Debugging Support for Reactive Programming with RxJS

Manuel Alabor

A summative thesis presented for the degree of
Master of Science FHO in Engineering

OST
Eastern Switzerland
University of Applied Sciences

FHO
Fachhochschule Ostschweiz

Supervisor	Prof. Dr. Markus Stolze
External Examiner	Johannes Rieken

Computer Science
Eastern Switzerland University of Applied Sciences
Switzerland
15 January 2022

v1.0.1

Declaration of Authorship

I, Manuel Alabor, declare that this thesis titled, "Debugging Support for Reactive Programming with RxJS" and the work presented in it are my own work. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the Eastern Switzerland University of Applied Sciences.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at the Eastern Switzerland University of Applied Sciences or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have clarified what others did and what I have contributed myself.

Manuel Alabor, 15 January 2022

Abstract

Software engineers use numerous software tools in their daily working routine. These tools help them to streamline complex and repetitive tasks. Integrated development environments bundle such utilities ready-to-hand. This way, engineers benefit from a seamless developer experience where every tool *feels* and *looks* like a part of its host application and is only a keypress away. Of course, debuggers are a vital component of this toolset.

Debuggers built into contemporary development environments are tailored to work best with programs following an imperative programming style. However, when used with different programming paradigms, such as reactive programming, these tools do not adequately assist the engineers. This is why software engineers resort to more simple debugging techniques like manual print statements instead.

This summative thesis documents the debugging techniques engineers employ to debug programs implemented using RxJS, a popular library providing reactive programming functionality for JavaScript. First, it reveals why engineers abstain from using specialized reactive debugging tools by identifying a critical success factor for such utilities: A reactive debugger must be ready-to-hand, integrating with the engineers' overall developer experience. Subsequently, the thesis illustrates the iterative research and development process of a ready-to-hand reactive debugger for Microsoft Visual Studio Code. "RxJS Debugging for Visual Studio Code" provides with *Operator Log Points* a novel reactive debugging utility. To my knowledge, this is the first reactive debugger that allows engineers to inspect RxJS applications' runtime behavior without leaving their development environment or adding manual print statements.

Acknowledgments

First and foremost, I would like to thank my supervisor Prof. Dr. Markus Stolze for many hours of inspiring discussions, his sincere feedback, and his constant support for my endeavors leading to this thesis. Thank you for sparking my enthusiasm for empirical research.

My further gratitude goes to all people who helped me during my research in any way. Thank you to all study participants, proofreaders, and everyone else I might have forgotten about.

Thank you to my wife and my daughter. You make all the things count.

Contents

Declaration of Authorship **i**

Abstract **ii**

Acknowledgments **iii**

1 Introduction **1**
- 1.1 Relevant Programming Paradigms . . . 2
- 1.2 Reactive Programming with RxJS . . . 3
- 1.3 Debugging Challenges of Reactive Programming . . . 4

2 Related Work **6**
- 2.1 Reactive Debugging . . . 6
- 2.2 Debugging as a Process . . . 6
- 2.3 Developer Experience . . . 6

3 Research Process **7**
- 3.1 Exploration . . . 7
- 3.2 Proof Of Concept . . . 8
- 3.3 Prototype . . . 9
  - 3.3.1 Communication with Node.js . . . 10
  - 3.3.2 Moderated Remote Usability Test . . . 11
- 3.4 Distribution . . . 11
  - 3.4.1 Community Reception . . . 11
  - 3.4.2 ISSTA '22 Research Paper . . . 12

4 Future Work **12**
- 4.1 Empirical Software Engineering . . . 12
  - 4.1.1 Open Science . . . 13
- 4.2 Open Source . . . 13
  - 4.2.1 User Onboarding after Installation (Issue #58) . . . 13
  - 4.2.2 Log Point History (Issue #44) . . . 13

5 Conclusion **14**

A Research Papers **16**
- A.1 Debugging of RxJS-Based Applications . . . 16
- A.2 Debugging Support for Reactive Programming . . . 27
  - A.2.1 Paper . . . 27
  - A.2.2 Supplementary Material . . . 36

B Comparative User Journey **54**

C RxJS Debugging for vscode **62**
- C.1 Major Release Milestone Plan . . . 62
- C.2 Feature Backlog . . . 64
- C.3 Release Tweet Stats . . . 66
- C.4 Visual Studio Marketplace . . . 68
- C.5 ANALYTICS.md . . . 71

C.6 CONTRIBUTING.md 74
C.7 CODE_OF_CONDUCT.md 76
C.8 ARCHITECTURE.md 78
C.9 CHANGELOG.md 81
C.10 Analytics Dashboard 83

D vscode-js-debug Pull Request: Reuse CDP Connection 85

E Marble Diagram Syntax 88

F Open Science 90

List of Figures 92

List of Listings 92

List of Tables 92

References 93

1 Introduction

Debugging is an essential part of a software engineer's daily job. Various techniques, some better suited for the task than others, help engineers explore the functionality of an unknown or malfunctioning program. Rather traditional debugging is done by interpreting memory dumps or the analysis of log entries. Sophisticated debugging solutions hook into a program at runtime and allow more detailed inspection and control [1,13].

Imperative programming languages like Java, C#, and Python dominated the mainstream software engineering industry over the last decades [6,16]. Because of the prevalence of imperative programming languages, integrated development environments like Eclipse, the JetBrains platform, or Microsoft Visual Studio provide specialized debugging utilities specifically tailored to imperative programming languages. This results in an excellent, fully integrated developer experience, where tool-supported debugging is only a keypress away.

The developer experience degrades rapidly when software engineers use programming languages and tools based on different programming paradigms such as reactive programming. Because of this, engineers tend to use simpler, less adept debugging techniques instead.

During my master studies research, I examined the necessity of paradigm-specific debugging utilities when software engineers debug programs based on RxJS[1], a library for reactive programming in JavaScript. During my research, I explored how professionals debug RxJS programs, what tools and techniques they employ, and why they prefer to use print statements instead of specialized debugging utilities requiring them to switch contexts. In doing so, I identified a key factor for the success of a debugging tool: It needs to be *ready-to-hand*, or its users will not use it at all.

Based on the premise of *readiness-to-hand*, I designed and implemented a novel debugging utility for reactive programming. *Operator Log Points* are available as an extension for Microsoft Visual Studio Code and provide the, to my knowledge, first fully integrated debugging utility for RxJS. Using Human-Computer Interaction methods, I examined the developer experience of operator log points. I successfully verified that the new utility replaces manual print statements and does not require engineers to change context. Thereby I proof that a ready-to-hand debugger for reactive programming is feasible.

This summative thesis contextualizes my research results documented in two research papers. I will complete this introduction with an overview of relevant programming paradigms, reactive programming with RxJS, and the challenges reactive programming provides for debuggers tailored to imperative programming. Relevant work will be discussed in Section 2, followed by a synopsis of the complete research process and its results in Section 3. Section 4 presents a list of opportunities for future work and highlights provisions taken to ensure the sustainability of the demonstrated results. Before the reader is left with the study of two research papers in the Appendix A, I will wrap up on the topic of debugging support for reactive programming with RxJS in Section 5.

[1]https://rxjs.dev/

1.1 Relevant Programming Paradigms

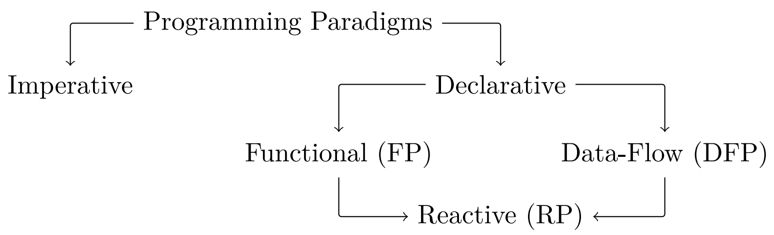


Figure 1: Taxonomy of relevant programming paradigms

On the way of producing the output for a given input, an imperatively implemented program keeps intermediate and final computational results in its state. The key concept of imperative programming languages like Java and C# is the assignment command. The assignment command modifies the programs state by changing the value assigned to a variable. Execution flow control commands, e.g., `if` and `while`, allow conditional and repeated execution of commands [26].

With a declarative programming language, computational results are carried explicitly from one program unit to the next instead of keeping them in extraneous state [11]. The source code of a declaratively implemented program is the blueprint of *what* the program is expected to accomplish eventually. In contrast, its imperative sibling resembles the precise step-by-step instruction on *how* the expected result must be achieved.

The Functional (FP) as well as the Data-Flow Programming (DFP) paradigm belongs to the family of declarative languages.

FP languages (e.g., Haskell and Erlang) are based on the concept of function and expression evaluation: Flow control statements are replaced with recursive function calls and conditional expressions [11,26]. Thus, a program's outcome results from its complete evaluation rather than its implicit state. With DFP, programs are modeled as directed graphs where a node represents an instruction of the program. The graph's edges describe how the data flows between its nodes [12]. Examples for DFP can be found in visual programming environments like Node-RED[2].

Reactive Programming (RP) combines FP and DFP. With RP, software engineers describe time-changing values and how they depend on each other using a Domain Specific Language (DSL) [23]. By doing so, they model a data-flow graph. A runtime environment interprets this graph and establishes a deterministic system state by executing necessary (re-)computations [2,3]. RP is usually not part of programming languages themselves. Instead, libraries and language extensions (e.g., Reactive for Haskell and REScala for Scala) provide RP features to their respective host programming language [7,22].

[2]https://nodered.org/

1.2 Reactive Programming with RxJS

RxJS provides RP features for JavaScript and TypeScript. It is an implementation of the ReactiveX API specification, where the *Observable*, "[..] a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming" [20], is the core concept.

Event Subscriptions

With the Observer pattern [8], observers subscribe to the notifications of a subject. Subscribers of an RxJS observable subscribe to the events of an observable likewise. Observables produce the following three event types:

1. `next` events carry produced values, e.g., the result of an HTTP request
2. `complete` events indicate that the observable finished its work and will not emit any further events in the future
3. `error` events notify subscribers about an error that occurred and that the observable will not emit any more events

Observables are push-based; they actively call the callback handler of their subscribers[3].

Operator Functions

An operator function subscribes to a source observable, modifies its events, and projects them to a target observable. Operator functions are the most powerful, yet most complex tool when working with observables. Listing 1 demonstrates how two simple operators filter and map values provided by a source observable. More complex operators allow for sophisticated constructions: E.g., `mergeMap`[4] composes higher-order observables to a new observable, or `retryWhen`[5] recovers an observable after it emitted an `error` event.

Listing 1 An observable emitting integers 1...8. Two operators process the integers before they are handed to the subscriber, which prints them to the console.

```
import { of, map, filter } from 'rxjs'

of(1, 2, 3, 4, 5, 6, 7, 8).pipe(
  filter(i => i % 2 === 0),        // Skip odd Integers
  map(i => i + 1),                 // Add 1 to Integer
).subscribe(i => console.log(i)); // Logs: 3, 5, 7, 9
```

Visualizing Observables with Marble Diagrams

Marble diagrams visualize observables graphically. Such diagrams help to understand the runtime behavior of an observable and its operators. Thus they are extensively used in the RxJS documentation.

[3]The Iterator pattern is pull-based, thus a counterexample to the push-based observable: The consumer has to actively poll (i.e., *pull*) the iterators `next` function to fetch a value [8].
[4]https://rxjs.dev/api/operators/mergeMap
[5]https://rxjs.dev/api/operators/retryWhen

Figure 2 shows the marble diagram for the observable implemented in Listing 1. Please refer to Appendix E for an in-depth look at the marble diagram syntax.

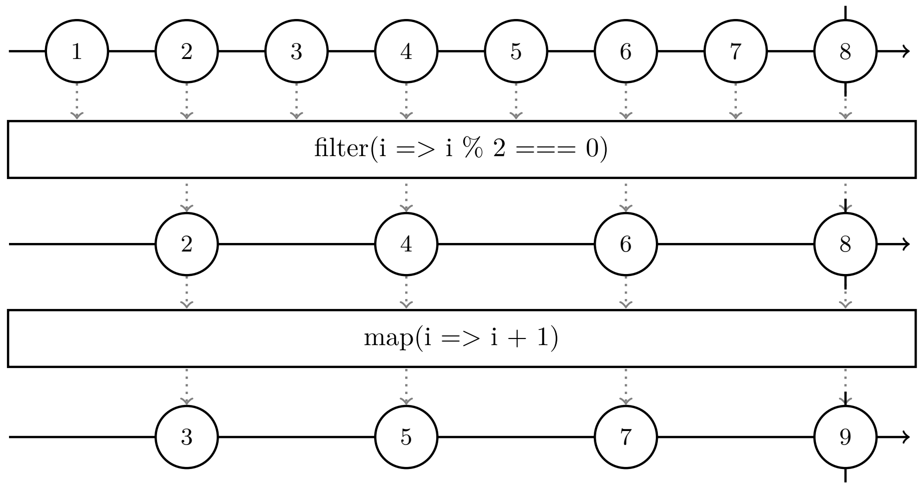


Figure 2: The marble diagram for Listing 1 shows three observables from top to bottom: The input observable emitting integers from 1 to 8, the intermediate result observable of the `filter` operator emitting only even integers, and the output observable emitting the even integers "plus 1". The two operators are shown in between the observables. The vertical line through the last marble of the input observable indicates that the input observable completed. Both operators forwarded this event to the output accordingly.

1.3 Debugging Challenges of Reactive Programming

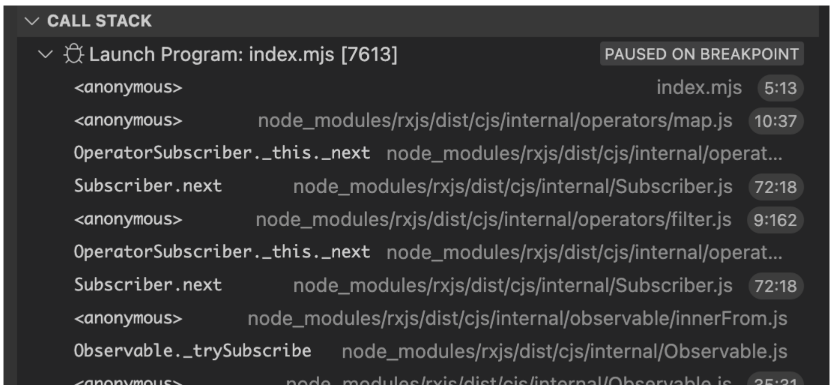
Listing 2 shows a reimplementation of Listing 1 using an imperative programming style. Software engineers use the debuggers built-in to their Integrated Development Environments (IDE) to follow the program's execution path. They pause the program's execution at a specific point of interest using breakpoints. Every time the debugger pauses program execution, the stack frame inspector provides details on what function calls lead to the execution of the current stack frame. Further, the values of all variables belonging to a stack frame are shown. Using the step controls, the engineers handle further program execution manually or resume "normal" execution eventually.

Listing 2 JavaScript program replicating Listing 1 using an imperative programming style.

```
for (let i = 1; i < 9; i++) {
  if (i % 2 === 0) {
    console.log(i + 1); // Logs: 3, 5, 7, 9
  }
}
```

Software engineers use the same imperative debugging techniques to debug RP programs like the one shown before in Listing 1: E.g., they add a breakpoint to the anonymous function passed to the `map` operator on Line 5 and run the program.

Again, the debugger provides a stack trace once program execution halts. Figure 3 depicts the debugger's major shortcoming when used with an RP program: The stack trace does not match the model of the data-flow graph described with the DSL. Instead, it reveals the inner, imperative implementation of RxJS' RP runtime. The debugger's step controls are ineffective since they operate on the imperative level as well. In this example, stepping to the following statement does not result in the debugger halting at Line 6. Instead, it leads the engineer to the internal implementation of RxJS.



```
CALL STACK
  Launch Program: index.mjs [7613]                        PAUSED ON BREAKPOINT
    <anonymous>                                                  index.mjs  5:13
    <anonymous>          node_modules/rxjs/dist/cjs/internal/operators/map.js  10:37
    OperatorSubscriber._this._next  node_modules/rxjs/dist/cjs/internal/operat...
    Subscriber.next          node_modules/rxjs/dist/cjs/internal/Subscriber.js  72:18
    <anonymous>        node_modules/rxjs/dist/cjs/internal/operators/filter.js  9:162
    OperatorSubscriber._this._next  node_modules/rxjs/dist/cjs/internal/operat...
    Subscriber.next          node_modules/rxjs/dist/cjs/internal/Subscriber.js  72:18
    <anonymous>        node_modules/rxjs/dist/cjs/internal/observable/innerFrom.js
    Observable._trySubscribe  node_modules/rxjs/dist/cjs/internal/Observable.js
    <anonymous>              node_modules/rxjs/dist/cjs/internal/Observable.js  35:31
```

Figure 3: The stack trace provided by the Microsoft Visual Studio Code debugger, after pausing program execution within the anonymous function on Line 5 in Listing 1.

A common practice to overcome this problem is the manual augmentation of the source code with print statements, as shown in Listing 3. This technique is often the last resort to debug RxJS programs. However, it is also regarded as a cumbersome and time consuming practice [2].

Listing 3 RxJS-based program from Listing 1 manually augmented with print statements.

```
import { of, filter, map, tap } from 'rxjs';

of(1, 2, 3, 4, 5, 6, 7, 8).pipe(
  tap(i => console.log(`A: ${i}`)), // <-- Added
  filter(i => i % 2 === 0),
  tap(i => console.log(`B: ${i}`)), // <-- Added
  map(i => i + 1),
  tap(i => console.log(`C: ${i}`))  // <-- Added
).subscribe(i => console.log(i));
```

2 Related Work

2.1 Reactive Debugging

The problem of imperative debuggers interpreting RP source code using the wrong model was subject to research efforts before. Salvaneschi et al. [24] coined the term *Reactive Debugging* and described a debugging utility specifically tailored to work with RP programs for the first time in their work. They provided the first implementation of such a debugger named *Reactive Inspector* for REScala.

Banken et al. [4] transferred former findings to RxJS. *RxFiddle* is a browser-based visualizer that takes a piece of isolated RxJS source code and displays its runtime behavior in two dimensions: A flow-graph shows all observables that get created and how they depend on each other. Additionally, the utility uses marble diagrams to show what events get emitted by an observable over time.

2.2 Debugging as a Process

Layman et al. [13] looked into how engineers debug programs. They formalized an iterative process model for the activity of debugging. During this process, engineers define and refine a hypothesis on the cause that triggered an unexpected behavior in a program. Ultimately, the process tries to validate that hypothesis. The debugging process after Layman et al. consists of three steps: Engineers start to (i) collect context information on the current situation (e.g., which particular program statements might be involved or what input caused the failure). This information then allows the software engineers to formulate a hypothesis on how the failure situation might be resolved. Next, they (ii) instrument the program, e.g., by adding breakpoints or modifying source code. They then (iii) test the instrumented program to validate their hypothesis. Step iii either proves their hypothesis correct, ending the debugging process, or yields new information for another iteration of hypothesis refinement and testing.

2.3 Developer Experience

Human-Computer Interaction (HCI) is the scientific examination of the interface between people and computers. HCI employs empirical research methods to review and verify the design of such interfaces [14].

While HCI is an academic research discipline, User Experience (UX) design is a more practice-oriented subject and draws from many tools originated in HCI. The International Organization for Standardization (ISO) defines UX as “a person’s perceptions and responses that result from the use or anticipated use of a product, system or service” [25].

Developer Experience (DX) is the umbrella term for the application of UX design principles and methodologies like User-Centered Design (UCD) [10] specifically for developers and software engineers [9,17].

3 Research Process

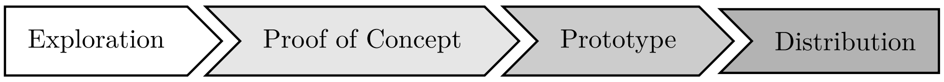


Figure 4: Research process in four phases

The research process is structured in four phases: (i) Exploration, (ii) Proof of Concept (PoC), (iii) Prototype, and (iv) Distribution. Methods originated in the fields of empirical software engineering [28] and HCI helped to verify the main artifacts listed in Table 1. The following four subsections highlight the most important results and deliveries of each project stage.

Table 1: Overview of all artifacts delivered per process phase.

Phase	**Artifact**
Exploration	Research Paper (Appendix A.1)
Proof of Concept	Cognitive Walkthrough (Appendix A.2.2) Comparative User Journey (Appendix B) PoC for RxJS Debugging Utility
Prototype	Usability Test Report (Appendix A.2.2) Minor Release "RxJS Debugging for vscode"
Distribution	Research Paper (Appendix A.2) Major Release "RxJS Debugging for vscode"

3.1 Exploration

I started with an analysis of what debugging tools and techniques software engineers use in their daily jobs. Data from five informal interviews and five written "war story" reports allowed me to build a first intuition in these regards. To verify the collected data, I set up a remote observational study with four subjects. In the study, two malfunctioning RxJS programs were presented and the subjects were asked to locate and fix the problems in the applications source code. To do so, they should use the debugging utilities they would use in their daily jobs as well. Figure 5 summarizes the results. All subjects used manual code modifications (i.e., print statements) to understand the behavior of the presented problems. Over half of them tried to use the imperative debugger of their IDE. The most pivotal insight was that, even though two subjects stated to know about specialized RxJS debugging tools, none of them used such during the study.

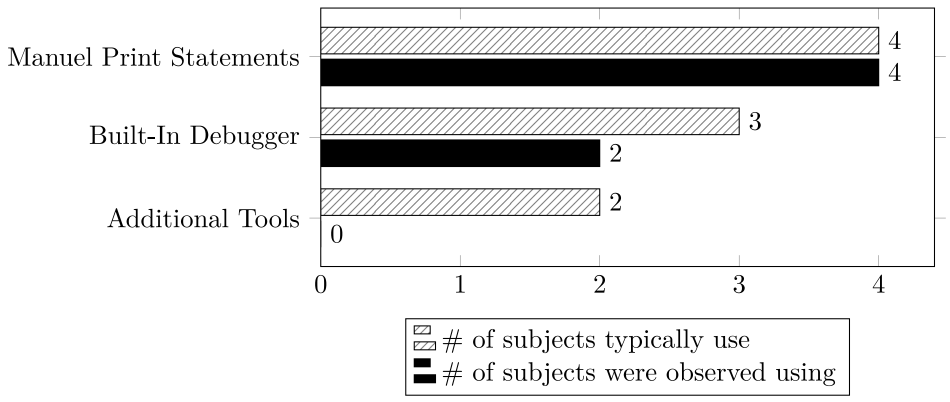


Figure 5: Comparison of what debugging techniques subjects stated to usually use, and what they were observed actually using during the observational study.

The results of the interviews, the analysis of the war story reports, and the interpretation of the observed behaviors during the observational study lead to the following two key take-aways:

1. The most significant challenge software engineers face when debugging RxJS-based programs is to know *when* they should apply *what* tool to resolve a problem the *best* way
2. Since engineers abstained from using specific RxJS debuggers, how can such utilities be provided without requiring them to switch context, thus be ready-to-hand?

I documented these results in the research paper "Debugging of RxJS-Based Applications" together with Markus Stolze [2]. The paper was published with the proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS' 20), where I also presented my findings. Furthermore, the published paper is available in Appendix A.1.

