MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Bug-reporting Component for the Web

Master's Thesis

BC. OLIVER SVETLÍK

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Bug-reporting Component for the Web

Master's Thesis

BC. OLIVER SVETLÍK

Advisor: RNDr. Vít Rusňák, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2023

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Oliver Svetlík

Advisor: RNDr. Vít Rusňák, Ph.D.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor, RNDr. Vít Rusňák, Ph.D., for his patient supervision, advice, and rich, invaluable feedback, which he provided on a regular basis.

I would also like to thank all respondents who participated in the user and developer testing of the implemented component for their valuable time and feedback.

Huge gratitude belongs also to my family, friends, colleagues and everyone else who supported and motivated me while I worked on this master's thesis and throughout my whole master's studies.

Abstract

The thesis maps existing solutions for user-friendly bug-reporting approaches through add-on web components that allow appending annotated images and screenshots to textual reports. Based on analyzing existing features and gaps, we implemented a Bug Reporting Tool (*BRT*) – a web component in React allowing users to comment, annotate and report issues on a webpage. The annotated screenshot, text comments, and related environmental data, e.g., the browser version, are then filed as issues in the common GitLab repository.

Keywords

bug reporting, issue tracking, visual annotations, annotation tool, React, GitLab

Contents

1 Introduction **1**

2 Survey of Existing Solutions **3**
2.1 Annotation Options 3
2.2 Gathered Data 5
2.3 Frontend Integration Methods 6
2.4 Integration with Issue Tracking Systems 8
2.5 Prices Comparison 9

3 Bug Reporting Tool Design **11**
3.1 Requirements 11
3.2 Architecture and Workflow 14

4 Implementation **16**
4.1 Used Technologies 16
4.2 *BRT* Decomposition 18
4.3 Integration 28
4.4 Challanges, Problems, and Optimizations 29

5 Installation **33**
5.1 GitLab Integration 33
5.2 GitHub Integration 35

6 Testing and Feedback **38**
6.1 User Testing 38
6.2 Developer Testing 47

7 Conclusion and Future Work **49**

Bibliography **50**

A Supplementary Materials **54**

1 Introduction

The bug reporting from users is a significant source of feedback, as developers might not think of all edge cases in the code-testing process. In the end, users are always the final consumers of any product. If consumers are unsatisfied, they will not use the product and may look for competing solutions.

Bug reporting tools with visual feedback options are designed for users and developers or other participators in developing the given application. Developers or employees from the quality assurance department can find these tools helpful when they create issues to track some bugs or enhancements of the application. On the other hand, user feedback is essential, as most applications use user-centric design.

The bug reporting process is not only about submitting a form with a user's comment. It also involves tracking these reports, mainly in the form of issues. Many paid tools, such as Jira [1], HubSpot [2], or Backlog [3], already exist for issue tracking. Nevertheless, also open-source alternatives such as BugZilla are available [4].

On the other hand, there are also tools for bug reporting, such as Marker.io [5], Bugherd [6], Ybug.io [7], or Usersnap [8]. However, all these tools are paid, and unlike issue-tracking platforms, there is no comparable open-source alternative in terms of capabilities. The lack of open-source alternatives is also one of the primary motivations for this thesis.

The thesis aims to explore the existing solutions for bug-reporting components and propose a feature-rich open-source implementation of a web component that can be connected to the existing issue-tracking subsystem of the GitLab platform. The component is named Bug Reporting Tool (*BRT*). It is developed in React and available as an NPM package for React web applications. However, this tool can be installed on any web application or static web page using an HTML `<script>` tag. Using *BRT*, it is possible to report bugs, as well as some enhancements to the web page. On top of the textual report, *BRT* allows the creation of multiple screenshots with annotations using five available annotation tools (area selection, area obfuscation, arrow, freehand, and text comment). Environment information such as a

version of the browser, or operating system, the size of the screen, and many more are also attached to the textual report.

The thesis is structured as follows. An overview of available bug-reporting solutions and an analysis of their features are described in Chapter 2. Chapter 3 describes the design of implemented bug reporting component and its architecture. Chapter 4 concerns the implementation details of *BRT*, including used technologies, problems encountered during the implementation, or its optimizations. Chapter 5 provides *BRT* installation instructions. Chapter 6 reports on the user testing and developers' feedback on *BRT*. Finally, Chapter 7 concludes the thesis, and discuss possible future work.

2 Survey of Existing Solutions

Many commercial paid bug-reporting tools already exist [9]. There are a few free or open-source ones, but none is as refined as their paid alternatives. In the same manner, also the range of provided functions varies. Some tools allow only textual feedback or also attachments of screenshots or other files. In our work, we focus on feature-rich tools that also allow annotating attached screenshots.

We surveyed available commercial and open-source tools. Their overview and comparison of their properties are described in Attachment A. Overall, we selected sixteen tools, all commercial, except for one. The following sections provide a brief overview of the main parameters based on the five categories used in the survey:

- annotation options,
- gathered data,
- frontend integration methods,
- integration with issue tracking systems,
- price.

2.1 Annotation Options

Extending the plain text input with annotated screenshot allow for a significantly higher delivery of information about the reported bug. Recent bug reporting tools tend to implement at least the support for attaching screenshots, but some integrate simple annotation editors directly. Users can then highlight the selected regions of interest and add various drawings or even emoticons.

There are two main approaches to highlighting an area with a bug: *HTML element selection* and *Screenshotting*. They usually use some overlay with dimmed colour over the whole screen.

HTML Element Selection tracks the exact position of the user's cursor in real time. From the cursor's position, it is possible to deduce over which HTML element the cursor is currently moving and highlight this HTML Element. When the cursor moves over another element, the highlighted element changes. After the user clicks, the

current element under the cursor is selected and highlighted permanently. Tools using HTML element selection usually do not support other annotations methods. One of the reasons is that it is descriptive enough, and there is no need for any additional annotations. The disadvantage of HTML element selection is that multiple HTML elements might be involved. For example, a user must take multiple actions to reproduce a bug, such as clicking on different buttons in a specific order and so on. In such cases, a user would have to choose at least the first parent element of all elements involved with the HTML element selection method. If these elements were spread around the whole page, even the HTML body might be needed to be chosen to wrap all involved elements. The wider the selected area is, the more text comments are necessary to describe the bug and the steps for its reproduction.

Screenshotting is principally based on taking a static screenshot or a video clip of the current view attached to the textual description of the bug report. It is usually implemented as a rectangle overlay with transparent background and bold border that delimits the selected region of interest. It is also possible to select multiple areas and use various annotation tools, such as arrows and in-annotation-tool comments, to visually describe steps of bug reproduction much better than with the HTML selection method, without the support of other annotations. Unlike the previous approach, there is no bonding to the specific HTML element. Therefore, it might be harder to find the source of the bug for the programmer. On the other hand, screenshot annotation is easier to use, even for less skilled users.

Further, we list the most commonly used visual annotation tools and their typical use:

Area selection allows marking a specific area or place in the screenshot, typically using a rectangular or circular drawing.

Arrow allow you to point out directly a specific area or place in the screenshot.

Straight line drawing can serve, for example, to underline some text.

Freehand drawing allows to create an arbitrary shape or mark some region.

Text comment inside of an annotated screenshot (not as the issue description) has two possible versions. It is either a text block where the comment is directly written on the canvas or an interactive pin with the text appended through a pop-up. The comment is not visible directly on the canvas but is attached to the issue description with reference to the pin ID.

Area obfuscation serves to hide sensitive information on the page or to hide disruptive surroundings. Most existing tools have it implemented as a drawn black rectangle, but for instance, Marker.io [5] implemented it with a rectangle that blurs the background behind it.

Emoticons as implemented in Marker.io [5] allow users to add emoticons directly to the canvas as in the Text block form of text comment annotation. In some other tools, emoticons can be appended only to the text comment.

Colour palette is used to change the annotation colours in several tools.

Table 2.1 presets the occurrence of individual annotation types in surveyed tools.

Table 2.1: Annotation types with the total amount of occurrences and all tools utilizing them

Annotation Type	Occurrence	Tools
HTML Element selection	5	[6], [10], [11], [12], [13]
Drawings	6	[5], [7], [8], [14], [15], [16]
Obfuscation	5	[5], [7], [8], [11], [15]
Text comment	6	[5], [7], [8], [14], [12], [15]
Emoticons	3	[5], [10], [8]

2.2 Gathered Data

Some more sophisticated bug reporting tools gather environment data and attach them to the issues. Such data help in increasing the reproducibility of the bug. The tools usually collect some of the following:

device type (mobile, desktop, etc..), timestamp, location (geographical location – country or the city), IP address, operating system and its version, web browser and its version, screen size, viewport size, pixel ratio and console output history.

Data such as location, IP address, or pixel ratio might help find the root cause of some very specific bugs, but they are probably unused in most cases. On the other hand, the type of device can be deduced from the operating system. The difference between viewport size and screen size is that viewport size represents the browser window size, and screen size means the whole screen size.

In Table 2.2, there are summarised occurrences of individual data types in surveyed tools.

Table 2.2: Data the tools are gathering with the total amount of occurrences and all tools gathering them

Data	Occurrence	Tools
Type of device	6	[5], [10], [17], [12], [13], [16]
Time	12	[7], [17], [18], [19], [8], [20], [21], [14], [12], [13], [15], [16]
Location	7	[7], [10], [8], [20], [21], [15], [16]
IP Address	2	[20], [21]
Operating System	11	[5], [7], [10], [21], [8], [20], [21], [12], [13], [15], [16]
Browser	11	[5], [7], [10], [17], [8], [16], [21], [12], [13], [15], [16]
Screen size	8	[5], [7], [8], [20], [14], [13], [15], [16]
Viewport size	7	[5], [7], [10], [8], [12], [13], [15]
Pixel ratio	3	[5], [7], [15]
Console history	3	[5], [7], [15]

2.3 Frontend Integration Methods

The tools usually provide two options for integrating them into web applications. One of them is an in-code modification. The second

option is a browser extension that the users must install beforehand. Some tools even combine both options.

The in-code integration is the most used option (14 tools out of 16). Yet, there are several ways of doing it. One possibility is to install a bug-reporting component by adding an HTML `<script>` tag with a JavaScript snippet directly into the HTML file. The advantages of this approach are that it is not dependent on any specific frontend framework, and it is pretty easy to install (basically, only copy-paste a code snippet). This integration option is supported by the majority of surveyed tools, including Marker.io [5], Hotjar [10], or Userback [15]. The following example demonstrates this installation option for the Userback [15] tool:

```
<script>
    window.Userback = window.Userback || {};
    Userback.access_token = '<unique_access_token>';
    (function(d) {
        var s = d.createElement('script');s.async = true;
        s.src = 'https://static.userback.io/widget/v1.js';
        (d.head || d.body).appendChild(s);
    })(document);
</script>
```

Another alternative is the public library for specific frontend frameworks, like React or Vue.js. Such libraries are published and can be downloaded from some package managers, mostly NPM. It is usually possible to install this library using a single command from the terminal. After installing the library, the developer must add this component to his application. However, the problem scales much better using this approach. For instance, in React, the component can be used as a wrapper around the whole application; hence, it will be available on every page of the application. It is supported, for example, in the most sophisticated tool from the mapped ones – Userback [15]. In-code integration is supported by fourteen of a total of sixteen mapped tools. A Userback [15] example for React follows.

```
import { UserbackProvider } from '@userback/react';
<UserbackProvider token="<unique_access_token>">
  <App />
</UserbackProvider>
```

Tools, such as Marker.io [5], UserSnap [8], Userback [15], and some others provide their bug-reporting tools as an extension to some widely used web browsers, such as Mozilla Firefox or Google Chrome (see Figure 2.1). It requires a registered account on a specific platform, where all the created issues are tracked. This is a great alternative for developers, or quality assurance employees, which want to report bugs in their project and do not want to edit the application, or do not want to expose the bug reporting tool to the public. This way, it is possible to annotate any web page the user opens in the browser, and it is not tied to a specific application either. The browser extension type of integration is less common, with seven tools utilizing this option.

