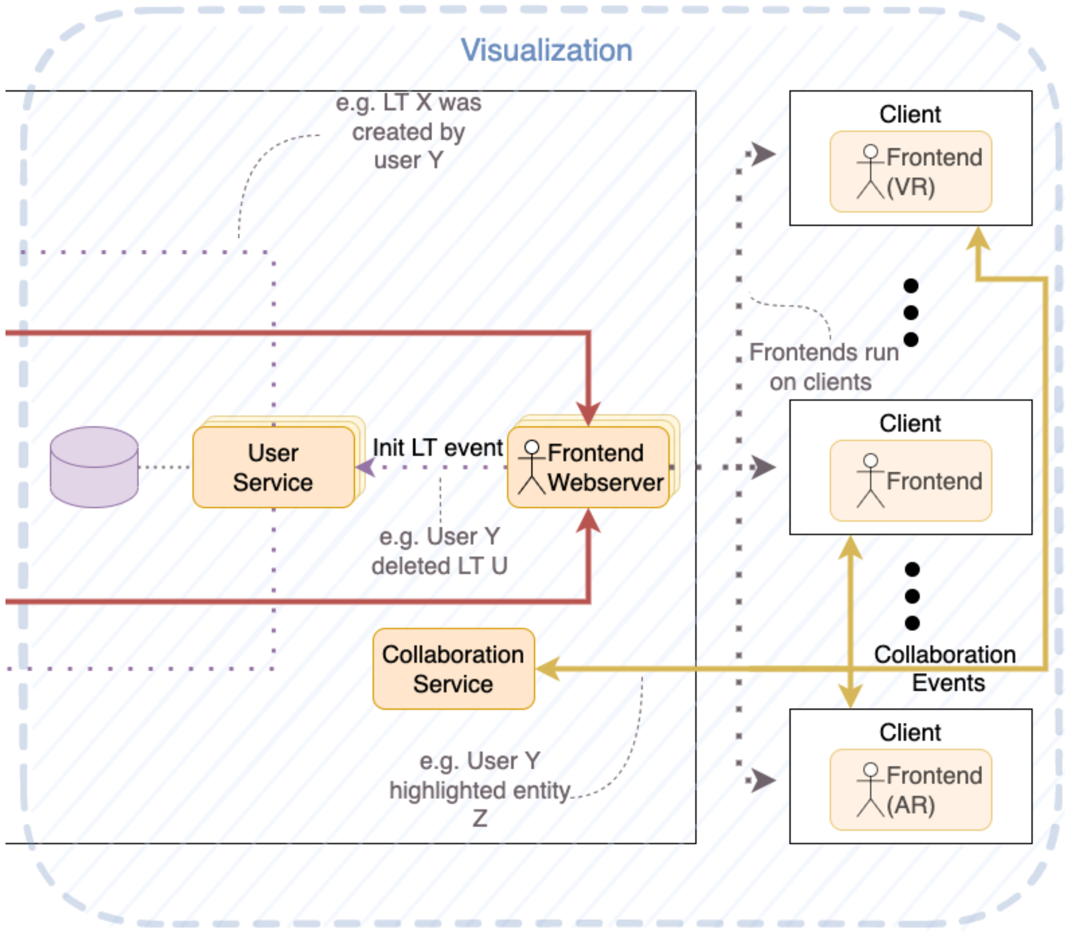


Figure 3.1. ExplorViz Architecture Concept: Visualization

3.2 VS Code Extension API

Visual Studio Code is a lightweight but powerful source code editor which is available for different operating systems like Windows, macOS, and Linux. It has built-in support for JavaScript, TypeScript, and Node.js but has a rich ecosystem of extensions for other languages and runtimes such as C++, C#, Java, Python, PHP, Go, and .NET [1].
The Extension API for VS Code has a multitude of possibilities to extend VS Code with an extension, e.g., changing the look of the IDE with themes, creating a webview of a webpage

[1]https://code.visualstudio.com/docs

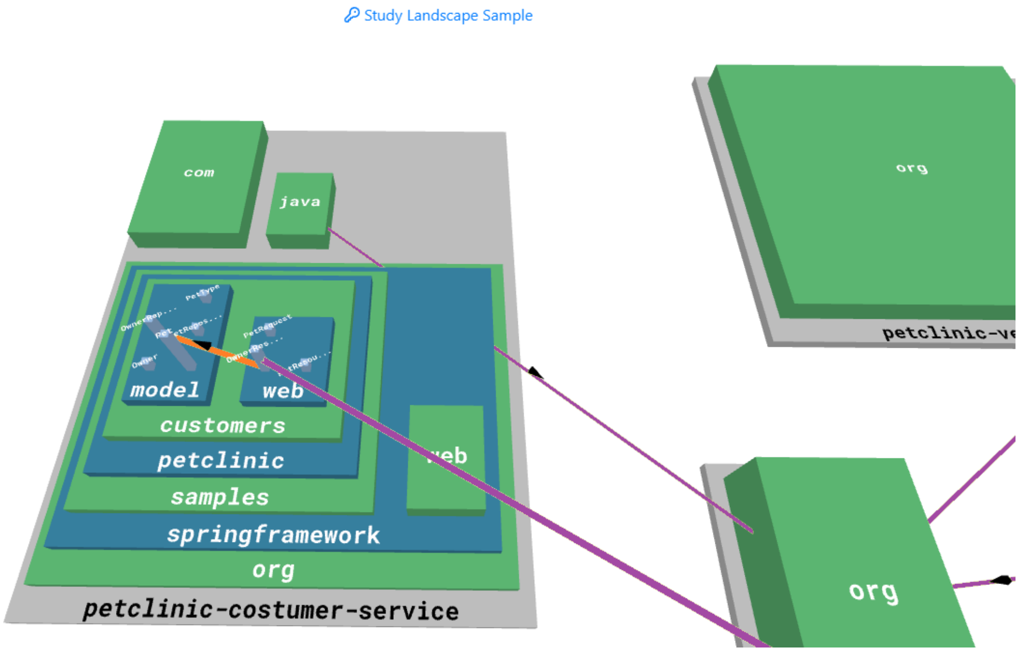


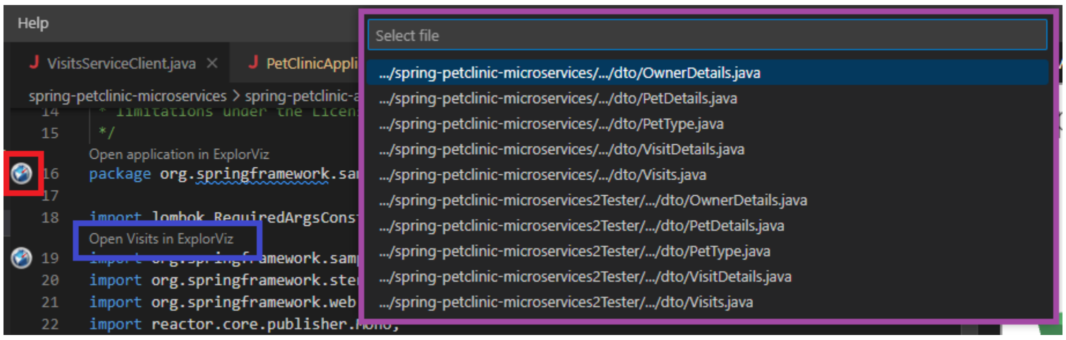
Figure 3.2. ExplorViz Landscape Example

built with HTML, CSS, and JavaScript, adding custom components for the views in the UI and provide support for a new programming language [13].
Especially do we need the VS Code Extension API to implement an extension to realize an embedded ExplorViz visualization to interact with in VS Code. Furthermore, do we use the VS Code API to visualize ExplorViz components within the IDE.

In Figure 3.3 are example features from the VS Code API like a selection prompt (violet), a gutter-icon (red) which can be added to a specific line in the code, and a codelense (blue) that acts as a visual indicator, but can also be used as a button to add functionality.

3.3 Express.js

Express.js is a server-side web framework for the JavaScript-based Node.js platform. It extends Node.js with tools that facilitate the development of modern web applications [5]. We use an Express.js application as our backend to enable communication between the ExplorViz frontend and the VS Code extension.


Help
VisitsServiceClient.java
PetClinicAppli
spring-petclinic-microservices > spring-petclinic-a
Open application in ExplorViz
Open Visits in ExplorViz
Select file

```
 * limitations under the Licen
 */
package org.springframework.sa

import lombok.RequiredArgsCons
import org.springframework.samp
import org.springframework.ste
import org.springframework.web
import reactor.core.publisher.Mono;
```

```
.../spring-petclinic-microservices/.../dto/OwnerDetails.java
.../spring-petclinic-microservices/.../dto/PetDetails.java
.../spring-petclinic-microservices/.../dto/PetType.java
.../spring-petclinic-microservices/.../dto/VisitDetails.java
.../spring-petclinic-microservices/.../dto/Visits.java
.../spring-petclinic-microservices2Tester/.../dto/OwnerDetails.java
.../spring-petclinic-microservices2Tester/.../dto/PetDetails.java
.../spring-petclinic-microservices2Tester/.../dto/PetType.java
.../spring-petclinic-microservices2Tester/.../dto/VisitDetails.java
.../spring-petclinic-microservices2Tester/.../dto/Visits.java
```

Figure 3.3. VS Code API Feature Preview

3.4 Socket.IO

Socket.IO is a JavaScript framework for real-time web applications. It provides bidirectional communication between web clients and servers using web sockets. The web clients running in the user's browser and servers running as a Node.js application use different parts of the framework but have a similar implementation process [15]. We use Socket.IO in conjunction with the Express.js framework to build the backend for the communication between ExplorViz and the VS Code extension.

3.5 Ember.js

Ember.js is a JavaScript framework for building web applications that work on different devices. It has a baked-in scalable UI architecture for a more efficient development approach [2]. The framework consists of features like Ember CLI as a toolkit to create, build, and develop Ember.js applications, a component system to reuse UI elements, a routing system for the application, and features to manage data inside the applications as well [14].

Ember.js is used for the ExplorViz frontend implementation, and we primarily need the Ember.js service and Ember.js Evented class. The service is an Ember.js object that lives throughout the duration of the application and can be made available in different parts of the application [14]. The class provides an internal event system, much like Socket.IO, that uses events and triggers to invoke application-scoped callbacks.

Chapter 4

Approach

In this chapter, we describe our approach to achieve the goals defined in Chapter 2 and which design decisions were made during the implementation. At first, we will discuss the communication model and technologies chosen to enable two-lane interactions from the ExplorViz frontend to the Extension, how to extend the existing frontend to support an VS Code extension, and what is needed for an extension to integrate the ExplorViz visualization.

4.1 Communication between ExplorViz Frontend and the VS Code Extension

To extend ExplorViz's versatility by integrating the visualization into VS Code, we have to establish a connection between the ExplorViz frontend and our VS Code extension. A reasonable thought would be extending the ExplorViz frontend to establish a connection via web technologies to the Visual Code extension or implementing a backend service integrated within the VS Code extension. Both options share the same flaw, as the ExplorViz frontend and the Visual Studio Code extension are pure frontend web applications. Integrating the backend service in the VS Code extension is also possible. However, the whole setup of the frontend communicating with the extension via a backend service would only work if every component runs on the computer locally because the backend has to be reachable in the network or over the internet.

Hence we want the ExplorViz frontend to be externally hosted; we will set up a standalone event-driven backend web server to handle the communications between the extension and frontend. Illustrated in Figure 4.1 is the communication flow of our different applications with a backend service as a proxy. The *IDE Ember.js Service* represents the modifications implemented in the ExplorViz frontend. The service can request interactions (green), which the VS Code extension performs and updates the current ExplorViz landscape data in the extension (violet). The *IDE Extension* can request interactions, which the frontend performs (yellow) and request a landscape data update (red).

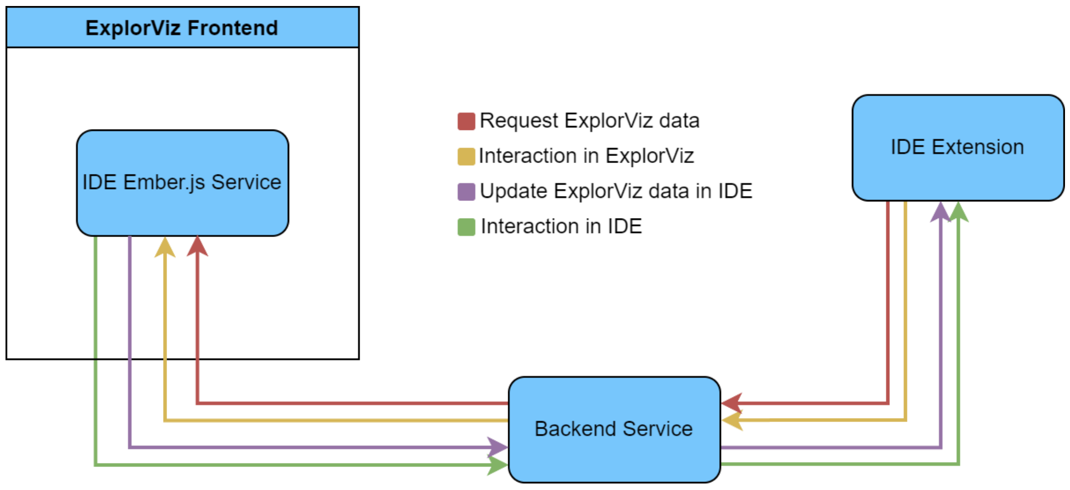


Figure 4.1. Communication diagram for ExplorViz Frontend and Extension

4.2 ExplorViz Frontend Extension

The ExplorViz frontend will be extended with web technologies like sockets to propagate the respective data model used within a frontend instance. Hence, we must understand how the application's data model works and what we especially need for our extension interactions to work. The following describes how we access the landscape data and interactions and how to communicate with the backend and extension.

Outline the Relevant ExplorViz Frontend Data Types

The ExplorViz frontend data we are looking for is split into two components, which we will discuss how to extract in Section 5.3.2 in the Implementation Chapter 5. The first component we need is an object called *ApplicationObject3D*, which is used as the main data object to represent the landscape data of a single foundation element in ExplorViz.

In Figure 4.2, one such instance of *ApplicationObject3D* would be a foundation which are components with a gray base foundation mesh, e.g., *petclinic-costumer-service*. Especially for a complete representation of the first data component, we need an array of all foundation instances of type *ApplicationObject3D* present in the current ExplorViz landscape.

The second component we need for our data model is the communication links representing method calls that are data-wise separated into cross-foundation communications and internal foundation communications. As shown in Figure 4.3, the cross-foundation communications (violet) represent communication between two classes inside distinct foundations or distributed system within the ExplorViz application for the specific project.