3.2 Proof Of Concept

Based on the learnings from the first phase, I started to compile ideas to help software engineers debug RxJS programs. It was essential that a potential solution:

1. Integrates seamlessly with an IDE
2. Is ready-to-hand, i.e. requires minimal to no effort from its users to get started with debugging

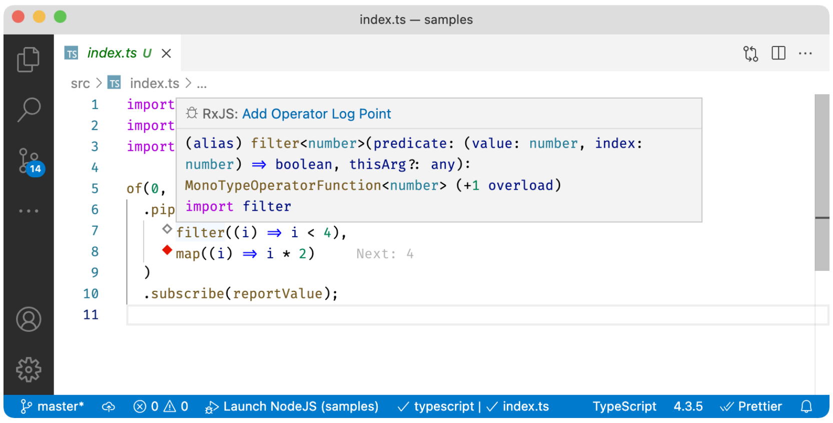
McDirmid [15] proposed with the concept of "probes" for live programming environments a way to trace variable values during runtime directly in the source code editor. Similarly, imperative debuggers provide log points, a special type of "breakpoint." Instead of halting the program, they print an arbitrary log entry to the debugging console. Using the debugging process by Layman et al. [13] as a mental model, I combined the two concepts and transferred them to the world of RP debugging: The *operator log point*[6] shows the events emitted by an operator during program execution in realtime.

After establishing the PoC for operator log points as an extension for Microsoft Visual Studio Code (vscode), I used the cognitive walkthrough method [14,27] to verify the utility. Its results, available as part of Appendix A.2.2, demonstrated that the proposed debugging utility replaces manual print statements in a scenario where engineers debug RxJS programs.

To convey the achieved improvement through operator log points effectively, I created a "comparative user journey." A basic user journey maps the touchpoints of a user with a product [21]. In my comparative journey, I correlate how a software engineer would solve a typical RxJS debugging task with an imperative debugger compared to an engineer having operator log points at their disposal. The result is available in Appendix B.

3.3 Prototype

Certain that operator log points satisfy all requirements defined in the previous stage of the process, I started with the actual implementation work for a production-ready vscode extension. Eventually, I released version 0.1.0 of "RxJS Debugging for vscode."



```
1  import
2  import
3  import
4
5  of(0,
6    .pip
7      filter((i) => i < 4),
8      map((i) => i * 2)       Next: 4
9    )
10   .subscribe(reportValue);
11
```

```
RxJS: Add Operator Log Point
(alias) filter<number>(predicate: (value: number, index:
number) => boolean, thisArg?: any):
MonoTypeOperatorFunction<number> (+1 overload)
import filter
```

Figure 6: A screenshot of the debugger extension prototype. Line 8 shows an enabled operator log point including a logged event. Operator log points are managed by hovering a log point suggestion with the cursor.

The prototype of the extension enabled engineers to debug RxJS-based applications running with Node.js. After they installed the extension, the debugger started to suggest operator log points with a small, diamond-shaped icon next to the respective operator. Next, the engineer launched their application using vscode's built-in JavaScript debugger. The RP debugger automatically augmented RxJS so it started to send event telemetry to vscode. The extension then

[6]Inspired by McDirmid [15], operator log points were called *probes* in the PoC and the early prototype of the extension. This name caused confusion with the test subjects in a later usability test. I renamed the utility based on the received feedback in turn.

displayed events (e.g., “Next: 4” in Figure 6 at the end of Line 8) for enabled operator log points in-line with the respective operator in the source code editor.

There were various challenges and tasks to solve during the Prototype phase. The following two sections present two highlights.

3.3.1 Communication with Node.js

One of the biggest challenges in implementing the prototype was to build a reliable way to communicate with RxJS running inside the Node.js process. I initially used a WebSocket to exchange messages with the JavaScript runtime. However, this proved to be prone to problems in numerous ways. E.g.:

- How should the extension discover the WebSocket server running in the other process?
- What if the network infrastructure prevents vscode from connecting to the WebSocket in the first place?
- Would a WebSocket-based solution work at all, when RxJS is running in a browser?

I decided to replace the WebSocket-based communication with something more suitable eventually.

With the intent to build a debugger that integrates with the IDE seamlessly, I looked into how vscode’s built-in JavaScript debugger, vscode-js-debug[7], communicates with the runtime environment. As it turned out, vscode-js-debug uses the Chrome DevTools Protocol[8] (CDP) to communicate with arbitrary JavaScript runtimes. Unfortunately, the debugger did not offer its CDP connection for reuse to other extensions. I reached out to the project maintainer and contributed this particular functionality as a new feature (Appendix D). By the time my contribution was released in April 2021, I had replaced the previous WebSocket-based communication channel with CDP. Furthermore, I had now a future-proof solution, which not only worked with RxJS running inside of a Node.js process, but also in any other JavaScript virtual machine that supports CDP (e.g., web browsers like Mozilla Firefox and Google Chrome).

Figure 7 depicts all relevant components involved in an RxJS RP debugging session. More details to the extensions architecture is available in Appendix C.8.

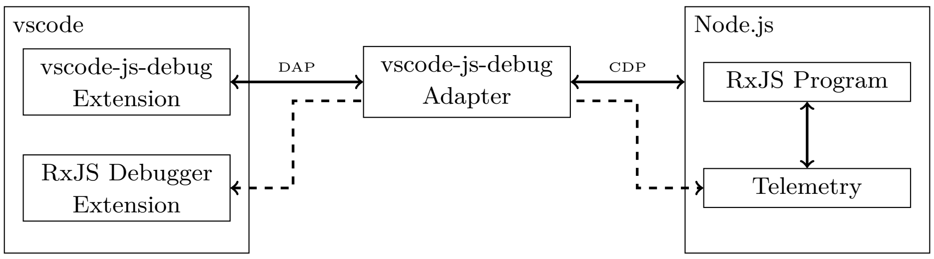


Figure 7: The *Telemetry* component instruments the *RxJS Program* (right). The *RxJS Debugger Extension* runs inside of the *vscode* process. The components communicate via a *CDP* channel established by *vscode-js-debug*.

[7] https://github.com/microsoft/vscode-js-debug
[8] https://chromedevtools.github.io/devtools-protocol/

3.3.2 Moderated Remote Usability Test

Once the main elements of the prototype were working sufficiently, I conducted a remote usability test [5,14,18,19] with three subjects. The goals of this study were:

1. To verify that operator log points can replace manual print statements in an actual programming scenario
2. To identify usability issues not detected during development
3. To collect unstructured feedback on prototype and gather ideas for its further development

Unfortunately, one subject could not get the extension prototype running on their machine. With the other two subjects left, I was able to verify the first two goals nonetheless. None of the participants used manual print statements during the usability test. Additionally, the evaluation of the test sessions revealed ten usability issues. Four of them prevailed for both subjects, hence I classified them as major. The complete list of identified usability issues is part of Appendix A.2.2.

I triaged the feedback from all three subjects and created items in the feature backlog for the upcoming Distribution phase accordingly. With this, the last goal was reached as well.

3.4 Distribution

The last process phase had two overarching goals:

1. To finalize the RP debugger prototype and release it to the community
2. To publish another research paper documenting the feasibility of a ready-to-hand, and fully IDE-integrated RP debugger

To get started, I defined the roadmap for the extensions 1.0.0 release, which is available in Appendix C.1. The following list contains three of its highlights:

- Support for the latest RxJS 7.x versions (only 6.6.7 was supported with the prototype)
- Debugging of web applications bundled with Webpack (only the Node.js virtual machine was supported so far)
- Resolve the four major usability issues identified during the Prototype stage

Version 1.0.0 of “RxJS Debugging for vscode” was finally released on the 2nd of December 2021 and was followed by three minor bugfix releases within six days.

3.4.1 Community Reception

On the day of release, I announced the debugger extension via its own Twitter account @rxjsdebugging. Until the 30th of December 2021, the tweet reached 77k impressions (Appendix C.3).

Further, the extension was downloaded 954 times (Appendix C.4), counted 51 unique users (Appendices C.5, C.10), and was featured in a live stream on Twitch[9].

Based on the results of the studies conducted before, I concluded that there was a real need for an integrated RP debugger for RxJS. The overall positive reception on RxJS Debugging for vscode was overwhelming nonetheless. However, the major release also revealed bugs and feature gaps in the extension. I resolved the most critical problems within a few days (see the changelog in Appendix C.9). In addition, I processed feedback using GitHub Discussions[10] and the feature backlog (Appendix C.2).

3.4.2 ISSTA '22 Research Paper

In contrast to the delivered practical effort, I wrote another research paper with Markus Stolze. The paper documents the latest advancements on the feasibility of an RP debugger that is ready-to-hand and fully integrated with an IDE. The latest version of this paper was submitted to the technical papers track of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis 2022 (ISSTA '22) at the time of publishing this thesis. The submitted paper includes revisions based on the feedback of a double-blind review with three reviewers and is available in Appendix A.2.

4 Future Work

4.1 Empirical Software Engineering

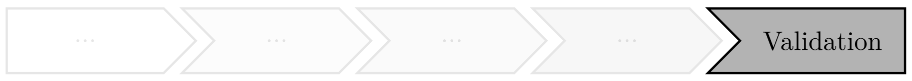


Figure 8: Empirical validation as the next step in a future follow-up.

RxJS Debugging for vscode provides a practical solution to the problems identified throughout the presented research process, and further empirical validation can now be carried out.

Operator log points were successfully tested using HCI methods during their development. However, a formal verification using empirical methods will yield useful insight into the presented debugging utility. The most important research question to answer in these regards is, how effectively operator log points can replace existing debugging tools (i.e., manual print statements and the built-in, imperative debugger tools).

With its major release, the debugging extension asks its users to opt-in for the anonymized collection of user behavior data. This data is available for further analysis as described in Appendix C.5. The accumulated data points allow

[9]David Müllerchen aka @webdave_de, a Google Developer Expert specialized on Angular development, hosted the live reaction stream on Twitch. Unfortunately, the recording of the stream is unavailable at this time.

[10]https://github.com/swissmanu/rxjs-debugging-for-vscode/discussions

conclusions on how software engineers use the extension. The data set might be evaluated on its own to derive improvements for the presented debugging utility or provide supportive arguments for a broader study as proposed above.

4.1.1 Open Science

All conducted studies (interviews, observational study, cognitive walkthrough and moderated remote usability test) and their results are documented in the respective research papers and their supplementary material available in Appendices A.1, A.2, B to encourage future research on RP debugging. In addition, a list of URLs leading to various GitHub repositories containing relevant artifacts and data sets is available in Appendix F.

4.2 Open Source

I developed the presented RxJS debugging extension with the intention to establish a sustainable open source project.

Three guides introduce new contributors to the project and to the extension's implementation and code organization details (Appendices C.6, C.7, C.8). The transparent project governance is built around the GitHub platform: The feature backlog and bug-tracking is based on GitHub Issues, Discussions help triage inquiries from users. Unit and integration tests, automatically executed using GitHub Actions, help keep the extension's main branch stable.

The feature backlog in Appendix C.2 contains ideas for practical-oriented future work. I present two features from this backlog in the following.

4.2.1 User Onboarding after Installation (Issue #58)

After an engineer installed the extension, they are left on their own to get started with debugging. Even though the readme file provides information to some extent, the onboarding experience for new users can be improved. With this feature, ways to enhance that experience should be explored and suitable measures be implemented eventually.

A contributor needs to understand the vscode extension API. However, profound knowledge of the extension's own source code is not required.

4.2.2 Log Point History (Issue #44)

Instead of showing only the latest emitted event from an enabled operator log point, the debugger should display all previously emitted events. This functionality would allow engineers to reconstruct the behavior of an operator without over and over replaying the failure scenario using the live system.

A contributor may start simply by implementing a list to display historical events in textual form. The list might then be gradually improved towards a graphical representation of the events using marble diagrams (Appendix E).

This feature requires a good understanding of the vscode extension API and in-depth knowledge of the debugging extensions codebase. However, all event data to populate a historical view is already present, and contributors can focus on implementing the best possible DX.

5 Conclusion

In this summative thesis, I presented the condensed results of my research on reactive debugging for programs based on RxJS, a popular library for reactive programming with JavaScript.

The results of interviews, war story reports, and an observational study revealed the major shortcoming of previously available RxJS debugging utilities. Even though software engineers might know them, they abstain from using them because they are not “ready-to-hand,” i.e., not integrated with the development environment they are working in and accustomed to. Instead, they use manual print statements.

With the concept of “readiness-to-hand” as a guiding light, I built a proof of concept implementation for a novel debugging utility to find relief from this problem: Operator log points debug RxJS operators without requiring the engineer to leave Microsoft Visual Studio Code. While refining the debugger iteratively, I employed a cognitive walkthrough, a comparative user journey, and a usability test at different stages of development to validate the utility’s capability of solving the problem of ready-to-hand reactive debugging.

I documented the results of my research together with Markus Stolze in two research papers: The first paper was published with the proceedings of the ACM REBLS ’20 workshop. The second report is in review for the technical papers track of the ACM ISSTA ’22 conference when publishing this thesis. Furthermore, I released “RxJS Debugging for Visual Studio Code,” the, to my knowledge, first RxJS-specific debugger that fully integrates with a development environment at the end of 2021.

By providing open access to all relevant material (studies, results, papers, source code, and project governance), academical- and practical-oriented future work is encouraged. To further facilitate potential contributions, I suggested concrete topics for researchers and engineers likewise.

A Research Papers

A.1 Debugging of RxJS-Based Applications

This paper was published with the proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '20), 16th November 2020.

Debugging of RxJS-Based Applications

Manuel Alabor
Eastern Switzerland University of Applied Sciences
Rapperswil, Switzerland
manuel.alabor@ost.ch

Markus Stolze
Eastern Switzerland University of Applied Sciences
Rapperswil, Switzerland
markus.stolze@ost.ch

Abstract

RxJS is a popular library to implement data-flow-oriented applications with JavaScript using reactive programming principles. This way of programming bears new challenges for traditional debuggers: Their focus on imperative programming limits their applicability to problems originated in the declarative programming paradigm. The goals of this paper are: (i) to understand how software engineers debug RxJS-based applications, what tools do they use, what techniques they apply; (ii) to understand what are the most prevalent challenges they face while doing so; and (iii) to provide a course of action to resolve these challenges in a future iteration on the topic. We learned about the debugging habits of ten professionals using interviews, and hands-on war story reports. Based on this data, we designed and executed an observational study with four subjects to verify that engineers predominantly augment source code with manual trace logs instead of using specialized debugging utilities. In the end, we identified the lack of fully integrated RxJS-specific debugging solutions in existing development environments as the most significant reason why engineers do not make use of such tools. We decided to elaborate on how to resolve this situation in our future work.

CCS Concepts: • **Software and its engineering**;

Keywords: reactive programming, debugging, empirical software engineering

ACM Reference Format:
Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-Based Applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '20), November 16, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427763.3428313

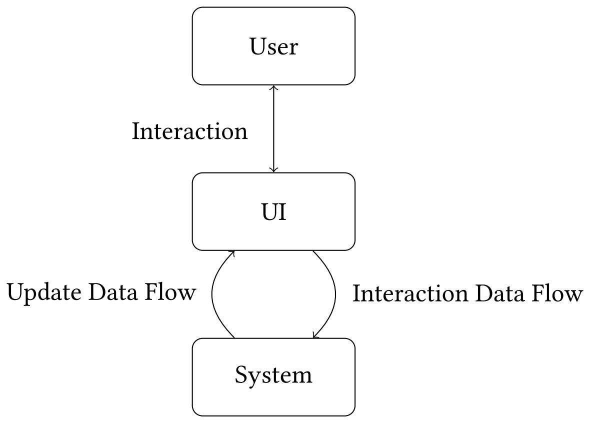


Figure 1. Basic data-flows in a UI Application.

1 Introduction

The (graphical) user interface (*UI* or *GUI*) of an application handles two constant flows of data: External user input (e.g. mouse, touch, or keyboard interaction) is interpreted and forwarded to the system. Once the system processed an interaction and updated its internal state accordingly, it notifies the UI about these changes, which are relayed to the user.

To implement the data-flows as shown in Figure 1 to drive a UI, the Observer design pattern[7] is often used and variations of the pattern are omnipresent today[4].

The Observer design pattern has its roots in the Object Oriented Programming paradigm (*OOP*), hence relies on imperative code constructs to handle a data-flow. Reactive Programming (*RP*) is another approach to realize such flows: It inherits the declarative way of implementing functionality from Functional Programming (*FP*), i.e., data-flows are described rather than implemented step by step[5]. RP functionality is usually available in the form of a library providing necessary abstractions, for imperative as well as declarative programming languages.

According to the IEEE Standard Glossary of Software Engineering, *debugging* is an activity "to detect, locate, and correct faults in a computer program."[1] From interpreting memory dumps, manually adding log statements to trace program execution up to the point where specialized debugging programs can interrupt a running process and interact with it on a low level, debugging utilities took different forms over time.

Modern IDEs and internet browsers ship with their own set of debugging tools. These debuggers are specialized in working with imperative, control-flow-oriented program code. The following example helps us to illustrate the implications of this: Assuming an engineer is inspecting a piece of code and wants to know which part of the program was executed right before. For a program implemented using the imperative paradigm, the call stack gives a clear answer to this question. Hence the stack frames represent each point in the program execution. In a data-flow-oriented program implemented using RP, the stack trace for a transformation function in the flow will not point to its logical predecessor. Instead, the stack frames lead to the internals of the RP runtime environment.

This example demonstrates the limits of a traditional control-flow oriented debugger, which cannot interpret RP abstractions. As a result, these debuggers are not able to give the correct answer to a data-flow-specific inquiry. There have been numerous efforts to provide engineers with improved debugging utilities for RP [17] [18] [6]. However, none of these have seen broad adoption by practitioners yet. To gain a better understanding of the underlying root causes, we conducted interviews with several software engineers and collected "war stories" about the challenges they face in their day-to-day jobs when using RP. Based on this collected evidence, we will validate their statements in an observational study using RxJS and search for an answer to our first research question:

- *RQ1: What challenges do software engineers face when debugging RxJS-based applications?*

In response to this, we are going to present a concept on how to resolve previously identified challenges and answer the second research question:

- *RQ2: How can the experience of software engineers during the debugging process of RxJS-based applications be improved?*

The implementation and validation of these proposals lead to our third and last research question, which will be investigated in our future research:

- *RQ3: What is the impact of proposed solutions on the debugging experience of software engineers?*

We will conclude this introductory section with the clarification of important terms and a view on known RP debugging utilities. Section 2 gives an overview of the insights from the conducted interviews and the collected war story reports. We present our observational study intended to validate results from the interviews and reports in Section 3, which allow us to answer RQ1. Before our final conclusion, we will answer RQ2 in Section 4 "Future Work" and review the threats to validity regarding our study in Section 5.

1.1 Reactive Programming

RP is a declarative programming paradigm that is strongly influenced by FP. While engineers use imperative programming languages to specify every step *how* a program has to do something, declarative languages allow to describe *what* the program should achieve ultimately. A runtime system then figures out a way to satisfy that description and executes it. RP functionality is usually provided in form of a language extension for a specific programming language (e.g. REScala for Scala[17]) or as a library (e.g. RxJS for JavaScript[15])

Either way, both usually provide a (i) domain specific language (*DSL*) to describe data-flow graphs, how they depend on each other and how data flowing through should be transformed. At program execution, a (ii) runtime environment evaluates these descriptions and creates a representation of the specified graphs. It then takes care that values are processed and propagated correctly through them as well as that a consistent system state[5] is always maintained.

1.2 ReactiveX and RxJS

"Reactive Extensions" (*ReactiveX*) is an open-source project. Its members and contributors created a generic description of a RP API. They further provide reference implementations of this API along with RP language extensions for various programming languages like Java, C#, or JavaScript[1]. ReactiveX summarizes the API as "…a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming"[14]. The core concept of the API specification is the Observable[2]: An observable can be composed with other observables to form a data-flow graph. Once an observable gets subscribed, it might push ("emit") an arbitrary number of values to the subscriber until it either completes, fails, or gets unsubscribed again. There is a multitude of operator functions available which allow the transformation of values and composition with other (higher-order[3]) observables. The mechanism of subscribing to an observable is closely related to the Observer design patterns `attach` method.

RxJS[15] is the reference implementation of the ReactiveX API specification for JavaScript. Its current major version 6 is implemented using TypeScript and is used by large projects like Angular[8]. Listing 1 shows an example of RP using RxJS in TypeScript.

The RxJS community uses *marble diagrams* as shown in Figure 2 to document [19] the runtime behavior of an observable visually. Unit test libraries[16] use this abstraction to encode the behavior of mocked observables or to describe assertions.

[1] http://reactivex.io/languages.html
[2] There is no known relation between ReactiveX' concept of the Observable and the deprecated Java class `java.util.Observable`.
[3] An observable emitted by another observable is considered a Higher-Order observable. This naming is related to the concept of higher-order functions in mathematics and computer science.

```
import { of } from 'rxjs';
import { filter , map } from 'rxjs/operators';

of(0, 1, 2, 3, 4).pipe(  // Create observable
   filter (i => i < 4),   // Omit integers >= 4
   map(i => i * 2)        // Multiply int. by 2
).subscribe(console.log) // Logs: 0, 2, 4, 6
```

Listing 1. Basic RxJS example creating an observable emitting four integers. Each integer is processed by two operators and finally written to the console.

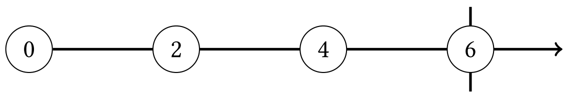


Figure 2. A Marble Diagram visualizing the observable in Listing 1. From left to right, each marble represents an emitted value. The vertical line at the last marble indicates that the observable completed after emitting 6.

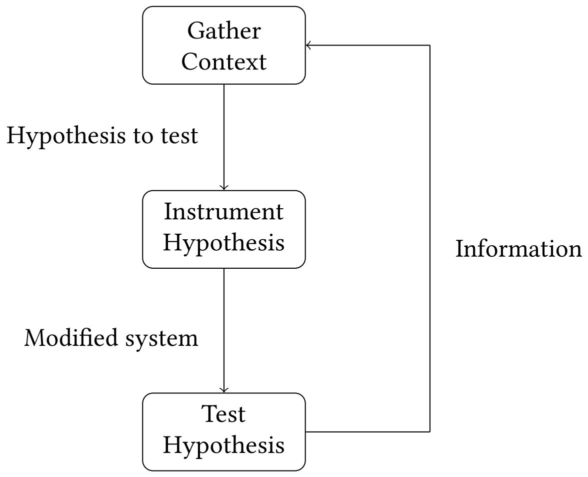


Figure 3. Debugging Process Model after Layman et al. [10]

1.3 Debugging Process Model

Layman et al. [10] use the debugging process model, an iterative hypothesis refinement process, to formalize the general activity of debugging a computer program in their paper.

The process consists of three steps and includes a feedback loop: After the engineer (i) gathered sufficient context information (e.g. ways to reproduce the failure or details about external factors) and understands the situation satisfactory, they generate a hypothesis on the origins of the bug or what impact a change made to the program might have. With the intent to proof their hypothesis, the engineer then (ii) instruments the defective program using suitable tools (e.g. adding log statements, setting breakpoints or removing code parts). Finally, the instrumented system gets (iii) challenged against the formed hypothesis. E.g., code statements are executed step by step using a debugger or trace logs are analyzed and compared against expected behavior. If the hypothesis turns out to be correct, the debugging process stops. If not, the newly gained knowledge about the problem is used to build a refined hypothesis and start a new iteration.

1.4 Debugger Concepts

Software engineers have many tools and utilities at hand, which help them to interact with and gain insight on the behavior of a defective program. Tools range from the instrumentation of source code with trace log statements (manually or automated) to specialized utilities allowing them to directly interact with a program at runtime.

Many of these specialized utilities differentiate themselves fundamentally in regards of the concepts they are built upon. We identified and will use the following three categories to structure them:

Traditional (i) *imperative-focused debuggers* provide the functionality to interact with programs at runtime: Once a breakpoint pauses the program execution, they provide access to the current call stack and the values assigned to variables of a given stack frame. Manual control of the program execution allows inspecting its behavior step by step as well as assigning new values to variables “on-the-fly.”