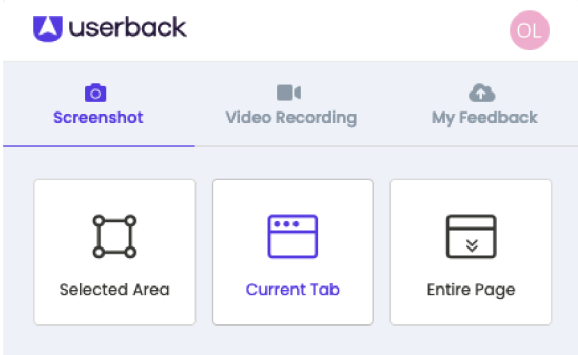

Figure 2.1: Userback extension for Mozilla Firefox

2.4 Integration with Issue Tracking Systems

Although most commercial bug-reporting tools provide their own issue-tracking systems, most projects will already use some issue-tracking system when they start looking for a visual bug-reporting tool.

It is the reason why all relevant tools have the option of integration with some third-party platforms. These platforms are mostly issue-tracking systems such as Jira [1], GitHub [22], GitLab [23], or Trello [24]. In these tools, a new issue is created every time feedback is submitted. Some of the tools have implemented an integration with communication platforms such as Slack or Microsoft Teams. Other platforms, such as WordPress or Shopify, belong to the list of integrable platforms as

well. All mapped tools except the open-source feedback.js [11] provide their own issue-tracking systems. Six tools support GitLab or GitHub integration.

2.5 Prices Comparison

The only open-source bug-reporting tool in the survey is feedback.js [11]. It offers a very basic annotation tool based on the HTML element selection principle. The tool does not gather any environmental data, and with the last commit in the code repository in 2012, it is already outdated.

A more significant part of available tools is commercial. There are usually more pricing programs per tool, even some freemiums occasionally. Their prices vary a lot based on the features they provide. Many platforms condition the price based on the maximum number of team members that can use the tool. Other tools are conditioned by the maximum amount of responses per month.

The maximum amount of monthly responses range from 50 to 1000 in freemium plans. In the cheapest paid programs of tools, the maximum amount of team members is in the range between three and ten. Some tools provide only their elementary and limited functions in their cheapest paid programs, where prices range from $15 to $70. However, there are also tools with no basic programs, with prices starting at $125, but these are considered enterprise with rich functionality.

Chapter Summary

All mapped bug-reporting tools support the most basic but crucial feature – a text description of the issue. Many tools provide information about the environment as part of the report. Another essential feature is the possibility of attaching screenshots with highlighted or annotated areas. Some tools allow attaching multiple screenshots, while others allow attaching only one. Some of them also provide an option to edit already submitted screenshots. Not many tools allow users to record a video in which users can reproduce the whole process of getting into the reported issue. It can be a very effective

way to report issues in complicated scenarios. Frontend integration is solved by in-code modification or installing a browser extension. All paid tools have implemented their issue-tracking systems but also allow users to integrate the tool with a limited amount of third-party issue-tracking systems. Most software projects already use some issue-tracking system for their internal issues, and hence it makes sense always to integrate bug-reporting tools with the issue-tracking systems they use. That is why issue-tracking systems, which come with these bug-reporting tools, might be considered unnecessary or redundant. It is worth mentioning that it also requires having a backend to implement its issue-tracking system, which is hardly possible to achieve for free, especially if scalability is one of the requirements.

3 Bug Reporting Tool Design

As it arises from Chapter 2, there is a significant gap with almost non-existent open-source bug-reporting tools. Therefore, we decided to fill this gap with a novel feature-rich open-source bug-reporting tool that will provide comparable capabilities to existing commercial alternatives. In this chapter, we define functional and non-functional requirements and the tool's architecture.

3.1 Requirements

Functional Requirements

The five main functional requirements include:

- The *BRT* will allow submitting textual reports to third-party issue-tracking systems.
- It will use the Screenshot approach (see Section 2.1) to capture the screenshot attached to the textual bug report.
- It will provide a set of tools to augment the attached screenshot with annotations.
- It will also collect selected environment data from the user's device.
- It will be integrated with the issue tracking system of GitLab and GitHub through their public APIs.

Screenshotting has been chosen since it is easier to use and allows for the integration of other annotation tools.

The annotation tools include two drawing tools (arrow and free-hand), the area obfuscation, and text comment. The drawing tools cover the two most common use cases: area designation and pointers. The area obfuscation will be realized as a black rectangle. For the text comment, the pin style was chosen mainly because pins occupy only minimal space on the canvas compared to text blocks.

With this somewhat limited set of features, it is still possible to create even complex annotations. Further graphical primitives (e.g., lines, polygons) could be added in the future. We intentionally omitted

emoticons, as they do not add any significant value to the annotation tool reports.

We selected only those environmental data that contextualize the image annotations and textual commentary. These include:

- timestamp,
- web browser name and version,
- operating system name and its version,
- screen and viewport sizes,
- browser console output history.

We omitted several other attributes since they are either irrelevant to the debugging process (e.g., reporter's location, IP address, pixel ratio) or can be derived from other collected data (device type).

Git has become a de facto standard versioning system for various projects nowadays. Platforms such as GitLab or GitHub build the whole development ecosystem around it, including their issue-tracking subsystems that also serve for managing bug reports. To integrate *BRT* with a third-party platform, a public API must be allowed to submit an issue and attach files to them. These are all reasons why the primary platform for integration was chosen to be GitLab and the secondary and optional GitHub.

GitLab API provides endpoints for both of these needs. To use these API endpoints, a user must authenticate using a token. At the time of writing this thesis, GitHub only provides an API endpoint for the first need – to create a new issue. As there is no option to upload a file attachment to the issue via its public API, the GitHub integration support was decided to be implemented only partially.

Non-functional Requiremenets

We also defined the following set of non-functional parameters:

- *BRT* component will be installable using in-code methods.
- it will be designed as a wrapper to wrap whole applications,
- it will be written in Typescript and React,
- it will be available as an NPM package for a better integration experience, and

- annotation tool will utilize `<svg>` element instead of `<canvas>` element for annotations.

Only in-code frontend integrations were selected for *BRT*. Integrating and installing the *BRT* as a React library and inserting the HTML script snippet into the specific HTML files should be possible.

Typescript is a superset of JavaScript, and it provides static typing and OOP (Object Oriented Programming) concepts, such as interfaces or generics to JavaScript. It is becoming a more and more preferred language for web development. The choice of React was made primarily based on our previous experience with development in it.

NPM registry was chosen as it is a widely used database for JavaScript open-source packages.

Annotation tools for web pages require appending either `<canvas>` or `<svg>` element to the web page. This element is the one where all annotations are captured. The `<canvas>` element is designed for drawings of graphics on web pages using JavaScript. Annotations can be drawn on the element itself without the need to append any extra elements. As a user can always resize the browser window, the `<canvas>` can cause scaling issues since the annotation tool must scale with the browser. However, when `<canvas>` is resized, its content is stretched or compressed, which is not an expected behaviour for annotations. Some existing bug-reporting tools use this approach for their annotation tools, but they need to take the screenshot upfront and use it as a static image background to prevent this scaling issue. Another candidate element is `<svg>`. It is the container for Scalable Vector Graphics (SVG). Unlike `<canvas>`, extra elements must be appended to the web page for every annotation. However, if the `<svg>` element resizes with the browser window, elements of annotations stay unmodified since all annotations are represented as vector graphics. Therefore, many bug-reporting tools use `<svg>` element as the solution. SVG is a declarative method of describing vectors, and it is a perfect fit with React's purposes [25]. These are all reasons why we chose to use `<svg>` for *BRT* as well.

3.2 Architecture and Workflow

BRT is an open-source library providing a web component for the frontend part of web applications. Figure 3.1 shows the high-level architecture of the *BRT*. It has two parts: the user interface and the business logic. The user interface provides forms to choose issue type (Bug or Enhancement) and submit textual reports, with an option to append multiple annotated screenshots.

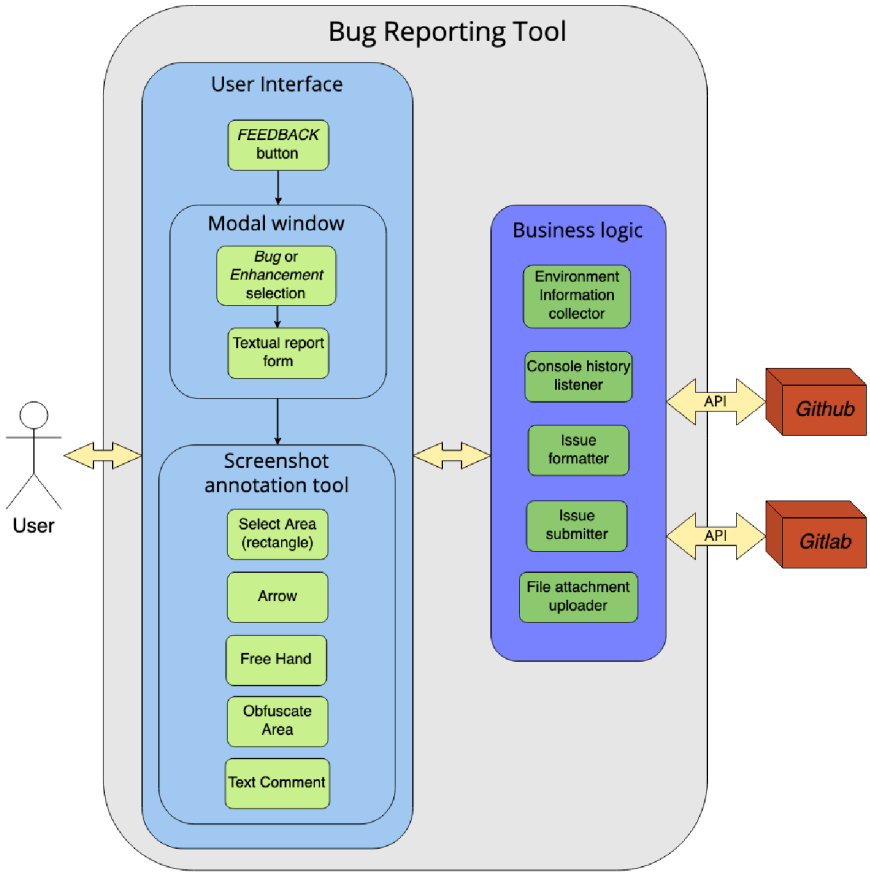

Figure 3.1: Bug Reporting Tool architecture

Behind the user interface, there is a business logic layer. All required environment information that the *BRT* is gathering is collected by designated functions here. In this layer, there is also a mechanism that listens for all console log-like function calls and stores the passed

arguments to track console history. There are also functions generating correct and uniformly formatted issue titles and descriptions. Issues must be created on issue-tracking systems, and functions securing this integration are also in the business logic layer. Also, attachments, such as screenshots or console history log files, must be uploaded, and this layer takes care of that too.