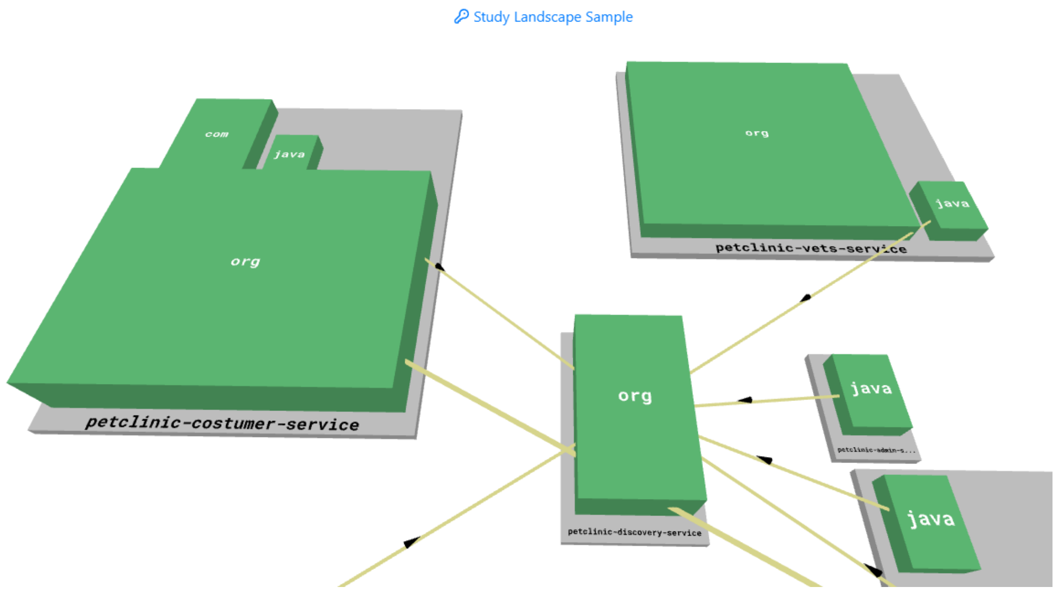


Figure 4.2. ExplorViz Frontend Example

The internal foundation communications (orange) are illustrated in Figure 4.3, which are method calls from one class to another inside the foundation.

To access and process the raw communications and foundations data, we have to extend the ExplorViz frontend with a custom Ember.js [2] service, which is also used for hooks and trigger points throughout the frontend. We have to process the raw ExplorViz data, which is based on the landscape components, to be suitable in size for sending it to our backend via web technologies. The concrete implementation for processing the raw data will be discussed further in Chapter 5.

Furthermore, the custom Ember.js service extends the ExplorViz frontend to communicate with our backend to process interactions triggered and received by the frontend or the extension with web technologies, e.g., a Socket.IO-client.

Access ExplorViz Frontend Interactions

An integral part of our Ember.js service is to expose and use interactions available within the ExplorViz frontend. Such interactions could be clicking on a landscape component within the frontend to highlight, open, or focus it. We do this by extending Ember.js components implemented within the frontend, such as the *BrowserRendering* class which handles the current data model for rendering and respective interactions on every component in the current landscape.

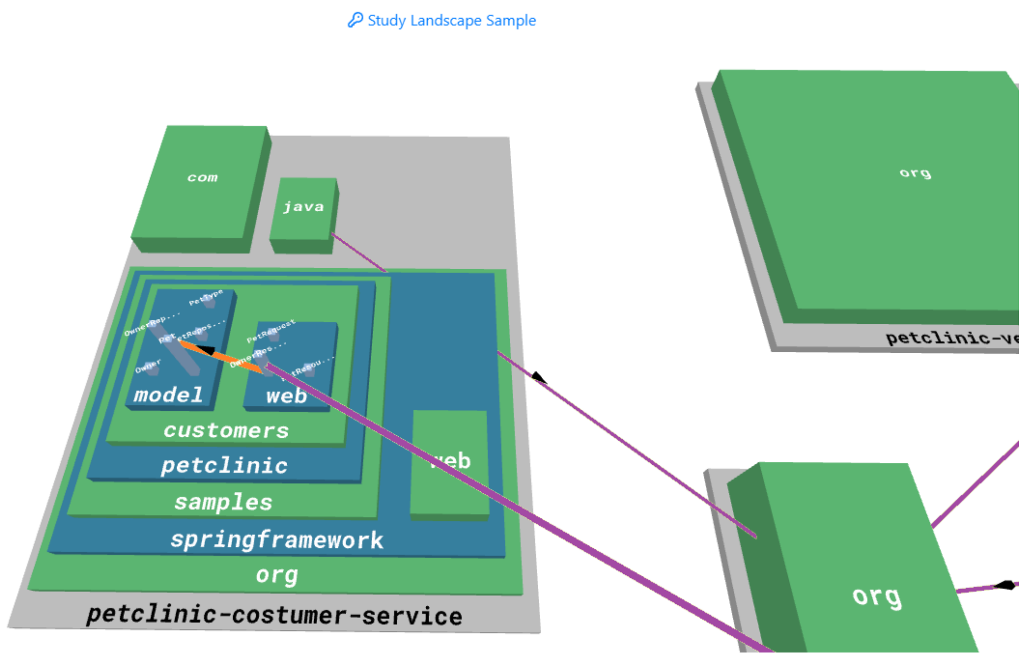


Figure 4.3. ExplorViz Frontend Communication Arrow Example

We can access those interactions by registering our custom Ember.js service within the *BrowserRendering* class to create hooks that we can use to mimic user interactions in the ExplorViz frontend propagated from our VS Code extension. Chapter 5 will go into detail about how and which interactions we choose to mimic and define the *BrowserRendering* class in context to our implementation.

At last, we will extend the Ember.js service with an additional settings tab in the already implemented menu of the ExplorViz frontend. This tab will provide a mockup to send monitoring data to the extension. The monitoring tool is a feature not currently implemented in ExplorViz but will provide info about classes or methods by any noteworthy metric, e.g., a long computation time.

4.3 Backend as a Proxy

As described earlier in this chapter, we want to implement a standalone backend service that handles the communication between the ExplorViz frontend and the VS Code Extension. The backend will not compute any data sent from the frontend or extension but only relays the data to the respective application. Multiple VS Code extension instances and ExplorViz frontend instances, respectively, can broadcast interactions to each other, with one backend instance handling the connections. We will implement the backend using an Express.js server based on Node.js while using Socket.IO events to transmit and receive data.

We will extend the ExplorViz ecosystem's visualization part as shown in Figure 4.4 where we deploy the *IDE Backend*① as a service that acts as a data proxy between new *frontend* and *VS Code Extension* clients②. Each *VS Code Extension* client is also a *frontend* client, and the backend can handle multiple instances respectively to proxy data and interactions.

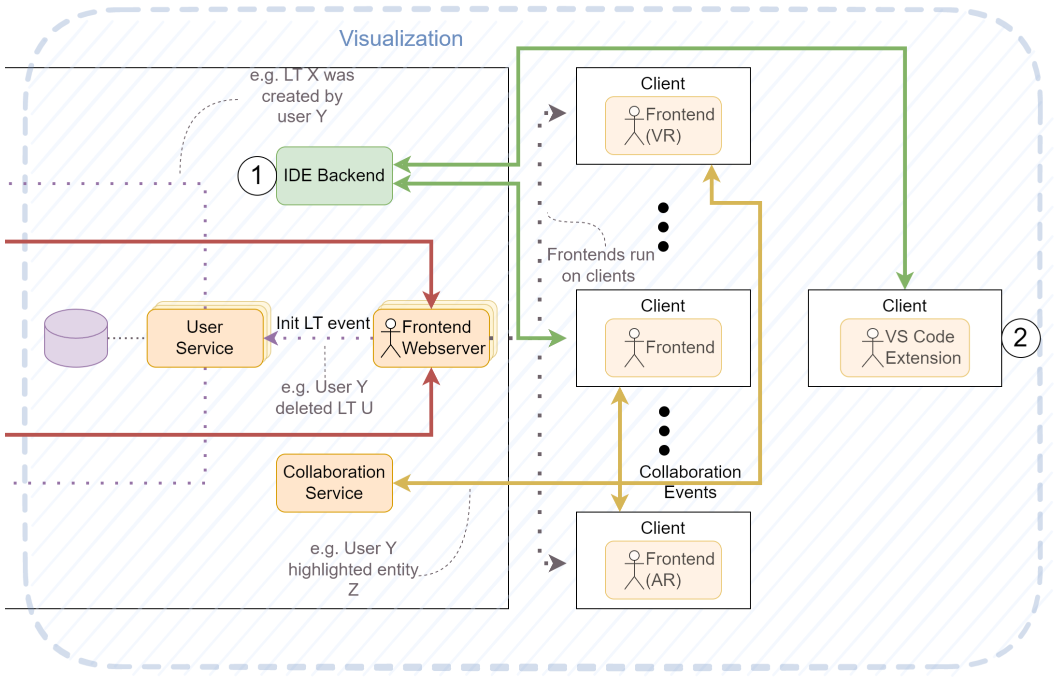


Figure 4.4. ExplorViz Architecture Concept: Visualization with IDE Extension

4.4 Visual Studio Code Extension

To integrate the ExplorViz frontend into a Visual Studio Code extension, we have to handle how we can display a frontend instance inside our extension without the context switch to a web browser. We also have to ensure socket events for data retrieval from the ExplorViz frontend and determine which functionality of the VS Code API we utilize to implement interactions triggered from the frontend and extension, respectively.

4.4.1 Integrating the ExplorViz Frontend in a VS Code Extension

Visual Studio Code is a lightweight, and extendable IDE built with web technologies. We will use a split window separating the IDE into two or more designated areas to integrate the ExplorViz frontend into Visual Studio Code without an external program like a standalone browser.

ExplorViz Frontend as a Split window

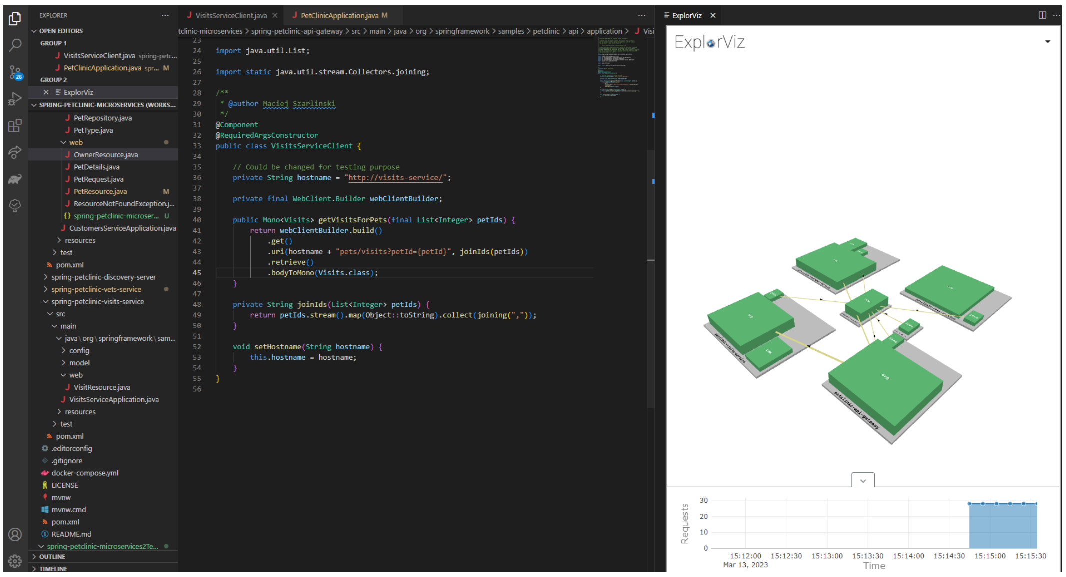
Figure 4.5 illustrates how the ExplorViz frontend will be displayed in Visual Studio Code. The IDE is split from left to right into the project file explorer, current open project files, and the ExplorViz frontend. The split section designated for ExplorViz in Figure 4.5 is not exclusively for an ExplorViz frontend, e.g., other files or extension editor-windows can use that split section as well [13]. The user has to manage the ExplorViz frontend window and open files after initialization for reasonable use like in other examples such as Markdown Preview Enhanced[1] or the LaTeX extension[2], which uses a split window as well.

The API for VS Code provides a feature to create a webview panel as a custom tab for a split window [13]. We can utilize this feature to create an iFrame[3] that embeds another HTML document within the webview HTML document. We set a URL as a source to the iFrame to show the content of a specific ExplorViz frontend. We have chosen an iFrame because other ways of including an external source would need the custom CSS and Javascript files mandatory to render the external website, especially the ExplorViz frontend, but with an iFrame, we can inject an external website into the webview panel as further described in Chapter 5.

4.4.2 Custom Visual Studio Code Commands and Utility functions

The VS Code API allows us to extend Visual Studio Code with custom commands [13]. Such commands can be used to open other files or create a webview but have many other possibilities for implementation. We need this utility to open a file inside our working directory as one of our interactions, which can be invoked from user interactions within

[1]https://marketplace.visualstudio.com/items?itemName=shd101wyy.markdown-preview-enhanced
[2]https://marketplace.visualstudio.com/items?itemName=mathematic.vscode-latex
[3]https://www.w3schools.com/tags/tag_iframe.ASP


EXPLORER
OPEN EDITORS
VisitsServiceClient.java
PetClinicApplication.java
ExplorViz
SPRING-PETCLINIC-MICROSERVICES
tclinic-microservices > spring-petclinic-api-gateway > src > main > java > org > springframework > samples > petclinic > api > application >
ExplorViz
Requests
Time
Mar 13, 2023

```java
import java.util.List;

import static java.util.stream.Collectors.joining;

/**
 * @author Maciej Szarlinski
 */
@Component
@RequiredArgsConstructor
public class VisitsServiceClient {

    // Could be changed for testing purpose
    private String hostname = "http://visits-service/";

    private final WebClient.Builder webClientBuilder;

    public Mono<Visits> getVisitsForPets(final List<Integer> petIds) {
        return webClientBuilder.build()
            .get()
            .uri(hostname + "pets/visits?petId={petId}", joinIds(petIds))
            .retrieve()
            .bodyToMono(Visits.class);
    }

    private String joinIds(List<Integer> petIds) {
        return petIds.stream().map(Object::toString).collect(joining(","));
    }

    void setHostname(String hostname) {
        this.hostname = hostname;
    }
}
```

Figure 4.5. Visual Studio Code split window with ExplorViz frontend

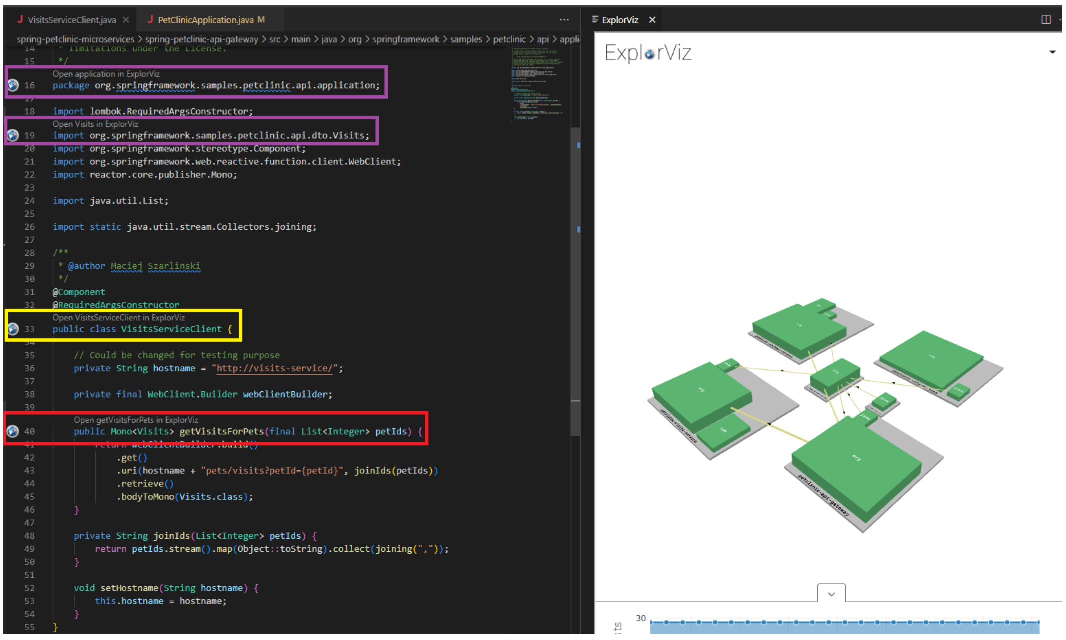
the ExplorViz frontend. Furthermore, examine the code currently open in the editor to filter for relevant classes and functions, which are both present in the ExplorViz frontend and the currently open file. We need this examination or simple symbol solving to send interaction commands to the ExplorViz frontend with codelenses.