RP provides its own set of challenges to debuggers: Call stacks expose internal invocations of the RP runtime system rather than, e.g., the predecessor transformation according to the data-flow graph. Further, breakpoints can only be used on the imperative parts of transformations and lack the functionality of interrupting execution when the RP runtime hits a specific node within the graph. A (ii) *reactive debugger* can interpret the underlying graph model of a RP runtime system. It leverages on it and provides specialized tools e.g., to navigate, visualize or instrument the data-flow graph [18] [6] [3].

Traditional, as well as reactive debuggers work with the *current* state of a program's execution only. They lack information about what happened before or what is going to happen in the future. This shortcoming is tedious, especially when debugging a problem depending on many complex circumstances in a system. An (iii) *omniscient debugger* [13] [12] does not interact with the executed program directly. Instead, it records runtime telemetry and provides an interface for later inspection. Engineers can “time travel” back and forth through the program execution trace without the hassle of reconstructing a given failure situation over and over again.

1.5 RxJS Debugging Utilities

1.5.1 rxjs-spy. *rxjs-spy*[2] is a logging library specialized on RxJS observables. Once an observable is tagged with an arbitrary string identifier, a monitor can generate trace logs whenever a value is emitted as well as when individual life cycle events occur (i.e. subscribe, unsubscribe, complete and error). Ideally, tagged observables are created during development when the data-flows are composed for the first time. Like `tap`[4], the `tag` operator in Listing 2 is completely transparent to the actual data-flow.

```
import { create } from 'rxjs-spy';
import { tag } from 'rxjs-spy/operators';

const spy = create(); // Create monitor
spy.log(/multiply/);  // Log tags matching
                      // RegEx /multiply/
interval(1000).pipe(
  map(i => i * 2),
  tag('multiply'),    // Tag with "multiply"
  map(i => i - 1),
  tag('subtract')     // Tag with "subtract"
  take(2)
).subscribe();
```

Listing 2. Application of rxjs-spy using its `tag` operator on Line 9 and 11.

The data-flow configuration in Listing 2 will produce a trace log as shown in Listing 3 eventually.

```
Tag = multiply; notification = subscribe
Tag = multiply; notification = next; value = 0
Tag = multiply; notification = next; value = 2
Tag = multiply; notification = unsubscribe
```

Listing 3. rxjs-spy execution trace log generated by default monitor in Listing 2 on Line 4.

Additional features are available through the library's console interface. E.g., a tagged observable can be paused, so values get collected rather than being emitted immediately. The engineer can then emitted these values one after another manually or resume all of them at once.

1.5.2 rxfiddle. *rxfiddle*, as proposed by Banken et al.[6] is the first reference implementation of their RP debugger architecture for the ReactiveX API specification. They describe a software design consisting of two independent components: The (i) *host instrumentation* augments a ReactiveX API implementation to emit events at runtime (e.g., emitting a value or life cycle events) and forwards them to the second component. The (ii) *visualizer* interprets the events and displays them along two dimensions: The StoryFlow graph[11] shows when an observable is created and how it interacts with other observables, whereas a marble diagram visualizes the values emitted over time for every observable.

The reference implementation supports event processing for the (outdated) RxJS major versions 4 and 5 only and is available as an online application[5]. A proof-of-concept implementation working on a local computer is available through the projects Git repository[6].

1.5.3 RxViz. *RxViz* is a visualizer utility available online[7]. It is an "animated playground for Rx observables"[3] and allows the visualization of RxJS observables using marble diagrams. Engineers implement or copy-paste data-flows in an editor window using JavaScript. A diagram is generated based on this code over a configurable time interval. The diagrams are rendered immediately and are available as downloadable SVG files.

1.5.4 rxjs-playground. Building on the basic concept of marble diagrams, *rxjs-playground*[8] is a sophisticated sandbox to simulate and visualize RxJS observables interactively in the browser. Users define editable and computed observables, represented as vertical marble diagrams: Values emitted by an editable observable can be created and modified either by interacting with its marble diagram directly or using a simple JSON syntax. The behavior of a computed observable is controlled by implementing functionality with TypeScript in the provided editor.

rxjs-playground renders the values and life cycle events for all observables in real-time, allowing quick iterations on a specific piece of code.

2 Interviews and War Stories

On the way of finding our interview partners and war stories reporters, we noticed it to be a challenge to find people who understand themselves as users of RP and related technologies. E.g., even though Angular makes heavy use of RxJS, we will see that many engineers do not directly interact with its abstractions when building "basic" UIs. In the end, we were able to conduct interviews with five engineers and collect reports on hands-on experiences from another five.

2.1 Interviews

We organized informal interviews, which allowed us to gain insight into how software engineers work with RP in their daily jobs. We talked to five engineers (following identified using the codes *I1* through *I5*) and asked them about the

[4]RxJS' `tap` operator is used to execute a side effect whenever an observable emits a value. It cannot modify or influence the emitted value nor the observable in any way.

[5]https://rxfiddle.net/
[6]https://github.com/hermanbanken/RxFiddle
[7]https://rxviz.com/
[8]https://hediet.github.io/rxjs-playground

technologies they use, what their personal experience with RP was, what they most liked and most disliked about it. We used video chat to conduct all interviews remotely. The interview with I4 was done in English, all others in Swiss-German. Therefore, quotes by I1, I2, I3, and I5 are translated statements.

Our first three interview partners I1 to I3 stated to work currently or more recently have worked with RxJS in conjunction with Angular and ngrx[9] to develop frontend web applications. I4 was a proficient RxJS user. Our fifth interview partner I5 was a backend engineer who used akka-streams[10] in Scala to model data-flows for a WebSocket-based[11], reactive API layer serving a web frontend application.

All interview partners pointed out that they like RP because it provides them with "[. . .] a good way for composing multiple data sources" (I1) and, combined with "[. . .] a statically typed language, RP guarantees some kind of basic formal correctness of a program" (I5). Hence a significant strength of RP seems to be the ability to describe complex data-flow constructs using a specialized DSL. However, they also pointed to current challenges: The learning curve can be steep for a novice engineer: "Being challenged with new abstractions [of Angular and ngrx] already, I experienced RxJS concepts and operators to be hard to convey" stated I3, giving lectures in frontend web application development.

It was interesting to hear that, especially in the area of developing web applications using Angular, our partners seemed not to have to work with pure RxJS code often. E.g., when using ngrx for state management, "The framework hides observables from its main API surface carefully, so you do not have to interact with them directly" (I2). As soon as our interviewees had to extend built-in functionalities with own features, e.g., a new effect[12], I1 and I2 valued the possibility to interact with underlying observables though.

When asked explicitly about what they dislike the most about RP, all interview partners, with no exception, emphasized the debugging process of an RP program as unsatisfactory. The fact that our interviewees remembered the search for a bug as something negative did not surprise, hence a bug is commonly something negative afflicted. It was remarkable though that statements like "In 99% of all cases, I add console.log statements manually and run the program over and over again, trying to understand what is happening" (I1) were prevalent and showed *why* our partners dislike RP debugging in particular. I1 to I3 mentioned the Redux DevTools[13] as particular helpful when debugging Angular/ngrx applications nonetheless. Further, I1 noted marble diagrams as valuable in order to understand how an RxJS observable works, whether during development or debugging.

[9]https://ngrx.io/
[10]https://akka.io/
[11]https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[12]https://ngrx.io/guide/effects
[13]https://chrome.google.com/webstore/detail/redux-devtools

2.2 War Stories

After we built an intuition for how software engineers work with RP by evaluating the interviews, we were interested in more RxJS-specific, hands-on experiences. We asked the engineering community via Twitter[14] about their personal, most recent RxJS debugging war story and sent out various emails with the same request. After reaching out, we were able to collect five responses in English: One by an RxJS core team member (R1), two from Angular Google Developer Experts (R2/3) and another set of reports by two software engineers (R4/5) building web and mobile applications using React and RxJS, which includes the author of this paper.

In their report, R3 focused on how they built code with improved testability because of recent changes in RxJS: "[. . .] in the beginning, it was very hard to write asynchronous tests [. . .]. I really disliked [. . .] you were forced to pass a TestScheduler". Allowing to pass the scheduler explicitly as parameter forced them to introduce code, which was only necessary for testing reasons they stated further. With its current major version, RxJS 6 improved profoundly on the `TestScheduler`. The runtime environment itself can be augmented with the scheduler now, which results in cleaner code.

Even though the share of non-productive, testing-related code necessary to build mature RxJS-based applications was mentioned to have decreased today, R2 as well as R4 and R5 commented on the common practice of manually modifying production code during debugging sessions, hence confirming earlier statements from our interview partners. R5 described a specific scenario where they suspected a problem within a complex observable composition: Having multiple asynchronous, remote data sources, they used observables to model the dependencies between them and implemented computations on their results as operators. On top, each data source could re-emit updated versions of previously requested information at any time. After a week in production, though tested thoroughly, the first of many bugs got reported: "Displayed numbers kept changing where we did not expected them to. In other places, they were not rerendered where they were supposed to, e.g. after we changed them in the system," they told us. The browser's debugger tools and its breakpoints did not help much since the operators were executed several times. Other parts of the stream were impossible to get a handle upon, even with conditional breakpoints. "I started to inspect the flow [. . .] with console.logs and later also using tags from rxjs-spy which exposed more detailed life cycle information." After a time-consuming log analysis, they finally were able to resolve all bugs. The log statements added were removed in the aftermath, though the rxjs-spy-related code was left in the observable stream in case they might be needed again in the future.

[14]https://twitter.com/swissmanu/status/1242429409208029185

In a related war story, R2 discloses similar invasive practices using an external tool when they implemented logic to request and cache batches of a remote resource: "It took quite some time to get it right and one of the most invaluable tools proved to be Stackblitz[15] which gave us the ability to quickly create smaller working examples and iterate on them." This sandboxed setting allowed them to run, debug, and iterate on selected pieces of a larger observable stream. Even though the final result had to be integrated back into the actual application, the extra effort was worth the result the engineer concluded in their report.

A final story describes yet another way of utilizing an external tool: Rather than using a dedicated sandbox to develop pieces of a more complex system, R5 used Rx Visualizer, an online visualization utility to generate marble diagrams from real code. Like in the report before, it was necessary to extract parts from the codebase of the actual application. Once done, the visualizations helped to understand when values got emitted, when subscriptions changed and when observables completed: "Marble diagrams were a huge help in order to understand detailed runtime and life cycle behaviors of the observable."

2.3 Insights

Software engineers value how they can describe data-flows using a RP DSL, even though the learning curve was perceived as steep. We heard some interesting reports on how RxJS is applied in a daily development environment: Marble diagrams help to understand an observables behavior and are useful to implement tests. Large frameworks like Angular hide some of the complexity of RxJS but allow engineers to make use of its full power once the pre-provided functionality needs to be extended.

Most participants considered a statically typed language like TypeScript, a fundamental necessity allowing them to implement data-flow graphs with minimal, formal correctness, as we heard in the interviews. When the engineers needed to interact with data-flow graphs at runtime, e.g. to debug the behavior of a specific part of a stream, we noticed throughout most reports that they were not 100% satisfied with the feature set they were provided by traditional debugging tools. Almost all of the reporters referred to the practice of modifying their source code manually and adding trace log statements where they assumed a problem to overcome the feature gaps in traditional debuggers. Listing 4 exemplifies two challenges when debugging a stream of observables with imperative-focused debugging tools.

Where a breakpoint can easily be added to Line 2 within the arrow function, this is impossible for `take` on Line 3. One would need to place the breakpoint within the operator's internal implementation instead, which can be cumbersome

```
1  interval(1000).pipe(
2    map(i => i * 2),
3    take(4),
4    tap(console.log)
5  ).subscribe(showValue); // Emits: 0, 2, 4, 6
```

Listing 4. An observable emitting a sequence of increasing integers every second. Traditional breakpoints are possible inside the arrow function on Line 2. Though a breakpoint can be added on Line 3, it will never be hit during the actual execution of the `take` operator. Line 4 shows a manually introduced trace log statement using the `tap` side effect operator.

in case the operator is used in a different stream as well. Once the breakpoint on Line 2 interrupts the execution of the program, we will notice another shortcoming related to this circumstance: Rather than representing the logical flow implemented using the DSL, the call stack as shown in Listing 5 points deep into RxJS' internal implementation. That is why a traditional debugger's step controls cannot operate on the data-flow graph; It just can not interpret this level of abstraction.

```
<anonymous> RxJS
    rxjs 6.5.2/internal/operators/map.js:49
    rxjs 6.5.2/internal/Subscriber.js:66
    rxjs 6.5.2/internal/observable/interval.js:23
    // ...
```

Listing 5. A call stack showing the internal RxJS execution stack for a breakpoint in the arrow function on Line 2 in Listing 4.

We learned that simple trace augmentation, like on Line 4 in Listing 4, logs emitted values only. Such trace logs might be helpful when debugging simple graph compositions. Though, they lack life cycle information of the underlying observables completely. The importance of such information was emphasized by R5 describing their usage of rxjs-spy: To know, when an observable gets subscribed and unsubscribed, when it completes or fails, helped them to solve complex problems multiple times. When dealing with higher-order observables, they value this information even as indispensable.

Finally, we understood that if a problem is hard to replicate within the actual application, the engineers use external sandbox development environments to isolate specific parts of an observable composition. These allow them to iterate on it faster than it would be possible otherwise.

After the evaluation of all reports and interviews, we speculate that software engineers truly lack, but also seldomly use debugging tools that can handle RP concepts provided by RxJS. Even though traditional debuggers might help to

[15]Stackblitz is a full JavaScript development environment available online https://stackblitz.com/

some extent, they do not provide all the information an engineer requires in a particular situation. Instead, they turn to manual trace log augmentation and extraction of source code as we saw repeatedly.

3 Validation

Almost all participants from our interviews and war story reports showed a tendency to manually modify source code with trace logs during the hypothesis instrumentation phase when debugging RxJS code. This practice is often not perceived as efficient since the evaluation and interpretation of trace information tends to be cumbersome and very time-consuming. Also, removing log statements after a successful debugging process might leave new bugs in production code if not done carefully. Like Banken et al. [6] before, we identified this technique as one of the primary debugging practices when software engineers work with RxJS-based code.

That is why we saw demand in validating this statement and previous findings about manual code modification for debugging reasons with an observational study. Our study sought to validate the following hypothesis:

- *Hypothesis: If software engineers must solve an RxJS-based problem, then they will instrument the code manually in order to understand its behavior.*

3.1 Study Design

The subjects for our study were required to have experience in developing applications with RxJS. We recruited four subjects willing to participate in our experiment. We were interested in seeing how the subjects apply debugging techniques they would use in everyday situations in their jobs. Hence we decided to conduct the experiment in a somewhat uncontrolled environment where the subjects used their own devices with their development environments of personal preference. Our objective for the experiment was communicated as broad as possible to prevent bias: "We are interested in how *you* debug a problem" did not mention our hypothesis by intention.

We planned to have a one-hour session for the actual experiment with each subject, followed-up by an unattended after-action survey. We executed the experiment in two consecutive blocks of 25 minutes each. We provided a ZIP file[16] containing the source code for two frontend web applications implemented using TypeScript and RxJS along with a Jest test suite at the start of a session. Each of these applications was rigged with two to three bugs, which we asked the subjects to identify and fix using whatever debugging techniques they prefer and commonly use. Where the first application required less complicated intervention to resolve the contained bugs, the second application demanded substantial modifications in the data-flow as it made heavy use of higher-order observables. The provided test suite allowed the subjects to understand the functional requirements of each application as well as to quickly verify their changes to be successful (or not).

A block was considered as complete once the test suite signaled all bugs as resolved, or the 25 minutes expired. We asked our subjects to act like in a pair programming situation where they "think out loud" their thought process. Though we refrained from answering any question related to the "where" a bug has to be expected.

We sent out the participant briefing document to all of our subjects a week before the experiment. We outlined the course of action and provided them with an example ZIP file. This file contained the same setup as the file provided at the experiment and allowed the subjects to get accustomed to things like starting the web applications or running the test suites.

We decided to monitor our subjects' progress remotely using voice chat and screen sharing due to the COVID-19 situation at the time of our study. Furthermore, this allowed us to record the sessions with relatively low technical effort for later evaluation.

The after-action survey[17] was provided within 24 hours after a subject's participation in the main part of the study. We asked the subjects about (Q1) if they currently use RxJS on or off their jobs, the (Q2) number of years they have experience with RxJS, in (Q3) which field (like frontend, backend or others) they use RxJS and finally (Q4) which tools and techniques they use to debug RxJS-based code. The respective answers allowed us to put the observed actions into perspective and detect potential irregularities in case a subject acted differently as they would have in a "real" situation.

3.2 Study Execution and Results

After the subjects got themselves accustomed to the application provided and understood its purpose, all of them used the test suite to gather context about what features do not work as expected initially. Further, all of them tried to recreate the failing behavior in the UI manually. We could not observe any of the subjects using external tools, e.g. RxViz, to inspect specific code parts in isolation in later iterations of the debugging process. Though, S4 noted that they would have usually started to decompose the problem into smaller pieces and observe their behavior in specific after the 25 minutes of the second block expired.

While all subjects added manual trace log statements to existing arrow functions or by adding `tap` operators in the instrument hypothesis phase, none of them used additional libraries like rxjs-spy for doing so. S2 and S4 used the traditional debugging tools provided by their browser or IDE to

[16] https://github.com/swissmanu/mse-pa1-experiment/archive/v1.0.2.zip

[17] https://github.com/swissmanu/mse-pa1-experiment/blob/f70102885be86fb2323b9516005e1d6dfeb9795b/after-action-survey-questions.md

Table 1. Observed practices and tool usage per subject.

Subject	Trace Logs	Debugger	Add. Tools
S1	X		
S2	X	X	
S3	X		
S4	X	X	Next step

Table 2. Results per subject for each presented problem.

Subject	Problem 1	Problem 2
S1	Time expired	Time expired
S2	Time expired	Time expired
S3	Time expired	Time expired
S4	Solved	Time expired

Table 3. After-action survey responses per subject.

Subject	Q1	Q2	Q3	Q4
S1	Yes	1 year	Frontend	Trace Logs, rxjs-spy
S2	No	> 3 years	Frontend	Debugger, Trace Logs
S3	Yes	2 years	Frontend	Debugger, Trace Logs, rxjs-spy
S4	No	2 years	Frontend, Backend	Debugger, Trace Logs

add breakpoints. Both of them commented on the inability of stack traces to interpret RP abstractions as unsatisfying. We could further observe a "trail-and-error" approach in later iterations of the debugging process. The subjects started to introduce modifications to the system, which they immediately tested against their latest hypothesis. Table 1 provides an overview on the complete collected data regarding used techniques and tools.

Only S4 was able to solve the first problem given, as shown in Table 2. None of the subjects was able to successfully identify and fix the bugs hidden in the second problem within time.

The survey responses available in Table 3 showed that S2, S3 and S4 had two or more years of experience with RxJS. Where all of them use RxJS to develop frontend applications, S4 declared having used RxJS for backend development as well. When asked what tools they usually use for debugging, S2, S3 and S4 stated to use the traditional debugger of their IDE. S1 and S3 leverage additional tracing functionality of rxjs-spy, and all four of our subjects use manual log statements.

3.3 Interpretation

We were able to observe how all subjects predominantly used manual source code augmentation by adding trace logs. Two of the subjects used traditional debugging utilities in order to inspect the program's state at runtime in addition. All subjects used the new information gained to refine their hypothesis about underlying problems before starting a new iteration in the debugging process. We could not observe the extraction to and reintegration from an external tool. All subjects exhibited the debugging behavior described in our hypothesis. Further, we could verify previous results by Banken et al. [6] successfully as well.

Even though S1 and S3 stated in the after-action survey to regularly use rxjs-spy for debugging RxJS programs, neither of them made use of this library during the experiment part of the study.

Interviewing professionals, consolidating RxJS hands-on experiences from the war stories, and evaluating the results from our observational study showed us that software engineers use a variety of practices, tools and utilities to debug RP programs. Beside the habit of adding trace logs manually, we saw them evidently trying to answer their debugging hypotheses using traditional, imperative-focused debugger utilities. The later way of debugging was repeatedly commented as unsatisfying as these utilities cannot handle RP constructs, and with this, cannot help to detect problems located within these at all. The former way, the introduction of manual log statements, was both described as the prevalent way of debugging RxJS or as "the last resort" when no other debugging technique helped before.

We heard further how engineers isolate specific observables from bigger data-flows and how they inspect those in sandboxed environments and visualizers. This helps them understanding the observable life cycle and value emitting behaviors better and iterate faster in order to resolve problems.

More than 50% of our 14 peers throughout the interviews, war story reports and the experiments after-action survey stated to know about specific RxJS RP debugging tools. It was apparent that all subjects during the observational study refrained from using any of them, though. It is our speculation that the subjects knowing about specific tools held themselves back from using them because they perceived the effort of setting them up (e.g., installing and configuring rxjs-spy) as too time-consuming. Not having the "right" tool available without significant additional effort is also what we interpret from the statement by S4: Though they would have started to extract parts of the data-flow and inspect it with other tools, they would have done so only after the 25 minutes of the block expired; Hence a more accessible way allowing such analyses would have influenced the behavior of the subject.

The best RP debugging tools are useless if either the hurdle to use them is too high, or engineers do not understand which particular part of the debugging process they can benefit from them. Salvaneschi et al. [18] provided in their previous study on the *Reactive Inspector* for *REScala* evidence on the effectiveness of a fully integrated RP debugging solution, which supports developers in their daily work using the Eclipse IDE. Hence, we can postulate an answer to our first research question RQ1: The most significant challenge software engineers face when debugging RxJS-based programs is to know *when* they should apply *what* tool to resolve their current problem in the *most efficient* way.

4 Future Work

We see the biggest shortcoming of current RxJS-oriented debugging solutions like rxjs-spy, RxFiddle, or RxViz in fact that they are not integrated in established development environments (e.g., IDEs or internet browser developer tools). This leads to the practice of manually augmenting code itself rather then working with it in a less obtrusive, fully integrated way as we were able to proof in our observational study. Using specialized utilities is an extra effort an engineer has to invest every time they want to debug a data-flow: Either tagging an observable for rxjs-spy or extracting parts of it to an external environment, all of these practices require engineers to "go the extra mile" in order to inspect the runtime behavior of an RxJS-based application. The additional effort might be neglectable when treating a rather complex data-flow composition. However, it holds back engineers from applying the tools to simple observables like in the first block of the experiment we conducted.

The observation that two out of four of our study subjects tried to debug an RP application with traditional, imperative-centric debugger utilities, as well as related statements from the interviews and war stories, strengthened our assumption regarding tool integration. Engineers expect the debugging tools they know and rely on to give correct insight on every program, no matter the paradigm (imperative or declarative) with which it got implemented.

This leads us to the answer to our second research question RQ2: We want to improve the experience of debugging RxJS-based applications by providing RP specific debugging utilities where software engineers expect them the most: Fully integrated with the traditional debugger they know from their IDE or browser developer tools[18].

[18]A positive example of such a seamless integration is the debugger of the Google Chrome developer tools: It combines call stack frames of asynchronously executed functions[9] seamlessly with those of synchronously executed code. This provides software engineers with a better understanding about which part of the program triggered the statement they currently inspect.

The answer on which RP debugging tool exactly (e.g., a full reactive debugger or a visualizer using marble diagrams) we are going to integrate, how such integration will look like in detail, how it will support engineers in a particular step of the debugging process, as well as the answer on RQ3 will be part of our future work on the topic of "Debugging RxJS-based Applications".

5 Threats to Validity

This study is subject to the following threats and limitations:

5.1 External Validity

The data we collected from interviews, war story reports, and the observational study is based on a sample population with 14 individuals. Hence, the results we conveyed in this paper are not representative and are not transferable to the entire software engineering population.