Figure 3.2 shows the user workflow. The entry point to the *BRT* is the floating FEEDBACK button at the page's edge. After a user clicks this button, a modal window opens with a selection dialogue, where the user chooses to report either a bug or an enhancement. Upon selecting one of the options, the content of the modal window changes to the textual report form with the mandatory fields Title and Description. There is also an email field, and if a developer wants to, he can configure the *BRT* to enforce email to be required too. The user can either send only the text report or attach and annotate screenshots by clicking the Annotate button. After clicking this button, the modal window closes with all filled fields saved, and the annotation tool appears. The annotation tool has a toolbar with all annotation types (area selection, arrow, freehand, area obfuscation, and text comment). There are also submit and close buttons. Both buttons close the annotation tool and re-open the modal window with the report form. If the screenshot was submitted, a screenshot tag appears in the form. By clicking the screenshot tag, a user can display an annotated screenshot to verify the result. There is also an icon to delete the screenshot on the tag. After a user clicks the Send button, the report is sent, and the bug reporting tool is closed.

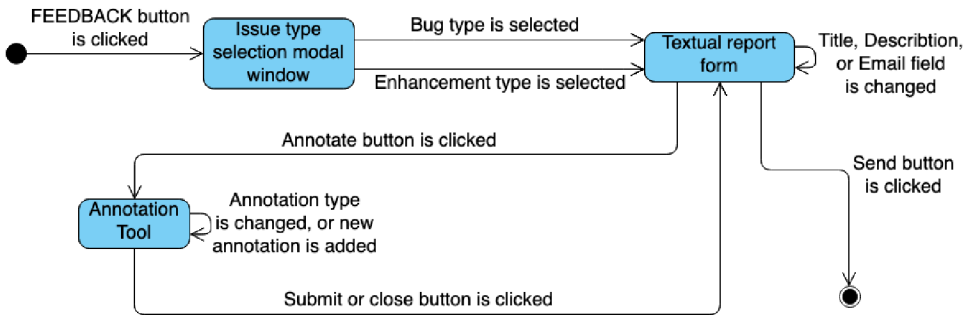

Figure 3.2: Bug Reporting Tool usage flow

4 Implementation

In this chapter, we present the implementation of *BRT*. We start with the overview of used technologies, followed by the source code folder structure, with a detailed description of the code decomposition and installation options. Finally, we provide details related to interesting or challenging parts of the library implementation.

4.1 Used Technologies

We discuss the technologies based on whether they were used directly to implement *BRT* (i.e., the core technologies), whether they were used to support the development process and those for deployment.

TypeScript is a typed superset of JavaScript, the most used programming language of the last few years. Every valid JavaScript code is, therefore, also a valid TypeScript code. TypeScript is statically typed, which avoids many unintended runtime exceptions or bugs. TypeScript brings well-known concepts from object-oriented programming such as classes, inheritance, interfaces and generics [26]. TypeScript has helped towards cleaner and more robust code, with minimum runtime exceptions.

React was chosen based on our previous experience, its documentation and its broad community of developers. It is a widely used open-source framework for frontend web development created by Meta Platforms. React leverages the idea of a component-based approach to build user interfaces, not only for web applications. React combines JavaScript or TypeScript code with an HTML-like JSX markup syntax. React is also known for a plethora of 3rd party open-source libraries that further simplify development.

For example, Material User Interface (MUI[1]) is a library containing predefined UI components (e.g., buttons, labels, text fields, blocks, and cards). The library is based on well-known Material Design created by Google designers. These components have a very refined, user-friendly design. Therefore we chose it for *BRT*.

1. https://mui.com/

Visual Studio Code (VS Code) was chosen as the primary IDE since it well supports web development of JavaScript applications and is easy to use. It also has a massive community of developers using it, so it is very easy to get support if needed. VS Code also offers plenty of plugins and also plugins for JavaScript development. For example, one of the plugins used during the implementation was Prettier, which automatically formatted code when the files were saved. Another helpful plugin was ESLint, a static analysis tool for JavaScript. It has also Git UI tools integrated, so all commits were possible to create and push comfortably through the GUI of VS Code.

ESLint was used for static analysis of the code [27]. As mentioned above, ESLint is available as VS Code plugin too. It shows warnings when some problematic pattern in code is found. The VS Code extension often offers a quick fix too.

Prettier is a code formatting plugin that helps ensure clean and consistent code. It is possible to configure Prettier to format the code every time the file is saved, and hence the correct format of the code is guaranteed. At the same time, it does not guarantee a clean code and can not substitute best coding practices [28]. To achieve clean code, developers must follow these practices, use proper names for variables, functions, or classes, have good function decomposition, and many more. Prettier is compatible with many languages, and among them, also with Typescript [29].

Rollup.js is a module bundler for JavaScript applications [30]. Every plugin for web applications is usually bundled into some more compact piece of code. Rollup serves exactly this purpose. It supports many formats such as CommonJS, ES modules, or IIFE[2]. ES modules is a standardized, official system for JavaScript modules.

Node Package Manager (NPM) [31] is an open-source project, serving the JavaScript community. It is not only a package manager for Node.js applications. One of the most known and used parts of NPM is **npm Registry**, which serves as a library of all publicly available and reusable libraries and packages of Node.js code, but also other various open-source projects using JavaScript or its flavours. An important complement to the npm Registry is **npm CLI**, which allows one to

2. Immediately-invoked Function Expression

download and install packages or even publish their own open-source packages.

GitHub is the hosting service for software development projects. It serves as a versioning system for git repositories too. For many reasons, GitHub was chosen as the versioning system for *BRT*. The main reason is that *BRT* provides also GitHub issue-tracking subsystem integration. GitHub is also the company behind NPM. The repository on GitHub is linked with the package after publishing to NPM, and the package is reachable from the GitHub repository's main page.

GitLab is a platform that hosts git repositories, similar to GitHub. It serves as a versioning system too. Because of its refined issue-tracking subsystem, it has been chosen as the main platform for *BRT* issue-tracking systems integration.

4.2 *BRT* Decomposition

In this section, we introduce the folder structure and describe the decomposition of *BRT* in detail.

The root folder contains configuration files for Typescript, NPM, ESLint, Prettier, Rollup, and Git. License and README files are here as well. The structure starts to nest in folders for Rollup build files (`dist/`), node modules, and the source code itself (see Figure 4.1).

The actual source code is grouped into five main subfolders. The `components` folder contains a separate folder subtree for the *BRT* part concerned with annotation-tool and another folder containing modals-related files and folders. Last but not least, it includes the component folders of the main component – `BugReportingTool`, and the UI entry point, which is `ReportBugButton`. There are context definitions in the homonymous folder. Folder `integration` contains folders `github` and `gitlab` grouping classes implementing issue controller interface for these platforms. Files describing models, implementing environment information collection, and issue description formatting are stored in `integration` subfolders. The `integration` folder contains an interface for issue controllers and their factory. Folder `models` contains files defining some new types or constants. In `styles` folder, there are CSS styles defined in two separate files for colours and modals. In the

```
BRT_ROOT/
├── dist/
├── node_modules/
├── src/
│   ├── components/
│   │   ├── annotations/
│   │   │   ├── AnnotationArea/
│   │   │   ├── Annotations/
│   │   │   ├── AnnotationTool/
│   │   │   ├── content/
│   │   │   │   ├── AnnotationAreaBorder/
│   │   │   │   ├── AnnotationAreaContent/
│   │   │   │   └── CloseButton/
│   │   │   ├── helpers/
│   │   │   ├── styles/
│   │   │   ├── tools/
│   │   │   │   ├── Arrow/
│   │   │   │   ├── DeleteButton/
│   │   │   │   ├── FreeHand/
│   │   │   │   ├── Obfuscation/
│   │   │   │   ├── SelectArea/
│   │   │   │   └── Text/
│   │   │   └── types/
│   │   ├── BugReportingTool/
│   │   ├── modals/
│   │   │   ├── FormModalContent/
│   │   │   ├── ModalController/
│   │   │   └── OptionsModalContent/
│   │   └── ReportBugButton/
│   ├── contexts/
│   ├── integration/
│   │   ├── github/
│   │   ├── gitlab/
│   │   ├── models/
│   │   └── utils/
│   ├── models/
│   └── styles/
├── LICENSE
├── README.md
└── ... other configuration files
```

Figure 4.1: *BRT* folder structure

source code folder, there is also a file defining the logic for the IIFE Rollup build, which is basically mounting the tool to the web page.

4.2.1 Components

Since React decomposes the code into so-called components, also *BRT* is decomposed into smaller related blocks – components.

Functional components have been chosen for *BRT*, instead of class components, as they have better code readability and lead to better performance [32]. These functional components are stateless functions returning JSX markup code, representing the component. The `useState()` hook is used to introduce states into the components. Most of the components contain their own states, with some local properties they need to store.

The main component is called `BugReportingTool` and encapsulates other building blocks which are decomposed by their main purpose. The feedback button is a floating element on the right side of the web page when the tool is closed. Integrations are controlled by one of `IIssueController` interface implementations. There is no need for any JSX component for integrations. Issue controllers are only OOP classes with methods for sending issues. The next building block is `ModalController`, which wraps and manages all modal windows. The biggest building block, controlling annotations, is wrapped in `AnnotationToolWrapper`.

One of the most significant environment information data sources is console output history. Unfortunately, it is impossible to extract complete console history since the browsers do not provide such functionality. To get at least some information, we implemented a workaround.

As `BugReportingTool` is the first component loaded, it redefines the console logging functions:

- `console.log()`
- `console.debug()`
- `console.info()`
- `console.warn()`
- `console.error()`

These functions need to be redefined so that every time they are called, they call themselves but also save their input to the state of `BugReportingTool` component. It is the only way to obtain the history as no function is solving this need.

Main Component

The main component, `BugReportingTool`, stores global states and flags. The global states are referenced and used across multiple children components, which are not even doing similar things. For example, screenshots are captured in the component dealing with annotations, but it needs to be accessible also in the component, which takes care of integrations with GitLab and GitHub, to ensure they are attached to the issue. The main component is the place where such properties are created and stored.

The flags represent global-wise indicators of the application state. For example, whether the tool is open, which issue type was selected in the modal window, if there is ongoing annotation, or whether the close annotation confirmation window is displayed. Also, all annotation objects and screenshots are stored here.

Feedback Button

The entry point of *BRT* is the floating FEEDBACK button (see Figure 4.2) that opens the initial modal window. The button is positioned on the right edge of the web browser by default and is present on every application page, which is wrapped in the main component wrapper `BugReportingTool`. If the *BRT* is installed using the HTML script tag, the Feedback button is present only on those pages where it was explicitly installed.