4.4.3 Interactions with the ExplorViz Frontend

We want to link the relevant landscape data provided by the connected ExplorViz frontend to currently active files inside VS Code.

In Figure 4.6, is an example of how we can show which classes, packages, and functions are directly relevant to the current frontend. The class *VisitsServiceClient* (yellow) has a decorater [13] which adds an icon with the ExplorViz logo to indicate that the marked Java class is present and accessible in the currently connected ExplorViz frontend instance. As another visual and now interactive medium, we add a codelense to relevant classes, methods, and packages that open and focus the respective element in the ExplorViz frontend.

We will also provide visual feedback for packages and methods with decorators and codelenses as shown in Figure 4.6 where the available packages (violet) and method calls (red) represent the current ExplorViz landscape data. The concrete implementation of how we examine the ExplorViz and VS Code editor data will follow in Chapter 5.


VisitsServiceClient.java
PetClinicApplication.java M
spring-petclinic-microservices > spring-petclinic-api-gateway > src > main > java > org > springframework > samples > petclinic > api > appli
ExplorViz

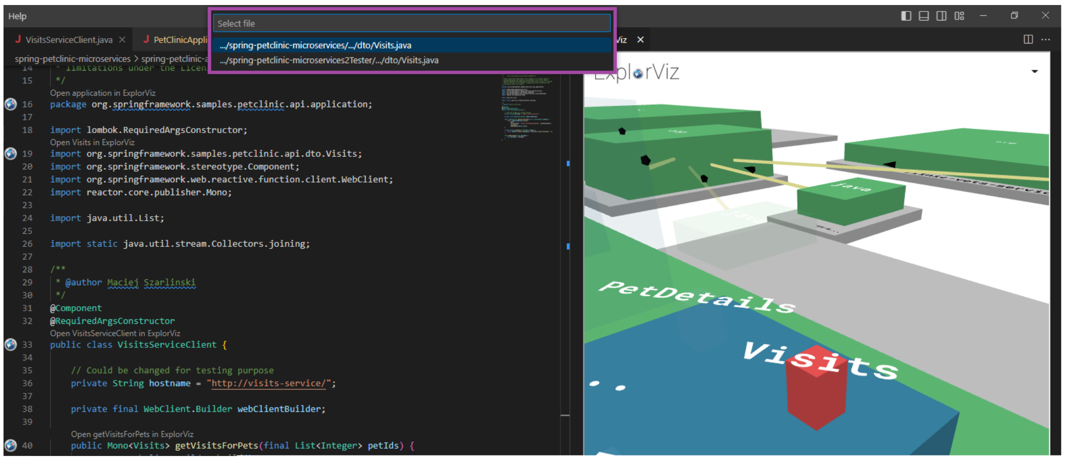
```
 * limitations under the License.
 */
Open application in ExplorViz
package org.springframework.samples.petclinic.api.application;

import lombok.RequiredArgsConstructor;
Open Visits in ExplorViz
import org.springframework.samples.petclinic.api.dto.Visits;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

import java.util.List;

import static java.util.stream.Collectors.joining;

/**
 * @author Maciej Szarlinski
 */
@Component
@RequiredArgsConstructor
Open VisitsServiceClient in ExplorViz
public class VisitsServiceClient {

    // Could be changed for testing purpose
    private String hostname = "http://visits-service/";

    private final WebClient.Builder webClientBuilder;

    Open getVisitsForPets in ExplorViz
    public Mono<Visits> getVisitsForPets(final List<Integer> petIds) {
        return webClientBuilder.build()
            .get()
            .uri(hostname + "pets/visits?petId={petId}", joinIds(petIds))
            .retrieve()
            .bodyToMono(Visits.class);
    }

    private String joinIds(List<Integer> petIds) {
        return petIds.stream().map(Object::toString).collect(joining(","));
    }

    void setHostname(String hostname) {
        this.hostname = hostname;
    }
}
```

Figure 4.6. Visual Studio Code Mockup of codelense

The codelenses and decorators are relevant for interactions from the extension to the ExplorViz frontend. However, we need additional inputs to consider interactions from the frontend to the extension. Firstly we have the case that the same codebase but with another root directory could be open in the current IDE session. For example, if the same classes and packages are used in distinct working directories. If we find duplicate classes or packages, we will show a selection based on that and the current ExplorViz landscape data.

In Figure 4.7 is an example where we clicked on the *visits* class in the ExplorViz frontend (red) and the selection (violet) prompted by the extension to select which Java file should be opened. This prompt illustrates the case when Java files have the same FQN within two distinct working directories.

Furthermore, can we click on packages in the ExplorViz frontend, which will prompt a selection that shows all available Java files inside the working directory respective to the given package's FQN. We especially have multiple results of the same Java file if they are redundant in two distinct working directories. In Figure 4.8, can we see the package that should be opened (red) and the selection (violet) that shows the available Java files in that directory or directories for redundant results.

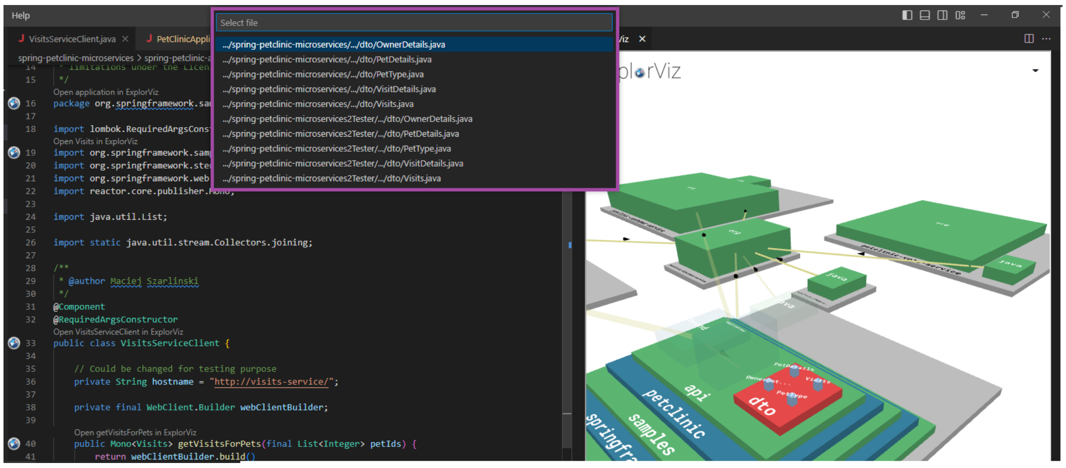


Select file

- .../spring-petclinic-microservices/.../dto/Visits.java
- .../spring-petclinic-microservices2Tester/.../dto/Visits.java

```
 * limitations under the Licen
 */
Open application in ExplorViz
package org.springframework.samples.petclinic.api.application;

import lombok.RequiredArgsConstructor;
Open Visits in ExplorViz
import org.springframework.samples.petclinic.api.dto.Visits;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

import java.util.List;

import static java.util.stream.Collectors.joining;

/**
 * @author Maciej Szarlinski
 */
@Component
@RequiredArgsConstructor
Open VisitsServiceClient in ExplorViz
public class VisitsServiceClient {

    // Could be changed for testing purpose
    private String hostname = "http://visits-service/";

    private final WebClient.Builder webClientBuilder;

    Open getVisitsForPets in ExplorViz
    public Mono<Visits> getVisitsForPets(final List<Integer> petIds) {
```

Figure 4.7. Visual Studio Code extension Java file selection from ExplorViz class



Select file

- .../spring-petclinic-microservices/.../dto/OwnerDetails.java
- .../spring-petclinic-microservices/.../dto/PetDetails.java
- .../spring-petclinic-microservices/.../dto/PetType.java
- .../spring-petclinic-microservices/.../dto/VisitDetails.java
- .../spring-petclinic-microservices/.../dto/Visits.java
- .../spring-petclinic-microservices2Tester/.../dto/OwnerDetails.java
- .../spring-petclinic-microservices2Tester/.../dto/PetDetails.java
- .../spring-petclinic-microservices2Tester/.../dto/PetType.java
- .../spring-petclinic-microservices2Tester/.../dto/VisitDetails.java
- .../spring-petclinic-microservices2Tester/.../dto/Visits.java

```
 * limitations under the Licen
 */
Open application in ExplorViz
package org.springframework.sa
 
import lombok.RequiredArgsCons
Open Visits in ExplorViz
import org.springframework.sam
import org.springframework.ste
import org.springframework.web
import reactor.core.publisher.Mono;

import java.util.List;

import static java.util.stream.Collectors.joining;

/**
 * @author Maciej Szarlinski
 */
@Component
@RequiredArgsConstructor
Open VisitsServiceClient in ExplorViz
public class VisitsServiceClient {

    // Could be changed for testing purpose
    private String hostname = "http://visits-service/";

    private final WebClient.Builder webClientBuilder;

    Open getVisitsForPets in ExplorViz
    public Mono<Visits> getVisitsForPets(final List<Integer> petIds) {
        return webClientBuilder.build()
```

Figure 4.8. Visual Studio Code extension Java file selection from ExplorViz package

Chapter 5

Implementation

5.1 Overview

This chapter will dive into the concrete implementation made for the backend, Visual Studio Code extension, and the additions made to the ExplorViz frontend with the approach steps from Chapter 4. We will start by defining mandatory data types and constants used in the backend, extension, and ExplorViz frontend, especially for communication purposes with Socket.IO.

5.2 Defining Relevant Data Types

At first, we define an enum with constants in Listing 5.1, so we can set a destination for our Socket.IO events to have simple event types for the backend that acts as a proxy to relay data of type *IDEApiCall* across the connected ExplorViz frontend and Visual Studio Code extension.

```
export enum IDEApiDest {
    VizDo = "vizDo",
    IDEDo = "ideDo",
}
```

Listing 5.1. IDEApiDest Definintion

Next, do we define a custom type *IDEApiCall* shown in Listing 5.2 that holds some more custom and basic types:

```
export type IDEApiCall = {
    fqn: string;
    meshId: string;
    occurrenceID: number;
    foundationCommunicationLinks: CommunicationLink[]
    action: IDEApiActions;
    data: OrderTuple[];
};
```

Listing 5.2. IDEApiCall Definintion

IDEApiCall.fqn Holds the full qualified name(FQN) from an ExplorViz frontend landscape component that was interacted with and triggers an interaction in the extension. Respectively if the user interacts with a codelense in the extension, the corresponding FQN will be stored to trigger an action in the ExplorViz frontend, like opening and focusing a component.

IDEApiCall.meshId The unique identifier for a landscape component in the ExplorViz frontend. The id is needed to access the component to compute any action, like a focus on that component.

IDEApiCall.occurrenceID The occurence for a software landscape that contains multiple applications, therefore multiple foundations. If the extension wants to perform an interaction like opening a component in ExplorViz, the *occurrenceID* is used to determine which component occurrence to open. The *occurrenceID* is a user selection when triggering the interactions within the extension as shown in Figure 5.2.

IDEApiCall.foundationCommunicationLinks The communication links from one foundation to another. This field is needed to distribute data dependent on the ExplorViz data lifecycle, which is described with the ExplorViz frontend modifications in Section 5.3.

```
export type CommunicationLink = {
    sourceMeshID: string;
    targetMeshID: string;
    meshID: string;
    methodName: string
}
```

Listing 5.3. CommunicationLink Definintion

The communication links within foundations are embedded in the landscape data, which we transform into the data fields from Listing 5.3 as follows:

CommunicationLink.meshID The identifier for the mesh of a communication link component within the ExplorViz frontend is used to compute user interactions on that component.

CommunicationLink.sourceMeshID The *meshID* of the origin mesh of the ExplorViz communication link component.

CommunicationLink.targetMeshID The *meshID* of the destination mesh of the ExplorViz communication link component.

CommunicationLink.methodName The name of the method that was called by the component with the *targetMeshID*

IDEApiCall.action The possible actions for our ExplorViz frontend and Visual Studio Code extension can be invoked via our Backend and, in some cases, also actions to invoke interactions within the IDE Ember.js service. The actions are defined as *IDEApiActions* in Listing 5.4 to set a standard for our socket events for interactions between ExplorViz and the extension, respectively.

```
export enum IDEApiActions {
    Refresh = 'refresh',
    GetVizData = 'getVizData',
    JumpToLocation = 'jumpToLocation',
    OpenAndFocusInExplorViz = 'OpenAndFocusInExplorViz',
    UpdateMonitoringData = 'UpdateMonitoringData',
    // SingleClickOnMesh = 'singleClickOnMesh',
    // ClickTimeline = 'clickTimeLine',
}
```

Listing 5.4. IDEApiActions Definintion

IDEApiActions.Refresh Used in the ExplorViz frontend to collect the current ExplorViz landscape data of the visualization, especially the *foundationCommunicationLinks* and provide updated ExplorViz landscape data for our VS Code extension. The backend handles the refresh event to distribute the data via the sockets.

IDEApiActions.GetVizData Suppose the VS Code extension is the destination. In that case, the socket event on this action will compute the current ExplorViz frontend data, which was transmitted and provides codelenses, code decorators, and especially interactions with the current ExplorViz landscape. The ExplorViz frontend, as the destination for this action, will collect and provide a trimmed version of the ExplorViz landscape components to prepare them for transmission over the socket. How this trimmed data looks will be discussed in Section 5.3.2.

IDEApiActions.JumpToLocation This action is used in two different segments within the applications. Firstly it is used in the VS Code extension as a receiving socket event to open a file inside the working directory depending on an interaction made in the ExplorViz frontend. Secondly, it is used within the frontend's Ember.js service with the Ember.js Evented framework to invoke an interaction in the extension after a landscape component in the frontend was clicked.

IDEApiActions.OpenAndFocusInExplorViz The VS Code Extension will send this action invoked by codelenses to open and focus a specific landscape component in the ExplorViz frontend.

IDEApiActions.UpdateMonitoringData This action updates the mocked-up monitoring data within the ExplorViz frontend for transmission to the VS Code extension.

Possible Additions: ClickTimeline and SingleClickOnMesh *ClickTimeline* and *SingleClick-OnMesh* are actions within the ExplorViz frontend which can be used for future interactions between the VS Code extension and ExplorViz. In Section 5.3, we show how to extend the ExplorViz frontend implementation to support interactions with a VS Code extension.

IDEApiCall.data The trimmed ExplorViz landscape data we use for transmission to the backend and VS Code extension is of type Listing 5.5, which is a tuple of the hierarchy model of the landscape components in the current ExplorViz frontend and the components present in the current frontend instance to ensure quick access for interactions like opening a landscape component.

```
export type OrderTuple = {
    hierarchyModel: ParentOrder;
    meshes: {
        meshNames: string[];
        meshIds: string[]
    };
};
```

Listing 5.5. OrderTuple Definintion

```
export type ParentOrder = {
    fqn: string;
    meshId: string;
    childs: ParentOrder[];
};
```

Listing 5.6. ParentOrder Definintion

The hierarchy model is of type Listing 5.6, which is a multi-leaf tree where each leaf holds the full qualified name, meshID, and children of an ExplorViz landscape component. An empty list of children in Listing 5.6 means that the current leaf is a class or a method.