5.2 Internal Validity

The observational study was executed in an uncontrolled environment. All subjects used their personal computers, running their own software development environments. We have no comparative data to measure how this design influenced the observed outcome, assuming that this setup diminishes the reproducibility of the experiment.

We noticed that the subjects needed time to understand the intention of the applications they were provided with before they were able to start with the actual debugging process. Since the amount of required time was different from subject to subject, we suspect it influenced the result of the experiment.

5.3 Construct Validity

The time limit of 25 minutes per experiment block bears the potential to put the subjects under time pressure. This risk might explain why we could not observe any more time-consuming debugging techniques (e.g., installing additional utilities like rxjs-spy) during the study.

6 Conclusion

In this paper, we have explored how software engineers debug data-flow-oriented programs implemented using RxJS. We presented an observational study to validate a hypothesis based on the outcome of ten individual interviews and hands-on experience reports from software engineering professionals. More than 50% of the 14 engineers we worked with during our research told us that they know of the existence of specific debugging tools for RP with RxJS. Nonetheless, the experiment conducted with four participants allowed us to prove that engineers augment source code manually with trace logs instead of using such specialized utilities.

We identified the fact that RxJS specific debugging tools are not tightly integrated with existing, traditional debuggers in IDEs and the developer tools of internet browsers as the main reason why software engineers do not use them more often. In order to lower the effort necessary to use specialized RP debugging tools for engineers, we declared the integration of such as the matter for our own future research.

Acknowledgments

We want to thank the engineers who participated in our study for their time.

References

[1] 1990. *IEEE Standard Glossary of Software Engineering Terminology.* https://doi.org/10.1109/IEEESTD.1990.101064

[2] 2019. *An example using the console API | rxjs-spy.* Retrieved 17-May-2020 from https://cartant.github.io/rxjs-spy/ Versioned as https://github.com/cartant/rxjs-spy/tree/2bffdee2d5f712d70583ef48297446bd31a9a6f4.

[3] 2020. *RxViz - Animated playground for Rx Observables.* Retrieved 16-May-2020 from https://rxviz.com/ Versioned as https://github.com/moroshko/rxviz/tree/51a737717a27f15b68f907b2329f7b0b6b11cb2b.

[4] Manuel Alabor. 2019. Reactive Applications in Frontend Engineering Today. (2019). https://github.com/swissmanu/mse-seminar-reactive-applications-in-frontend-engineering-today/releases/tag/v1.0.1.

[5] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[6] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE âĂŹ18).* Association for Computing Machinery, New York, NY, USA, 752âĂŞ763. https://doi.org/10.1145/3180155.3180156

[7] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Education India.

[8] Google. 2020. *Angular - Observables in Angular.* Retrieved 24-Apr-2020 from https://angular.io/guide/observables-in-angular Versioned as https://github.com/angular/angular/blob/64ac1062489bbc97a0d4b95af5ce9566091fe044/aio/content/guide/observables-in-angular.md.

[9] Google. 2020. *JavaScript Debugging Reference.* Retrieved 16-Aug-2020 from https://developers.google.com/web/tools/chrome-devtools/javascript/reference#call-stack Versioned as https://web.archive.org/web/20200713153259/https://developers.google.com/web/tools/chrome-devtools/javascript/reference.

[10] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.* IEEE, 383âĂŞ392. https://doi.org/10.1109/ESEM.2013.43

[11] Shixia Liu, Yingcai Wu, Enxun Wei, Mengchen Liu, and Yang Liu. 2013. StoryFlow: Tracking the Evolution of Stories. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of IEEE InfoVis 2013)* 19, 12 (2013), 2436–2445.

[12] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. *CoRR* abs/1705.05937 (2017). arXiv:1705.05937 http://arxiv.org/abs/1705.05937

[13] G. Pothier and E. Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Software* 26, 6 (2009), 78–85.

[14] ReactiveX. 2020. *ReactiveX.* Retrieved 24-Apr-2020 from http://reactivex.io/ Versioned as https://web.archive.org/web/20200419004415/http://reactivex.io/.

[15] ReactiveX. 2020. *RxJS - Introduction.* Retrieved 15-Aug-2020 from https://rxjs.dev/guide/overview Versioned as https://github.com/ReactiveX/rxjs/blob/46e35f71b02d02c5a7d7f426e78eadd625c1a67a/docs_app/content/guide/overview.md.

[16] ReactiveX. 2020. *RxJS - Testing RxJS Code with Marble Diagrams.* Retrieved 16-May-2020 from https://rxjs-dev.firebaseapp.com/guide/testing/marble-testing Version 6.5.5-local+sha.7e4589a1.

[17] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity* (Lugano, Switzerland) *(MODULARITY âĂŹ14).* Association for Computing Machinery, New York, NY, USA, 25âĂŞ36. https://doi.org/10.1145/2577080.2577083

[18] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE âĂŹ16).* Association for Computing Machinery, New York, NY, USA, 796âĂŞ807. https://doi.org/10.1145/2884781.2884815

[19] Andre Staltz and Contributors. 2019. *RxJS Marbles.* Retrieved 16-May-2020 from https://rxmarbles.com

A.2 Debugging Support for Reactive Programming

The review version of this paper (Appendix A.2.1) and its supplementary material (Appendix A.2.2) were submitted to the Technical Papers track at the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis 2022 (ISSTA '22).

A.2.1 Paper

Debugging Support for Reactive Programming

Feasibility of a Ready-to-hand Debugger for RxJS

Anonymous Author(s)

ABSTRACT

Debugging reactive data-flow-oriented applications is a cumbersome task. Unfortunately, modern development environments provide only suitable tools to debug control-flow-oriented programs. As a result, software engineers utilizing RxJS, a popular library for reactive programming in JavaScript, use inapt debugging tools, utilities outside of their accustomed IDE, or antiquated debugging practices like manual print statements. This paper presents two contributions to reactive debugging: (i) Operator log points, a novel debugging utility for reactive programming, make manual print statements obsolete. We implement them for RxJS as an extension for Microsoft Visual Studio Code. By doing so, we integrate the utility with the workflow of software engineers seamlessly, thus (ii) proof the feasibility of a ready-to-hand debugging utility for reactive programming by existence.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Data flow languages*; *Software maintenance tools*; • **Human-centered computing** → *Human computer interaction (HCI)*; *User centered design*.

KEYWORDS

reactive programming, reactive debugging, human computer interaction, user centered design

ACM Reference Format:

Anonymous Author(s). 2022. Debugging Support for Reactive Programming: Feasibility of a Ready-to-hand Debugger for RxJS. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When software engineers look at the source code of an existing application, they want to understand how the program was implemented technically. They do this either because they want to get themselves acquainted with a new code base they never worked with before (e.g., during onboarding of a new team member) or, more often, because someone reported an unexpected application behavior (e.g., the program crashed). Inspecting source code at runtime is commonly known as “debugging” [1]. Layman et al. [6] formalized an iterative process model (see Figure 1) by dividing the broader task of debugging into three steps: The engineer uses (i) gathered context information to build a hypothesis on what the problem at hand might be. They then (ii) instrument the program using appropriate techniques to prove their hypothesis. Eventually, they (iii) test the instrumented program. If the outcome proves the hypothesis to be correct, the process ends. Otherwise, the engineer uses gained insight as input for the next iteration.

ISSTA 2022, 18-22 July, 2022, Daejeon, South Korea
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

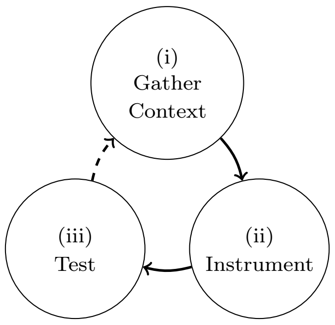


Figure 1: Iterative Debugging Process after Layman et al.: Gather context to formalize hypothesis, instrument and test system to prove hypothesis, resulting in a new iteration or a confirmed hypothesis.

The most basic debugging technique for instrumentation and testing is manually adding print statements to the source code: They generate execution logs when placed across the program’s code and allow to reconstruct its runtime behavior. However, the number of generated log entries increases, the required amount of work to analyze the logs gets out of hand quickly. This is why specialized debugging utilities provide tools to interact with a program at runtime: After interrupting program execution with a breakpoint, they allow engineers to inspect stack frames, inspect and modify variables, step through successive source code statements, or resume program execution eventually. These utilities work best with imperative or control-flow-oriented programming languages since they interact with statements and stack frames of the debugged program.

Modern IDEs enable software engineers to debug programs, no matter what programming language they are implemented with, using one generalized user interface (UI). The result is a unified user experience (UX) where debugging support is only a click away.

However, by adopting control-flow-oriented debugging utilities into their workflows, software engineers face a new problem when working with reactive programming (RP). Salvaneschi et al. [16] described this shortcoming of traditional debuggers when confronted with RP and coined the concept of *RP Debugging*. Later, Banken et al. [3] proposed a solution for debugging RxJS RP programs in an external visualizer utility.

Alabor et al. [2] examined the RP debugging habits of software engineers in an observational study. They replicated the observation by Salvaneschi et al. and observed that even engineers aware of RP debugging tools did not use them. Instead, these engineers used manual print statements.

Within this paper, we are going to present two contributions to the field of RP debugging:

1. *Operator log points* are a novel utility for debugging RP programs. They make manual print statements obsolete by providing specialized log points for RP applications.
2. By implementing operator log points in *RxJS Debugging for Visual Studio Code*, an extension for Microsoft Visual Studio Code[1] (vscode), we provide a proof by existence for the feasibility of a ready-to-hand RP debugging utility. Software engineers can debug RxJS programs without learning new UX patterns or additional setup effort.

Before we do a deep-dive on the functionality of operator log points in Section 4, we present an example for the primary challenge of RP debugging in Section 2 and discuss related work in Section 3. Next, we give an overview of performed usability inspections and validations in Section 5. Finally, we consider threats to validity regarding the usability tests in Section 6 and introduce topics for future work in Section 7.

2 RP DEBUGGING: THE HARD WAY

A primary characteristic of RP is the paradigm shift away from imperatively formulated, control-flow-oriented code (see Listing 1) to declarative, data-flow-focused source code [16]. Instead of instructing the computer how to do what, i.e., one step after another, we use RP abstractions to describe the transformation of a continuous flow of data.

RxJS implements reactive sources with *Observables*. An observable generates five types of life cycle events: Once a consumer (i) *subscribes* to an observable, the observable starts to (ii) *emit* values, (iii) *completes* (e.g., when a network request has been completed), fails with an (iv) *error*, or may get (v) *unsubscribed*. Engineers use *Operators* to transform these events on their way through the data-flow graph. An operator modifies values, composes other observables, or changes how life cycle events get forwarded (e.g., catch an error and emit an empty value instead). Listing 2 shows

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

```
import reportValue from './reporter';

for (let i = 0; i < 5; i++) {
  if (i < 4) {
    reportValue(i * 2);
  }
}
```

Listing 1: Basic example of imperative-style/control-flow-oriented programming in JavaScript: Multiply integers between 0 and 4 for every value that is smaller than 4 and call reportValue with the result.

```
1 import reportValue from './reporter';
2 import { of } from 'rxjs';
3 import { filter, map } from 'rxjs/operators';
4
5 of(0, 1, 2, 3, 4).pipe( // Observable with ints 0..4
6   filter (i => i < 4),  // Operator omitting 4
7   map(i => i * 2),      // Operator multiplying by 2
8 ).subscribe(reportValue)
```

Listing 2: Basic RP example implemented with RxJS in JavaScript: Generate a data-flow of integers from 0 to 4, skip values equal or larger then 4, multiply these values by 2 and call reportValue with each resulting value.

an example of a source observable, two operators, and one consumer.

Traditional debuggers reach their limitations when facing data-flow-oriented code: While we can navigate through the successive iterations of the *for* loop in Listing 1 using the step controls of the debugger, this is not possible for the transformations described in Listing 2. Assuming we set a breakpoint within the lambda function passed to *filter* on Line 6, stepping over to the next statement will not lead to the lambda of *map* on Line 7 as one might expect. Instead, the debugger continues in the internal implementations of *filter*, part of the RxJS RP runtime. With a deeper understanding of the difference between control- and data-flow-oriented programming, this might look plausible. However, previous research [2,3,16] revealed that software engineers expect different behavior from the debugging tools they have at hand. As a direct consequence, engineers fall back to the problematic debugging technique of adding manual print statements, as exemplified in Listing 3 on the next page.

3 RELATED WORK

Salvaneschi et al. [16] identified the divergence between a control-flow-oriented debugger's expected and actual behavior as one of their key motivations for RP debugging. The stack-based runtime model of control-flow-oriented debuggers does not match the software engineers' data-flow-oriented mental model of the program they are debugging. Because the debugger has a "lack of abstraction," it cannot interpret

```
1  import reportValue from './reporter';
2  import { of } from 'rxjs';
3  import { filter, map, tap } from 'rxjs/operators';
4
5  of(0, 1, 2, 3, 4).pipe(
6    tap(console.log),      // <-- Print Statement
7     filter (i => i < 4),
8    tap(console.log),      // <-- Print Statement
9    map(i => i * 2),
10   tap(console.log),      // <-- Print Statement
11 ).subscribe(reportValue)
```

Listing 3: Manually added print statements on Lines 6, 8 and 10 to debug a data-flow implemented with RxJS in JavaScript.

high-level RP abstractions and works on the low-level implementations of the RP runtime extension instead. Salvaneschi et al. proposed *Reactive Inspector* [15], the first specialized RP debugging solution for RP programs implemented with REScala, an RP extension for the Scala programming language. Integrated with the Eclipse IDE, the utility provides a wide range of RP debugging functionalities like the visualization of data-flow graphs and the information that traverses through them. Reactive breakpoints allow to interrupt program execution once a graph node reevaluates its value.

Since then, RP has gained more traction across various fields of software engineering. With a shared vision on how to surface RP abstractions at the API level, *ReactiveX*[2] consolidated numerous projects under one open-source organization. Together, its members provide RP extensions for many of today's mainstream programming languages like Java, C#, and Swift. For the development of JavaScript-based applications, software engineers can rely on RxJS[3]. Angular by Google is one of the more popular adopters of this library and uses RxJS to model asynchronous operations like fetching data in web frontend applications.

Two years after Salvaneschi et al. proposed RP Debugging, Banken et al. [3] showed that debugging RxJS-based RP programs is quite similar to REScala-based ones. They were able to categorize the debugging motivations of their study participants into four main, overarching themes. These directly correlate with the debugging issues identified by Salvaneschi et al. earlier, as we show in Table 1.

Banken et al. provided a debugger in the form of an isolated visualizer: *RxFiddle*. The browser-based application visualizes the runtime behavior of an RxJS program in two dimensions: A central (i) data-flow graph shows which elements in the graph interact with each other, and a dynamic (ii) marble diagram[4] represents the processed values over time.

[2]http://reactivex.io/
[3]https://rxjs.dev
[4]Marble diagrams are a visualization technique used throughout the ReactiveX community to graphically describe the behavior of observable-based data-flow graphs. A marble represents a life cycle event, e.g., an emitted value. Multiple marbles are arranged on a thread from left to right, indicating the point in time when the respective life cycle event happened. See https://rxmarbles.com/ for examples.

Table 1: Correlation of debugging issues identified/solved by Salvaneschi et al. with overarching debugging motivations by Banken et al.

Salvaneschi et al.	Banken et al.
Missing dependencies	Understanding dependencies between observables
Bugs in signal expressions	Finding bugs and issues in reactive behavior
Understanding RP programs	Comprehending behavior of operators in existing code
	Gaining high-level overview of the reactive structure
Performance Bugs	-
Memory and Time Leaks	-

Both Salvaneschi et al. and Banken et al. suggested technical architectures for RP debugging systems. Both suggestions can be summarized as distributed systems consisting of two main components: The (i) RP runtime is instrumented to produce debugging-relevant events (e.g., value emitted or graph node created). These events get processed by the (ii) debugger, which provides a UI to inspect the RP program's state.

Another two years after Banken et al. published their work, Alabor et al. [2] examined the state of RxJS RP debugging. Software engineers still struggled to use appropriate tools to debug RxJS programs according to the interviews they conducted. The authors performed an observational study and found instances of engineers who knew about RP-specific debugging tools but abstained from using them during the experiment. They credited this circumstance to the fact that the IDEs of their subjects did not provide suitable RP debugging utilities ready-to-hand.

Alabor et al. conclude that knowing the correct RP debugging utility (e.g., *RxFiddle*) is not enough. The barrier to using such utilities must be minimized. I.e., RP debugging utilities must be fully integrated into the IDE to live up to their full potential, so using them is ideally only an engineer's keypress away and adheres to accustomed, known UX patterns.

4 AN RXJS DEBUGGER READY-TO-HAND

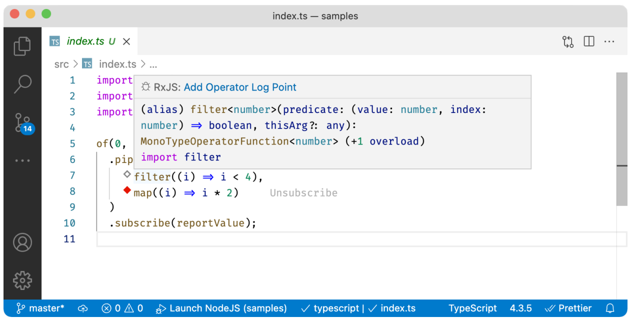
We translated these findings into the central principle for the design of our RP debugger for RxJS: *Ready-to-hand*. Software engineers should always have the proper debugging tool available, no matter what programming paradigm they are currently working with. Further, this tool should integrate with the engineer's workflow seamlessly.

4.1 Operator Log Points

Operator log points combine the concept of log points as known from control-flow-oriented debuggers with live *probes*, formerly proposed by McDirmid [7][5] for RP programs. They display life cycle events produced by an RxJS operator directly within the source code editor.

Possible operator log points are suggested ready-to-hand through an icon annotation within the code editor, next to the respective operator. While the software engineer instruments the source code to prove their debugging hypothesis, they can enable a log point by hovering the mouse pointer over its associated annotation and selecting the *Add Operator Log Point* action (see Figure 2). When ready to test their hypothesis, the engineer starts the RxJS program using the built-in JavaScript debugger; no extra effort is required. Once the program is running, each enabled operator log point displays the life cycle events together with the source code that produced them. Engineers are free to enable or disable additional log points during the debugging session; the life cycle event display will adapt accordingly.

Once finished debugging, the software engineer stops the program. Contrary to manual print statements, no clean-up work is necessary afterward since operator log points do not require any code modifications.


index.ts — samples

index.ts U

src > index.ts > ...

```
1   import
2   import
3   import
4
5   of(0,
6     .pip
7       filter((i) => i < 4),
8       map((i) => i * 2)      Unsubscribe
9     )
10    .subscribe(reportValue);
11
```

RxJS: Add Operator Log Point

```
(alias) filter<number>(predicate: (value: number, index:
number) => boolean, thisArg?: any):
MonoTypeOperatorFunction<number> (+1 overload)
import filter
```

Figure 2: *RxJS Debugging for vscode* used to debug code from Listing 2. A diamond icon indicates operator log points: A grey outline represents a suggested log point (Line 7), a filled, red diamond an enabled log point (Line 8). The source code editor shows life cycle events at the end of the respective line (Line 8, "Unsubscribe"). Log points are managed by hovering the respective icon and selecting the appropriate action.

4.2 Suggesting a Log Point

Log points for operators are automatically suggested while the software engineer edits the source code of an RxJS program. To interpret the programs code semantically, the debugger extension leverages on the TypeScript[6] programming language toolchain.

We use the TypeScript parser to continuously evaluate source code, which results in an abstract syntax tree (AST). Along with the semantical structure of the program, the AST contains type and positional information for every parsed token. The extension processes the type information to detect all present RxJS operator functions. For every operator function found, the positional information allows to annotate the relevant source code in the editor with an icon.

4.3 Architecture

The technical architecture of *RxJS Debugging for vscode* (see Figure 4) is a refined version of the system proposed by Banken et al. [3].

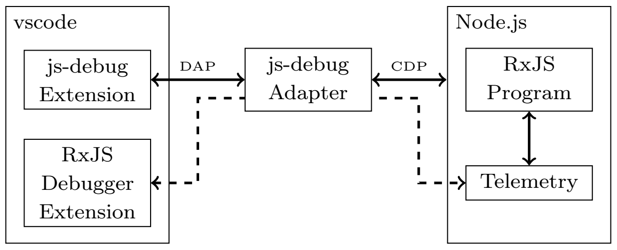


Figure 3: The *Telemetry* component instruments the *RxJS program* (right). The *RxJS Debugger Extension* runs inside of the vscode process. The two components communicate with each other by reusing the CDP communication channel established by the generic vscode JavaScript debugger called *js-debug*.

JavaScript virtual machines (VM) like V8 (used in Google Chrome or Node.js) or SpiderMonkey (used in Mozilla Firefox) implement the Chrome DevTools Protocol (CDP)[7]. Debugging tools like vscode's built-in JavaScript debugger use CDP to connect and debug JavaScript programs. RxFiddle by Banken et al. [3] uses WebSockets to exchange relevant data. We leverage the CDP connection established by the vscode's JavaScript debugger, making the system more robust since we do not need to maintain an additional channel for debugger communication.

5 USABILITY INSPECTION AND VALIDATION

We followed a User-Centered Design (UCD) [5] approach in three iterations to conceptualize and implement our debugging utility. The relevant methods we applied helped us to keep our efforts aligned with our main goal: To establish a debugging utility that is ready to hand and does not requiry any extra learning or setup procedures.

After sketching a rough proof of concept (PoC) in the first step, we performed a cognitive walkthrough [17] to validate our idea of replacing manual print statements with operator log points. The resulting data helped us to build a prototype of the extension. Next, we used this prototype to conduct a moderated remote usability test with three subjects. This allowed us to uncover pitfalls in the UX concept and find

[5]As a matter of fact, operator log points were originally called *operator probes*, but got renamed after initial confusion with our test users.
[6]TypeScript is a strongly typed programming language that compiles to JavaScript https://www.typescriptlang.org/

[7]https://chromedevtools.github.io/devtools-protocol/

Table 2: Cognitive walkthrough action sequence with eight steps.

Step	Task
1	Open File
2	Navigate to Operator
3	Open Code Actions
4	Create Operator Log Point
5	Open Operator Log Point Monitor
6	Launch Application
7	Interact with Application
8	Interpret Runtime Behavior

misconceptions early in the development process. Finally, we used the results of these sessions for further refinement. We completed the first minor version of the RxJS RP debugger, which we released to the Visual Studio Marketplace in May 2021[8].

We used the test cases created by Alabor et al. [2] for both the cognitive walkthrough and the remote usability test.

5.1 Cognitive Walkthrough

We concluded the first iteration of our development process with a PoC demonstrating the basic concept of operator log points.

Looking for an informal, expert-driven usability inspection method [8], we found the cognitive walkthrough [17] to be a good fit in this early stage of development. We prepared the profile of a typical user for the RP debugger as input to the inspection. Based on this profile and the debugging process by Layman et al. [6], we created the action sequence available in Table 2. We performed the walkthrough using the *Problem 1* web application by Alabor et al. [2].

The cognitive walkthrough revealed six usability issues, as summarized in Table 3. The full inspection report, including the complete user profile, is available on Github[9].

5.2 Moderated Remote Usability Test

After the initial validation using the cognitive walkthrough, we completed the development of the refined prototype, ready to test with real users.

5.2.1 Study Design. "Think aloud" tests for high functionality systems benefit from at least five test subjects or more [9]. The feature spectrum of the RP debugger prototype is small; hence the probability of finding major usability issues with a smaller subject population is high. Therefore, we decided to work with three individual subjects for our study.

Participants, recruited via Twitter, were required to have worked with RxJS during the past year and use vscode as their primary IDE. We sent out a PDF containing a short

[8]An anonymized excerpt of the extensions Marketplace presence is available in the supplementary material.
[9]The report is available in the supplementary material.