Figure 4.2: Feedback button

Modal Windows

The workflow of *BRT* utilizes modal windows a lot. There is a modal dialogue window shown when the user clicks the FEEDBACK button (see Figure 4.3). The content of this modal is defined in `OptionsModalContent`. This content is displayed in the parent component `ModalController`.

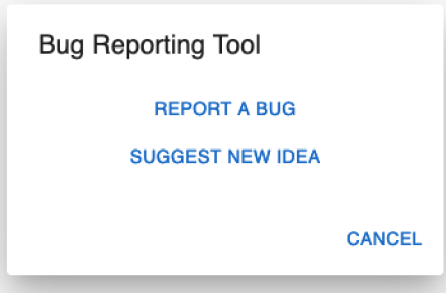

Figure 4.3: Modal window with `OptionsModalContent`

When the issue type – Bug or New idea is selected, the modal window content changes to the `FormModalContent` with the form for issue details (see Figure 4.4). These details are the title, description, and email address.

The email address is not required by default, but it can be configured to be required by passing `isEmailRequired` property to the main component `BugReportingTool`. The Title and Description are required without any option to configure them. In `FormModalContent`, there is also a button to start the annotation tool – ANNOTATE and another button to submit the issue – SEND.

`ModalController` takes care of which modal content should be displayed. It is achieved thanks to conditional rendering in React, as there are conditions on whether or not to display the content based on the flags defined in the main component accessible from `ToolContext`.

This approach mainly ensures that the screen does not flick when the content of the modal window changes. Material UI modal windows show and hide with a transition of background colour effect,

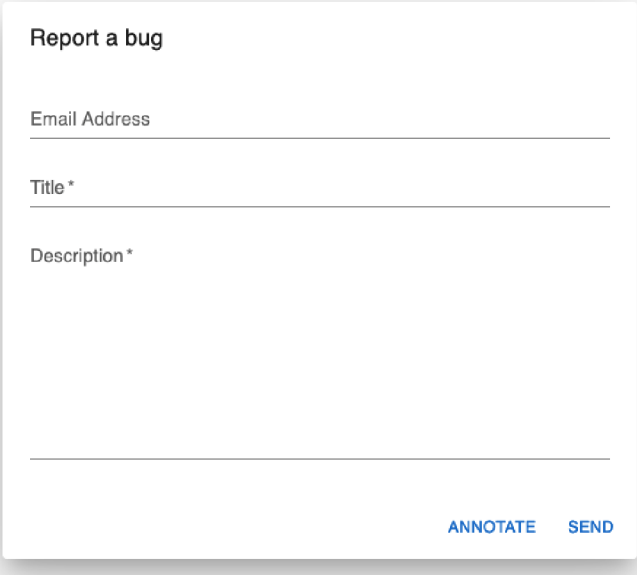

Figure 4.4: Modal window with `FormModalContent` and *Bug issue* type selected

so it seems that the background gets darker smoothly or disappears smoothly. Using two separate modal windows for each of the contents would cause an undesirable background flick effect.

Annotations

The *Annotation Tool* is the name for the whole *BRT* part concerning annotations. It is the core feature of *BRT* and is accessible from `FormModalContent` by clicking the ANNOTATE button. It is made of multiple tools representing individual annotation types: select area, arrow, freehand, obfuscate area, and text comment.

There are multiple levels of abstraction in the Annotation Tool. The first and the most outer one is `AnnotationToolWrapper` which is introduced only to wrap the whole `AnnotationTool` into a provider of `AnnotationContext`, where all states, used exclusively by components in `annotations/.` subtree, are accessible from.

The root component of all annotation tools is `AnnotationTool`. It calls hooks which return mouse handlers responsible for the creation of annotations and handlers managing the relocation of annotations, and it stores the state of the currently chosen tool.

Whether the Annotation Tool is displayed or not depends on `isOngoingAnnotation` state stored in global `ToolContext`. The `AnnotationTool` component displays `AnnotationArea`, which is responsible for showing all created annotations. It also displays `AnnotationToolContent`, a controller for all the annotation tool's buttons, the toolbar with the annotation tool's buttons, the close button and the borders.

Figure 4.5: The Annotation Tool usage example on simulated Google home screen page.

In Figure 4.5, there is an Annotation Tool used on the simulated Google home screen to distinguish the web page content from what is added by *BRT*.

The red borders indicate an active Annotation Tool. The toolbar with the annotation tools is aligned in the top left corner; the close button is in the upper right one. There are also several annotations demonstrating the implemented tools:

- An area selection tool is used on the part of the Google logo. It can be seen that everything outside this selection is in less striking colours, as there is a dimmed overlay.
- An arrow connecting comment no. 1 and the area selection annotation.
- A freehand drawing in the right upper part.
- An obfuscation tool is used to hide Pictures button right of the Gmail button (top right corner).
- There are also two comments. The open one (relates to the pin no. 1) describes the selected area. The second one is closed and may contain a message itself.

The annotations can be moved using the drag-and-drop mechanism. It is possible to delete them individually by clicking on a trash can icon, which shows when a user hovers over the annotation element with a mouse cursor.

The text comment can be opened and closed by clicking on the respective pin, and it is possible to close it also by clicking the green tick button on the comment itself. The comment is draggable only by grabbing it by its pin.

The toolbar is draggable vertically with limits on the window size. If it is dragged across the whole screen to the right side, the toolbar gets pinned to the right edge of the window and is again draggable vertically. The reason is that the toolbar may unintentionally cover important parts of the web page which a user might want to see or even annotate.

When a user clicks the close button, the confirmation modal appears so the user can confirm that this was intentional. The close button will cancel all the annotations.

The green tick flag in the toolbar saves all the annotations into an image file and appends the screenshot to the bug report. After the screenshot is submitted, the Annotation Tool closes, and the modal window with the form reappears. The data provided earlier are asynchronously loaded (the submission process does not block the tool's UI), and the user can add more screenshots after the previous one gets submitted.

The submitted screenshot contains text comment annotations represented in the closed state as numbered pins. The texts are attached to the main issue description referenced in the respective screenshots. The elements of the `AnnotationToolContent` component (e.g., borders, close button, and the toolbar) are omitted in the screenshot too.

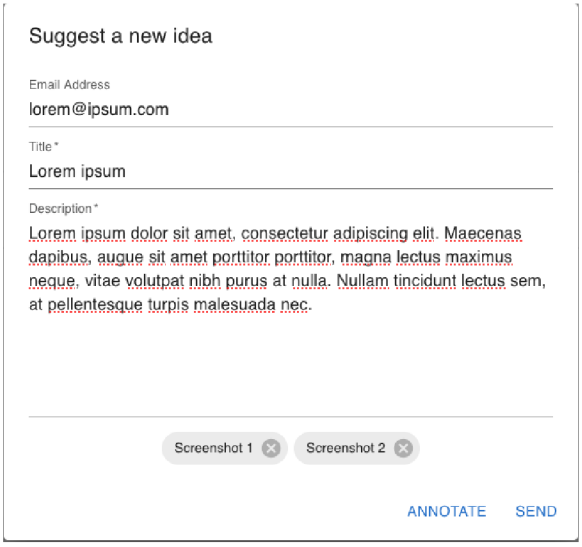

Figure 4.6: Example of the form modal window after submission of two screenshots

All screenshots can be reviewed or deleted using tags, which are displayed in the form after the screenshot is submitted and loaded (see Figure 4.6). When a user clicks the SEND button, the modal window closes, and issue creation gets requested. After the browser receives a response from the GitLab or GitHub server, whether the issue was created successfully, the Snackbar in the lower right corner with success or error message displays respectively (see Figure 4.7).

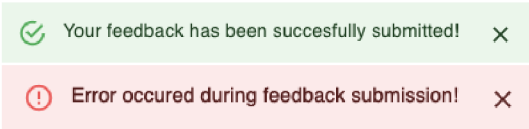

Figure 4.7: Snackbar with the successful (upper) and unsuccessful (lower) issue creation message

4.2.2 Contexts

A lot of properties of components are stored in states of particular components. As the nesting depth of components is deep in some cases, these properties are passed through multiple components, which creates a sort of boilerplate code. To avoid that, *BRT* utilizes contexts.

Context allows us to share its properties across all components in its scope. Suppose component A, which is composed of components B and C, is wrapped in the context. In that case, it is easily possible to access properties of context also in components B and C without the need to pass these properties to both of them. The `useContext()` hook can be used to get these properties.

Tool Context

In `BugReportingTool` component, there are multiple states that describe properties, which are requested globally in many places down the component decomposition tree. Hence they have been wrapped into the `ToolContext` accessible from all *BRT* components. These properties are:

- list of all annotations and their setter,
- list of all screenshots and their setter,
- information if there is an ongoing annotation and its setter, and
- console output history.

Annotation Context

`AnnotationContext` contains only properties requested only by components in `AnnotationToolWrapper` component tree. It holds the following information:

- annotation ID in hand or the next free annotation ID,
- function definition to add a new annotation,
- list of all selected areas and their setter,
- list of all open comments and their setter,
- boolean flag whether the new annotation is being created at the moment.

4.3 Integration

Integration with GitLab and GitHub requires calling their public REST APIs. Classes implementing the `IIssueController` have been created, with one required method from the mentioned interface – `newIssue(IssueInfo)`.

Also, the `IssueControllerFactory` is implemented following the *Factory* pattern. This factory has a method returning the appropriate issue controller based on the platform it is given. This factory is used in `BugReportingTool` component to obtain the correct issue controller.

4.3.1 GitLab Integration

GitLab has a separate API endpoint to create an issue and another endpoint to upload a file attachment.

To create a new issue, there exists `/api/v4/projects/PROJECT_ID/issues` [33] endpoint where issue details can be provided through query parameters. For the purpose of *BRT*, the required parameters are title, description and labels.

Endpoint `/api/v4/projects/PROJECT_ID/uploads` [34] serves to upload a screenshot to the server. Call with the POST method to this endpoint returns a response with an uploaded screenshot URL. On top of the full URL address and other information, the response also contains the URL in markdown format, which can be used directly in the issue description. The screenshot link is then automatically expanded into the picture.

To be able to use these endpoints to create new issues and upload files, the developer must generate an access token to grant *BRT* write rights to the scoped project API. The required scope, which sets the access token's permission level, is called *api*.

4.3.2 GitHub Integration

An API endpoint to create a new issue on GitHub is `/repos/OWNER/REPOSITORY_NAME/issues` [35]. This endpoint accepts issue details through the request call body. In the body, relevant attributes for *BRT* purpose are title, body and labels. Unfortunately, GitHub does not support direct uploads of screenshots or any files through its API.

4.4 Challanges, Problems, and Optimizations

During the implementation, we encountered a lot of challenges and difficulties. The most challenging was implementing text comment annotation type behaviour to be as intuitive as possible. There were also troubles with annotation tool performance. The implementation of GitHub integration was the most problematic one. We describe all these topics in this section.