In Listing 5.7, we have socket calls that we use throughout the applications with the definitions from Listing 5.1 and Listing 5.2, which are described as follows:

```
socket.on(IDEApiDest.VizDo, (data: IDEApiCall) => {...});
socket.emit(IDEApiDest.VizDo, data);
socket.broadcast.emit(IDEApiDest.VizDo, data);
```

Listing 5.7. Basic Socket.IO calls

socket.on(...) The backend listens to events if the ExplorViz frontend or the extension should perform an interaction using the *IDEApiDest* and relay the data as *IDEApiCall*. The VS Code extension and ExplorViz frontend will also have *socket.on(...)* calls but without the relay feature and will be mentioned in their respective sections.

socket.broadcast.emit(IDEApiDest, IDEApiCall) Broadcast to any connected Socket.IO client, which will perform the actions defined in the *IDEApiCall* if the destination meets the *IDEApiDest*

socket.emit(IDEApiDest, IDEApiCall) It will be used to trigger interactions to the backend for relaying data tasks, e.g., refreshing ExplorViz landscape data or sending data provided by a monitoring tool.

This concludes the data types used by the ExplorViz frontend, VS Code extension, and backend to send interactions and data across the sockets. In the following section, we will discuss the core features of our extension to the ExplorViz frontend implementation.

5.3 ExplorViz Frontend Extension

We extend the ExplorViz frontend[1] to support interactions with our VS Code extension and send data to our backend. We firstly create a custom Ember.js service [2] that lives throughout the duration of the ExplorViz frontend application and can be made available in different parts of the application to create custom hooks for already implemented features to interact with landscape components within the frontend and transmitting data from ExplorViz for further computation of our trimmed ExplorViz landscape data.

5.3.1 Custom Ember.js Service with Evented

We extend the ExplorViz frontend Ember.js implementation with a service called *ide-api.ts*(*IDEAPI*) that extends the Ember.js service with Evented [2]. Additionally, use it in already implemented features from the frontend to distribute the ExplorViz landscape data and foundation communication links to create callbacks for component interactions we want to use for the ExplorViz frontend and VS Code extension interactions.

[1]https://github.com/ExplorViz/frontend

```
import Service from '@ember/service';
import Evented from '@ember/object/evented';
...

export default class IDEApi extends Service.extend(Evented) {
    constructor(
      // handleSingleClickOnMesh: (meshID: string) => void,
      handleDoubleClickOnMesh: (meshID: string) => void,
      lookAtMesh: (meshID: string) => void,
      getVizData:
        (foundationCommunicationLinks: CommunicationLink[]) => VizDataRaw
    ) {
      super();

      // feature implementations...
    }
}
```

Listing 5.8. ExplorViz IDE API Service Base Implementation

The Listing 5.8 is the basic implementation for an Ember.js service that uses Evented. The constructor takes specific methods as callbacks used by the frontend to, e.g., focus a landscape component in ExplorViz so we can invoke it in further feature implementations within the service. The callback *handleDoubleClickOnMesh* from Listing 5.8 is used to open one layer within an ExplorViz foundation component to open it and must be called multiple times if the destination component is nested inside the foundation. The *getVizData* callback provides the current landscape data with internal communication links. It is possible to extend the constructor with more callbacks for future or already existing ExplorViz features to extend the interactions possible with the VS Code extension. We make additions the part *feature implementations* from Listing 5.8 in the following sections with important code snippets to illustrate how we implemented the necessary features and how we can use this service within already existing ExplorViz implementations.

5.3.2 Access and Prepare ExplorViz Landscape Data and User Interactions

A crucial part of our communication with the backend and, therefore, VS Code extension is the current data of the ExplorViz landscape that represents the runtime behavior. We also want to invoke functionality and features, like user interactions on a landscape component, already present in the ExplorViz frontend, which we hand over to our service to use in custom interactions with our VS Code extension.

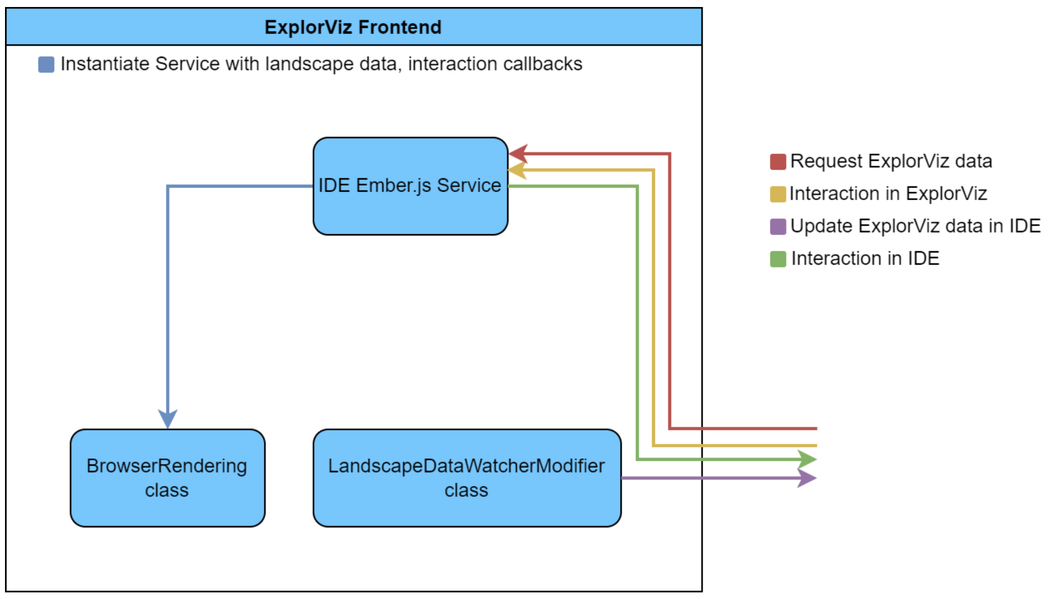

ExplorViz Frontend
🟦 Instantiate Service with landscape data, interaction callbacks
IDE Ember.js Service
BrowserRendering class
LandscapeDataWatcherModifier class
🟥 Request ExplorViz data
🟨 Interaction in ExplorViz
🟪 Update ExplorViz data in IDE
🟩 Interaction in IDE

Figure 5.1. ExplorViz frontend Modifications

Illustrated in Figure 5.1 are the different classes we must modify in the ExplorViz frontend:

▻ In the *BrowserRendering*[2] class we have access to the landscape data and user interactions. An instance of the *ApplicationRenderer*[3] class is part of *BrowserRendering* and is used to get the landscape data.

▻ The *LandscapeDataWatcherModifier*[4] class is used to get the cross foundation communication links and update the extension data according to the internal ExplorViz frontend update rate.

ExplorViz Landscape Data Access

In Section 4.2, we defined different data endpoints we have to implement to collect the current foundations and communication links from the ExplorViz frontend.

We determined that the current ExplorViz landscape data can be accessed within the class *BrowserRendering* with features from the already implemented *ApplicationRenderer*

[2]repo:ex-frontend/app/components/visualization/rendering/browser-rendering.ts
[3]repo:ex-frontend/app/services/application-renderer.ts
[4]repo:ex-frontend/app/modifiers/landscape-data-watcher.ts

class. We add a new function *getVizData* in Listing 5.9 to the *BrowserRendering* class with an *@action* decorator, which makes the function callable by reference inside the class [2].

```
// repo:ex-frontend/app/components/visualization/rendering/browser-rendering.ts
...
export default class BrowserRendering extends Component<BrowserRenderingArgs> {
    ...
    @action
    getVizData(foundationCommunicationLinks: CommunicationLink[]): VizDataRaw {
        const openApplications: ApplicationObject3D[]
            = this.applicationRenderer.getOpenApplications();
        const communicationLinks: CommunicationLink[]
            = foundationCommunicationLinks;
        openApplications.forEach(element => {
            ... // prepare ExplorViz data

        });
        return {
            applicationObject3D: openApplications,
            communicationLinks: communicationLinks
        };
    }
    ...
}
```

Listing 5.9. ExplorViz BrowserRendering getVizData Implementation 1

The *this.applicationRenderer* reference in Listing 5.9 of type *ApplicationRenderer* has a function *getOpenApplications* which returns the current state of the ExplorViz frontend as an array of *ApplicationObject3D* which we will use in the next section to compute our trimmed landscape data that we use for the communication with our backend and VS Code extension. The *foundationCommunicationLinks* parameter will be used to merge the communication links within the *ApplicationObject3D* with the foundation links that we extract from another class *LandscapeDataWatcherModifier*, which is used to update the internal ExplorViz landscape data. The ExplorViz frontend has an internal data update rate of 10 seconds that invokes the *LandscapeDataWatcherModifier* class and sends the updated landscape data to the VS Code extension.

```
// repo:ex-frontend/app/modifiers/landscape-data-watcher.ts
...
import {refreshVizData} from 'explorviz-frontend/services/ide-api';

export default class LandscapeDataWatcherModifier extends Modifier<Args> {
    ...
    @restartableTask *handleUpdatedLandscapeData(): Generator<any, any, any> {
        ...
        let cls: CommunicationLink[] = [];
        communicationLinks.forEach(element => {
            // convert ExplorViz DrawableClassCommunication to CommunicationLink
            // push results to cls
        });
        refreshVizData(IDEApiActions.Refresh, cls);
        ...
    }
    ...
}
```

Listing 5.10. ExplorViz LandscapeDataWatcherModifier class extension

We modify the LandscapeDataWatcherModifier class handleUpdatedLandscapeData function as in Listing 5.10 with a socket call to the backend to refresh the ExplorViz landscape data and *foundationCommunicationLinks*. The extension to Listing 5.10 also converts the currently available *communicationlinks* between foundations to our custom *CommunicationLink* type defined in Listing 5.3 to reduce size for suitable transmission via our backend.

ExplorViz Landscape Data Preparation

The type of the tuple returned by the *getVizData* function from Listing 5.9 is defined in Listing 5.11 which is the parameter for the function *VizDataToOrderTuple* in Listing 5.12 to convert the *CommunicationLink* and *ApplicationObject3D* arrays to our custom type *OrderTuple* from Listing 5.5.

```
export type VizDataRaw = {
    applicationObject3D: ApplicationObject3D[],
    communicationLinks: CommunicationLink[]
}
```

Listing 5.11. ExplorViz VizDataRaw Definintion

```
...
export default class IDEApi extends Service.extend(Evented) {
    ...
    function VizDataToOrderTuple(vizData: VizDataRaw): OrderTuple[] {
    const vizDataOrderTuple: OrderTuple[] = [];
    vizData.applicationObject3D.forEach(element => {
        const orderedParents = getOrderedParents(element.dataModel);
        const meshes = getFqnAndIDsForMeshes(orderedParents);
        let tempOT:
            OrderTuple = { hierarchyModel: orderedParents, meshes: meshes }
        tempOT =
            addCommunicationLinksToOrderTuple(tempOT, vizData.communicationLinks)
        vizDataOrderTuple.push(tempOT)
    })

    return vizDataOrderTuple;
}
...
}
```

Listing 5.12. ExplorViz IDE API Service extended with VizDataToOrderTuple

The implementation in Listing 5.12 uses several helper functions which extract different parts from the *ApplicationObject3D* to create an array of *OrderTuple* that we can use as a data field in the *IDEApiCall*.

We first extract the multi-leaf tree *ParentOrder* with a helper function *getOrderedParents*, which takes the essential parts from the *ApplicationObject3D*. The *ApplicationObject3D* is similar to the *ParentOrder* but is specific to some ExplorViz frontend classes we are not interested in, so we homogenize and cut out data for a tree with reduced size and cohesive type. The *meshes* array of an *OrderTuple* is a flattened variant of the *ParentOrder* tree to directly get a *meshID* with the *FQN* without searching the tree to perform interactions in some cases. Lastly, we add the internal and cross foundation *communicationlinks* to the *OrderTuple* and push the final results to our return value *vizDataOrderTuple*.

We have implemented the data callback for the *IDEAPI* service. However, we also have to modify the *BrowserRendering* class to prepare the callbacks needed for interactions with the ExplorViz frontend and VS Code extension.

Access and Preparation for ExplorViz User Interactions

The ExplorViz frontend has multiple user interactions possible over the web client. We want to simulate those interactions by clicking on, for example, a codelense in our VS Code extension which should highlight, focus, or open a landscape component in the frontend.

In the class *BrowserRendering* of the ExplorViz frontend, we already implemented the *getVizData* function as a callback that can be passed to the *IDEApi* service. We also have several functions already implemented in the frontend that perform interactions on landscape components. Namely, do we have *handleDoubleClickOnMesh*, a function from the *BrowserRendering* class that opens a landscape component in the frontend and *this.cameraControls.focusCameraOn* that controls the Three.js scene camera to focus on a specific component. For both actions, we respectively implement a new action that can be passed to our service. Those are *handDoubleClickOnMeshIDEAPI* Listing 5.13 and *lookAtMesh* Listing 5.14 that both take the *meshID* and extract the respective mesh from the landscape data on which we can perform 3D object-specific functions like *handleDoubleClickOnMesh* or *this.cameraControls.focusCameraOn*.

Our new actions share the same scope with the *getVizData* function that we implemented earlier in Listing 5.9.

```
@action
handleDoubleClickOnMeshIDEAPI(meshID: string) {
    let mesh = this.applicationRenderer.getMeshById(meshID)
    if (mesh?.isObject3D) {
        this.handleDoubleClickOnMesh(mesh)
    }
}
```

Listing 5.13. ExplorViz BrowserRendering extended with handDoubleClickOnMeshIDEAPI

```
@action
lookAtMesh(meshId: string) {
    let mesh = this.applicationRenderer.getMeshById(meshId)
    if (mesh?.isObject3D) {
        this.cameraControls.focusCameraOn(1, mesh);
    }
}
```

Listing 5.14. ExplorViz IDE API Service extended with lookAtMesh

We could extend more actions with a callback for our service if we wish to implement additional features that the ExplorViz frontend is using, for example, implementing a callback when the frontend opens a pop-up window to perform an interaction in the VS Code extension.

5.3.3 Instantiating the IDEAPI Service

The additions to the *BrowserRendering* class in the previous sections are used to instantiate the IDEAPI service. In Listing 5.15 is the addition we finally have to make to instantiate the *IDEAPI* service within the ExplorViz frontend.

```
import IDEApi from 'explorviz-frontend/services/ide-api';
...
export default class BrowserRendering extends Component<BrowserRenderingArgs> {
    ...
    @service
    ideApi: IDEApi;

    constructor(owner: any, args: BrowserRenderingArgs) {
        ...
        this.ideApi = new IDEApi(
            this.handleDoubleClickOnMeshIDEAPI,
            this.lookAtMesh,
            this.getVizData
        );
    }
    ...
}
```

Listing 5.15. ExplorViz IDE API Instantiation

This concludes almost all mandatory implementations to already implemented ExplorViz frontend features that we need to access the ExplorViz landscape data and interactions in the *IDEAPI* service and invoke interactions in the VS Code extension and ExplorViz frontend. Now we define the events for the Socket.IO client and Ember.js Evented events in the *IDEAPI* base implementation from Listing 5.8.

5.3.4 Socket.IO Client Events

The communication with our backend is done with a Socket.IO client[5] that we initiate during the activation of the *IDEAPI* service. In the service's constructor from Listing 5.16, we add socket events that our service listens to if the backend relays or sends a request to the *IDEAPI* service. We only add one *IDEApiDest* event and filter the specific action with a switch case to determine which *IDEApiAction* is requested within the sent *IDEApiCall* data. Before we enter the switch case to check the actions requested, we call the *getVizData* and *VizDataToOrderTuple* functions to collect and prepare the data in the form of *IDEApiCall*.

We specifically have the two *IDEApiActions OpenAndFocusInExplorViz* and *GetVizData* that the *IDEAPI* service will execute and perform on requests. The first will open a specific landscape component within the ExplorViz frontend based on the *FQN* and *occurrenceID* and perform the callbacks on landscape components added in the *BrowserRendering* class.