Table 3: UX issues identified using cognitive walkthrough inspection.

Step	Issue
3	The user might know code actions, indicated through the yellow light bulb icon, for providing refactoring and quick fix options. It is questionable if they would expect operator log point options in here as well.
4	When enabling an operator log point, the user does not get any confirmation that this action was successful. Exception: The list of enabled operator log points in the debugging view is visible.
5	The monitoring pane, showing logs for enabled operator log points, must be opened manually. The user might not be aware of this after enabling a log point.
5	The monitoring pane is empty initially. Users might not know what to do next after opening it.
7	The user might not interact with the RP program in the opened default browser in order to get live feedback in the monitoring pane.
7	The opened default browser might overlay the monitor pane in vscode. Because of this, the user might miss on the live trace of values and life cycle events.

briefing and a prototype description a week before the actual test session. The briefing contained information about software requirements (Zoom, Node.js, npm/Yarn, and vscode) and details on what the subjects might encounter during their test session. Here, we emphasized the importance of "think aloud" [4,11], the practice of continuously verbalizing thoughts without reasoning about them.

5.2.2 Study Execution. At the start of a test session, we provided each participant with a ZIP file[10] containing the *Problem 2* web application by Alabor et al. [2] and the packaged version of the debugger extension prototype[11]. While the subject prepared their development environment, we started the video, screen, and audio recording with their consent. Also, we gave a scripted introduction to the code base they just received.

The participants had 25 minutes to resolve as many bugs as possible using the debugger prototype. Rather than tracking each subject's success rate of fixed defects, we emphasized detecting usability issues in their workflow instead.

5.2.3 Study Evaluation. One participant could not get the prototype extension up and running on their system, which means we had only two valid data sets for further evaluation after study execution. We categorized the observed usability issues by debugging process phase (i.e., gather context, instrument hypothesis, and test hypothesis) and task (e.g., "Setup Environment," "Manage Log Points," or "Interpret Log"). From a total of 10 issues, we observed four being a

[10]*This link might reveal the author(s) identity/identities https://github.com/ANONYMOUS*
[11]*This link might reveal the author(s) identity/identities https://github.com/ANONYMOUS*

Table 4: Major UX issues observed during usability test sessions.

Phase	Task	Issue
Instru.	Setup	Participant starts the application in debugging mode, even though they have started it before.
Instru.	Manage	Participant unable to find log point list in debugging view.
Test	Interpret	Participant has difficulties to make a connection from a log point to the generated log entry.
Test	Interpret	Participant interprets logged value as the "input" of the instrumented operator.

problem for both remaining study subjects. Thus we prioritized them as "major." The full usability issue report is available on Github[12]. Table 4 presents the four major issues.

5.3 Utilization

5.3.1 Application of Results. We applied the results from the cognitive walkthrough and the usability tests to refine and complete the RxJS RP debugger presented in Section 4. For example, both the PoC and the prototype had an extra view for displaying the output of a log point, visually disconnecting them from each other. We classified this circumstance as prone to confuse the user during the walkthrough but did not change the prototype yet. The usability tests with real subjects confirmed our suspicion, however. Because of this, we changed the UI for the final, current version and introduced the inline display for log point output directly in the code editor. Another example of an improvement is how the debugger suggests operator log points: The subjects were unaware that suggested log points were available via the code action menu, even though this is an established UX pattern in vscode. Therefore, we removed the suggestions from this menu and introduced the diamond-shaped indicator icon, which is always visible.

5.3.2 Concept Verification. The applied inspection and verification methods, in combination with the practical implementation of the debugger, deliver the existence proof for the feasibility of a ready-to-hand RP debugging utility. Even though the usability test revealed four major usability issues, we successfully verified that operator log points resolve the problems previously identified by Alabor et al. [2].

6 THREATS TO VALIDITY

The results of the usability test are subject to the following threats and limitations:

6.1 Internal Validity

We performed the usability test in an uncontrolled, remote environment, and all participants used their own computers and software installations. The downside of this is the early failure of one subject, which could not get the prototype extension running on their system resulting in an invalid data set. Even though we could have prevented this situation in a controlled lab environment, we consciously decided to take this risk and, in turn, get more realistic results from users working in the context of their accustomed development environment.

[12]The report is available in the supplementary material.

6.2 External Validity

Due to the circumstance that one study participant could not set up the prototype extension, we ended up having only two valid data sets after the remote usability test. Two test subjects should have allowed us to find around 50% of all usability issues present [10]. Because the two remaining subjects share four of 10 issues, we are confident that we identified the most critical usability problems nonetheless.

6.3 Construct Validity

We carefully moderated the test session once test subjects fell silent for more than 10 seconds and reminded them to "think aloud." Even though the participants told us that "speaking to themselves" created an unfamiliar environment for them, we expect the moderation techniques used [4] to minimize any influences on the results.

7 FUTURE WORK

There are several ways how future work can contribute to the efforts presented in this paper.

7.1 Field Test

Version 0.1.2 of *RxJS Debugging for vscode* can debug RxJS programs running in the Node.js JavaScript VM. The major release 1.0.0 generalizes this solution further and brings operator log points to RxJS applications running in web browsers. Thus, we expect installations of the debugger to increase further since more software engineers can benefit from its features.

We see the opportunity for a comprehensive field test on how engineers use the novel RP debugger once its next iteration is available. Usage statistics provided through the planned analytics reporting module will prove helpful in these regards.

7.2 Visualizer Component

Banken et al. [3] proposed visualization techniques for RxJS data-flow graphs in *RxFiddle*. The debugging utility we presented in this paper benefits from the integration of such a visualizer. The graphical representation of an observable graph helps novice engineers to understand RxJS concepts better, and experienced engineers get a new angle on the composition of multiple observables when debugging.

7.3 Record and Replay

A software engineer can record the behavior of a RP program and replay that data independently as many times as they wish later [12]. Such a function would allow two things: During debugging, the engineer can rerun a recorded failure scenario without depending on external systems like remote APIs. Further, recorded data might be used for regression testing to verify that a modified program still works as expected [13].

7.4 Time Travel Debugging

Contrary to regular control-flow-oriented debuggers, omniscient [14], or *time travel* debuggers cannot only step forward but also backward in time. This is because they rely on recorded data rather than a currently running program. Once there is a way to record, store and replay debugging data as suggested before, time travel debugging is a possible next step. Software engineers can then manually navigate through recorded data and observe how individual system parts react to the stimuli.

8 CONCLUSION

We presented *operator log points* as a novel debugging utility for programs implemented using reactive programming in this paper. With *RxJS Debugging for vscode*, we demonstrated how operator log points replace manual print statements for RxJS-based programs. We developed the debugger using a user-centered design process facilitating usability inspection and validation methods, which allowed us to identify and resolve four major usability issues. In addition, we successfully verified that the proposed utility fulfills the requirement of readiness-to-hand, i.e., that it integrates seamlessly with software engineers' daily workflows and does not require additional learning or setup effort.

REFERENCES

[1] 1990. *IEEE standard glossary of software engineering terminology.* IEEE. DOI:https://doi.org/10.1109/IEEESTD.1990.101064

[2] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN international workshop on reactive and event-based languages and systems* (REBLS 2020), Association for Computing Machinery, 15–24. DOI:https://doi.org/10.1145/3427763.3428313

[3] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging data flows in reactive programs. In *Proceedings of the 40th international conference on software engineering*, ACM, 752–763. DOI:https://doi.org/10.1145/3180155.3180156

[4] T. Boren and J. Ramey. 2000. Thinking aloud: Reconciling theory and practice. *IEEE Transactions on Professional Communication* 43, 3 (September 2000), 261–278. DOI:https://doi.org/10.1109/47.867942

[5] Kim Goodwin. 2009. *Designing for the digital age: How to create human-centered products and services.* Wiley Pub.

[6] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE international symposium on empirical software engineering and measurement*, IEEE, 383–392. DOI:https://doi.org/10.1109/ESEM.2013.43

[7] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software - onward! '13*, ACM Press, 53–62. DOI:https://doi.org/10.1145/2509578.2509585

[8] Jakob Nielsen. 1994. Usability inspection methods. In *Conference companion on human factors in computing systems*, 413–414.

[9] Jakob Nielsen. 1994. Estimating the number of subjects needed for a thinking aloud test. *International Journal of Human-Computer Studies* 41, 3 (September 1994), 385–397. DOI:https://doi.org/10.1006/ijhc.1994.1065

[10] Jakob Nielsen and Thomas K. Landauer. 1993. A mathematical model of the finding of usability problems. In *Proceedings of the SIGCHI conference on human factors in computing systems - CHI '93*, ACM Press, 206–213. DOI:https://doi.org/10.1145/169059.169166

[11] Mie Nørgaard and Kasper Hornbæk. 2006. What do usability evaluators do in practice?: An explorative study of think-aloud testing. In *Proceedings of the 6th ACM conference on designing interactive systems - DIS '06*, ACM Press, 209. DOI:https://doi.org/10.1145/1142405.1142439

[12] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability: Extended technical report. *arXiv:1705.05937 [cs]* (May 2017). Retrieved from http://arxiv.org/abs/1705.05937

[13] Ivan Perez and Henrik Nilsson. 2017. Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages* 1, ICFP (August 2017), 1–27. DOI:https://doi.org/10.1145/3110246

[14] Guillaume Pothier and Éric Tanter. 2009. Back to the future: Omniscient debugging. *IEEE Software* 26, 6 (November 2009), 78–85. DOI:https://doi.org/10.1109/MS.2009.169

[15] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on modularity - MODULARITY '14*, ACM Press, 25–36. DOI:https://doi.org/10.1145/2577080.2577083

[16] Guido Salvaneschi and Mira Mezini. 2016. Debugging for reactive programming. In *Proceedings of the 38th international conference on software engineering - ICSE '16*, ACM Press, 796–807. DOI:https://doi.org/10.1145/2884781.2884815

[17] Cathleen Wharton, John Rieman, Lewis Clayton, and Peter Polson. 1994. *The cognitive walkthrough: A practitioner's guide*. Institute of Cognitive Science, University of Colorado.

A.2.2 Supplementary Material

Debugging Support for Reactive Programming

Supplementary Material

Anonymous Author(s)

Contents

Introduction **3**

Cognitive Walkthrough **4**
Persona "Frank Flow" 4
Profile 4
Goals 4
Frustrations 4
Setup 5
Context 5
User 5
Task 5
Environment 5
Walkthrough 5
Open File 5
Navigate to Operator 5
Open Code Actions 6
Create Probe for Operator 6
Open Observable Probe Monitor 8
Launch Application 8
Interact with Application 8
Interpret Runtime Behavior 10
Failure Stories 13

Usability Test **14**
Observed Issues 14

Marketplace Presence **15**

References **17**

Introduction

This document complements the research paper “Debugging Support for Reactive Programming: Feasibility of a Ready-to-hand Debugger for RxJS” during the double-blind review process.

Cognitive Walkthrough

The cognitive walkthrough report formally follows the guide by Wharton et al. [2]. Further, the report refers to “operator log points” as “probes.” This is because we called the log point concept differently during the proof of concept phase and later transitioned to the more intuitive name, based on usability test results.

Persona “Frank Flow”

Profile

- Age: 29 years
- Gender: Male
- Education: BSc in Computer Science
- Occupation: Frontend Software Engineer at ReactiBank

Frank started to work for ReactiBank 2 years ago as a frontend software engineer. As part of a small, interdisciplinary team of 7 people, Frank’ and his team are responsible for developing and maintaining a trading application. This application relies heavily on real-time data, so the group decided to use reactive programming principles throughout the application. Frank knows traditional programming paradigms and the related debugging tools from his studies and personal experiences. He built up knowledge on RP and RxJS for the frontend part of their application after joining the team quickly, however.

Today, Frank uses RxJS efficiently to build new features. He can solve simple problems reported by the product owner on his own. Working on more complicated issues is still something Frank struggles with: He often feels like his knowledge of traditional programming techniques and its debugging utilities are not enough. These tools feel “out of place” to him and do not provide the answers he is looking for. Frank does not like that, eventually, he has to consult one of his colleagues who have experience in RxJS for a longer time.

Goals

- Make complex business domains simple and easy to use for everyone
- Build beautiful, responsive and easy-to-use user interfaces
- Be a fully productive member of the team
- Understand RxJS in complex setups better and deepen knowledge on it

Frustrations

- Known debugging utilities seem unfit to provide answers regarding RP code

Setup

Context

This cognitive walkthrough is based on the first problem given to subjects during the observational study of Alabor et al. [1].

User

See Section "Persona Frank Flow".

Task

After I started the "Problem 1" application and inspected its UI, I was able to observe multiple, unexpected updates rendered in quick succession after I clicked the reset button. Based on this evidence, I formulate my first debugging hypothesis: I suspect that the `flatMap` operator on Line 18 in the file `index.ts` does create multiple observables, which do not get unsubscribed when the reset button is clicked. This results in the observed behavior eventually. To proof my hypothesis, I want to inspect the life cycle events of the created observables more closely.

Environment

Visual Studio Code with enabled TypeScript support is installed. The prototype of the RxJS debugging extension is installed as well. The source code of "Problem 1" [1] is present. Further, an internet browser (e.g. Mozilla Firefox or Google Chrome) is present.

Walkthrough

Open File

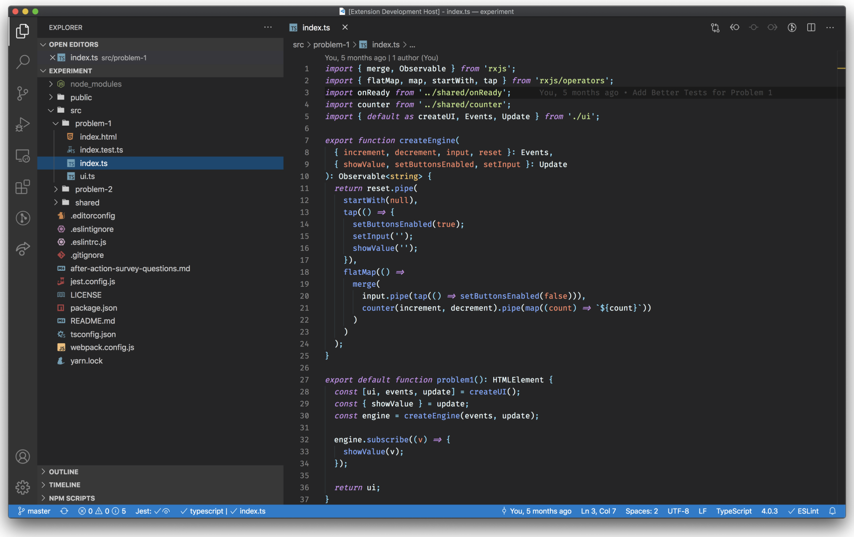
Open `index.ts` in Visual Studio Code.

- Visual Studio Code: Shows contents of `index.ts` file.
- Success story:
  - We can expect the user to open `index.ts` since he already suspects a problem within this file as stated in the original task.

Navigate to Operator

Move cursor the `flatMap` operator on Line 18.

- Visual Studio Code: Shows code actions icon in front of Line 18.
- Success story:



```
You, 5 months ago | 1 author (You)
1  import { merge, Observable } from 'rxjs';
2  import { flatMap, map, startWith, tap } from 'rxjs/operators';
3  import onReady from '../shared/onReady';      You, 5 months ago • Add Better Tests for Problem 1
4  import counter from '../shared/counter';
5  import { default as createUI, Events, Update } from './ui';
6
7  export function createEngine(
8    { increment, decrement, input, reset }: Events,
9    { showValue, setButtonsEnabled, setInput }: Update
10 ): Observable<string> {
11   return reset.pipe(
12     startWith(null),
13     tap(() => {
14       setButtonsEnabled(true);
15       setInput('');
16       showValue('');
17     }),
18     flatMap(() =>
19       merge(
20         input.pipe(tap(() => setButtonsEnabled(false))),
21         counter(increment, decrement).pipe(map((count) => `${count}`))
22       )
23     )
24   );
25 }
26
27 export default function problem1(): HTMLElement {
28   const [ui, events, update] = createUI();
29   const { showValue } = update;
30   const engine = createEngine(events, update);
31
32   engine.subscribe((v) => {
33     showValue(v);
34   });
35
36   return ui;
37 }
```

Figure 1: Visual Studio Code after opening the `index.ts` file.

    – The original task clearly describes the hypothesis regarding this line/piece of source code. Hence, navigating here seems the natural course of action for the user.

Open Code Actions

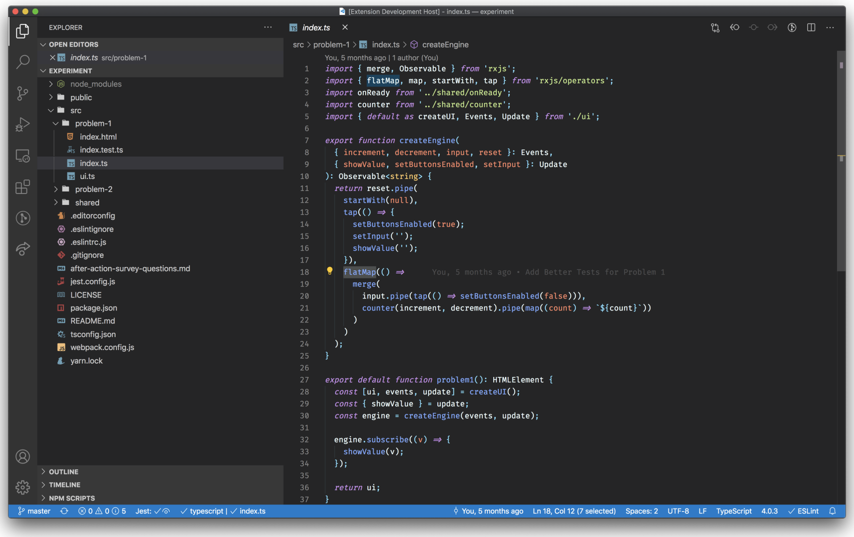
Open the code actions menu by clicking the yellow light bulb icon.

- Visual Studio Code: Shows available code actions.
- Failure story:
    - Will the user know that the correct action is available?
        - The user might know code actions for providing options to refactor a piece of code or quick fixes for code linting problems. It is questionable if he will expect functionality to inspect parts of a data flow graph here.

Create Probe for Operator

Select "Probe Observable..." code action from the related menu.

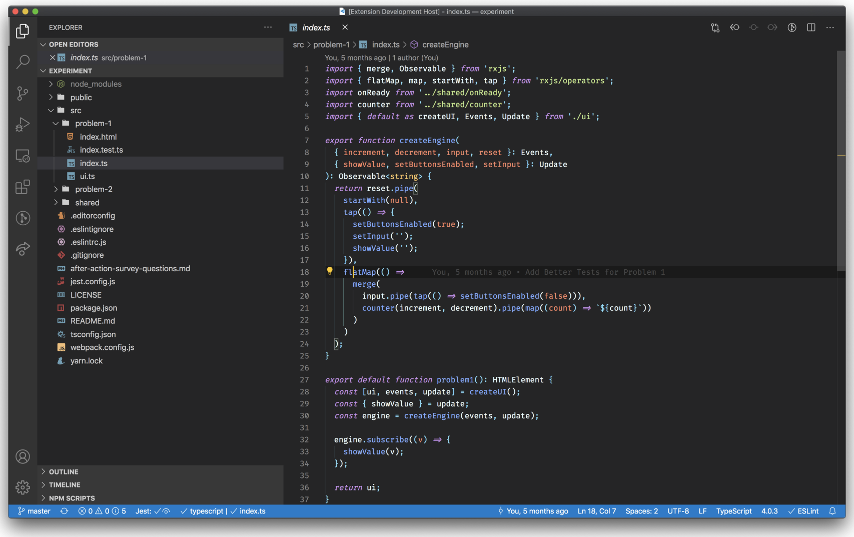
- Visual Studio Code: Adds `flatMap` operator on Line 18 to "Observables" list in debugging view.
- Failure story:



```
src > problem-1 > index.ts > createEngine
You, 5 months ago | 1 author (You)
 1  import { merge, Observable } from 'rxjs';
 2  import { flatMap, map, startWith, tap } from 'rxjs/operators';
 3  import onReady from '../shared/onReady';
 4  import counter from '../shared/counter';
 5  import { default as createUI, Events, Update } from './ui';
 6
 7  export function createEngine(
 8    { increment, decrement, input, reset }: Events,
 9    { showValue, setButtonsEnabled, setInput }: Update
10  ): Observable<string> {
11    return reset.pipe(
12      startWith(null),
13      tap(() => {
14        setButtonsEnabled(true);
15        setInput('');
16        showValue('');
17      }),
18      flatMap(() =>       You, 5 months ago • Add Better Tests for Problem 1
19        merge(
20          input.pipe(tap(() => setButtonsEnabled(false))),
21          counter(increment, decrement).pipe(map((count) => `${count}`))
22        )
23      )
24    );
25  }
26
27  export default function problem1(): HTMLElement {
28    const [ui, events, update] = createUI();
29    const { showValue } = update;
30    const engine = createEngine(events, update);
31
32    engine.subscribe((v) => {
33      showValue(v);
34    });
35
36    return ui;
37  }
```

```
You, 5 months ago   Ln 18, Col 12 (7 selected)   Spaces: 2   UTF-8   LF   TypeScript   4.0.3   ✓ ESLint
```

Figure 2: Visual Studio Code after navigating cursor to the `flatMap` operator on Line 18.



```
src > problem-1 > index.ts > createEngine
You, 5 months ago | 1 author (You)
 1  import { merge, Observable } from 'rxjs';
 2  import { flatMap, map, startWith, tap } from 'rxjs/operators';
 3  import onReady from '../shared/onReady';
 4  import counter from '../shared/counter';
 5  import { default as createUI, Events, Update } from './ui';
 6
 7  export function createEngine(
 8    { increment, decrement, input, reset }: Events,
 9    { showValue, setButtonsEnabled, setInput }: Update
10  ): Observable<string> {
11    return reset.pipe(
12      startWith(null),
13      tap(() => {
14        setButtonsEnabled(true);
15        setInput('');
16        showValue('');
17      }),
18      flatMap(() =>       You, 5 months ago • Add Better Tests for Problem 1
19        merge(
20          input.pipe(tap(() => setButtonsEnabled(false))),
21          counter(increment, decrement).pipe(map((count) => `${count}`))
22        )
23      )
24    );
25  }
26
27  export default function problem1(): HTMLElement {
28    const [ui, events, update] = createUI();
29    const { showValue } = update;
30    const engine = createEngine(events, update);
31
32    engine.subscribe((v) => {
33      showValue(v);
34    });
35
36    return ui;
37  }
```

```
You, 5 months ago   Ln 18, Col 7   Spaces: 2   UTF-8   LF   TypeScript   4.0.3   ✓ ESLint
```

Figure 3: Visual Studio Code indicating available code actions on Line 18 using a yellow light bulb icon.

    - If the correct action is taken, will the user see that things are going ok?
        - The “Observables” list is part of the debugging view of Visual Studio Code. The user will not get any feedback that his action “Probe Observable...” was successful without changing the view manually to debugging and expanding the “Observables” panel in the lower left.

Open Observable Probe Monitor

Open the “Observable Probe Monitor” view using command palette.

- Visual Studio Code: Shows empty “Observable Probe Monitor” view
- Failure story:
    - Will the user know that the correct action is available?
        - The user might not be aware that the “Observable Probe Monitor” view is hidden within the command palette. Hence, they might feel lost after adding the observable probe in the previous step.
    - If the correct action is taken, will the user see that things are going ok?
        - The user might get confused by the “Observable Probe Monitor” being blank by default.