4.4.1 Text Comment Annotation User Experience

The pin representation was chosen for text comment annotation type, as it scales better with longer comments. Each text comment pin has a thought bubble with a text field assigned, which can be hidden (or closed). It was challenging to implement intuitive behaviour. The pin is simply represented by an SVG circle. However, the thought bubble is implemented as a `foreignObject` made of Material UI `Card` and `TextField` elements. There is no built-in logic for this type of component, so the whole behaviour had to be implemented from scratch. As there are mouse event listeners on the parent SVG element, it was initially creating new text comments every time the user clicked even on the existent thought bubble of text comment because the mouse click event was propagated to the parent element. It was basically impossible to type any input to the text comment. The expected behaviour is that the clicked text comment activates itself for the input. It was fixed by stopping event propagation to the parent SVG element.

There were also other challenges. For example, when a user has a text comment open in Userback's [15] annotation tool, and the user clicks outside the currently opened text comment, the currently active comment moves to the new place, where the user clicked. So to create a new text comment, a user must close the active one. It is not intuitive behaviour, so in *BRT*, we decided to implement it differently. It does not matter whether a user has opened any comments, but whenever he clicks on a non-occupied area by any annotations, a new text comment is created. This way, it is possible to have opened multiple text comments.

A text comment is possible to be opened by clicking on the pin. In the same way, it is possible to close already opened text comments

(a user can use a green tick button inside the thought bubble of the comment as well). And since the user could relocate the text comment by grabbing it, it sometimes happened that the cursor unintentionally moved by a few pixels. As a result, it became very difficult to open or close the comment. To resolve this issue, we implemented the 5-pixel movement threshold. If the cursor distance between mouse `down` and `up` events is smaller or equal to five pixels, it is considered that it did not move, and a user just clicked. By empirical tests, it turned out to be an appropriate value for the threshold.

4.4.2 Low Performance of Annotation Tool

The creation and relocation of annotations require a lot of rendering since there is an animation of annotations being created and also moved to new positions when the user moves a cursor.

In the initial implementation, the same mechanism was used for the creation and relocation of annotations. The mechanism works in the following way: Every time a user moves the cursor while creating a new annotation or moving an existing one, all necessary properties are recalculated based on the difference between the cursor's previous and actual x and y-axis. For example, in the area selection annotation type, the x, y, *width* and *height* properties are recalculated. In the case of the freehand drawing annotation type, when creating a new one, the new point is appended to the path [36]. However, when relocating it, the x and y axis must be recalculated for every single point of the path, which leads to a performance decrease as it has $O(n)$ time complexity, where n is the number of points at the path. On top of that, there was needed a separate recalculation function for each annotation type, as all of them have different properties.

The solution for this issue was to use CSS transform property with matrix function where the used matrix looks as follows.

$$\begin{pmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{pmatrix}$$

The tx and ty parameters define a shift on the x-axis by tx pixels and a shift on the y-axis by ty pixels. Using this approach, there is no need for recalculation of every single point in the case of freehand

drawing, and the time complexity is $O(1)$. Furthermore, there is no need for separate recalculation functions for each annotation type as they can all use the relocation by matrix independently of their properties.

4.4.3 Unsuccessful GitHub Integration

As mentioned in section 4.3.2, GitHub does not support file uploads intended for issue attachments. There are several workarounds, e.g., using third-party cloud storage for file uploads. However, this increases the workflow complexity and adds another dependency on such storage service. Another approach might be to use a dedicated GitHub repository to store the attachment files. Yet, storing screenshots somewhere else from the issue would make an undesired mess and further complicates the issue management. The third workaround is to use releases and their assets. Releases in the GitHub context are used mainly to bundle iterations of the project, and for example, binaries can be linked to them by uploading them as their assets. These assets can be any files, even pictures or screenshots. Furthermore, releases can be marked as drafts, which means they are not publicly visible, so they do not create an undesired mess in the repository. The idea behind this workaround is to create a new draft release which will serve as a screenshot storage in the form of assets. To create the draft release, there exists `/repos/OWNER/REPOSITORY_NAME/releases` [37] endpoint, and to upload an asset `/repos/OWNER/REPOSITORY_NAME/releases/RELEASE_ID/assets` [38]. However, neither this workaround is a viable option as it turned out. This workaround theoretically works and even practically, but only under certain circumstances.

First, it was tested alone, separately from the *BRT* component. For that, the TypeScript script was created, and function fetch was utilized. It was ready for relocation into the *BRT* GitHub issue controller. Calling GitHub API endpoints responsible for draft release creation and assets upload from the script resulted in success.

As the workaround was tested and it was functional, the script was refactored and inserted into the GitHub issue controller of *BRT*. There was no difference between the TypeScript code and *BRT* GitHub issue controller method implementation, only some refactoring changes.

However, calling the endpoint for assets upload raised a CORS error[3] because this GitHub endpoint does not support cross-origin requests, as was later revealed. This means until GitHub changes its CORS policy, it is not possible to upload screenshots as assets directly from the *BRT* component, as it will always be opened in the browser where the CORS policy applies.

The only option is to have a server from which the assets upload calls would be made, as the CORS policy applies only to browsers. The *BRT* component would have to call the endpoint of this server, and the server would call `/assets` endpoint without getting the CORS error. This can be considered as possible future work for BRT. For now, the *BRT* supports creating issues on GitHub, but the annotation tool was disabled.

3. If the server does not allow cross-origin requests and a request is made from a different origin to the server, the CORS error appears. This CORS policy applies only to requests made from browsers.

5 Installation

BRT is available as an NPM open-source library for React. It can be installed from the terminal using the command:

```
npm install @xsvetli1/bug-reporting-tool
```

This installation option depends on the target application's frontend framework as it is compatible with React only. There are two prerequisites for this installation option. The developer must be authenticated to GitHub Packages and add the GitHub registry to the npm configuration. To authenticate, the developer may generate GitHub Access Token with scope `read:packages` and add it to the NPM configuration by running the following command:

```
npm config set //npm.pkg.github.com/:_authToken=GITHUB_PACKAGES_TOKEN
```

To add a GitHub registry to the NPM configuration developer may run the following command:

```
npm config set @GITHUB_USERNAME:registry=https://npm.pkg.github.com
```

For applications using another frontend framework, or for those which do not use any framework and work only as static web pages, there is an option to append HTML `<script>` tag directly to HTML files of pages where developers intend to add this component. This way, the component will be visible only on these pages, not in the whole application. The script tag automatically appends the *BRT* to the page. This is achieved by an IIFE[1] Rollup bundle format.

Further, we describe the steps for integrating the *BRT* with GitLab and GitHub since each platform requires additional properties to be set.

5.1 GitLab Integration

There are three required properties to be set:

- `server` is a Javascript URL object wrapping the GitLab server address;

1. Immediately Invoked Function Expression

- `projectId` – ID of the GitLab project that can be found in the General tab of repository settings;
- `authToken` that can be generated in the Access Tokens tab of repository settings. Note that the minimum scope required to be set is `api` (see Figure 5.1). The *authToken* is not the same as `GITHUB_PACKAGES_TOKEN` from the *BRT* installation process.

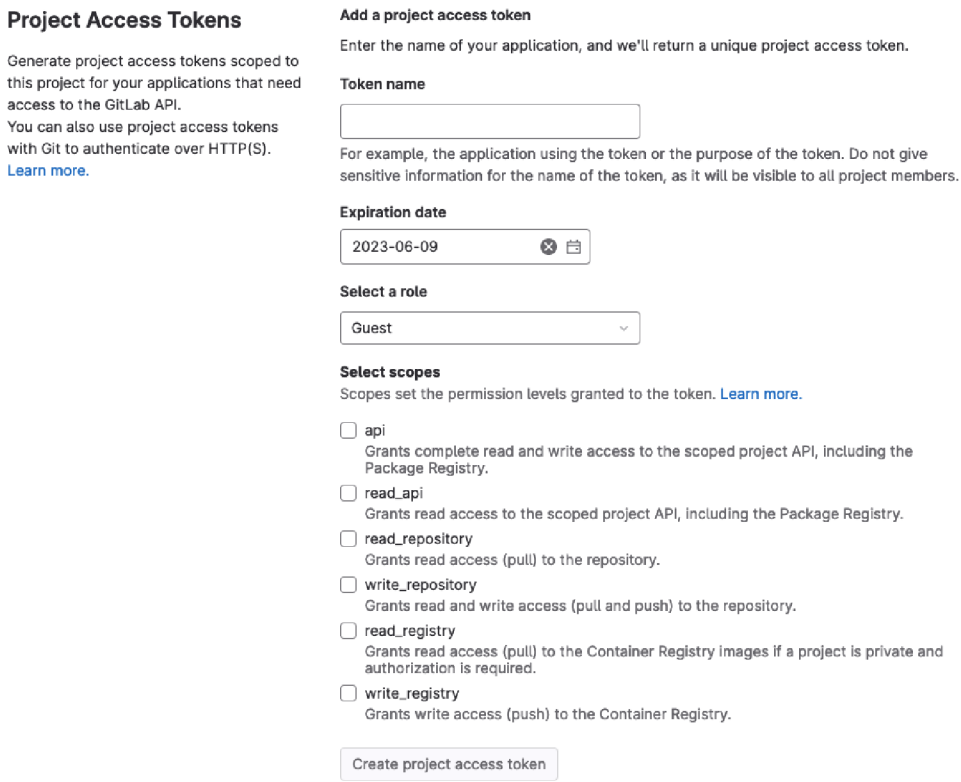

Figure 5.1: Gitlab Project Access Tokens screen

React Usage

To add *BRT* to React application, the whole application must be wrapped in `BugReportingTool` component. GitLab properties must be passed into the component, as in the following example:

```
<BugReportingTool
  platform={"Gitlab"}
  platformProps={{
    server: new URL(GITLAB_SERVER_URL),
    projectId: PROJECT_ID,
    authToken: AUTHENTICATION_TOKEN
  }}
  isRequiredEmail
>
  // ... app code goes here
</BugReportingTool>
```

The `isRequiredEmail` line can be omitted, and the email will not be the required field.

HTML Usage

To utilize *BRT* via appending a script to an HTML file, the following code snippet must be inserted into desired HTML files, and platform properties must be set to the correct values:

```
<script>
    window.BugReportingToolProps = { platformProps: {} };
    BugReportingToolProps.platform = "Gitlab";
    BugReportingToolProps.platformProps.server = new URL(GITLAB_SERVER_URL);
    BugReportingToolProps.platformProps.projectId = PROJECT_ID;
    BugReportingToolProps.platformProps.authToken = AUTHENTICATION_TOKEN;
    // uncomment following line to make the email required
    // BugReportingToolProps.isEmailRequired = true;
    (function () {
      var script = document.createElement("script");
      script.async = true;
      script.src =
        "https://cdn.jsdelivr.net/gh/xsvetli1/bug-reporting-tool/dist/iife/index.js";
      (document.head || document.body).appendChild(script);
    })();
</script>
```

5.2 GitHub Integration

Since GitLab and GitHub API differs, the properties are different too. There are three required properties to be set:

- *owner* is the GitHub repository owner's username;
- *repository* is the GitHub repository name;
- *authToken* is the authentication token that can be generated on GitHub repository Settings. The *authToken* is again not the same as `GITHUB_PACKAGES_TOKEN` from the *BRT* installation process.