[5]https://www.npmjs.com/package/Socket.IO-client

```
import { io } from 'socket.io-client';
let socket = io(httpSocket);
...
export default class IDEApi extends Service.extend(Evented) {
    constructor(...) {
        socket.on(IDEApiDest.VizDo, (data: IDEApiCall) => {
            const vizDataRaw = getVizData(foundationCommunicationLinksGlobal);
            const vizDataOrderTuple = VizDataToOrderTuple(vizDataRaw)

            switch (data.action) {
                case IDEApiActions.OpenAndFocusInExplorViz:
                    OpenObject(
                        handleDoubleClickOnMesh,
                        data.fqn,
                        data.occurrenceID,
                        lookAtMesh,
                        vizDataOrderTuple
                    );
                    break;
                case IDEApiActions.GetVizData:
                    // IDEApiCall.action: IDEApiActions.GetVizData
                    emitToBackend(IDEApiDest.IDEDo, IDEApiCall);
                    break;
            }
        });
    }
}
```

Listing 5.16. ExplorViz IDE API Socket

The *OpenObject* function called in Listing 5.16 resets the current foundations to ensure a correct visual representation and opens components recursively until the FQN from the *IDEApiCall* is found. The second action that can be requested from the VS Code extension sends the current representation of the landscape data to the backend, which relays the *IDEApiCall* to the VS Code extension.

With this Socket.IO client setup, we can request user interactions, usually made with the ExplorViz frontend web client. Especially can the VS Code extension invoke user interactions to, e.g., open a landscape component in the frontend. We also need an implementation that invokes user interactions made with the ExplorViz frontend to request an interaction in the VS Code extension where the Ember.js service extension Evented comes into play.

5.3.5 Ember.js Evented Hooks and Triggers

Ember.js Evented works similarly to Socket.IO events but is scoped to the current application. In Listing 5.17, we make our final addition to the *IDEAPI* constructor, which defines an Evented event that computes if the *JumpToLocation* action is triggered. The *JumpToLocation* event is triggered when the user single-clicks on a landscape component in ExplorViz and uses the *meshID* to emit the relevant data for the *JumpToLocation* action to the backend, which relays the request to the VS Code extension to open the specified package's, class's or method's file location.

```
...
export default class IDEApi extends Service.extend(Evented) {
    constructor(...) {
        socket.on(IDEApiDest.VizDo, (data: IDEApiCall) => {...});

        this.on(IDEApiActions.JumpToLocation, (object) => {
            const vizDataRaw: VizDataRaw =
                getVizData(foundationCommunicationLinksGlobal);
            const vizDataOrderTuple: OrderTuple[] =
                VizDataToOrderTuple(vizDataRaw);

            // IDEApiCall.action: IDEApiActions.JumpToLocation
            // IDEApiCall.meshId: getIdFromMesh(object),
            emitToBackend(IDEApiDest.IDEDo, IDEApiCall);
        });
    }
}
```

Listing 5.17. ExplorViz IDE API Evented

The trigger for the *JumpToLocation* event uses the already instantiated *IDEAPI* service in *BrowserRendering* with one addition to the action that handles a single click in the ExplorViz frontend. In Listing 5.18, we add a *trigger(...)* call on the *IDEApi* instance to trigger the specified *IDEApiAction* event. For further similar interactions, we have to implement another Evented event in the *IDEApi* and trigger the respective *IDEApiActions* within a function as shown in Listing 5.18.

```
// repo:ex-frontend/app/components/visualization/rendering/browser-rendering.ts
...
export default class BrowserRendering extends Component<BrowserRenderingArgs> {
    ...
    @action
    handleSingleClick(intersection: THREE.Intersection) {
        this.ideApi.trigger(IDEApiActions.JumpToLocation, intersection.object);
        ...
    }
    ...
}
```

Listing 5.18. BrowserRendering Ember Evented Trigger

With the Socket.IO client events and the events defined with Evented, we can request and perform interactions with the VS Code extension and include additional ExporViz features if necessary. Lastly, we will implement another feature that will support delivering ExplorViz analytics data to the VS Code extension.

5.3.6 Monitoring Mockup

A future feature for ExplorViz will be a tool that analyzes Java classes and methods with metrics that determines if any of these should be reviewed by a developer. Such metrics could be a runtime analysis that returns methods that exceed or take a specified amount of resources and computation time.

The return data for such analytics could look like Listing 5.19 where we define fields to specify a component by FQN and a description. The FQN is needed to open the specific class or method inside the IDE and the description is the results from the monitoring tool.

We add a new *ArSettingsSelector* class with *xr-vscode-extension-settings.ts* and Ember.js template component with *xr-vscode-extension-settings.hbs* to the ExplorViz frontend that has an action Listing 5.20 to send the data provided by the tool to the VS Code extension. This class and the respective template are already used in the frontend to render a menu item in the Sidebar menu of the ExplorViz frontend, which we took as a guideline to add a menu item for the extension. A button added in the Ember.js template invokes the action and passes the monitoring data to the VS Code extension, where it can be visualized.

```
@action
monitoring() {
    const monitoringData: MonitoringData[] = [
        {
            fqn: "mockup␣FQN",
            description: "mockup␣description"
        }
    ]
    sendMonitoringData(monitoringData);
}
```

Listing 5.19. MonitoringData

We call the exported function *sendMonitoringData* from the *IDEApi* service in the *@action* from Listing 5.19 which is an *socket.emit* with destination *IDEApiDest.IDEDo*, action

IDEApiActions.UpdateMonitoringData and the monitoring data. The backend then relays the provided data to the VS Code extension.

```
export type MonitoringData = {
    fqn: string,
    description: string
}
```

Listing 5.20. MonitoringData Menu Action

5.4 Express.js Backend Server

The backend for the communication between the ExplorViz frontend and VS Code extension is implemented with the Express.js framework, which uses a Node.js web server and provides additional support to implement sockets with Socket.IO.

```
import express from "express";
import { Server } from "socket.io";
import http from "http";
import { IDEApiActions, IDEApiCall, IDEApiDest } from "./types";

const backend = express();
const server = http.createServer(backend);

const port = 3000;
const corsExplorVizHttp = "http://localhost:4200";
const maxHttpBufferSize = 1e8;

const io = new Server(server, {
    maxHttpBufferSize: maxHttpBufferSize,
    cors: {
        origin: corsExplorVizHttp,
        methods: ["GET", "POST"],
    },
});
```

Listing 5.21. Backend Express Server with Socket.IO

In Listing 5.21, we first define the Express.js backend, which is used to create a Node.js HTTP server. We also defined some variables to specify the port, CORS policy, and buffer size. Those variables can be adjusted with environment variables for deployment. We defined a maximum buffer size for the maximum amount of data possible to send over the socket because the standard size is 1MB which is easily exceeded for larger ExplorViz landscapes. A socket with Socket.IO automatically aborts the connection if the buffer size is too big [15]. Lastly, we create the Socket.IO server *io*, which we use to define the events on connected sockets that relay data from the ExplorViz frontend and VS Code extension, respectively.

```
io.on("connection", (socket) => {
    socket.on(IDEApiDest.VizDo, (data: IDEApiCall) => {
        console.log("vizDo", data);
        socket.broadcast.emit(IDEApiDest.VizDo, data);
    });
    socket.on(IDEApiDest.IDEDo, (data: IDEApiCall) => {
        console.log("ideDo", data);
        socket.broadcast.emit(IDEApiDest.IDEDo, data);
    });
});
```

Listing 5.22. Backend Socket Events

In Listing 5.22 we defined the two events on connected sockets that depend on the *IDEApiDest*, which describes the recipient for the action set in the *IDEApiCall*. The backend then broadcasts the data to the destination with the attached *IDEApiCall.data*, and the receiving socket event handles the action and *IDEApiCall.data*.

5.5 Visual Studio Code Extension

We implement a Visual Studio Code extension to integrate the live trace visualization ExplorViz into software development. Starting with the integration of the ExplorViz frontend into the extension with an iFrame, creating custom VS Code commands that we use to request landscape data and invoke interactions from the frontend, handling user input for interactions, establishing Socket.IO events, creating visual feedback in the IDE that represents the runtime behavior and implementing the interactions the ExplorViz frontend can request to the VS Code extension.

5.5.1 Basic Custom Visual Studio Code Extension

The basic implementation for our extension Listing 5.23 consists of an *activate* function where we have access to the *context* of VS Code, which includes currently open editors, the working directory, and more to use for specific purposes. The *activate* function is called on specific parameters set in the *package.json* and starts the extension with the available context. Those parameters are, for example, custom commands we implement that are callable from the VS Code command context menu *Default:F1*, or the extension could activate when the IDE is started.

```
import * as vscode from "vscode";
export async function activate(context: vscode.ExtensionContext) {
    ...
}
```

Listing 5.23. Extension Basic Implementation

5.5.2 ExplorViz Frontend as an iFrame

Visual Studio Code can split editors into tabs, which we use with the *WebviewPanel* feature from the VS Code API that can render HTML with CSS and Javascript [13]. To integrate the ExplorViz frontend without the heavy CSS and Javascript overhead that would be necessary, we integrated the frontend with an iFrame that includes a website to an HTML document. For that, we have to implement our first custom Visual Studio Code command, which creates a webview and sets the HTML content, including the iFrame Listing 5.24 that includes the ExplorViz frontend URL in the *body* HTML tag.

```
<iframe src="${websiteUrl}" width="100%" height="100%"></iframe>
```

Listing 5.24. Extension Webview iFrame

5.5.3 Custom IDE Commands

To add a new command to the extension, we have to modify the *package.json* with a *contributes.commands* field and add additional commands we want to implement to the array. Especially do we add a *webview* and *OpenInExplorViz* Listing 5.25 command to open the extensions web view in a split window and to open a landscape component in the ExplorViz frontend while interacting with the Java code.

```
{
    ...
    "contributes": {
        "commands": [
            {
                "command": "explorviz-vscode-extension.webview",
                "title": "ExplorViz␣webview"
            },
            {
                "command": "explorviz-vscode-extension.OpenInExplorViz",
                "title": "OpenInExplorViz"
            },
    }
    ...
}
```

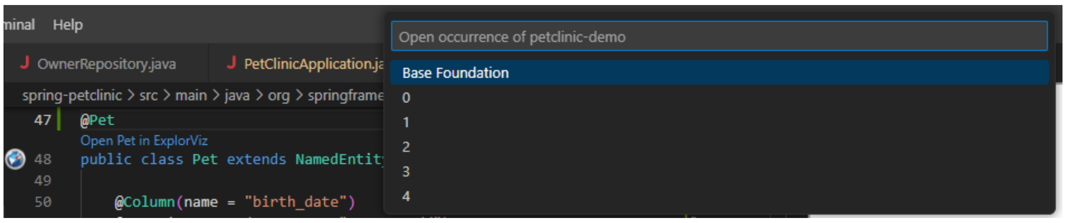
Listing 5.25. Extension Package Commands

The first command *webview* Listing 5.26 uses the VS Code API to register a new command that can be invoked in Visual Studio Code. We also create a new window that represents the web view with the ExplorViz frontend included with an iFrame. The iFrame is set by the function *getWebViewContent*, which returns an empty HTML document with only an iFrame in the body tag and some additional CSS styling.

```
let webview = vscode.commands.registerCommand(
    "explorviz-vscode-extension.webview",
    function () {
        vscode.window.showInformationMessage("Show␣ExplorViz␣Visualization");
        let panel = vscode.window.createWebviewPanel(...);
        panel.webview.html = getWebviewContent();
    }
);
context.subscriptions.push(webview);
```

Listing 5.26. Extension Webview Command

For the second command, which requests the *IDEApiActions.OpenAndFocusInExplorViz* to the connected ExplorViz frontend that opens a specific package, method, or class chosen in VS Code, we must first determine if the frontend landscape data uses occurrences of multiple package instances in a foundation. The VS Code API provides a selection tool to select between the occurrences as shown in the illustration Figure 5.2. The *OpenAndFocusIn-ExplorViz* command is used as a callback in every codelense as shown in Figure 4.6 from the approach, which we will implement in Section 5.5.4.


minal
Help
OwnerRepository.java
PetClinicApplication.ja
spring-petclinic > src > main > java > org > springframe
Open occurrence of petclinic-demo
Base Foundation
0
1
2
3
4

```
@Pet
Open Pet in ExplorViz
public class Pet extends NamedEntit

    @Column(name = "birth_date")
```

Figure 5.2. Select Occurrence to open in ExplorViz

```
let OpenInExplorViz = vscode.commands.registerCommand(
    "explorviz-vscode-extension.OpenInExplorViz",
    function (name: string, fqn: string, vizData: OrderTuple[]) {
      // Check if the ExplorViz uses occurences
      // if yes show selection which to open
      // Request OpenAndFocusInExplorViz action with FQN
    }
  );
  context.subscriptions.push(OpenInExplorViz);
```

Listing 5.27. Extension OpenInExplorViz Command

Both commands are scoped within the *activate* function from Listing 5.23 and push the commands to the current context of Visual Studio Code to make them accessible during the extensions runtime.

Update ExplorViz Landscape Data in the Extension

The ExplorViz frontend *IDEApi* service in Section 5.3.2 updates the current landscape data corresponding to the updates that occur in the frontend implementation and sends it to the extension for further processing and visual feedback. We can also use the already implemented sockets from the *IDEApi* service to send a *IDEApiActions.GetVizData* action to the ExplorViz frontend to refresh the landscape data manually. The *refreshVizData* function in Listing 5.28 is doing such an emit with the ExplorViz frontend as the *IDEApiDest*. We could call the refresh function with, for example, a keyboard shortcut or specific events that trigger in Visual Studio Code, as in Listing 5.28 where we use the VS Code API to add an event to the *activate* function that fires if the user saves the current document.

```
vscode.workspace.onDidSaveTextDocumen(async (e) => {
    refreshVizData();
});
```

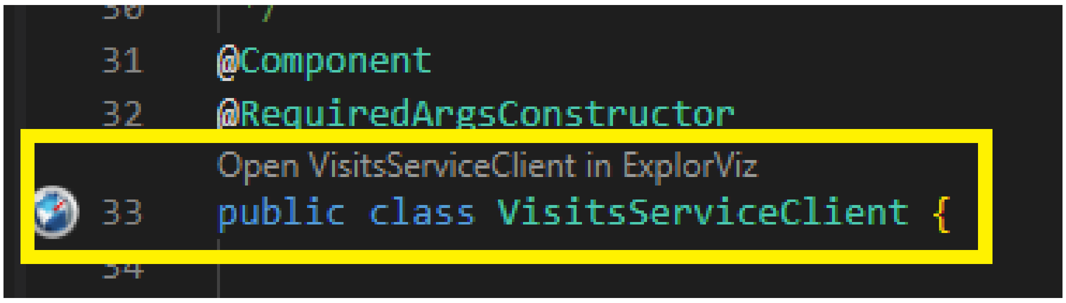
Listing 5.28. Extension Refresh ExplorViz Landscape Data

5.5.4 User Interactions with the ExplorViz Frontend

With the VS Code command *OpenAndFocusInExplorViz*, we defined the interaction side from the extension to the ExplorViz frontend. We will invoke that action with a codelense, a piece of actionable contextual information that modifies the current active Java file in VS Code dependent on the provided landscape data. We will also implement VS Code decorators to add icons and highlight certain code lines and snippets. Finally, we have to add the interaction *IDEApiActions.JumpToLocation* invoked from the frontend's *IDEApi* service to open the file from the class specified in the *IDEApiCall*.