Launch Application

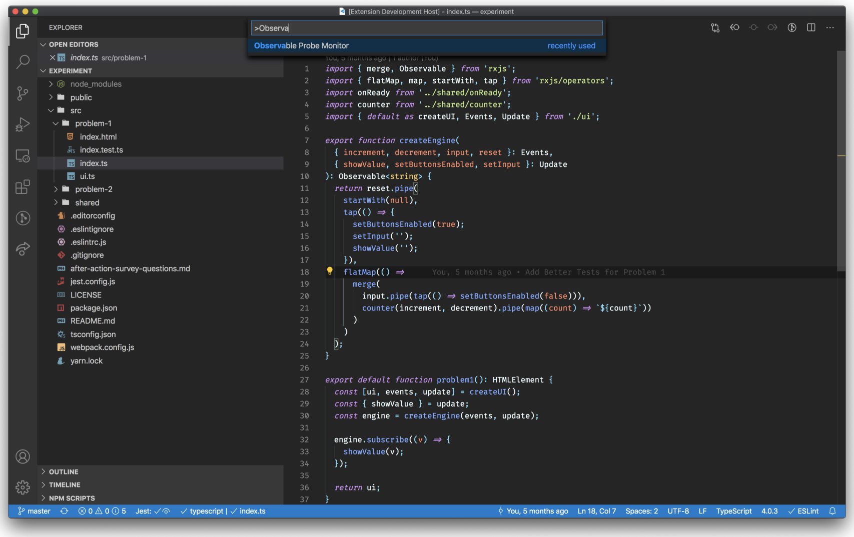
Execute “Problem 1” launch configuration

- Visual Studio Code: Opens default browser showing “Problem 1”
- Default Browser: Shows “Problem 1” UI
- Success story:
    - The users previous experience with Visual Studio Code launch configuration allows assuming this the natural course of action in order to prepare himself for further inspection of the application.

Interact with Application

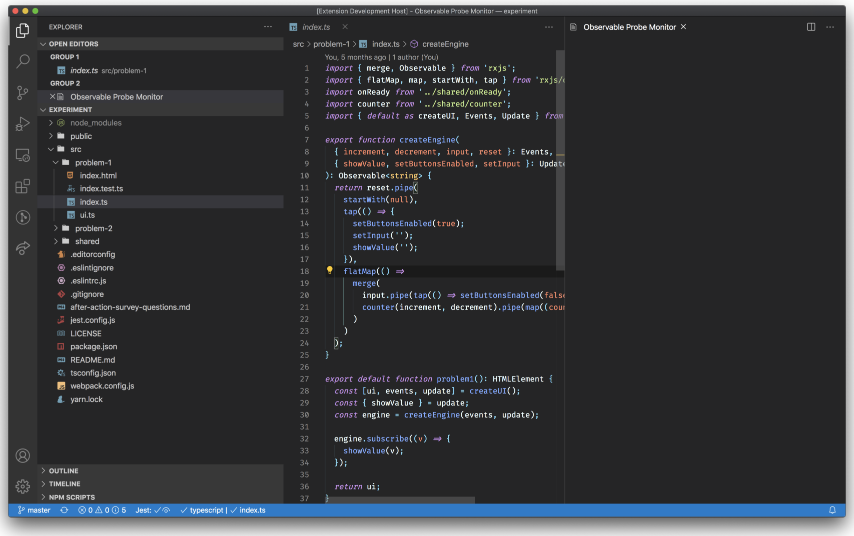
Interact with “Problem 1” in the default browser.

- Visual Studio Code: “Observable Probe Monitor” provides live telemetry information about values and life cycle events produced by the `flatMap` operator.
- Failure story:



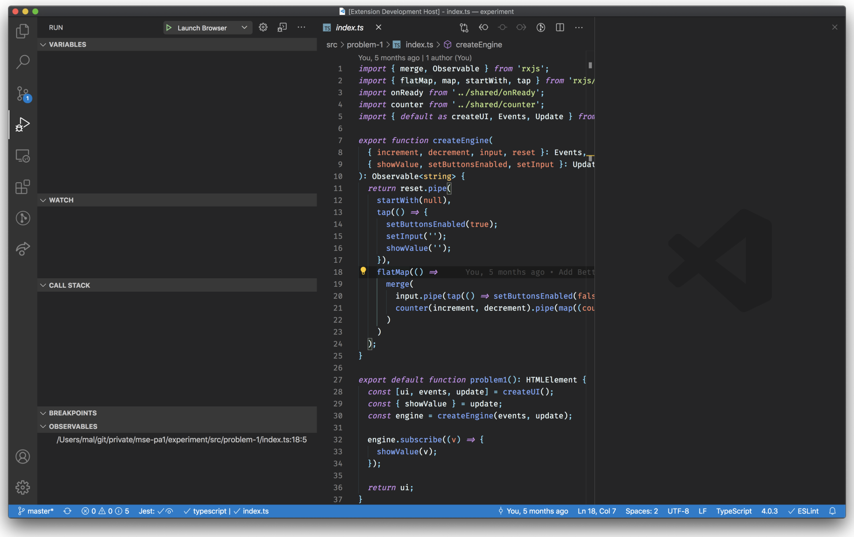
```
import { merge, Observable } from 'rxjs';
import { flatMap, map, startWith, tap } from 'rxjs/operators';
import onReady from '../shared/onReady';
import counter from '../shared/counter';
import { default as createUI, Events, Update } from './ui';

export function createEngine(
  { increment, decrement, input, reset }: Events,
  { showValue, setButtonsEnabled, setInput }: Update
): Observable<string> {
  return reset.pipe(
    startWith(null),
    tap(() => {
      setButtonsEnabled(true);
      setInput('');
      showValue('');
    }),
    flatMap(() =>        You, 5 months ago • Add Better Tests for Problem 1
      merge(
        input.pipe(tap(() => setButtonsEnabled(false))),
        counter(increment, decrement).pipe(map((count) => `${count}`))
      )
    )
  );
}

export default function problem1(): HTMLElement {
  const [ui, events, update] = createUI();
  const { showValue } = update;
  const engine = createEngine(events, update);

  engine.subscribe((v) => {
    showValue(v);
  });

  return ui;
}
```

Figure 4: Visual Studio Codes command palette menu showing the “Observable Probe Monitor” command.



```
import { merge, Observable } from 'rxjs';
import { flatMap, map, startWith, tap } from 'rxjs/
import onReady from '../shared/onReady';
import counter from '../shared/counter';
import { default as createUI, Events, Update } from

export function createEngine(
  { increment, decrement, input, reset }: Events,
  { showValue, setButtonsEnabled, setInput }: Updat
): Observable<string> {
  return reset.pipe(
    startWith(null),
    tap(() => {
      setButtonsEnabled(true);
      setInput('');
      showValue('');
    }),
    flatMap(() =>
      merge(
        input.pipe(tap(() => setButtonsEnabled(fals
        counter(increment, decrement).pipe(map((cou
      )
    )
  );
}

export default function problem1(): HTMLElement {
  const [ui, events, update] = createUI();
  const { showValue } = update;
  const engine = createEngine(events, update);

  engine.subscribe((v) => {
    showValue(v);
  });

  return ui;
}
```

Figure 5: Visual Studio Code showing the empty Observable Probe Monitor on the right pane.



```
/Users/mal/git/private/mse-pa1/experiment/src/problem-1/index.ts:18:5
```

```
You, 5 months ago | 1 author (You)
import { merge, Observable } from 'rxjs';
import { flatMap, map, startWith, tap } from 'rxjs/
import onReady from '../shared/onReady';
import counter from '../shared/counter';
import { default as createUI, Events, Update } from

export function createEngine(
  { increment, decrement, input, reset }: Events,
  { showValue, setButtonsEnabled, setInput }: Updat
): Observable<string> {
  return reset.pipe(
    startWith(null),
    tap(() => {
      setButtonsEnabled(true);
      setInput('');
      showValue('');
    }),
    flatMap(() =>          You, 5 months ago • Add Bett
      merge(
        input.pipe(tap(() => setButtonsEnabled(fals
        counter(increment, decrement).pipe(map((cou
      )
    )
  );
}

export default function problem1(): HTMLElement {
  const [ui, events, update] = createUI();
  const { showValue } = update;
  const engine = createEngine(events, update);

  engine.subscribe((v) => {
    showValue(v);
  });

  return ui;
}
```

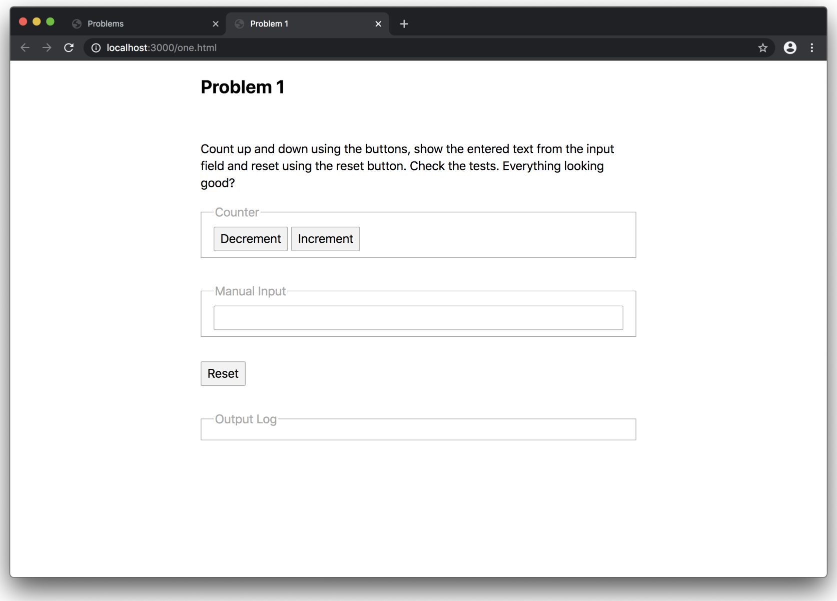
Figure 6: Visual Studio Code showing the debugging view after launching “Problem 1.”

- Will the user know that the correct action will achieve the desired effect?
    - ∗ The user might not be aware that he is expected to interact with “Problem 1” in the default browser in order to get live feedback in the “Observable Probe Monitor.”
- If the correct action is taken, will the user see that things are going ok?
    - ∗ The default browser might overlay Visual Studio Code and the “Observable Probe Monitor” view. This is why the user might miss the live trace of values and life cycle events displayed in the “Observable Probe Monitor.”

Interpret Runtime Behavior

Interpret the live trace of emitted values and life cycle events in the “Observable Probe Monitor” view

- Visual Studio Code: Provides detail information to a traced item
- Success story:
    - The original task states that the user is interested in more close information regarding the `flatMap` operator. Since the “Observable

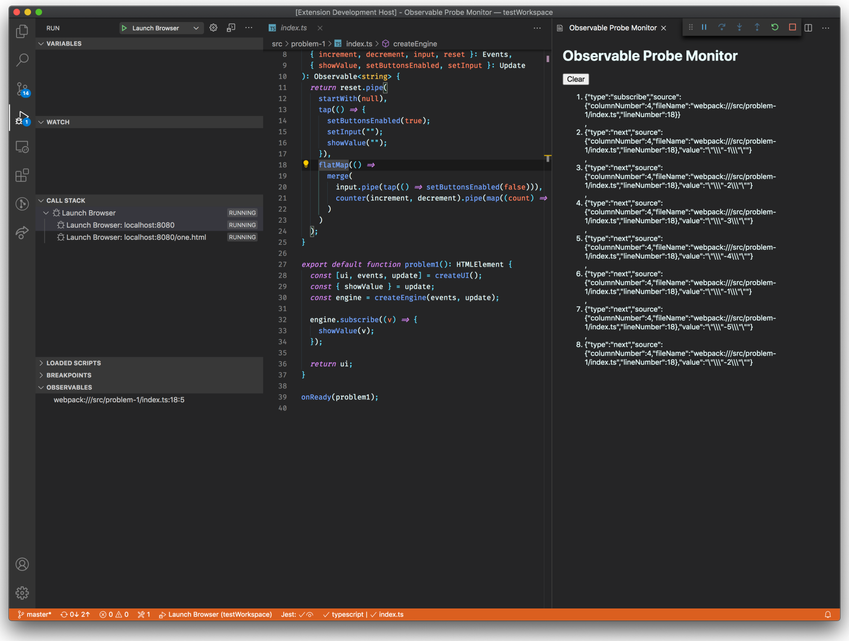


Problem 1

Count up and down using the buttons, show the entered text from the input field and reset using the reset button. Check the tests. Everything looking good?

Figure 7: Google Chrome displaying the user interface of "Problem 1" ready to receive interactions.

Probe Monitor" provide such information in real-time, we can expect the user to use this information accordingly.


[Extension Development Host] - Observable Probe Monitor — testWorkspace
RUN
Launch Browser
VARIABLES
WATCH
CALL STACK
Launch Browser RUNNING
Launch Browser: localhost:8080 RUNNING
Launch Browser: localhost:8080/one.html RUNNING
LOADED SCRIPTS
BREAKPOINTS
OBSERVABLES
webpack:///src/problem-1/index.ts:18:5
index.ts
src > problem-1 > index.ts > createEngine
Observable Probe Monitor
Clear

```
  { increment, decrement, input, reset }: Events,
  { showValue, setButtonsEnabled, setInput }: Update
): Observable<string> {
  return reset.pipe(
    startWith(null),
    tap(() => {
      setButtonsEnabled(true);
      setInput("");
      showValue("");
    }),
    flatMap(() =>
      merge(
        input.pipe(tap(() => setButtonsEnabled(false))),
        counter(increment, decrement).pipe(map((count) =>
      )
    )
  );
}

export default function problem1(): HTMLElement {
  const [ui, events, update] = createUI();
  const { showValue } = update;
  const engine = createEngine(events, update);

  engine.subscribe((v) => {
    showValue(v);
  });

  return ui;
}

onReady(problem1);
```

Observable Probe Monitor

```
1. {"type":"subscribe","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18}}
   ,
2. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-1\\\"\""}
   ,
3. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-2\\\"\""}
   ,
4. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-3\\\"\""}
   ,
5. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-4\\\"\""}
   ,
6. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-1\\\"\""}
   ,
7. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-5\\\"\""}
   ,
8. {"type":"next","source":
   {"columnNumber":4,"fileName":"webpack:///src/problem-
   1/index.ts","lineNumber":18},"value":"\"\\\"-2\\\"\""}
```

Figure 8: Visual Studio Code showing live telemetry in the "Observable Probe Monitor."

Failure Stories

This is a summary of all failure stories identified during the cognitive walkthrough.

Step	Failure Story
Open Code Actions	The user might know code actions for providing options to refactor a piece of code or quick fixes for code linting problems. It is questionable if he will expect functionality to inspect parts of a data flow graph here.
Create Probe for Operator	The “Observables” list is part of the debugging view of Visual Studio Code. The user will not get any feedback that his action “Probe Observable...” was successful without changing the view manually to debugging and expanding the “Observables” panel in the lower left.
Open Observable Probe Monitor	The user might not be aware that the “Observable Probe Monitor” view is hidden within the command palette. Hence, they might feel lost after adding the observable probe in the previous step.
Open Observable Probe Monitor	The user might get confused by the “Observable Probe Monitor” being blank by default.
Interact with Application	The user might not be aware that he is expected to interact with “Problem 1” in the default browser in order to get live feedback in the “Observable Probe Monitor.”
Interact with Application	The default browser might overlay Visual Studio Code and the “Observable Probe Monitor” view. This is why the user might miss the live trace of values and life cycle events displayed in the “Observable Probe Monitor.”

Usability Test

Observed Issues

These are all usability issues identified during the usability test sessions.

Participant(s)	Phase	Task	Problem
P2, P3	Instrument Hypothesis	Environment Setup	Subject starts the application in debugging mode, even though they have started it before already.
P2, P3	Instrument Hypothesis	Manage Log Points	Subject unable to find log point list in debugging view.
P2	Instrument Hypothesis	Manage Log Points	Subject unable to identify already defined log points.
P2	Instrument Hypothesis	Interpret Log	Subject cannot find “Clear” button to clear the log before starting a new debugging iteration.
P3	Instrument Hypothesis	Manage Log Points	Subject cannot add log point to an observable.
P3	Instrument Hypothesis	Manage Log Points	Subject cannot add log point by clicking the editors gutter.*(Regular break points are added here)*
P2, P3	Test Hypothesis	Interpret Log	Subject has difficulties to make a connection from a log point to the generated log entry.
P2, P3	Test Hypothesis	Interpret Log	Subject interprets logged value as the “input” of the instrumented operator.
P2	Test Hypothesis	Interpret Log	Subject is overwhelmed by multiple log entries generated by multiple log points.
P3	Test Hypothesis	Interpret Log	Subject does not see log entries when running the unit test suite.

Marketplace Presence

The next page shows an excerpt of the Visual Studio Marketplace presence of the “RxJS Debugging for Visual Studio Code” extension as of 2021-12-23.

Visual Studio | **Marketplace** Sign in

Visual Studio Code > Debuggers > RxJS Debugging for Visual Studio Code New to Visual Studio Code? Get it now.

RxJS Debugging for Visual Studio Code

[redacted] | 922 installs | ★★★★★ (1) | Free

Add non-intrusive debugging capabilities for RxJS applications to Visual Studio Code.

Install Trouble Installing?

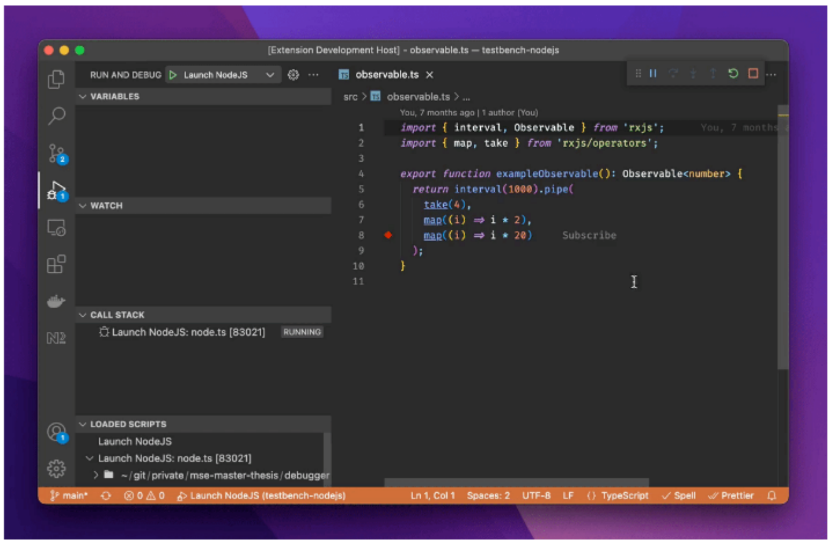
Overview Version History Q & A Rating & Review

RxJS Debugging for Visual Studio Code

VS Marketplace v1.1.1 Follow @rxjsdebuggung

> Never, ever use `tap(console.log)` again.

Add non-intrusive debugging capabilities for RxJS applications to Visual Studio Code.


[Extension Development Host] - observable.ts — testbench-nodejs
RUN AND DEBUG
Launch NodeJS
VARIABLES
WATCH
CALL STACK
Launch NodeJS: node.ts [83021] RUNNING
LOADED SCRIPTS
Launch NodeJS
Launch NodeJS: node.ts [83021]
~/git/private/mse-master-thesis/debugger
observable.ts
src > observable.ts > ...
You, 7 months ago | 1 author (You)
main* 0 0 Launch NodeJS (testbench-nodejs) Ln 1, Col 1 Spaces: 2 UTF-8 LF TypeScript Spell Prettier

```typescript
import { interval, Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';

export function exampleObservable(): Observable<number> {
  return interval(1000).pipe(
    take(4),
    map((i) => i * 2),
    map((i) => i * 20)    Subscribe
  );
}
```

Features

- RxJS debugging, fully integrated with Visual Studio Code
- Works with RxJS 6.6.7 and newer
- Support for Node.js and Webpack-based RxJS applications

Requirements

- Visual Studio Code 1.61 or newer
- RxJS 6.6.7 or newer
- To debug NodeJS-based applications:
  - Node.js 12 or newer
- To debug Webpack-based web applications:
  - Webpack 5.60.0 or newer
  - The latest @rxjs-debugging/runtime-webpack Webpack plugin (see here for setup instructions)

Categories

Debuggers

Tags

javascript javascriptreact typescript typescriptreact

Works with

Universal

Resources

Issues
Repository
Homepage
License
Changelog
Download Extension

Project Details

[redacted]
Last Commit: 2 weeks ago
11 Pull Requests
19 Open Issues

More Info

Version	1.1.1
Released on	17/05/2021, 15:58:57
Last updated	08/12/2021, 15:30:44
Publisher	[redacted]
Unique Identifier	[redacted]
Report	Report Abuse

References

[1] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN international workshop on reactive and event-based languages and systems* (REBLS 2020), Association for Computing Machinery, 15–24. DOI:https://doi.org/10.1145/3427763.3428313

[2] Cathleen Wharton, John Rieman, Lewis Clayton, and Peter Polson. 1994. *The cognitive walkthrough: A practitioner's guide.* Institute of Cognitive Science, University of Colorado.

B Comparative User Journey

The Comparative User Journey is publicly accessible on alabor.me and GitHub:

- https://alabor.me/research/user-journey-debugging-of-rxjs-based-applications/
- https://github.com/swissmanu/alabor.me/tree/master/research/user-journey-debugging-of-rxjs-based-applications

User Journey: Debugging of RxJS-Based Applications

Manuel Alabor
Eastern Switzerland University of Applied Sciences
manuel.alabor@ost.ch

This document describes two user journeys of Frank Flow, a software engineer working on an application implemented using RxJS. The application has a web-based user interface and was part of a study by Alabor et al. [1]. Frank will use different debugging techniques and utilities to solve the task given to him. The journeys follow the debugging process model proposed by Layman et al. [2].

Actor

Frank Flow started to work for ReactiBank 2 years ago as a frontend software engineer. As part of a small, interdisciplinary team of 7 people, Frank' and his team are responsible for developing and maintaining a trading application. This application relies heavily on real-time data, so the group decided to use reactive programming principles throughout the application. Frank knows traditional programming paradigms and the related debugging tools from his studies and personal experiences. He built up knowledge on RP and RxJS for the frontend part of their application after joining the team quickly, however.

Today, Frank uses RxJS efficiently to build new features. He can solve simple problems reported by the product owner on his own. Working on more complicated issues is still something Frank struggles with: He often feels like his knowledge of traditional programming techniques and its debugging utilities are not enough. These tools feel "out of place" to him. All to often, they do not provide any helpful insights and turn out to be time-sink. Frank does not like that, eventually, he has to consult one of his colleagues who have experience in RxJS for a longer time.

Context

Frank participates in a study about RxJS-specific debugging techniques. He is asked to work on an application allowing a user to change and display a value. The user can achieve a change of the value either by two buttons to increase and decrease a numeric value or by entering an arbitrary text in an input field. The two buttons get disabled once a text is entered. There is a third button labeled "Reset", which clears all user input and reverts the application to its initial state.

Goal

Frank receives a bug report for the application: Once a user had clicked the reset button, the application started to behave strangely by showing multiple values in quick succession when the increase or decrease button was clicked. It is Franks task to analyse the bug and find a solution.

User Journey

Beginning from the same point, we are going to present two distinct user journeys in which Frank uses different tools to achieve his goal:

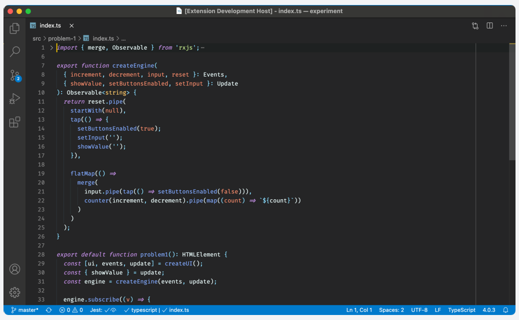
Branch A on the left describes a debugging session using traditional debugging techniques and practices as described by Alabor et al. [1]. Frank will use imperative debugging utilities built-in to Visual Studio Code and manual code modification to reach his goal.

Branch B on the right describes how Frank uses the prototype of an extension for Visual Studio Code providing a less invasive debugging technique for RxJS-based source code.