To generate one, the developer must click on their own profile photo and navigate to Settings > Developer Settings > Personal access tokens > Tokens (classic) > Generate new token > Generate new token (classic) (see Figure 5.2). If the component will be integrated with a public repository, the `public_repo` is minimal required scope. However, if the repository is private, the minimal required scope is `repo`.

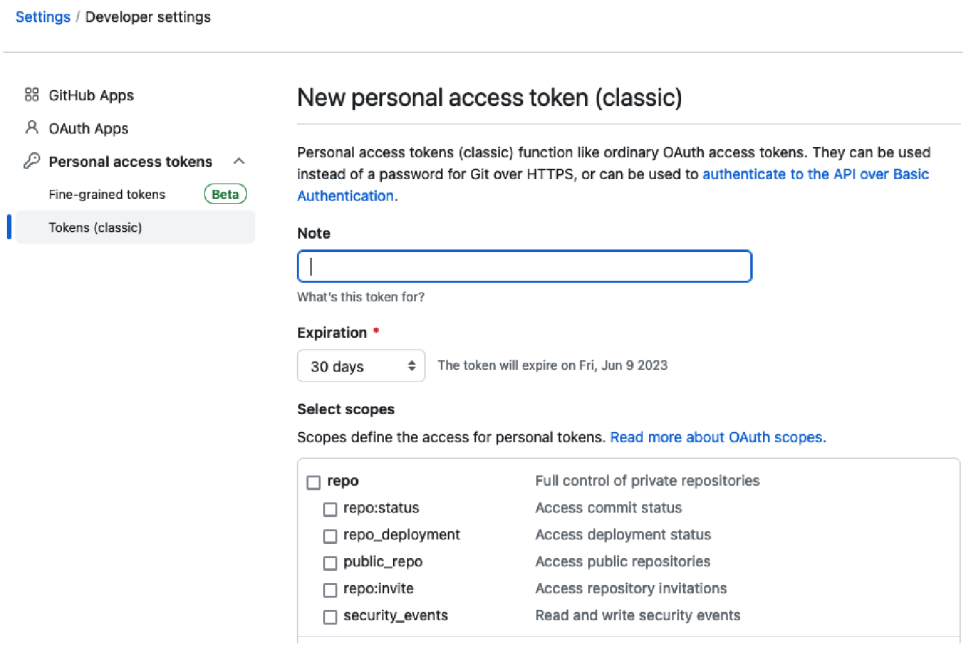

Figure 5.2: GitHub Project Access Tokens screen

React Usage

Same as for GitLab, to attach *BRT* to React application with GitHub integration, the application must be wrapped in `BugReportingTool` component with the properly set properties:

```
<BugReportingTool
  platform={"Github"}
  platformProps={{
    owner: GITHUB_USERNAME,
    repository: GITHUB_REPOSITORY_NAME,
    authToken: AUTHENTICATION_TOKEN,
  }}
  isRequiredEmail
>
  // ... app code goes here
</BugReportingTool>
```

Likewise, the line `isRequiredEmail` is not mandatory; if removed, the email will not be enforced as a required field.

HTML Usage

The HTML usage is almost identical, except for the GitHub-specific properties:

```
<script>
    window.BugReportingToolProps = { platformProps: {} };
    BugReportingToolProps.platform = "Github";
    BugReportingToolProps.platformProps.owner = GITHUB_USERNAME;
    BugReportingToolProps.platformProps.repository = GITHUB_REPOSITORY_NAME;
    BugReportingToolProps.platformProps.authToken = AUTHENTICATION_TOKEN;
    // uncomment following line to make the email required
    // BugReportingToolProps.isEmailRequired = true;
    (function () {
      var script = document.createElement("script");
      script.async = true;
      script.src =
        "https://cdn.jsdelivr.net/gh/xsvetli1/bug-reporting-tool/dist/iife/index.js";
      (document.head || document.body).appendChild(script);
    })();
</script>
```

6 Testing and Feedback

In order to get feedback, we launched user and developer testing. We prepared a fictional scenario to test the *BRT* by user respondents. Respondents were supposed to use *BRT* on a prepared webpage. The whole scenario was described in the assignment and placed in the header of the mentioned webpage. After respondents completed the assignment, they were asked to fill out the feedback form. Other respondents were chosen for developer testing. For them, the assignment was to try to install *BRT* and test it by submitting some reports and validating that submission was successful. Developer respondents were asked to provide feedback by submitting the form as well.

6.1 User Testing

A total of twenty-one respondents participated in user testing. They were aged from eighteen to sixty years, with an average age of 29 (see Figure 6.2), 61.9% males, 33.3% females, 4.8% preferred not to say (see Figure 6.1). 71.4% of the respondents studied or worked in information technologies. Respondents were of diverse user experience levels. There were 33.3% of respondents with no issue reporting experience, 38.1% reported at least one and at most five issues in the last three years, and 28.6% of respondents reported ten and more issues in the same time period (see Figure 6.3).

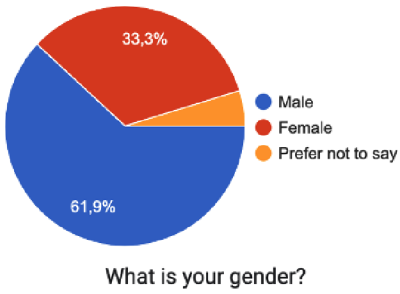

Figure 6.1: Age of user testing respondents

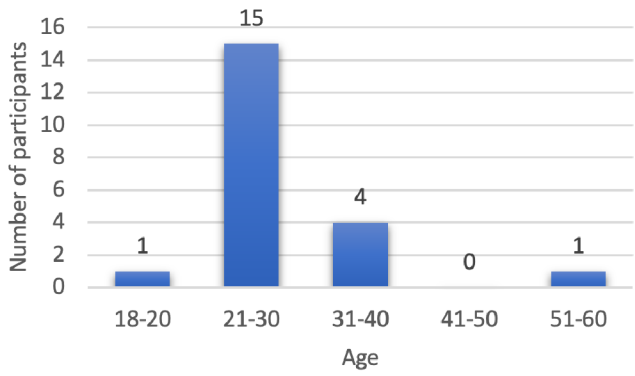

Figure 6.2: Age distribution of user testing respondents

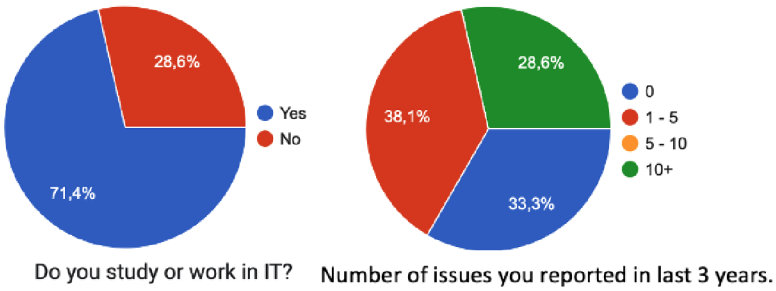

Figure 6.3: User testing respondents' background and experience

Assignment

We prepared a fictional scenario for respondents, which was hosted on our faculty web page[1], with *BRT* installed. There was a screenshot from the Stripe payment gateway copied from the Stripe documentation, which served as a base for our scenario (see Figure 6.4).

1. https://www.fi.muni.cz/ xsvetli1/

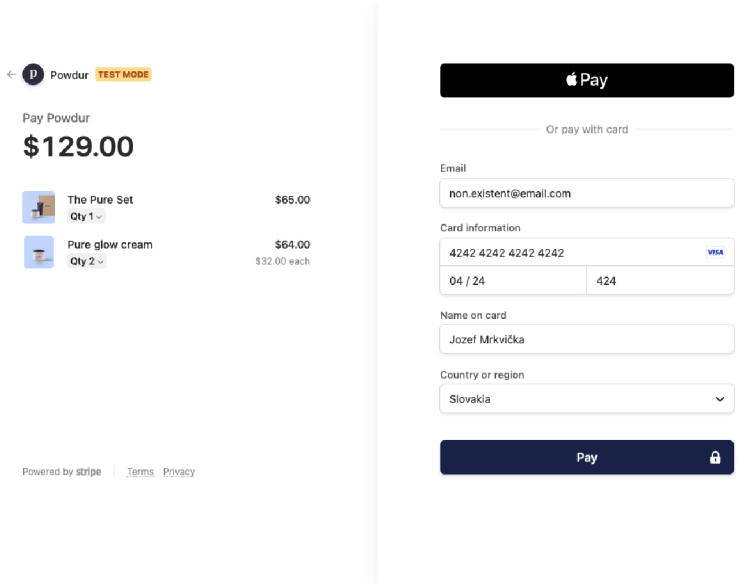

Figure 6.4: Base for the scanario

The scenario had the following wording:

You are buying cosmetics online and paying via a payment gateway. You already entered your email and credit card details. However, when you click the Pay *button, nothing happens, and you do not get any response. It does not redirect you to the merchant or show any error message.*

This scenario was designed in such a way that all annotation types should be helpful in reporting the issue. The area selection is helpful in highlighting the elements of interest. Arrows and freehand annotations can be used as pointers, text comments to provide the local description, and even obfuscation to hide sensitive information.

In the assignment, we asked respondents to use *BRT* and submit a report about the issue from the scenario. We stressed to respondents not to forget to annotate the web page. We emphasized that they should act as if the details pre-filled in the payment gateway were their own and that some developers might see them in the screenshots of the report. We did so to impale them to use also the area obfuscation. We also asked them first to try all annotation tools, not to miss anything they might find useful. We provided exact steps on how to start processing the assignment by scrolling down and clicking the

floating FEEDBACK button on the right. We also asked participants to fill out the feedback form after submitting the report.

Results

All annotation types were utilized by at least ten user respondents. The least used annotation type was freehand drawing, with ten respondents using it. The most used annotation types were area selection, area obfuscation and text comment, with fourteen respondents using them (see Figure 6.5).

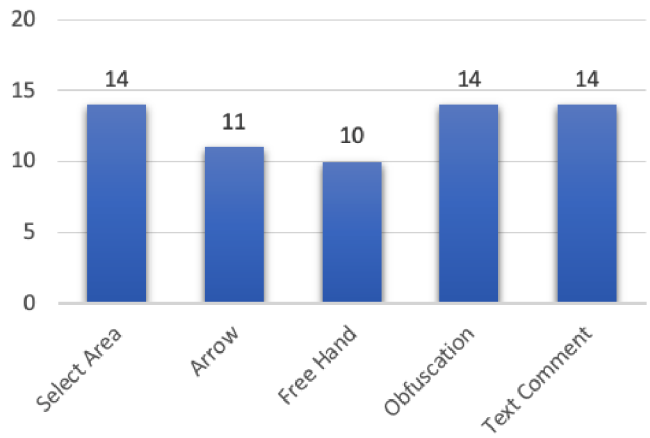

Figure 6.5: Amount of respondents using individual annotation types

For each annotation type which respondents used, we asked them three questions, which they answered by providing a number in the range between one (the least) to ten (the most):

- *Suitability:* How much was the given annotation type the right choice for your intention?
- *Intuitiveness:* How intuitive was the usage of the given annotation type?
- *Usability:* How easy was it to use the given annotation type?