Codelenses, Decorations and Analytics

The codelense *Open VisitsServiceClient in ExplorViz* as shown in Figure 5.3 is a feature provided by the VS Code API, which we also can use as an actionable button to invoke functions.



```
@Component
@RequiredArgsConstructor
Open VisitsServiceClient in ExplorViz
public class VisitsServiceClient {
```

Figure 5.3. Codelense Example

We implemented a new class *ExplorVizAPICodeLensProvider*, which implements the *vscode.CodeLensProvider* interface from the VS Code API to add codelenses representing the current ExplorViz landscape data and the currently active file in Visual Studio Code. We built a simple symbol solver to analyze the current file based on a regular expression that detects Java classes and methods currently available in the file. We calculate the FQN of such classes and methods with the package and respective names to compare the collected matches with the current landscape data sent by the ExplorViz frontend. If the FQNs match components from the ExplorViz frontend, we add a codelense to that specific line as shown in Listing 5.29. Furthermore, we add a callback function to each codelense that invokes the *OpenInExplorViz* command from Section 5.5.3 with the name and FQN for that match. We use the name to define the codelense description, e.g., *Open VisitsServiceClient in ExplorViz* where *VisitsServiceClient* is the corresponding class name.

```
const codeLens = new vscode.CodeLens(new vscode.Range(i, 0, i, 0), {
    title: "Open␣" + match.name + "␣in␣ExplorViz",
    command: "explorviz-vscode-extension.OpenInExplorViz",
    arguments: [match.name, match.fqn, this.vizData],
});
codeLenses.push(codeLens);
```

Listing 5.29. Extension Codelense Implementation Example

The *ExplorVizAPICodeLensProvider* adds a decorator to all code lines with matches also available in the connected ExplorViz frontend. Decoraters like the ExplorViz logo in

Figure 5.3 are also a feature from the VS Code API to modify the IDE with additional visual context.

We also create a *diagnosticCollection* to process the monitoring data provided by an analytics tool. The *diagnosticCollection* will be used to modify the existing terminal, which includes a *problems* tab that lists current errors and warnings from other VS Code extensions [13]. We add the given data from the *MonitoringData* to give textual feedback about the received analytics from ExplorViz.

Go to file location

For the action *IDEApiActions.JumpToLocation* invoked by the ExplorViz frontend *IDEApi* service, which will open the corresponding file if the *IDEApiCall.FQN* matches a Java file in the currently available working directories of the Visual Studio Code workspace, we implement the function *goToLocationsByMeshId*. The function scans the currently open workspace, which could have multiple directories, and creates the FQN based on the folder structure for each Java file inside the workspace. We also check the current Visual Studio Code workspace for multiple matches, e.g., when the same class with the same FQN is inside two distinct directories. Suppose multiple Java files were found in a package or the *IDEApiCall.FQN* matches multiple files in the workspaces, we prompt a selection to choose which file to open as in the illustrations Figure 4.7 and Figure 4.8 from the approach.

Lastly, we define the events for the *Socket.IO client*, which invoke the corresponding actions requested to or invoked from the frontend *IDEAPI* service.

5.5.5 Socket.IO Client Events

Setting up the events for the Socket.IO client is done with the same logic we used to define the events in the *IDEApi* service in Listing 5.30. We define one main event *IDEApiDest.IDEDo* to only act if the VS Code extension is requested to do computations or interactions. Then we define a switch case to determine which action is requested from the *IDEApi*. With the *UpdateMonitoringData* action, we receive the event when the user clicks the button that we defined as a mockup in the ExplorViz frontend and writes the data to a globally defined variable which is used to include the VS Code API analytics on the next *GetVizData* action. On the *JumpToLocation* action, we invoke the already implemented *goToLocationsByMeshId*. At last, on the *GetVizData* action, we update the current codelenses, decorations, and monitoring data to represent the newly received landscape data from the ExplorViz *IDEApi*.

```
socket.on(IDEApiDest.IDEDo, (data) => {
    switch (data.action) {
        case IDEApiActions.UpdateMonitoringData:
            monitoringData = data.monitoringData;
            break;

        case IDEApiActions.JumpToLocation:
            goToLocationsByMeshId(data.meshId, data.data, isMethod);
            break;

        case IDEApiActions.GetVizData:
            provider = new ExplorVizApiCodeLensProvider(...);
            codeLensDisposable.dispose();
            codeLensDisposable = vscode.languages.registerCodeLensProvider(
                "java",
                provider
            );
            context.subscriptions.push(codeLensDisposable);
            break;
    }
});
```

Listing 5.30. Extension Socket.IO Client Events

5.6 Concluding the Implementation

We have established the types we used across our applications and implemented a custom Ember.js service within the ExplorViz frontend. It prepares the ExplorViz landscape data for socket communication and passes them to the VS Code extension. We also added interactions that the frontend computes and hooks to invoke interactions in the VS Code extension. Furthermore, we implemented a Visual Studio Code extension that integrated the ExplorViz frontend into the IDE. We added interactions that invoke callbacks in the frontend to interact with landscape components and implemented interactions invokable from ExplorViz to open a file corresponding to a landscape component. We implemented a backend that only relays data and interaction requests to the extension and frontend, respectively. The final communication flow for our applications and ExplorViz frontend modifications are illustrated in Figure 5.4.

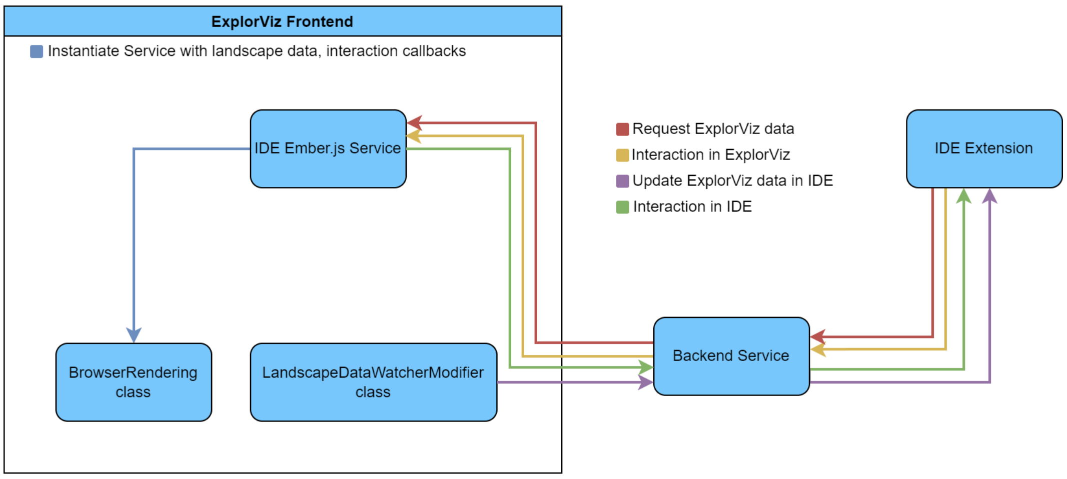


Figure 5.4. Communication diagram for ExplorViz Frontend and Extension: IDE Ember.js Service

Chapter 6

Evaluation

This chapter shows the application of our implementation using two exemplary software systems, i.e., the monolithic and distributed versions of the Spring PetClinic demonstrating the interactions and features we implemented. We use the ExplorViz frontend repository, which includes a mocked ExplorViz backend that visualizes different sample projects. The ExplorViz frontend, mocked ExplorViz backend, VS Code extension, and IDE backend service will run locally for this. We will also conclude some limitations we encountered while evaluating the sample applications.

6.1 Codebase for Sample Projects

We clone the repositories *spring-petclinic*[1] and *spring-petclinic-microservices*[2], which include the necessary codebase to interact with the ExplorViz sample applications *Increasing Landscape Sample* and *Study Landscape Sample*. We also copy each repository cloned this way and save them in a new Visual Studio Code workspace where the file explorer is illustrated in Figure 6.1.

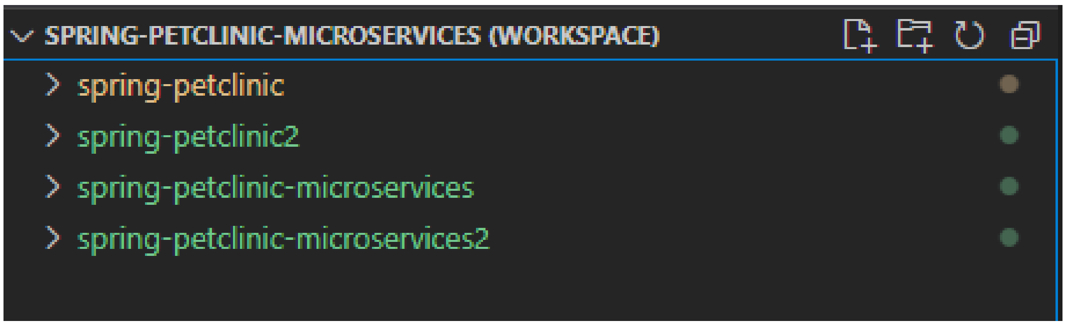

SPRING-PETCLINIC-MICROSERVICES (WORKSPACE)
spring-petclinic
spring-petclinic2
spring-petclinic-microservices
spring-petclinic-microservices2

Figure 6.1. Workspace Setup

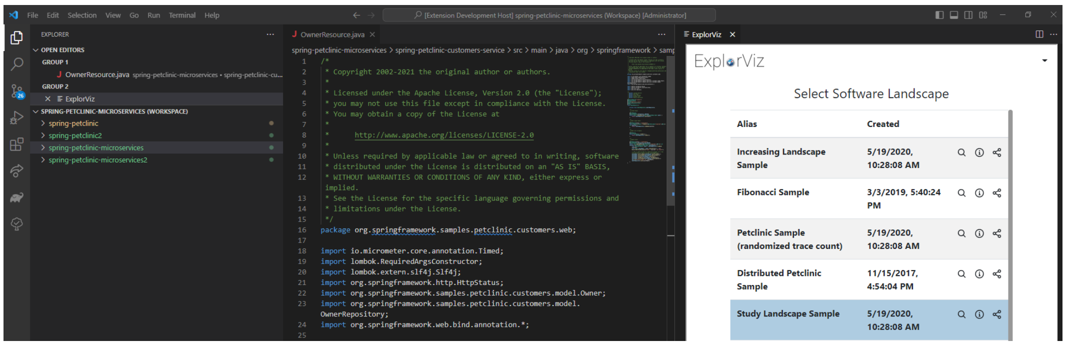
[1]https://github.com/spring-projects/spring-petclinic
[2]https://github.com/spring-petclinic/spring-petclinic-microservices

6.2 Setup ExplorViz Frontend, VS Code Extension and Backend

After setting up the codebase, we clone the ExplorViz frontend[3], VS Code extension[4], and backend[5]. According to the ExplorViz frontend documentation in the *README.md* we set up the mocked backend, which uses docker, and the frontend, which uses the Ember CLI and Node.js. The VS Code extension and the backend also need Node.js to install the necessary packages. After installing the packages in the respective directories for the VS Code extension and the backend with *npm install*, we start the backend with *npm start* and the extension by pressing *F5* within the extensions workspace, which starts the debug mode for the extension.

After starting the debug mode for the VS Code extension, a new instance of Visual Studio Code will open. We have to manually open the sample VS Code workspace that we saved in Section 6.1. The workspace will be open for future debug sessions but should be opened initially.

Suppose the ExplorViz frontend, mocked ExplorViz backend, IDE backend, and the extension debugging instance are running locally. In that case, we open any file in the VS Code instance with our sample codebase, press *CTRL+SHIFT+P* to show all available commands, and run the *ExplorViz webview* command to open the ExplorViz frontend in a right-hand tab. The final VS Code instance(*Sample Instance*) is illustrated in Figure 6.2, which we use as a baseline to showcase the following feature demonstrations in the software landscapes *Increasing Landscape Sample* and *Study Landscape Sample*.


File
Edit
Selection
View
Go
Run
Terminal
Help
[Extension Development Host] spring-petclinic-microservices (Workspace) [Administrator]
EXPLORER
OPEN EDITORS
GROUP 1
OwnerResource.java spring-petclinic-microservices • spring-petclinic-cu...
GROUP 2
ExplorViz
SPRING-PETCLINIC-MICROSERVICES (WORKSPACE)
spring-petclinic
spring-petclinic2
spring-petclinic-microservices
spring-petclinic-microservices2
OwnerResource.java
spring-petclinic-microservices > spring-petclinic-customers-service > src > main > java > org > springframework > samp
ExplorViz
ExplorViz
Select Software Landscape

```
/*
 * Copyright 2002-2021 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.springframework.samples.petclinic.customers.web;

import io.micrometer.core.annotation.Timed;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.samples.petclinic.customers.model.Owner;
import org.springframework.samples.petclinic.customers.model.
OwnerRepository;
import org.springframework.web.bind.annotation.*;
```

Alias	Created
Increasing Landscape Sample	5/19/2020, 10:28:08 AM
Fibonacci Sample	3/3/2019, 5:40:24 PM
Petclinic Sample (randomized trace count)	5/19/2020, 10:28:08 AM
Distributed Petclinic Sample	11/15/2017, 4:54:04 PM
Study Landscape Sample	5/19/2020, 10:28:08 AM

Figure 6.2. Extension Sample Workspace

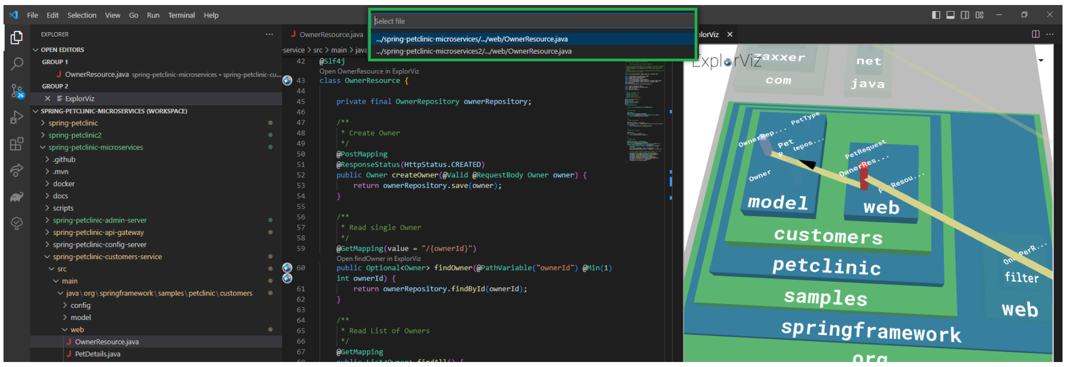
[3]https://github.com/ExplorViz/frontend
[4]https://github.com/ExplorViz/vs-code-extension
[5]https://github.com/ExplorViz/vs-code-backend

6.3 Example: Study Landscape Sample

In the *Sample Instance*, we open the *Study Landscape Sample*, which shows the visualization where we will demonstrate some of the interactions we implemented in the *IDEApi* service and the VS Code extension.

Codelenses with Package, Import, Class and Method Interactions

We start with the possible *IDEApiActions.JumpToLocation* interactions to open Java class files while interacting with the ExplorViz frontend. We first open all components in the frontend from the *Sample Instance* by right-clicking the free space around the foundations and selecting *Open All Components*. Our first interaction is made by a single left-click on the *OwnerResource* class located in the *petclinic-costumer-service* foundation (red) as shown in Figure 6.3.