Step 1 **Build Hypothesis**

After Frank recreated the behavior reported in the bug report, he starts analyzing the source code of the application. In this process, he formulates his first hypothesis about what could cause the unexpected behavior: Frank suspects the `flatMap` operator in Line 19 to be responsible.

Frank wants to have a closer look on the operators runtime behavior, which is why he decides to use debugging utilities.

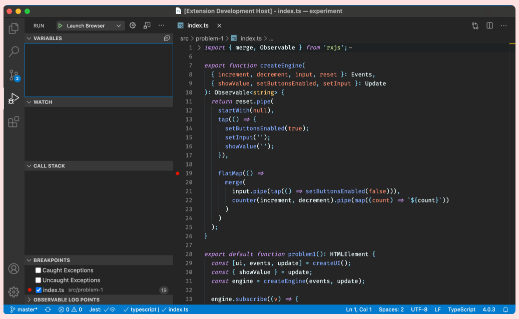

[Extension Development Host] - index.ts — experiment
index.ts
src > problem-1 > index.ts > ...

```
 1 > import { merge, Observable } from 'rxjs';…
 6
 7   export function createEngine(
 8     { increment, decrement, input, reset }: Events,
 9     { showValue, setButtonsEnabled, setInput }: Update
10   ): Observable<string> {
11     return reset.pipe(
12       startWith(null),
13       tap(() => {
14         setButtonsEnabled(true);
15         setInput('');
16         showValue('');
17       }),
18
19       flatMap(() =>
20         merge(
21           input.pipe(tap(() => setButtonsEnabled(false))),
22           counter(increment, decrement).pipe(map((count) => `${count}`))
23         )
24       )
25     );
26   }
27
28   export default function problem1(): HTMLElement {
29     const [ui, events, update] = createUI();
30     const { showValue } = update;
31     const engine = createEngine(events, update);
32
33     engine.subscribe((v) => {
```

master* ⊗ 0 ⚠ 0 Jest: ✓ typescript | ✓ index.ts Ln 1, Col 1 Spaces: 2 UTF-8 LF TypeScript 4.0.3

Step 2.A **Instrument Hypothesis**

Using the built-in debugger of Visual Studio Code, Frank adds a breakpoint to Line 19 where the `flatMap` operator is called with the intention to stop the program execution every time a values "flows" through the operator.

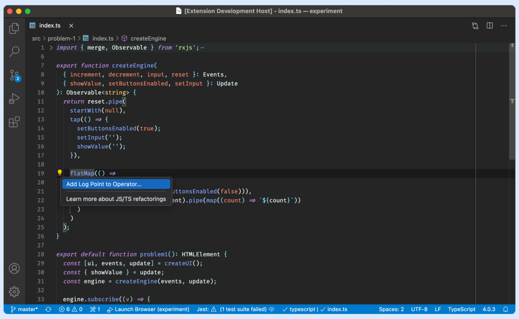


[Extension Development Host] - index.ts — experiment

```
 1 > import { merge, Observable } from 'rxjs';…
 6
 7   export function createEngine(
 8     { increment, decrement, input, reset }: Events,
 9     { showValue, setButtonsEnabled, setInput }: Update
10   ): Observable<string> {
11     return reset.pipe(
12       startWith(null),
13       tap(() => {
14         setButtonsEnabled(true);
15         setInput('');
16         showValue('');
17       }),
18
19       flatMap(() =>
20         merge(
21           input.pipe(tap(() => setButtonsEnabled(false))),
22           counter(increment, decrement).pipe(map((count) => `${count}`))
23         )
24       )
25     );
26   }
27
28   export default function problem1(): HTMLElement {
29     const [ui, events, update] = createUI();
30     const { showValue } = update;
31     const engine = createEngine(events, update);
32
33     engine.subscribe((v) => {
```

Step 2.B **Instrument Hypothesis**

Frank navigates to the `flatMap` operator in Line 19 and selects the "Add Log Point to Operator..." code action provided by the Visual Studio Code extension.

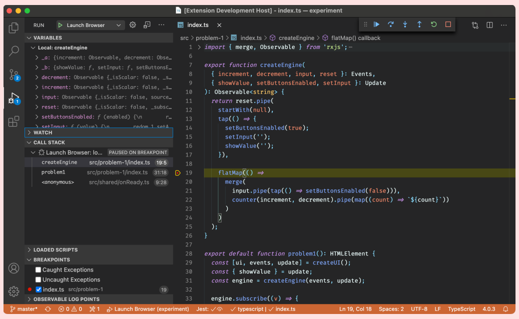

Add Log Point to Operator...
Learn more about JS/TS refactorings

Step 3.A **Test Hypothesis**

Frank launches the application. Before the web browser can display anything, the debugger in Visual Studio Code halts the program execution at the breakpoint in Line 19.

Contrary to Franks expectation, the application was already paused during the creation of the `flatMap` operator rather than when a value was processed by it.

Frank resumes the program execution, though the breakpoint never pauses the application again.

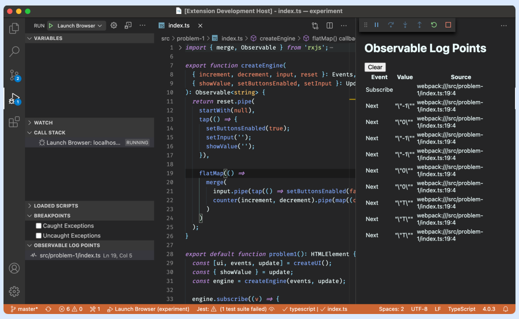


Step 3.B **Test Hypothesis**

Frank launches the application. While reproducing the reported bug in the browser, the extension produces a log of all events detected at the `flatMap` operator in Line 19.

Frank recognizes a peculiar pattern: After the reset button was clicked, the log point reports multiple values emitted for each click on the increase/decrease button.

This confirms Franks hypothesis about the `flatMap` operator: After the user clicked the reset button, the operator emits multiple values.

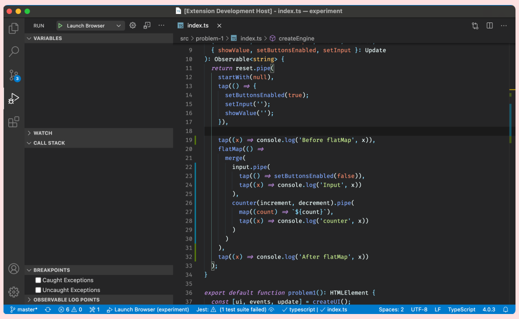


Observable Log Points

Event	Value	Source
Subscribe		webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"T\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"T\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"T\""	webpack:///src/problem-1/index.ts:19:4

Step 4.A **Instrument Hypothesis**

After Franks first failed instrumentation attempt using breakpoints, he decides to add trace log statements using the `tap` operator manually. He adds multiple of them on Lines 19, 24, 28, and 32.


[Extension Development Host] - index.ts — experiment

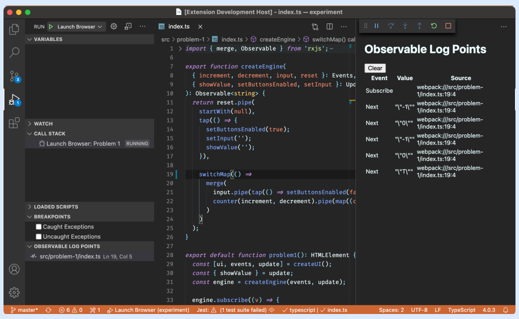
src > problem-1 > index.ts > createEngine

```
 9    { showValue, setButtonsEnabled, setInput }: Update
10  ): Observable<string> {
11    return reset.pipe(
12      startWith(null),
13      tap(() => {
14        setButtonsEnabled(true);
15        setInput('');
16        showValue('');
17      }),
18
19      tap((x) => console.log('Before flatMap', x)),
20      flatMap(() =>
21        merge(
22          input.pipe(
23            tap(() => setButtonsEnabled(false)),
24            tap((x) => console.log('Input', x))
25          ),
26          counter(increment, decrement).pipe(
27            map((count) => `${count}`),
28            tap((x) => console.log('counter', x))
29          )
30        )
31      ),
32      tap((x) => console.log('After flatMap', x))
33    );
34  }
35
36  export default function problem1(): HTMLElement {
37    const [ui, events, update] = createUI();
```

Step 4.B **Resolve Bug**

After researching the RxJS documentation, Frank realizes that the `flatMap` operator [3] does not unsubscribe observables it created earlier. This sounds like a reasonable explanation for the observed behavior.

Frank replaces the faulty operator with the `switchMap` [4] operator.


[Extension Development Host] - index.ts — experiment

src > problem-1 > index.ts > createEngine > switchMap() cal

```
 1  > import { merge, Observable } from 'rxjs';
 6
 7  export function createEngine(
 8    { increment, decrement, input, reset }: Events,
 9    { showValue, setButtonsEnabled, setInput }: Upd
10  ): Observable<string> {
11    return reset.pipe(
12      startWith(null),
13      tap(() => {
14        setButtonsEnabled(true);
15        setInput('');
16        showValue('');
17      }),
18
19      switchMap(() =>
20        merge(
21          input.pipe(tap(() => setButtonsEnabled(fa
22          counter(increment, decrement).pipe(map((c
23        )
24      )
25    );
26  }
27
28  export default function problem1(): HTMLElement {
29    const [ui, events, update] = createUI();
30    const { showValue } = update;
31    const engine = createEngine(events, update);
32
33    engine.subscribe((v) => {
```

Observable Log Points

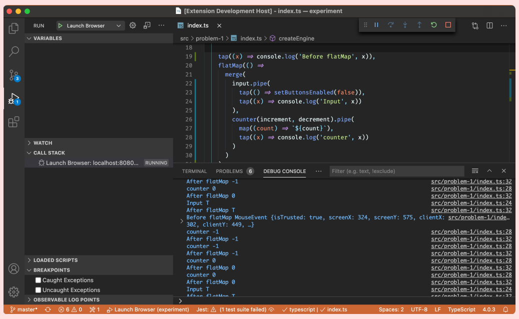
Event	Value	Source
Subscribe		webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"T\""	webpack:///src/problem-1/index.ts:19:4

Step 5.A **Test Hypothesis**

Frank launches the application again. The manually added code generates the expected trace log in the debuggers console as Frank executes the steps necessary to recreate the reported bug.

Once the application produced enough logs, Frank starts to analyse them. Even though it is hard to reassign a log entry to a piece of code and a related action, Frank recognizes a peculiar pattern after some time: After the reset button was clicked, the log statement on Line 32 is executed multiple times, even though the increase/decrease button gets clicked only once.

This confirms Franks hypothesis about the `flatMap` operator: After the user clicked the reset button, the operator emits multiple values.


[Extension Development Host] - index.ts — experiment
index.ts
src > problem-1 > index.ts > createEngine
RUNNING
TERMINAL
PROBLEMS 6
DEBUG CONSOLE
Filter (e.g. text, !exclude)

```
    tap((x) => console.log('Before flatMap', x)),
    flatMap(() =>
      merge(
        input.pipe(
          tap(() => setButtonsEnabled(false)),
          tap((x) => console.log('Input', x))
        ),
        counter(increment, decrement).pipe(
          map((count) => `${count}`),
          tap((x) => console.log('counter', x))
        )
      )
```

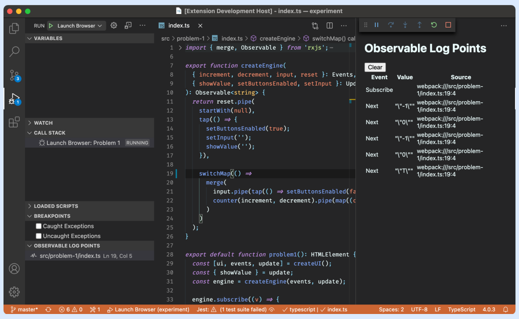
```
After flatMap -1                                              src/problem-1/index.ts:32
counter 0                                                     src/problem-1/index.ts:28
After flatMap 0                                               src/problem-1/index.ts:32
Input T                                                       src/problem-1/index.ts:24
After flatMap T                                               src/problem-1/index.ts:32
> Before flatMap MouseEvent {isTrusted: true, screenX: 324, screenY: 575, clientX: src/problem-1/inde…
302, clientY: 449, …}
counter -1                                                    src/problem-1/index.ts:28
After flatMap -1                                              src/problem-1/index.ts:32
counter -1                                                    src/problem-1/index.ts:28
After flatMap -1                                              src/problem-1/index.ts:32
counter 0                                                     src/problem-1/index.ts:28
After flatMap 0                                               src/problem-1/index.ts:32
counter 0                                                     src/problem-1/index.ts:28
After flatMap 0                                               src/problem-1/index.ts:32
Input T                                                       src/problem-1/index.ts:24
```

Step 5.B **Verify**

Frank launches the application again and repeats the steps necessary to reproduce the reported bug.

The application appears to work correctly now. A quick look at the log point monitor confirms that the previously observed behavior of multiple values emitted is fixed.

Frank has reached his goal successfully.


[Extension Development Host] - index.ts — experiment
index.ts
src > problem-1 > index.ts > createEngine > switchMap() cal
RUNNING
Clear

```
import { merge, Observable } from 'rxjs';…

export function createEngine(
  { increment, decrement, input, reset }: Events,
  { showValue, setButtonsEnabled, setInput }: Upd
): Observable<string> {
  return reset.pipe(
    startWith(null),
    tap(() => {
      setButtonsEnabled(true);
      setInput('');
      showValue('');
    }),

    switchMap(() =>
      merge(
        input.pipe(tap(() => setButtonsEnabled(fa
        counter(increment, decrement).pipe(map((c
      )
    )
  );
}

export default function problem1(): HTMLElement {
  const [ui, events, update] = createUI();
  const { showValue } = update;
  const engine = createEngine(events, update);

  engine.subscribe((v) => {
```

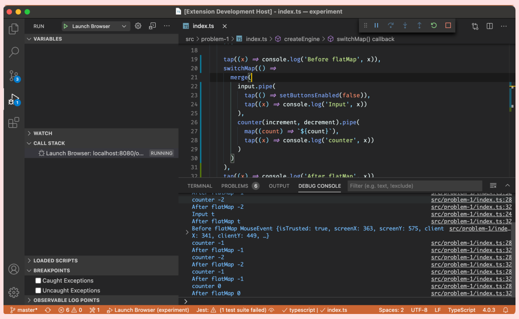
Observable Log Points

Event	Value	Source
Subscribe		webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"-1\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"0\""	webpack:///src/problem-1/index.ts:19:4
Next	"\"T\""	webpack:///src/problem-1/index.ts:19:4

Step 6.A **Resolve Bug**

After researching the RxJS documentation, Frank knows that the `flatMap` operator [3] does not unsubscribe observables it created earlier. This sounds like a reasonable explanation for the observed behavior.

Frank replaces the faulty operator with the `switchMap` [4] operator.



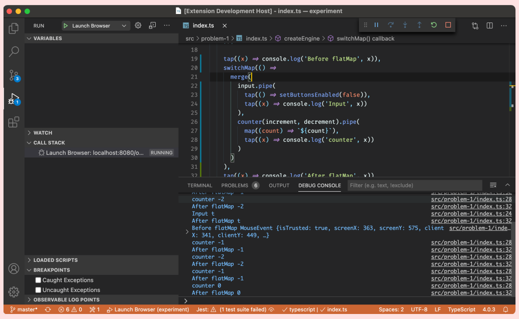
```
    tap((x) => console.log('Before flatMap', x)),
    switchMap(() =>
      merge(
        input.pipe(
          tap(() => setButtonsEnabled(false)),
          tap((x) => console.log('Input', x))
        ),
        counter(increment, decrement).pipe(
          map((count) => `${count}`),
          tap((x) => console.log('counter', x))
        )
      )
    ),
    tap((x) => console.log('After flatMap', x))
```

```
counter -2                                                              src/problem-1/index.ts:28
After flatMap -2                                                        src/problem-1/index.ts:32
Input t                                                                 src/problem-1/index.ts:24
After flatMap t                                                         src/problem-1/index.ts:32
> Before flatMap MouseEvent {isTrusted: true, screenX: 363, screenY: 575, client  src/problem-1/inde…
X: 341, clientY: 449, …}
counter -1                                                              src/problem-1/index.ts:28
After flatMap -1                                                        src/problem-1/index.ts:32
counter -2                                                              src/problem-1/index.ts:28
After flatMap -2                                                        src/problem-1/index.ts:32
counter -1                                                              src/problem-1/index.ts:28
After flatMap -1                                                        src/problem-1/index.ts:32
counter 0                                                               src/problem-1/index.ts:28
After flatMap 0                                                         src/problem-1/index.ts:32
```

Step 7.A **Verify**

Frank launches the application again and repeats the steps necessary to reproduce the reported bug.

The application appears to work correctly now. A look at the trace log confirms that the previously observed behavior of multiple values emitted is gone.



```
    tap((x) => console.log('Before flatMap', x)),
    switchMap(() =>
      merge(
        input.pipe(
          tap(() => setButtonsEnabled(false)),
          tap((x) => console.log('Input', x))
        ),
        counter(increment, decrement).pipe(
          map((count) => `${count}`),
          tap((x) => console.log('counter', x))
        )
      )
    ),
    tap((x) => console.log('After flatMap', x))
```

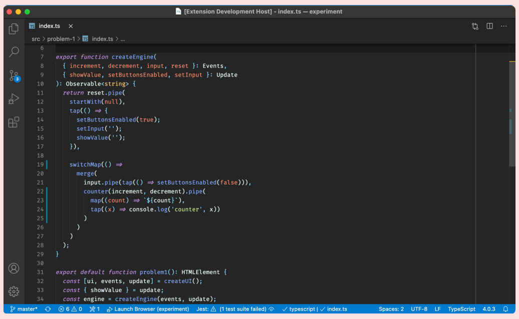
```
counter -2                                                              src/problem-1/index.ts:28
After flatMap -2                                                        src/problem-1/index.ts:32
Input t                                                                 src/problem-1/index.ts:24
After flatMap t                                                         src/problem-1/index.ts:32
> Before flatMap MouseEvent {isTrusted: true, screenX: 363, screenY: 575, client  src/problem-1/inde…
X: 341, clientY: 449, …}
counter -1                                                              src/problem-1/index.ts:28
After flatMap -1                                                        src/problem-1/index.ts:32
counter -2                                                              src/problem-1/index.ts:28
After flatMap -2                                                        src/problem-1/index.ts:32
counter -1                                                              src/problem-1/index.ts:28
After flatMap -1                                                        src/problem-1/index.ts:32
counter 0                                                               src/problem-1/index.ts:28
After flatMap 0                                                         src/problem-1/index.ts:32
```

Step 8.A **Revert Hypothesis Instrumentation**

Frank removes all `tap` operators he added solely for the trace log generation, except the one in Line 24.

Another engineer notices this leftover during the code review, fortunately. Frank removes the forgotten statement afterwards.

Finally, Frank has reached his goal.


[Extension Development Host] - index.ts — experiment
index.ts
src > problem-1 > index.ts > ...

```
 6
 7  export function createEngine(
 8    { increment, decrement, input, reset }: Events,
 9    { showValue, setButtonsEnabled, setInput }: Update
10  ): Observable<string> {
11    return reset.pipe(
12      startWith(null),
13      tap(() => {
14        setButtonsEnabled(true);
15        setInput('');
16        showValue('');
17      }),
18
19      switchMap(() =>
20        merge(
21          input.pipe(tap(() => setButtonsEnabled(false))),
22          counter(increment, decrement).pipe(
23            map((count) => `${count}`),
24            tap((x) => console.log('counter', x))
25          )
26        )
27      )
28    );
29  }
30
31  export default function problem1(): HTMLElement {
32    const [ui, events, update] = createUI();
33    const { showValue } = update;
34    const engine = createEngine(events, update);
```

master* ⊗ 6 ⚠ 0 1 Launch Browser (experiment) Jest: ⚠ (1 test suite failed) ✓ typescript | ✓ index.ts Spaces: 2 UTF-8 LF TypeScript 4.0.3

References

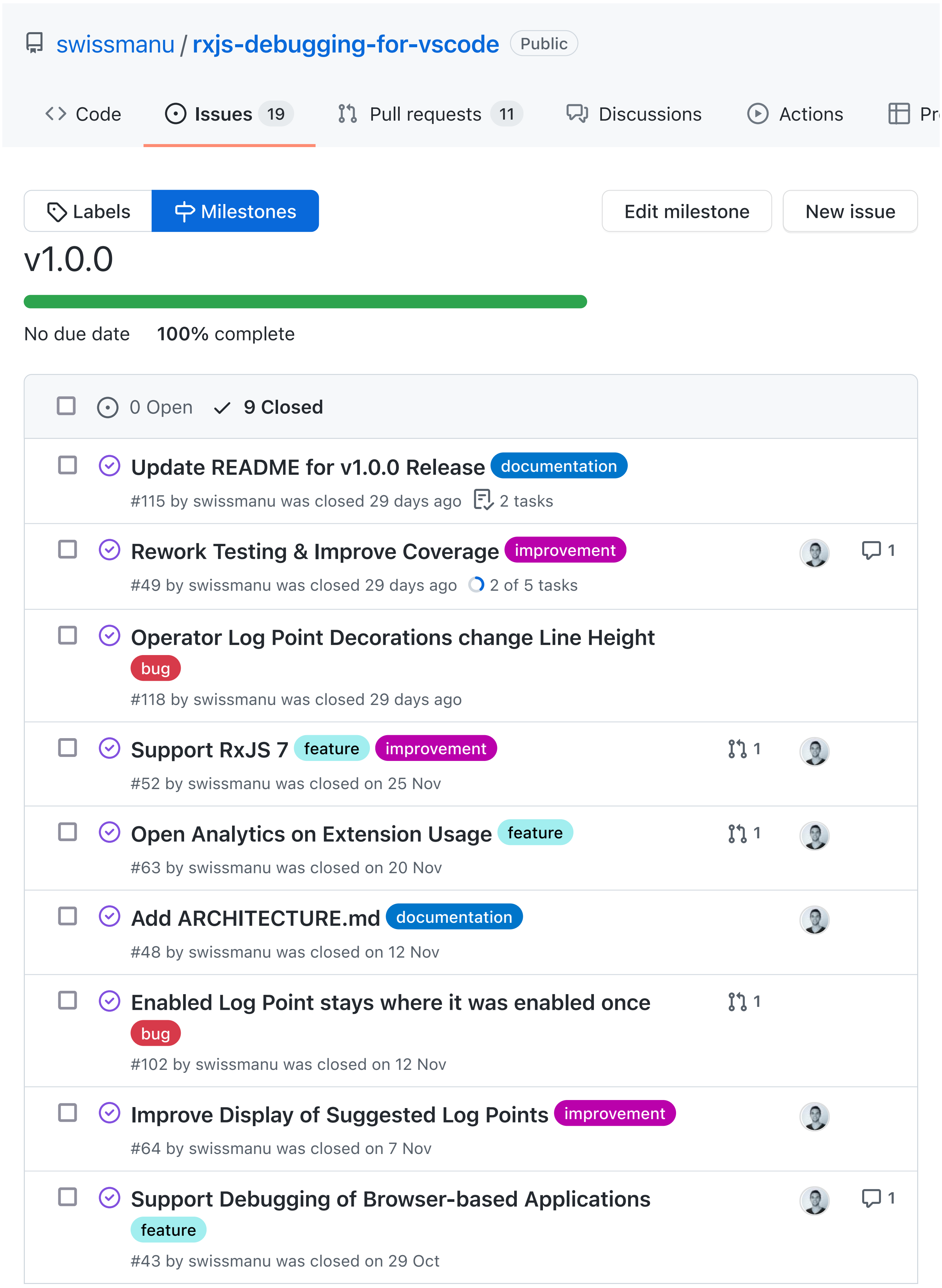
1. Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2020). Association for Computing Machinery, New York, NY, USA, 15–24. DOI: https://doi.org/10.1145/3427763.3428313
2. L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, 2013, pp. 383-392, doi: 10.1109/ESEM.2013.43 .
3. ReactiveX. 2020. RxJS - flatMap. Retrieved 05-December-2020 from https://rxjs.dev /api/operators/flatMap. Version cbc77213e97ecc00d90a65ecf18707b76ebfe7fc.
4. ReactiveX. 2020. RxJS - switchMap. Retrieved 05-December-2020 from https://rxjs.dev /api/operators/switchMap. Version cbc77213e97ecc00d90a65ecf18707b76ebfe7fc.