We summarized answers by computing the weighted arithmetic mean of points for each annotation type and each question (see Table 6.1). These results imply that all annotation tools are intuitive to

use, but not all of them are always the right choice for the intended use or are easy to use. For example, the freehand annotation type is very intuitive with a weighted arithmetic mean score of 9.00 out of 10 (which is still the lowest score in the Intuitiveness category), but it gained the lowest score in the question, of whether it was the right choice with 7.20 points out of 10 and also in the question of easiness to use (8.50 out of 10), which makes it lowest ranked annotation type in all categories and hence it gained the lowest total score too. Paradoxically, the highest-ranked annotation type is Text Comment which was the most complex for implementation, and therefore we expected to face most issues right with this annotation type. It had a total score of 9.62 out of 10, being the most intuitive and easy-to-use annotation type. The highest-ranked annotation type in suitability is area obfuscation, followed by Comment. Overall, all annotation types gained a total score higher than 8.00 points which we consider to be a great success.

Table 6.1: Weighted arithmetic mean from all points gained in each category for each annotation type

	Select Area	Arrow	Free Hand	Obfuscate Area	Comment
Suitability	8.71	8.73	7.20	9.71	9.36
Intuitivness	9.21	9.45	9.00	9.36	9.71
Usability	9.36	9.00	8.50	9.36	9.79
Total score	9.10	9.06	8.23	9.48	9.62

The respondents could provide comments on each annotation type they used. Respondents identified several issues and bugs, suggested improvements, and provided compliments of features they liked.

In the case of Select Area, respondents experienced behaviour that was not caught during the development. After the area is selected, it is not possible to annotate anything else on top of the selected area, as the Select Area relocation drag-and-drop mechanism is activated every time a user tries to do so. This issue was caught by two different respondents. One respondent experienced that it was not possible to delete the selected area if the rectangle was too narrow. The next revealed bug was that with an active annotation tool, it is possible

to scroll the webpage in the background, which is not expected behaviour. One respondent did not like the bright red colour of the annotation tool. Two respondents suggested it would be nice if the area could be resized. There was also a suggestion to implement a feature that allows going a step back (CTRL+Z). Another respondent wrote it would be convenient to change the cursor to the + symbol. There were also positive comments regarding user-friendliness or that background outside the selected area is blurred and that the selected area is movable and deletable.

Found issues for arrows are that they were not possible to be deleted if they were too small (as the trash can icon does not appear on the hover event of the arrow's tip). Respondents did not like that they could not change the directions of arrows. Some of them suggested implementing this feature. They appreciated that arrows are movable and deletable.

Respondents using freehand drawings were bothered by an unintentional drag-and-drop mechanism activation when they wanted to add another freehand drawing (similar to the area selection issue). There were also suggestions for new functionality, namely the possibility of changing the line width and implementing an eraser.

The obfuscation area and area selection annotations have very similar behaviour. Hence they share the same issues. Respondents encountered issues when they drew a very narrow rectangle, which was more of a line than a rectangle. The problem was that it was tough to delete it, as the trash can icon appears only on hovering exactly over the rectangle, without any margin. They lacked the possibility to resize the obfuscated area. Apparently, it was not clear how to use area obfuscation, as some respondents thought it works as a line or a freehand drawing, but all of them concluded that a rectangle makes more sense. Again one respondent lacked a step-back feature. There was a unique suggestion to implement a feature to have the option to change the shape of obfuscation. Some respondents recognized the possibility of moving and deleting the obfuscated area, and others liked it without providing any critics.

For the text commentary annotation type, no bugs or issues were reported, only a few comments. Respondents liked that comments were numbered and that they were movable and deletable. They also recognized that they could hide comments, which *"helped massively*

with making annotation easy to read and orientate in it" (P16). They appreciated they could create new text comments with just one click. P9 also realized important reasoning, why we actually chose the pin style of comments, as one can fit more text in it this way without overlapping the image. There was one suggestion for improvement by allowing users to connect commentary to other annotations, for instance, with an arrow, to make it one element instead of two.

Apart from comments regarding specific annotation types, respondents also provided general comments at the end. P1 was not sure what screenshot was actually being sent. This probably occurred because screenshot tags are not yet correctly displaying the screenshots in all web browsers (for example, Google Chrome). On hovering over the toolbar buttons, an explanatory text appears. However, it is under the toolbar itself, as it was found out by one respondent. There were also the following suggestions:

- To have multiple layers of annotations, to avoid situations like comments stuck behind the selected area.
- To add a cancel button to modals, as less experienced users might not know, the modal disappears after clicking outside of it.
- To have a colour palette and some help pop-up buttons to display examples and tutorials on how to use the tool.
- To add more annotation tools and clarify what email should be entered in the form.
- The overall experience could be improved by providing some live demos of tool usage.

There were also complimentary comments, recognizing intuitiveness, user-friendliness, ease of use, usefulness, and necessity.

Examples of Screenshots Annotated by Respondents

In Figure 6.6, we can see the usage of the annotation tool, where the selected area (red border) is used to define the whole area of interest. The issue is described using text comments pointing to related elements using arrow annotations. Area obfuscation is used to hide sensitive information.

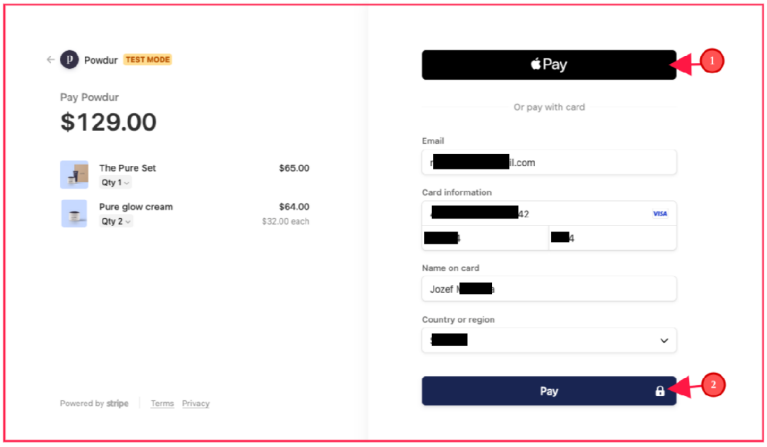

Figure 6.6: Annotated screenshot by a respondent. Comments:
1. Nothing happens after clicking;
2. Nothing happens after clicking here either.

In Figure 6.7, we can see the usage of only two wider obfuscated areas serving the same purpose as in Figure 6.6. However, the selected areas are concentrated on much more specific elements with comments nearby without the need for any additional pointers. This usage is much simpler and cleaner, but the message of it is the same.

Figure 6.8 demonstrates how the thought process differs among the respondents as different respondents used different annotation types for the same purpose. In this example, a freehand drawing is used to obfuscate sensitive information.

User Testing Summary

After analyzing the user testing, we conclude it brought a lot of valuable feedback, either of what is done well as it is, what could be done better, or what should be fixed. It also provided insight into how different reporters might think and use different annotation types for various purposes. The testing itself revealed wide space for possible future work. After receiving this feedback, we fixed two most critical bugs – enabled page scrolling with an open annotation tool and unintentional drag-and-drop mechanism activation in the Select Area annotation type. The former bug was resolved by setting the body ele-

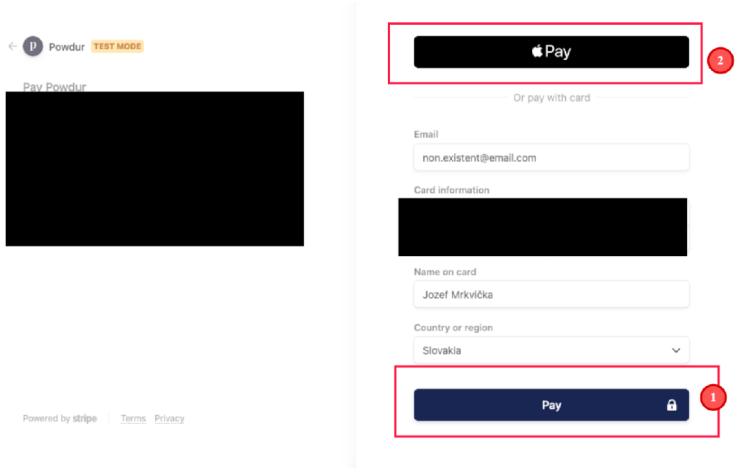

Figure 6.7: Annotated screenshot by a respondent. Comments:
1. After filling all the necessary information, the 'Pay' button is locked, even though it should not be;
2. Not working, it will not redirect anywhere.

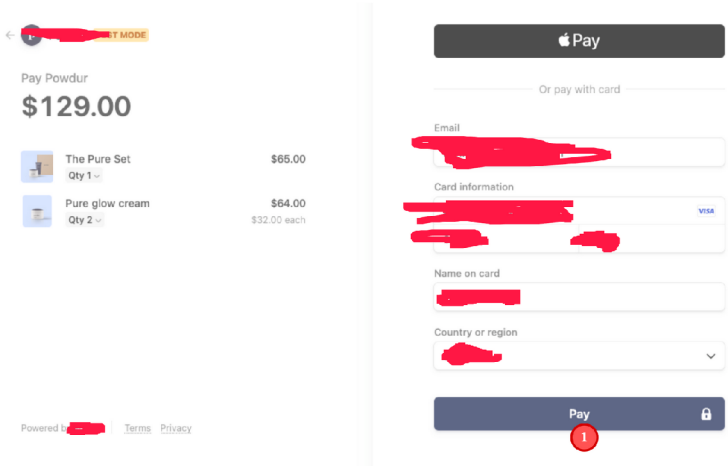

Figure 6.8: Annotated screenshot by a respondent. Comments:
1. This is a button I am talking about; *Respondent referenced a Pay button in the general description.*

ment's CSS `overflow` property to `hidden` when the annotation tool is active. The latter one required more complex change in the `SelectArea` component, where the `rect` SVG element had to be replaced with two `path` elements – one for the actual border and another for the thicker transparent border, serving as a grab handle.

6.2 Developer Testing

We asked two respondents for developer testing. Both of them were males and students at the Faculty of Informatics, Masaryk University. Their goal was to try to install *BRT* to their own projects, following the tutorial in the README file. After successful installation, they should have integrated it either with GitLab or GitHub issue-tracking subsystem and tested whether it is working. In the end, they were asked to fill out the form with questions regarding the overall installation experience.

They both used a NPM library installation option to install the *BRT*, but only one of them was successful. The respondent who was unable to install it used React version 16 in his application, which turned out to be incompatible with the *BRT*, which uses React version 18. Even though he was not able to install the *BRT*, he provided valuable feedback regarding the README file in the repository and revealed that some dependencies of *BRT* should be changed to developer dependencies, as developers utilizing *BRT* do not need them, and they are used only during the build process. He suggested that README should explain the meaning of constant `GITHUB_REPOSITORY_NAME` more as it is unclear whether it should also contain the prefix with the owner of the repository or not.