File
Edit
Selection
View
Go
Run
Terminal
Help
EXPLORER
OPEN EDITORS
GROUP 1
OwnerResource.java spring-petclinic-microservices • spring-petclinic-cu...
GROUP 2
ExplorViz
SPRING-PETCLINIC-MICROSERVICES (WORKSPACE)
spring-petclinic
spring-petclinic2
spring-petclinic-microservices
.github
.mvn
docker
docs
scripts
spring-petclinic-admin-server
spring-petclinic-api-gateway
spring-petclinic-config-server
spring-petclinic-customers-service
src
main
java \ org \ springframework \ samples \ petclinic \ customers
config
model
web
OwnerResource.java
PetDetails.java
OwnerResource.java
service > src > main > java
Select file
.../spring-petclinic-microservices/.../web/OwnerResource.java
.../spring-petclinic-microservices2/.../web/OwnerResource.java
ExplorViz
zaxxer
com
net
java
model
web
customers
petclinic
samples
springframework
org
filter
web

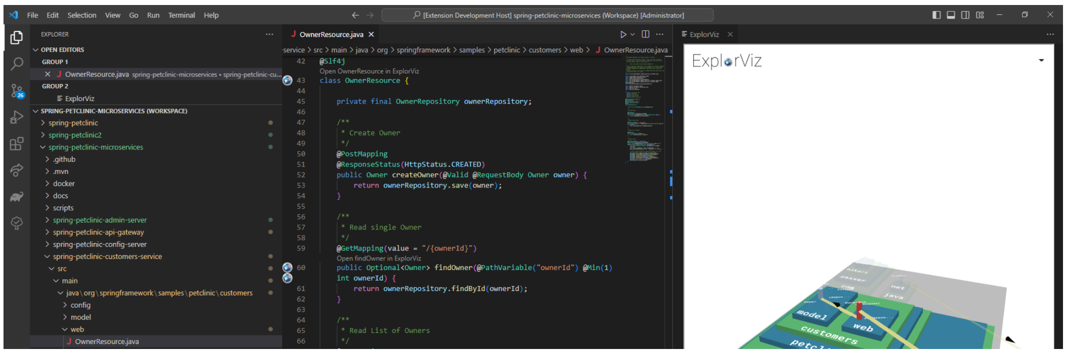
```
@Slf4j
Open OwnerResource in ExplorViz
class OwnerResource {

    private final OwnerRepository ownerRepository;

    /**
     * Create Owner
     */
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Owner createOwner(@Valid @RequestBody Owner owner) {
        return ownerRepository.save(owner);
    }

    /**
     * Read single Owner
     */
    @GetMapping(value = "/{ownerId}")
    Open findOwner in ExplorViz
    public Optional<Owner> findOwner(@PathVariable("ownerId") @Min(1)
    int ownerId) {
        return ownerRepository.findById(ownerId);
    }

    /**
     * Read List of Owners
     */
    @GetMapping
```

Figure 6.3. Interaction from ExplorViz to Extension with Prompt: Class

We are prompted to select a Java file we want to open specific to the respective workspace (green) because we created a duplicate of the *spring-petclinic-microservices* directory to demonstrate the possibility if we have the same FQN in distinct workspaces. We select the non-duplicate workspace, and the *OwnerResource.java* file will open in the left-hand tab.

In the newly opened file, we see codelenses created by the VS Code extension, which indicate what classes, packages, and methods are present and can be interacted with in the ExplorViz frontend. Illustrated in Figure 6.4, we have codelenses and a line icon for the *OwnerResource* class and the *findOwner* method.

That concludes one interaction cycle of a possible ExplorViz frontend to VS Code extension interaction. Respectively, we use the codelenses from Figure 6.4 to interact with the frontend. Before we click on a codelense, we reset the current foundation by double-clicking on the *petclinic-costumer-service* foundation, which closes all opened components to demonstrate that a component can be hidden and still interacted with. Now we click on



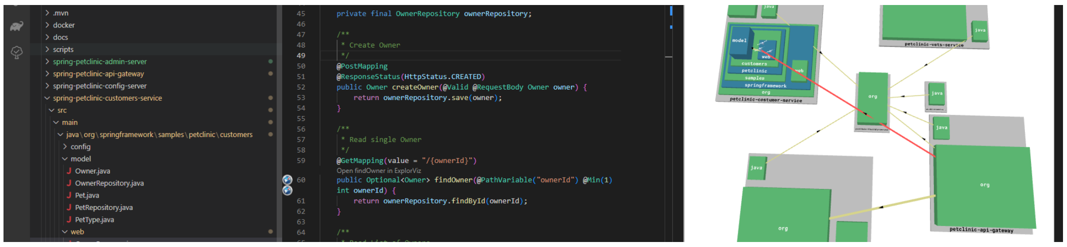
```
File  Edit  Selection  View  Go  Run  Terminal  Help
[Extension Development Host] spring-petclinic-microservices (Workspace) [Administrator]

EXPLORER
OPEN EDITORS
  GROUP 1
    OwnerResource.java spring-petclinic-microservices • spring-petclinic-cu...
  GROUP 2
    ExplorViz
SPRING-PETCLINIC-MICROSERVICES (WORKSPACE)
  > spring-petclinic
  > spring-petclinic2
  v spring-petclinic-microservices
    > .github
    > .mvn
    > docker
    > docs
    > scripts
    > spring-petclinic-admin-server
    > spring-petclinic-api-gateway
    > spring-petclinic-config-server
    v spring-petclinic-customers-service
      v src
        v main
          v java\org\springframework\samples\petclinic\customers
            > config
            > model
            v web
              OwnerResource.java
```

```java
service > src > main > java > org > springframework > samples > petclinic > customers > web > OwnerResource.java
@Slf4j
Open OwnerResource in ExplorViz
class OwnerResource {

    private final OwnerRepository ownerRepository;

    /**
     * Create Owner
     */
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Owner createOwner(@Valid @RequestBody Owner owner) {
        return ownerRepository.save(owner);
    }

    /**
     * Read single Owner
     */
    @GetMapping(value = "/{ownerId}")
    Open findOwner in ExplorViz
    public Optional<Owner> findOwner(@PathVariable("ownerId") @Min(1)
    int ownerId) {
        return ownerRepository.findById(ownerId);
    }

    /**
     * Read List of Owners
     */
```

Figure 6.4. OwnerResource.java Codelenses

the *Open OwnerResource in ExplorViz* codelense above the *OwnerResource* class definition in our *Sample Instance*. This opens the *OwnerResource* and parent components within the visualization, and the camera moves to and focuses the component. Furthermore, can we click on the codelense *Open findOwner in ExplorViz* above the *findOwner* definition in the *Sample Instance* to switch the camera focus to the communication link from Figure 6.5 (red). This communication link is a *findOwner* method call from the foundation *petclinic-api-gatway's* nested *CustomersServiceClient* component which invokes *OwnerResource.findOwner* from the *petclinic-costumer-service's OwnerResource* component.

In the *OwnerResource.java* file from the *Sample Instance*, we also have a codelense for the package and some for the imports, which indicate that those components are visualized in the currently connected ExplorViz frontend and are interactable. Clicking those codelenses will open and focus the respective component. Single-clicking on the components within the *Sample Instance's* visualization will open the respective file in Visual Studio Code.



```
> .mvn
> docker
> docs
> scripts
> spring-petclinic-admin-server
> spring-petclinic-api-gateway
> spring-petclinic-config-server
v spring-petclinic-customers-service
  v src
    v main
      v java\org\springframework\samples\petclinic\customers
        > config
        v model
          Owner.java
          OwnerRepository.java
          Pet.java
          PetRepository.java
          PetType.java
        v web
```

```java
    private final OwnerRepository ownerRepository;

    /**
     * Create Owner
     */
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Owner createOwner(@Valid @RequestBody Owner owner) {
        return ownerRepository.save(owner);
    }

    /**
     * Read single Owner
     */
    @GetMapping(value = "/{ownerId}")
    Open findOwner in ExplorViz
    public Optional<Owner> findOwner(@PathVariable("ownerId") @Min(1)
    int ownerId) {
        return ownerRepository.findById(ownerId);
    }

    /**
```

Figure 6.5. OwnerResource.java Method Codelense Interaction

Monitoring Data and Analytics

To send a monitoring data mockup to the extension using the VS Code API analytics, we have to open the Extension Settings within the ExplorViz frontend of our *Sample Instance*. Right-clicking the free space within the visualization opens the context menu we already used to open all components. We select the *Open Sidebar* option, which opens a menu where we select the *Extension Settings*. In the *Extension Settings*, we press the *Monitoring Tool* button as illustrated in Figure 6.6.

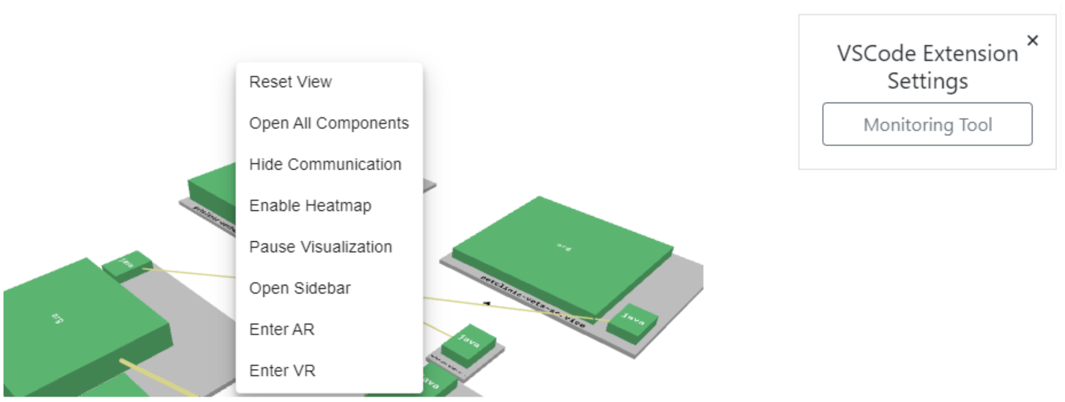
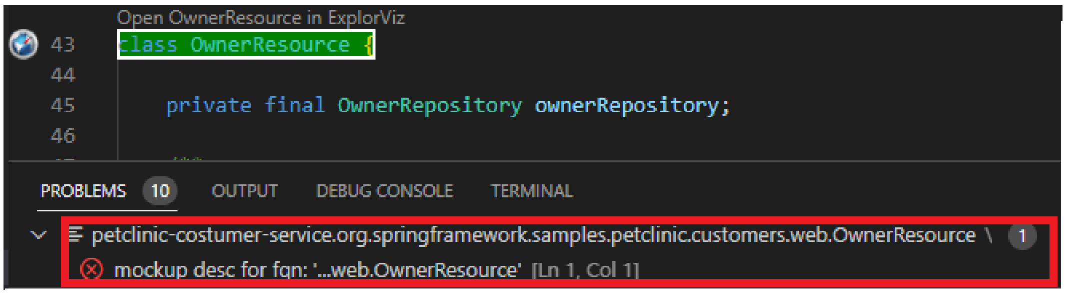


Figure 6.6. Extension Settings for Monitoring Data Mockup

In the next update cycle of the ExplorViz frontend, the extension adds the analytics for the monitoring data, which highlights the class *OwnerResource* (green) with a decorater and adds an entry to the problems tab (red) with the FQN and description for that monitoring dataset.



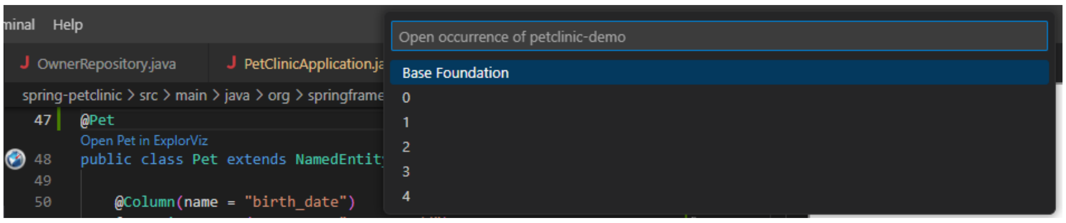
```
Open OwnerResource in ExplorViz
class OwnerResource {

    private final OwnerRepository ownerRepository;

```

```
PROBLEMS 10    OUTPUT    DEBUG CONSOLE    TERMINAL
petclinic-costumer-service.org.springframework.samples.petclinic.customers.web.OwnerResource \ 1
  mockup desc for fqn: '...web.OwnerResource' [Ln 1, Col 1]
```

Figure 6.7. Extension Analytics with Monitoring Data Mockup

6.4 Example: Increasing Landscape Sample

We switch the current software landscape in the *Sample Instance* to the *Increasing Landscape Sample* to demonstrate how we can interact with multiple occurrences of an application. We open the *Pet.java* file inside the *spring-petclinic* codebase by opening all components in the *Sample Instance* visualization and selecting the *Pet* component in the *owner* package. The codelenses update according to the new ExplorViz landscape, and we click the *Open Pet in ExplorViz* codelense above the *Pet* class definition, which prompts the selection shown in Figure 6.8. Selecting an item will open and focus that occurrence of the *Pet* class in the ExplorViz visualization.


minal Help
OwnerRepository.java
PetClinicApplication.ja
spring-petclinic > src > main > java > org > springframe

```
@Pet
Open Pet in ExplorViz
public class Pet extends NamedEntit

    @Column(name = "birth_date")
```

Open occurrence of petclinic-demo
Base Foundation
0
1
2
3
4

Figure 6.8. Increasing Landscape Sample: Occurences selection

6.5 Conclusion

We successfully set up an ExplorViz frontend, ExplorViz backend, the Visual Studio Code extension, and the IDE backend locally to test interaction variants, visual IDE additions, and analytics for the petclinic sample projects. We found some limitations worth mentioning during the evaluation and further testing of the different implementations.

6.5.1 Limitations

- We sparsely tested the different applications on a Linux operating system and mostly did all the testing on Windows.
- The Express.js backend server has a *maxHttpBufferSize* variable, which defines the maximum possible data sent with Socket.IO. The maximum is currently set to 100MB, which was never surpassed during testing, but it could be possible for bigger projects to reach the maximum.
- For applications that grow during runtime like the *Increasing Landscape Sample*, we have found that the occurrences of methods if interacted with the respective codelense are not correct because the communication links are only in the first occurrence. Furthermore,

we did not test occurrences with a software landscape that contains multiple applications, therefore, multiple foundations, such as the distributed version of the Spring PetClinic.

- ▻ The ExplorViz frontend sometimes combines communication links into a group visualized as one communication link instead of multiple per method. In that case, only the first communication link of the group is sent to the VS Code Extension for interactions and visual additions to the IDE.
- ▻ We did not test the ExplorViz AR and VR capabilities with the implementations made to the ExplorViz frontend because of technical boundaries.

Chapter 7

Related Work

Combining software visualization and software development is the main subject of this thesis, with a focus on integrating live trace visualization into software development. We will focus on related work that combines the two aspects differently by integrating development tools inside a visualization tool.

7.1 SEE

SEE is a multi-user multi-purpose software visualization tool based on the city metaphor that allows users to immerse into a virtual room. It also traces the software's evolution and analyzes the runtime behavior [10]. SEE uses a code-viewer to show the source code of an implementation entity, as illustrated in Figure 7.1 [9]. The visualization is based on the *Unity* game engine used to develop a shared three-dimensional world where users can view, move, and arrange source code files interactively [10].