The respondent, who successfully installed *BRT*, was using React version 18 in his application. For him, the README instructions were straightforward. However, he also encountered problems with installing the library. He was not authenticated to GitHub in CLI in the beginning. Hence he had to generate a new GitHub access token for reading packages to be able to install the *BRT*. It turns out he had to set the access token inside the NPM config, together with the Github registry, as *BRT* is stored in the Github registry, not the default NPM registry. After he did so, he finally successfully installed *BRT*. He was

using IntelliJ IDEA for development, and when he tried to automatically import the main component – `BugReportingTool`, the wrong path was selected, and hence it did not work. After fixing the import path, the *BRT* started working as expected. Thanks to his feedback, we understood what obstacles developers may encounter, and hence we are able to improve the installation tutorial in the README, to cover all blind spots in the installation process. He tested the *BRT* integrating GitHub, as well as, GitLab, and both worked as expected. During the testing process, he encountered one of the issues that user testers faced: the unintentional drag-and-drop mechanism activation when trying to annotate over a selected area. We also received positive feedback in the final form question, in his words: *I consider it to be a very promising and useful tool that might find a lot of applications in modern software*.

After receiving the feedback, we solved the main issues the respondents found during the installation process by enriching the README in the *BRT* repository. Namely, we described the meaning of all constants, even proposed `GITHUB_REPOSITORY_NAME`. We also mentioned that *BRT* is compatible with React version 18 or higher. There were reported obstacles with the installation process because of missing Github registry configuration and required Github authentication. Therefore we added steps on how to add the GitHub registry and GitHub Packages token to the NPM configuration. To resolve all issues, we also added the exact import statement to the React usage examples to prevent wrong import path issues in the future. We believe the installation process will now be more straightforward and flowing.

7 Conclusion and Future Work

One of this thesis's goals was to survey existing solutions for user-friendly bug-reporting approaches on websites through add-on web components that allow appending annotated images and screenshots to the textual report. The second goal was to design and implement our own solution in the form of a web component based on the analysis of the survey. The implementation focused on user-friendliness, ease of integration with the GitLab platform, and future extendability with new features and other issue-tracking platforms. The resulting component was published as an open source under an LGPL-3.0 license in the NPM. The last goal was to include user testing to detect bugs and implementation issues. The implemented *Bug Reporting Tool* (*BRT*) component was evaluated from two perspectives: via user testing and by developers. The former evaluation aimed at the identification of issues and bugs in the implementation and assessment of the intuitiveness, usability and suitability of the particular annotation tools. The latter provided insights into frequently omitted developers' viewpoints while using third-party libraries in their projects. Both user and developer testing brought interesting suggestions, and the feedback was also very positive.

The output of the thesis is a novel, fully functional open-source web component with all properties described in the first paragraph. In the implemented web component, there is still room for improvement. The installation process for developers might be simplified by providing a more advanced tutorial, covering and answering all questions that might arise during the installation. The user experience for users working with *BRT* may be improved by removing all issues found during the user testing process and also by implementing all reasonable feature suggestions. The interestingness of *BRT* and demand for *BRT* might be increased by adding more platforms that can integrate with this component. However, thanks to easy GitLab integration, this component can be helpful also for other students of Masaryk University for their school projects, as well as for the broad open-source community versioning their projects on GitLab.

Bibliography

1. ATLASSIAN CORPORATION. *Jira | Issue & Project Tracking Software | Atlassian* [online]. [visited on 2023-05-13]. Available from: https://www.atlassian.com/software/jira.
2. HUBSPOT, INC. *Issue Tracking Software | HubSpot* [online]. [visited on 2023-05-13]. Available from: https://www.hubspot.com/products/service/issue-tracking.
3. NULAB INC. *Project Management Software for Virtual Teams | Backlog | Nulab* [online]. [visited on 2023-05-13]. Available from: https://nulab.com/backlog/.
4. BHATTACHARYA, Sweta; SUJA, Radha; JAT, DS. Comparative analysis of bug tracking tools. *International Journal of Pharmacy and Technology*. 2016, vol. 8, pp. 4989–4998. ISSN 0975-766X. Available also from: https://www.researchgate.net/profile/Suja-Radha/publication/316888056_Comparative_analysis_of_bug_tracking_tools/links/5a1648720f7e9bc6481c8afa/Comparative-analysis-of-bug-tracking-tools.pdf.
5. MARKER.IO. *Marker.io: Visual Website Feedback & Bug Reporting tool* [online]. [visited on 2023-05-07]. Available from: https://marker.io/.
6. BUGHERD. *Website Annotation Tool | BugHerd* [online]. [visited on 2023-05-07]. Available from: https://bugherd.com/website-annotation-tool.
7. YBUG S.R.O. *Visual bug reporting and feedback tool – Ybug.io* [online]. [visited on 2023-05-07]. Available from: https://ybug.io/.
8. USERSNAP. *Usersnap | Your #1 Customer Feedback Software* [online]. [visited on 2023-05-07]. Available from: https://usersnap.com/.
9. HAIJE, Erin Gilliam. *Top 21 Visual Feedback Tools – an overview* [online]. 2022. [visited on 2023-05-07]. Available from: https://mopinion.com/top-21-visual-feedback-tools-an-overview.

10. HOTJAR LTD. *Hotjar: Website Heatmaps & Behavior Analytics Tools* [online]. [visited on 2023-05-07]. Available from: https://www.hotjar.com/.

11. VON HERTZEN, Niklas. *feedback.js – Feedback form with screenshot* [online]. [visited on 2023-05-07]. Available from: https://github.com/niklasvh/feedback.js.

12. PASTEL LABS INC. *Pastel | Fastest visual website feedback tool for web designers, developers and agencies* [online]. [visited on 2023-05-07]. Available from: https://usepastel.com/.

13. DGRIGG DEVELOPMENT INC. *Pageproofer | The easy way to leave visual feedback on a website!* [online]. [visited on 2023-05-07]. Available from: https://pageproofer.com/.

14. ZIPBOARD. *Digital Content Review and Approval Solution | zipBoard* [online]. [visited on 2023-05-07]. Available from: https://zipboard.co/.

15. USERBACK PTY LTD. *Home | Userback* [online]. [visited on 2023-05-07]. Available from: https://www.userback.io/.

16. LOGICINFLUX LTD. *FeedbackRig – Visual Feedback for Your Website and web apps.* [online]. [visited on 2023-05-07]. Available from: https://www.feedbackrig.com/visual-feedback.html.

17. AUDIENCEPROJECT. *UserReport: Simple user engagement tools, that help you improve* [online]. [visited on 2023-05-07]. Available from: https://www.userreport.com/.

18. TYPEFORM SL. *Typeform: People-Friendly Forms and Surveys* [online]. [visited on 2023-05-07]. Available from: https://www.typeform.com/.

19. PROPROFS. *Survey Maker: Online Survey Software for Free* [online]. [visited on 2023-05-07]. Available from: https://www.proprofssurvey.com/.

20. FEEDBACKIFY LIMITED. *Feedbackify!* [online]. [visited on 2023-05-07]. Available from: https://www.feedbackify.com/.

21. 3CG ONSITE LLC. *UseResponse – Customer Service Software* [online]. [visited on 2023-05-07]. Available from: https://www.useresponse.com/.

22. GITHUB, INC. *GitHub Docs* [online]. [visited on 2023-05-07]. Available from: `https://docs.github.com`.

23. GITLAB, B. V. *GitLab Documentation* [online]. [visited on 2023-05-07]. Available from: `https://docs.gitlab.com/`.

24. ATLASSIAN CORPORATION. *Manage Your Team's Projects From Anywhere | Trello* [online]. [visited on 2023-05-13]. Available from: `https://trello.com/`.

25. BERTOLI, Michele. Scalable Vector Graphics. In: BERTOLI, Michele. *React Design Patterns and Best Practices: Build Modular Applications That Are Easy to Scale Using the Most Powerful Components and Design Patterns That React Can Offer You Right Now*. 1st ed. Birmingham, UK: Packt Publishing, 2017, chap. 6, pp. 147–149. ISBN 978-1786464538.

26. BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. Understanding TypeScript. In: JONES, Richard (ed.). *ECOOP 2014 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN 978-3-662-44202-9.

27. BERTOLI, Michele. ESLint. In: BERTOLI, Michele. *React Design Patterns and Best Practices: Build Modular Applications That Are Easy to Scale Using the Most Powerful Components and Design Patterns That React Can Offer You Right Now*. 1st ed. Birmingham, UK: Packt Publishing, 2017, chap. 2, pp. 38–45. ISBN 978-1786464538.

28. BERTOLI, Michele. *React Design Patterns and Best Practices: Build Modular Applications That Are Easy to Scale Using the Most Powerful Components and Design Patterns That React Can Offer You Right Now*. 1st ed. Birmingham, UK: Packt Publishing, 2017. ISBN 978-1786464538.

29. PRETTIER CONTRIBUTORS. *Prettier · Opinionated Code Formatter* [online]. [visited on 2023-05-15]. Available from: `https://prettier.io/`.

30. ROLLUP CONTRIBUTORS. *Rollup | Rollup* [online]. [visited on 2023-05-13]. Available from: `https://rollupjs.org/`.

31. GITHUB, INC. *npm Docs* [online]. [visited on 2023-05-07]. Available from: `https://docs.npmjs.com/`.

32. SHARMA, Rohit. *Class Components and Functional Components in Reactjs* [online]. 2018. [visited on 2023-05-07]. Available from: https://www.letsreact.org/class-components-and-functional-components-in-reactjs/.

33. GITHUB, INC. *Issues API | GitLab* [online]. [visited on 2023-05-13]. Available from: https://docs.gitlab.com/ee/api/issues.html#new-issue.

34. GITLAB, B.V. *Issues API | GitLab* [online]. [visited on 2023-05-13]. Available from: https://docs.gitlab.com/ee/api/projects.html#upload-a-file.

35. GITHUB, INC. *Issues – GitHub Docs* [online]. [visited on 2023-05-13]. Available from: https://docs.github.com/en/rest/issues/issues?apiVersion=2022-11-28#create-an-issue.

36. SPIESS, Philipp. *How to Build a Freehand Drawing Using React* [online]. 2017. [visited on 2023-05-07]. Available from: https://pspdfkit.com/blog/2017/how-to-build-free-hand-drawing-using-react/.

37. GITHUB, INC. *Releases – GitHub Docs* [online]. [visited on 2023-05-13]. Available from: https://docs.github.com/en/rest/releases/releases?apiVersion=2022-11-28#create-a-release.

38. GITHUB, INC. *Release Assets – GitHub Docs* [online]. [visited on 2023-05-13]. Available from: https://docs.github.com/en/rest/releases/assets?apiVersion=2022-11-28#upload-a-release-asset.

A Supplementary Materials

The thesis archive in the Information System of Masaryk University (IS MU) contains:

- **source-code.zip** – a complete source code of the implementation part of this thesis,
- **bug_reporting_tools_survey.xlsx** – a table containing all research data of existing solutions for bug-reporting.

Additionally, the most recent version of the code is available at `https://github.com/xsvetli1/bug-reporting-tool` and the package is available at `https://github.com/xsvetli1/bug-reporting-tool/pkgs/npm/bug-reporting-tool`.