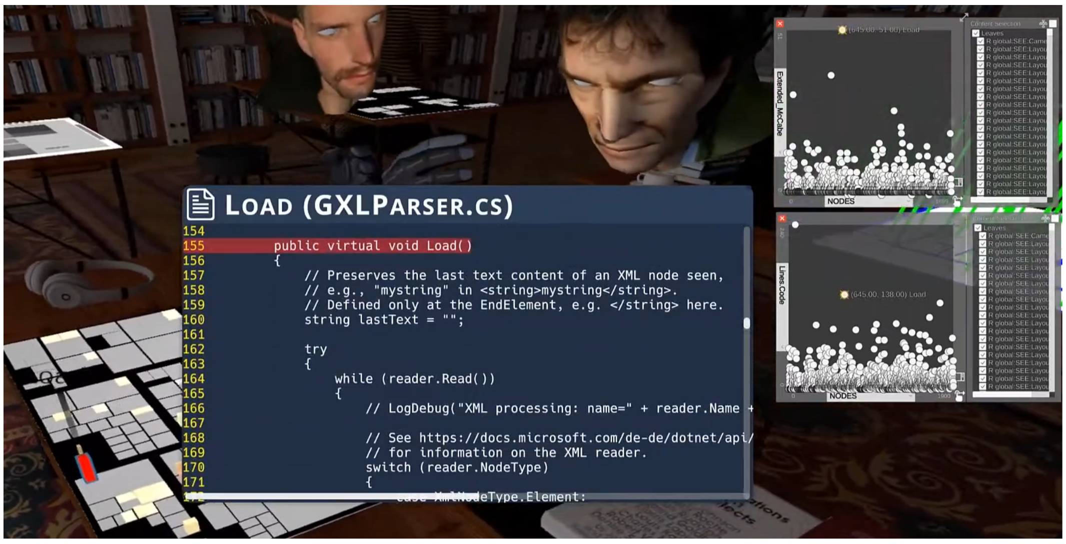
Integrating ExplorViz into a VS Code extension preserves the IDE as a development tool instead of a code viewer. Compared to SEE, we are not fully immersed in a virtual room to navigate in, but we can still use the ExplorViz runtime behavior as a city landscape.

7.2 Code Park

Code Park is a novel tool that visualizes codebases to improve a programmer's understanding of an existing codebase. The visualization is made in a 3D game-like environment accessible in a top-down view Figure 7.2, which shows the entire codebase, and an ego-centric view Figure 7.3, which allows examining the concrete implementation. The top-down view shows each class as a room the user can enter to examine the respective variables, member functions, etc., on the inside walls [8].

In contrast to ExplorViz, Code Park visualizes the codebase for the developer to examine and uses a different visualization approach and no runtime behavior. We still maintain the IDE as a fully-fledged development tool while interacting with the ExplorViz city landscape.

[1]https://youtu.be/V2WpRtKF5wM
[2]https://youtu.be/LUiy1M9hUKU
[3]https://youtu.be/LUiy1M9hUKU



LOAD (GXLPARSER.CS)

```
        public virtual void Load()
        {
            // Preserves the last text content of an XML node seen,
            // e.g., "mystring" in <string>mystring</string>.
            // Defined only at the EndElement, e.g. </string> here.
            string lastText = "";

            try
            {
                while (reader.Read())
                {
                    // LogDebug("XML processing: name=" + reader.Name +

                    // See https://docs.microsoft.com/de-de/dotnet/api/
                    // for information on the XML reader.
                    switch (reader.NodeType)
                    {
                        case XmlNodeType.Element:
```

Figure 7.1. Source code viewer in SEE[1]

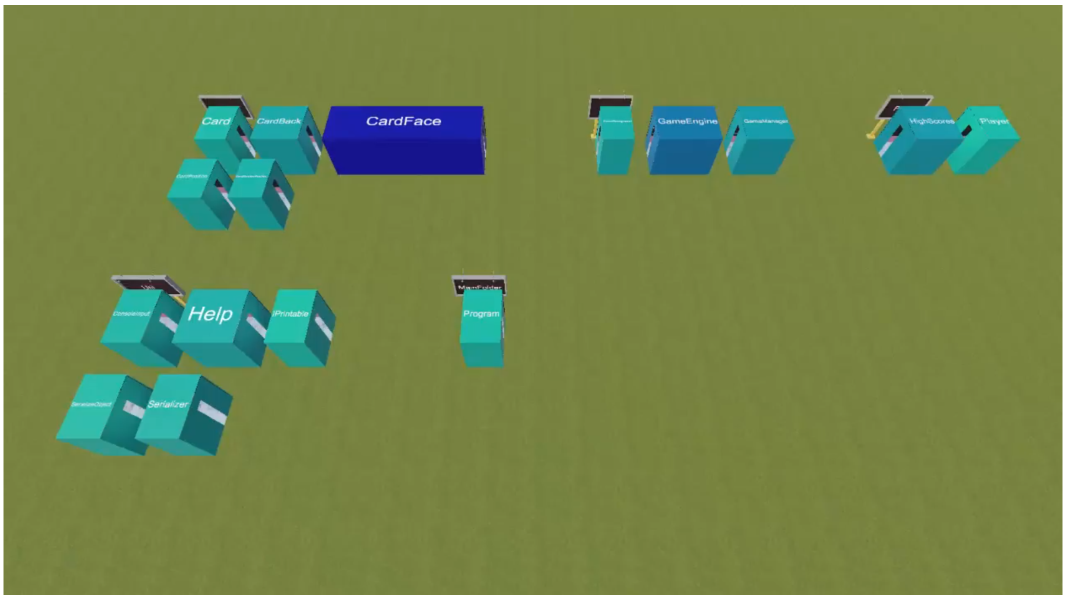


Figure 7.2. Code Park - Park View[2]

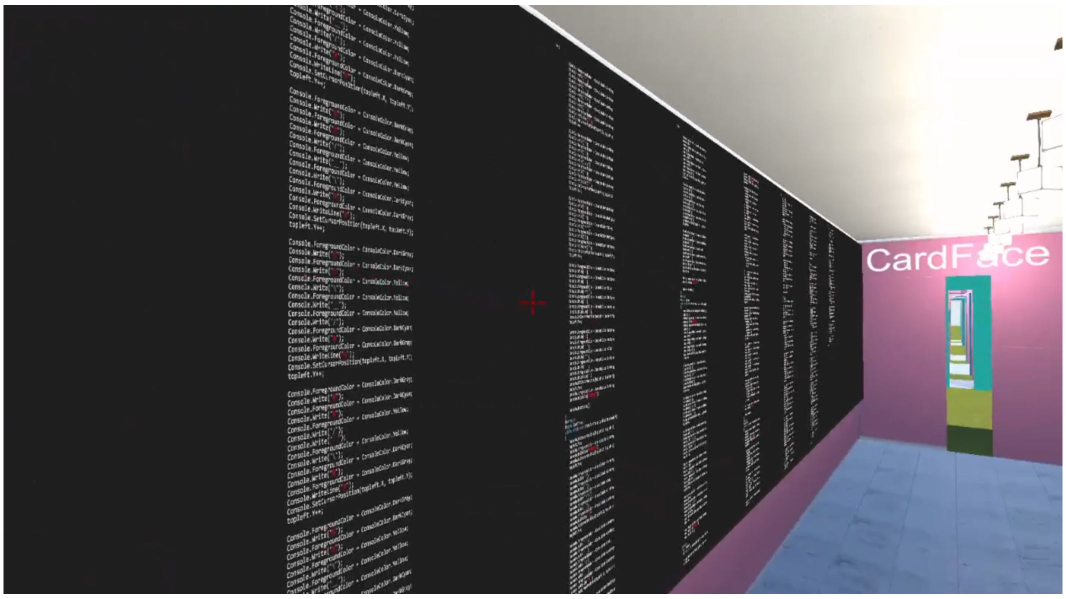
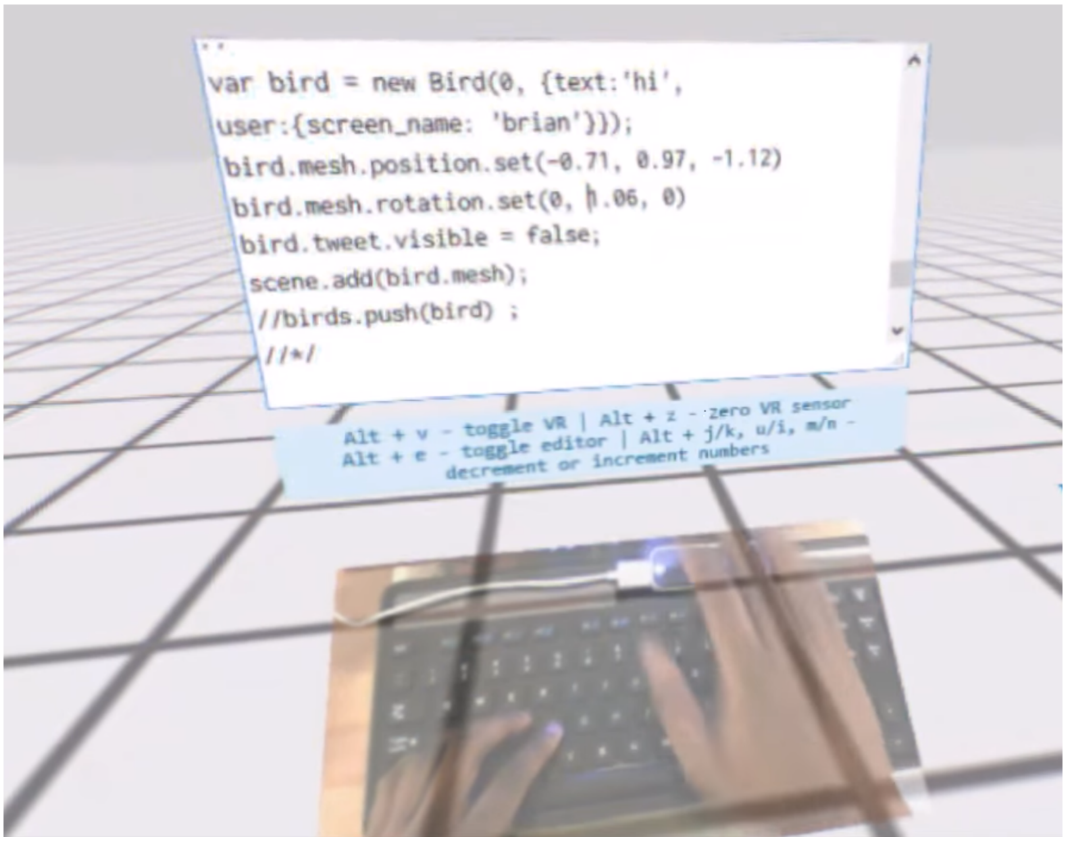


Figure 7.3. Code Park - Room View[3]

7.3 RiftSketch

RiftSketch is a web-based live coding environment for Virtual Reality that allows users to describe a 3D scene using the Three.js library. Complementary to hand-gesture features to modify parts of the code, it also assists the interaction with a keyboard by using a mounted web camera on the VR headset to show the keyboard in Virtual RealityFigure 7.4 [1].

RiftSketch, compared to ExplorViz, allows coding in a visualization tool limited to a Three.js scene which computes changes at runtime. RiftSketch allows implementation changes in the visualization and is not only a code viewer.



```
var bird = new Bird(0, {text:'hi',
user:{screen_name: 'brian'}});
bird.mesh.position.set(-0.71, 0.97, -1.12)
bird.mesh.rotation.set(0, 1.06, 0)
bird.tweet.visible = false;
scene.add(bird.mesh);
//birds.push(bird) ;
//*/
```

Alt + v - toggle VR | Alt + z - zero VR sensor
Alt + e - toggle editor | Alt + j/k, u/i, m/n -
decrement or increment numbers

Figure 7.4. RiftSketch Example: https://youtu.be/SKPYx4CEIlM

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis goal was to integrate ExplorViz a live trace visualization into software development to enhance the versatility of ExplorViz as a software development tool and preserve the IDE as toolkit for development. We successfully implemented a Visual Studio Code extension capable of integrating the ExplorViz frontend, performing user interactions on a connected frontend, and providing visual feedback with codelenses and decorations representing the current packages, classes, and methods of the runtime behavior. We extended the ExplorViz frontend implementation with a new Ember.js service that prepares the current ExplorViz landscape data and user interactions to be accessible via a backend service that relays the data to the extension and ExplorViz frontend, respectively. We evaluated our implementation with sample interactions to demonstrate that it is possible to open a landscape component with an IDE user interaction, add a visual representation of the current runtime behavior and interact with a landscape component to open the respective source code implementation.

Lastly, we will mention possible future work applicable for our different implementations.

8.2 Future Work

- Integrate ExplorViz into another more applicable IDE for Java development, for example, IntelliJ IDEA. The implemented backend service and ExplorViz frontend modifications can be reused. The new integration has to implement the IDE interactions and visual source code feedback.
- Additional interaction can be added to the extension and Ember.js service depending on the developer's needs and findings during further usage.
- The backend service can handle multiple frontend and extension clients but broadcasts one interaction to all receiving destinations. Modifications to the backend service that map socket clients to specific connectable sessions are necessary for a granular collaboration.

- ▻ The symbol solver in the extension currently looks for packages, imports, class definitions, and basic method definitions and can be extended to include class instantiations and method calls. We currently use a regular expression combined with cases to filter specific patterns from the active document, which can be modified or implemented differently.

Bibliography

[1] Anthony Elliott, Brian Peiris, and Chris Parnin. “Virtual reality in software engineering: affordances, applications, and challenges”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Volume 2. 2015, pages 547–550. DOI: 10.1109/ICSE.2015.191 (cited on page 53).

[2] *Ember js*. URL: https://emberjs.com/ (visited on 11/24/2022) (cited on pages 8, 11, 23, 26).

[3] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. “Hierarchical software landscape visualization for system comprehension: a controlled experiment”. In: *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. 2015, pages 36–45. DOI: 10.1109/VISSOFT.2015.7332413 (cited on page 5).

[4] Florian Fittkau et al. “Live trace visualization for comprehending large software landscapes: the explorviz approach”. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013, pages 1–4. DOI: 10.1109/VISSOFT.2013.6650536 (cited on page 5).

[5] Evan M. Hahn. *Express in action*. Manning, 2016 (cited on page 7).

[6] Wilhelm Hasselbring, Alexander Krause, and Christian Zirkelbach. “Explorviz: research on software visualization, comprehension and collaboration”. In: *Software Impacts* 6 (2020), page 100034 (cited on page 5).

[7] Wilhelm Hasselbring, Alexander Krause-Glau, and Marcel Bader. “Collaborative software visualization for program comprehension”. In: (2022), pages 1–12 (cited on page 5).

[8] Pooya Khaloo et al. “Code park: a new 3d code visualization tool”. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 2017, pages 43–53. DOI: 10.1109/VISSOFT.2017.10 (cited on page 51).

[9] Rainer Koschke and Marcel Steinbeck. “Modeling, visualizing, and checking software architectures collaboratively in shared virtual worlds”. In: *European Conference on Software Architecture*. 2021 (cited on page 51).

[10] Rainer Koschke and Marcel Steinbeck. “See your clones with your teammates”. In: *2021 IEEE 15th International Workshop on Software Clones (IWSC)*. 2021, pages 15–21. DOI: 10.1109/IWSC53727.2021.00009 (cited on page 51).

[11] Alexander Krause, Malte Hansen, and Wilhelm Hasselbring. “Live visualization of dynamic software cities with heat map overlays”. In: *2021 Working Conference on Software Visualization (VISSOFT)*. 2021, pages 125–129. DOI: 10.1109/VISSOFT52517.2021.00024 (cited on page 5).

[12] Alexander Krause-Glau and Wilhelm Hasselbring. “Scalable collaborative software visualization as a service: short industry and experience paper”. In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pages 182–187. DOI: 10.1109/IC2E55432.2022.00026 (cited on page 5).

[13] *Vs code api*. URL: https://code.visualstudio.com/api (visited on 03/26/2023) (cited on pages 3, 7, 14, 15, 36, 40).

[14] *What is ember?* URL: https://guides.emberjs.com/release/getting-started/ (visited on 11/24/2022) (cited on page 8).

[15] *What socket.io is*. URL: https://Socket.IO/docs/v4/ (visited on 03/26/2023) (cited on pages 8, 35).

[16] Xin Xia et al. “Measuring program comprehension: a large-scale field study with professionals”. In: *IEEE Transactions on Software Engineering* 44.10 (2018), pages 951–976 (cited on page 1).