C RxJS Debugging for vscode

C.1 Major Release Milestone Plan

The following screenshot was created at the 31th of December 2021, 10:00 CET. The milestone plan is publicly accessible on GitHub:

- https://github.com/swissmanu/rxjs-debugging-for-vscode/milestone/2?closed=1



v1.0.0

No due date **100%** complete

0 Open ✓ **9 Closed**

Update README for v1.0.0 Release documentation
#115 by swissmanu was closed 29 days ago 2 tasks

Rework Testing & Improve Coverage improvement 1
#49 by swissmanu was closed 29 days ago 2 of 5 tasks

Operator Log Point Decorations change Line Height bug
#118 by swissmanu was closed 29 days ago

Support RxJS 7 feature improvement 1
#52 by swissmanu was closed on 25 Nov

Open Analytics on Extension Usage feature 1
#63 by swissmanu was closed on 20 Nov

Add ARCHITECTURE.md documentation
#48 by swissmanu was closed on 12 Nov

Enabled Log Point stays where it was enabled once bug 1
#102 by swissmanu was closed on 12 Nov

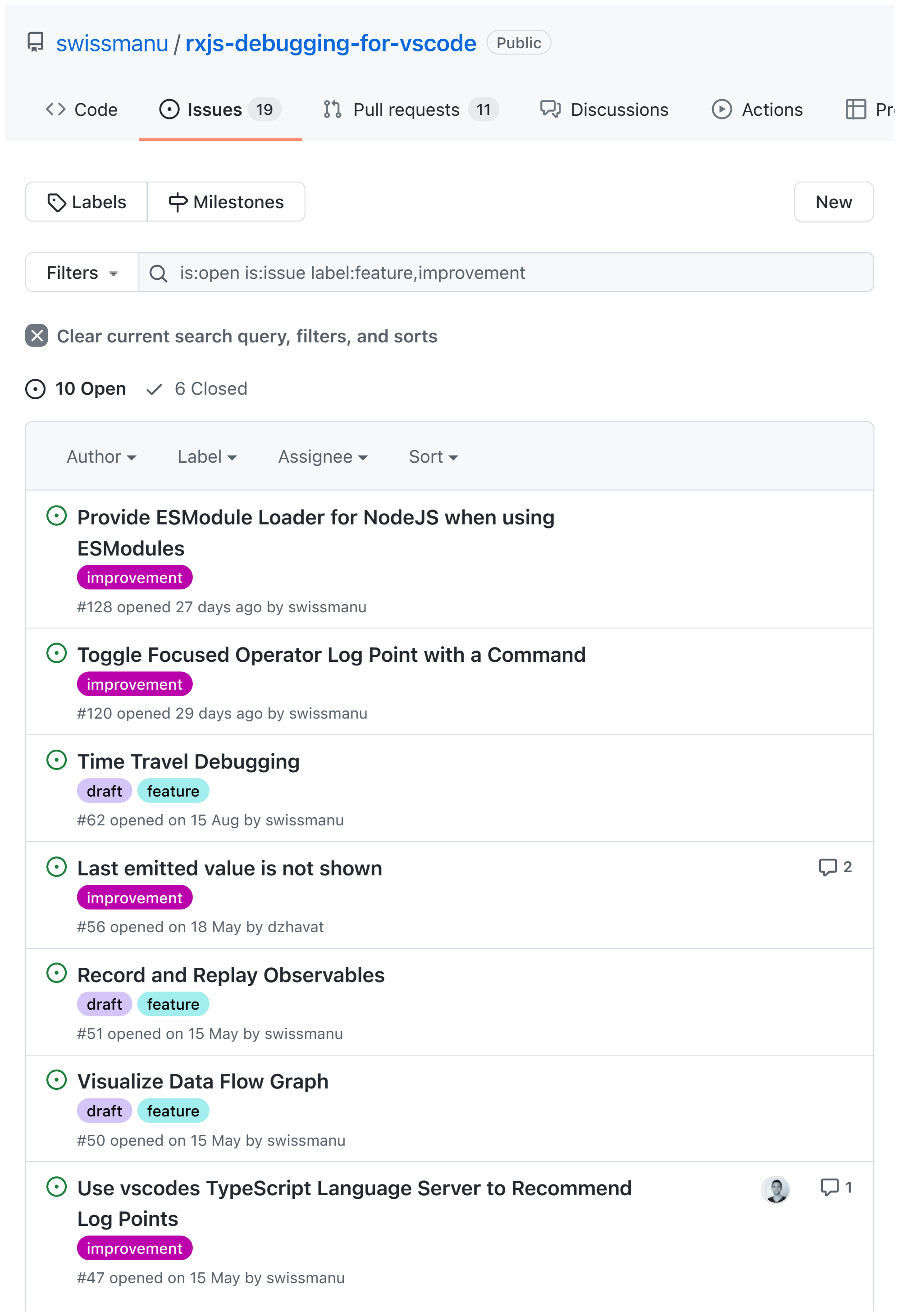
Improve Display of Suggested Log Points improvement
#64 by swissmanu was closed on 7 Nov

Support Debugging of Browser-based Applications feature 1
#43 by swissmanu was closed on 29 Oct

C.2 Feature Backlog

The following document was created at the 31th of December 2021, 10:00 CET. The most current feature backlog is publicly accessible on GitHub:

- https://github.com/swissmanu/rxjs-debugging-for-vscode/issues?q=is%3Aopen+is%3Aissue+label%3Afeature%2Cimprovement



is:open is:issue label:feature,improvement

Clear current search query, filters, and sorts

10 Open ✓ 6 Closed

Provide ESModule Loader for NodeJS when using ESModules
improvement
#128 opened 27 days ago by swissmanu

Toggle Focused Operator Log Point with a Command
improvement
#120 opened 29 days ago by swissmanu

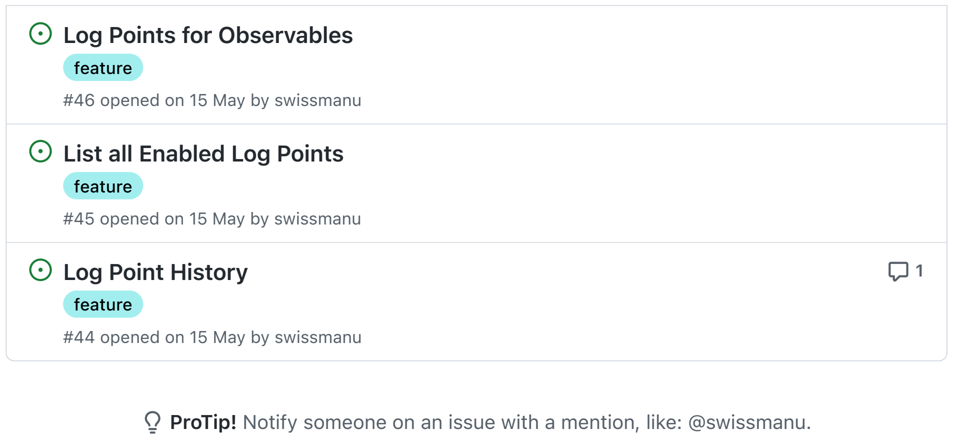
Time Travel Debugging
draft feature
#62 opened on 15 Aug by swissmanu

Last emitted value is not shown 2
improvement
#56 opened on 18 May by dzhavat

Record and Replay Observables
draft feature
#51 opened on 15 May by swissmanu

Visualize Data Flow Graph
draft feature
#50 opened on 15 May by swissmanu

Use vscodes TypeScript Language Server to Recommend Log Points 1
improvement
#47 opened on 15 May by swissmanu



Log Points for Observables
feature
#46 opened on 15 May by swissmanu

List all Enabled Log Points
feature
#45 opened on 15 May by swissmanu

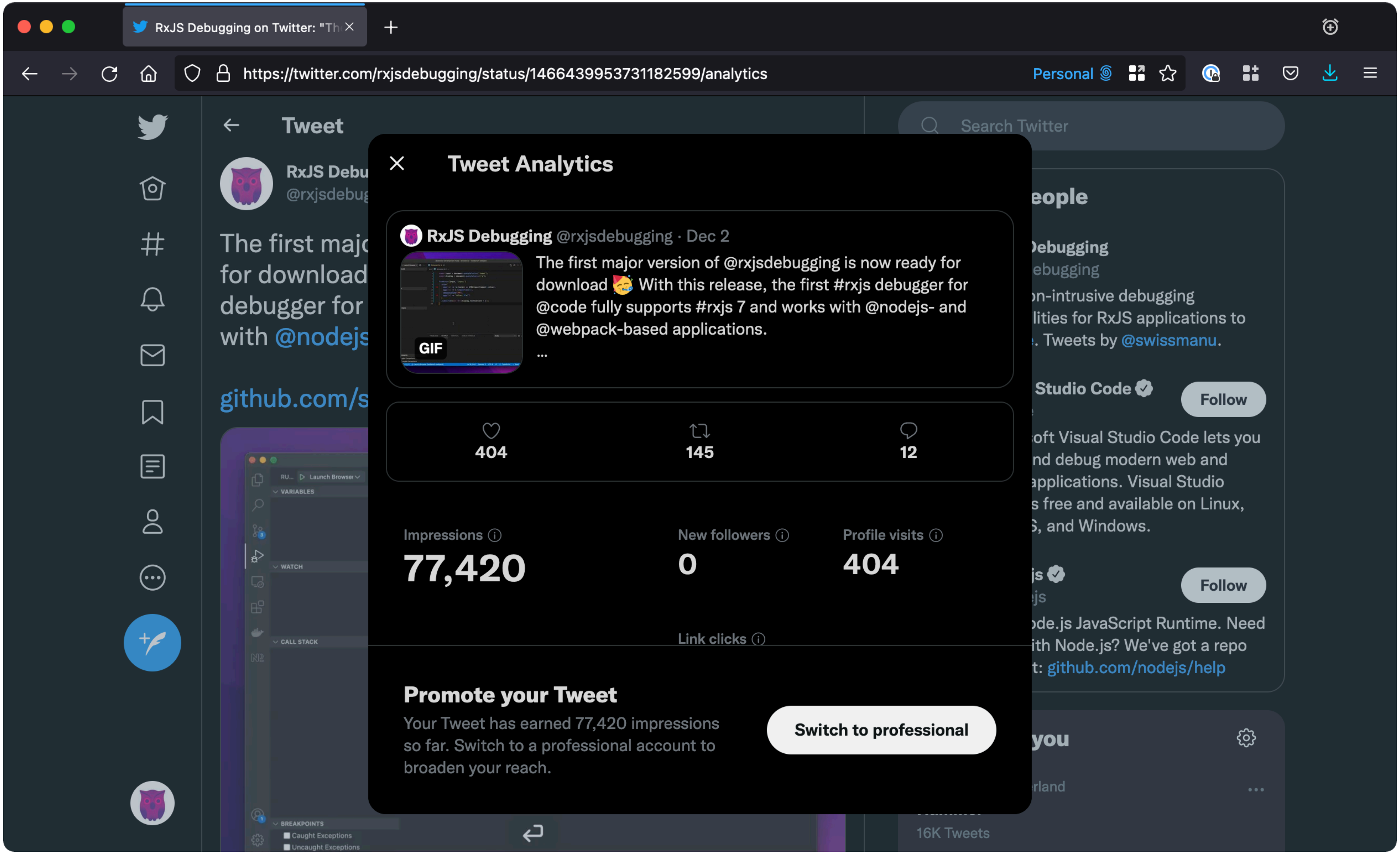
Log Point History 1
feature
#44 opened on 15 May by swissmanu

ProTip! Notify someone on an issue with a mention, like: @swissmanu.

C.3 Release Tweet Stats

The following screenshot was taken at the 30th of December 2021, 19:00 CET. The publicly accessible tweet is available on Twitter:

- https://twitter.com/rxjsdebugging/status/1466439953731182599



RxJS Debugging on Twitter: "Th

https://twitter.com/rxjsdebugging/status/1466439953731182599/analytics

Tweet Analytics

RxJS Debugging @rxjsdebugging · Dec 2

The first major version of @rxjsdebugging is now ready for download 🥳 With this release, the first #rxjs debugger for @code fully supports #rxjs 7 and works with @nodejs- and @webpack-based applications.
...

404 145 12

Impressions
77,420

New followers
0

Profile visits
404

Link clicks

Promote your Tweet

Your Tweet has earned 77,420 impressions so far. Switch to a professional account to broaden your reach.

C.4 Visual Studio Marketplace

The following screenshots were taken at the 30th of December, 19:00 CET. The most current version of the Marketplace page is available here:

- https://marketplace.visualstudio.com/items?itemName=manuelalabor.rxjs-debugging-for-vs-code

Visual Studio | Marketplace   Sign in

Visual Studio Code > Debuggers > RxJS Debugging for Visual Studio Code   New to Visual Studio Code? Get it now.

RxJS Debugging for Visual Studio Code

Manuel Alabor | 954 installs | ★★★★★ (1) | Free

Add non-intrusive debugging capabilities for RxJS applications to Visual Studio Code.

Install   Trouble Installing?

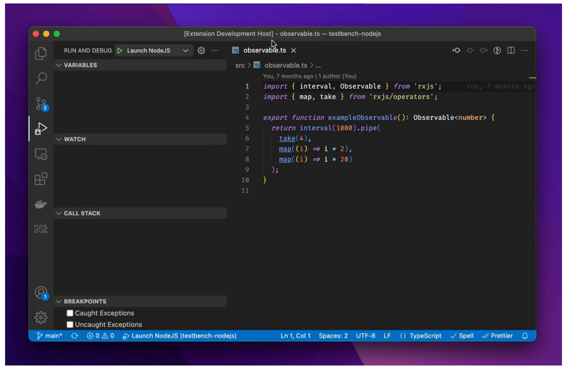
Overview   Version History   Q & A   Rating & Review

RxJS Debugging for Visual Studio Code

VS Marketplace v1.1.1   Follow @rxjsdebugging

> Never, ever use `tap(console.log)` again.

Add non-intrusive debugging capabilities for RxJS applications to Visual Studio Code.



```
import { interval, Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';

export function exampleObservable(): Observable<number> {
  return interval(1000).pipe(
    take(4),
    map((i) => i * 2),
    map((i) => i * 20)
  );
}
```

Features

- RxJS debugging, fully integrated with Visual Studio Code
- Works with RxJS 6.6.7 and newer
- Support for Node.js and Webpack-based RxJS applications

Requirements

- Visual Studio Code 1.61 or newer
- RxJS 6.6.7 or newer
- To debug NodeJS-based applications:
  - Node.js 12 or newer
- To debug Webpack-based web applications:
  - Webpack 5.60.0 or newer
  - The latest @rxjs-debugging/runtime-webpack Webpack plugin (see here for setup instructions)

Usage

Operator Log Points

Operator log points make manually added `console.log` statements a thing of the past: RxJS Debugger detects operators automatically and recommends a log point. Hover the mouse cursor on the operator to add or remove a log point to the respective operator:

Categories

Debuggers

Tags

javascript   javascriptreact   typescript   typescriptreact

Works with

Universal

Resources

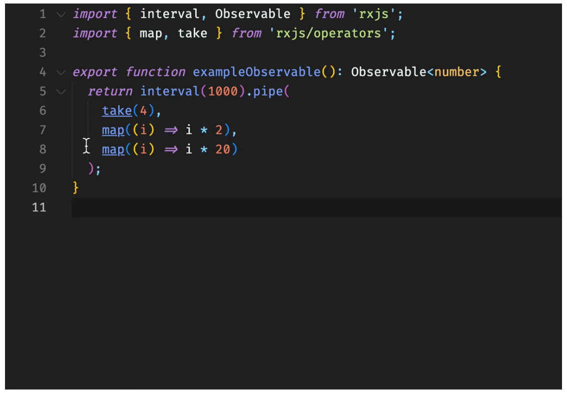
Issues
Repository
Homepage
License
Changelog
Download Extension

Project Details

swissmanu/rxjs-debugging-for-vscode
Last Commit: 3 weeks ago
11 Pull Requests
19 Open Issues

More Info

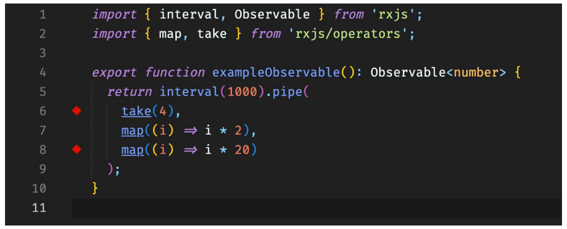
Version	1.1.1
Released on	17/05/2021, 15:58:57
Last updated	08/12/2021, 15:30:44
Publisher	Manuel Alabor
Unique Identifier	manuelalabor.rxjs-debugging-for-vs-code
Report	Report Abuse



```
import { interval, Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';

export function exampleObservable(): Observable<number> {
  return interval(1000).pipe(
    take(4),
    map((i) => i * 2),
    map((i) => i * 20)
  );
}
```

Once you launch your application with the JavaScript debugger built-in to Visual Studio Code, enabled log points display events of interest inline in the editor:

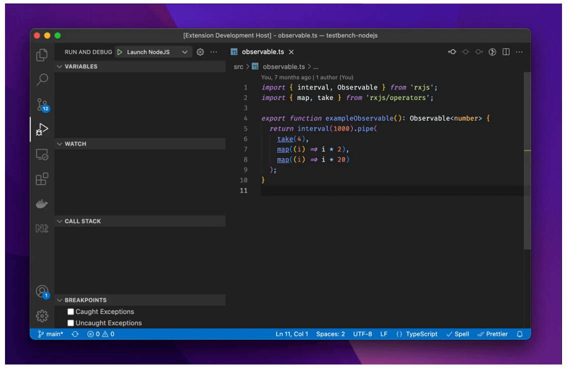
- Subscribe
- Emitted values (next, error, complete)
- Unsubscribe



```
import { interval, Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';

export function exampleObservable(): Observable<number> {
  return interval(1000).pipe(
    take(4),
    map((i) => i * 2),
    map((i) => i * 20)
  );
}
```

By default, RxJS Debugger clears logged events from the editor after you stop the JavaScript debugger. You can customize this behavior in the settings.

Finally, you can toggle gutter indicators for recommended log points via the command palette:



```
import { interval, Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';

export function exampleObservable(): Observable<number> {
  return interval(1000).pipe(
    take(4),
    map((i) => i * 2),
    map((i) => i * 20)
  );
}
```

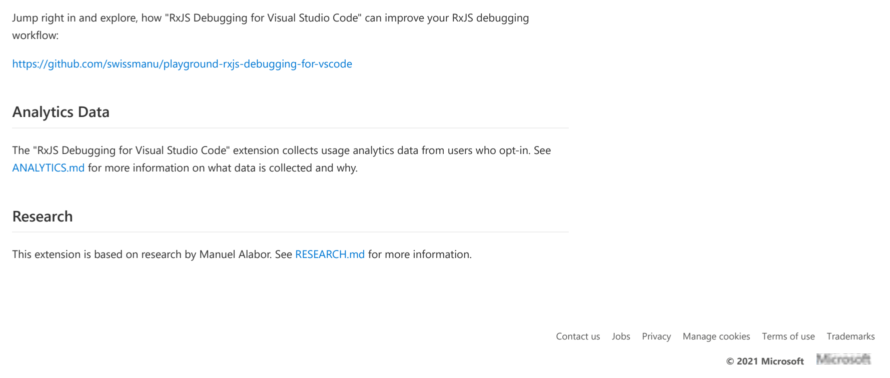
Roadmap & Future Development

Refer to the milestones overview for planned, future iterations. The issue list provides an overview on all open development topics.

Contributing

"RxJS Debugging for Visual Studio Code" welcomes any type of contribution! ❤️ Have a look at CONTRIBUTING.md for further details.

Playground



Jump right in and explore, how "RxJS Debugging for Visual Studio Code" can improve your RxJS debugging workflow:

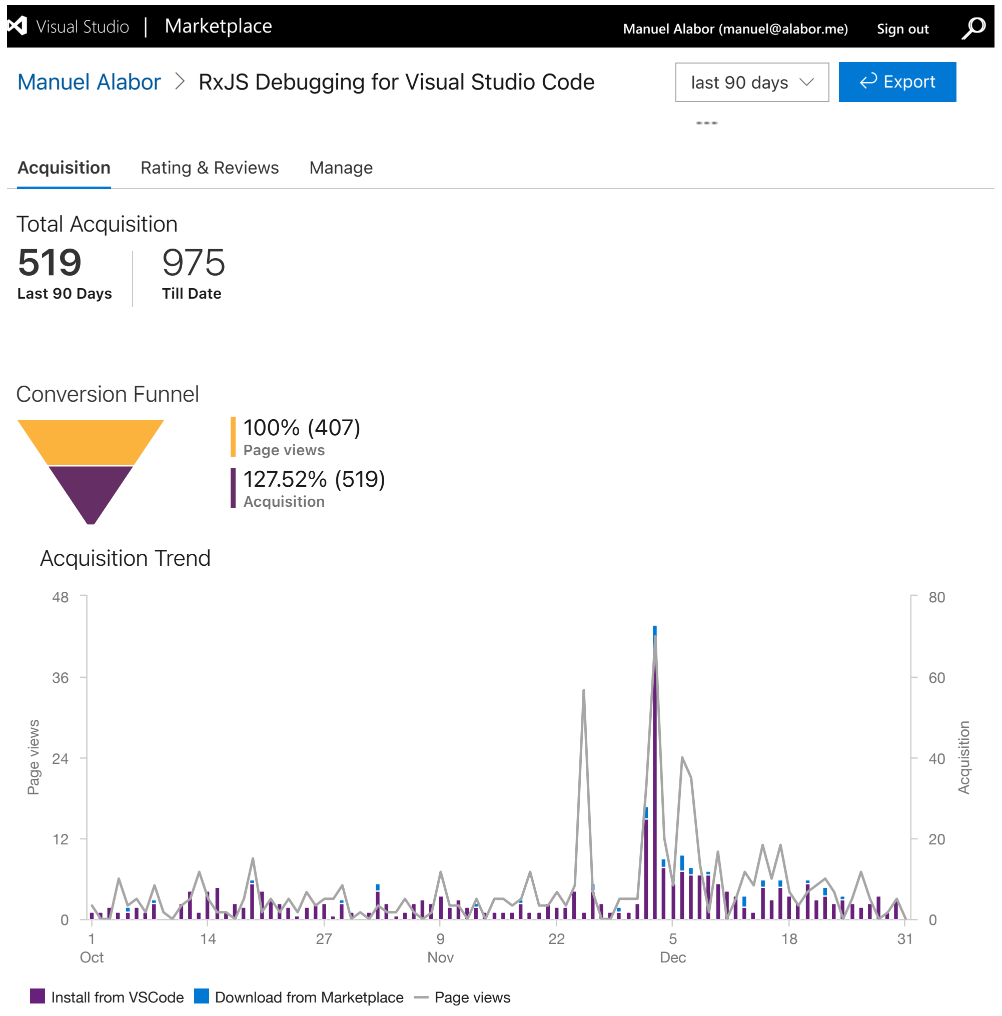
https://github.com/swissmanu/playground-rxjs-debugging-for-vscode

Analytics Data

The "RxJS Debugging for Visual Studio Code" extension collects usage analytics data from users who opt-in. See ANALYTICS.md for more information on what data is collected and why.

Research

This extension is based on research by Manuel Alabor. See RESEARCH.md for more information.


Visual Studio | Marketplace
Manuel Alabor (manuel@alabor.me)
Sign out
Manuel Alabor > RxJS Debugging for Visual Studio Code
last 90 days
Export
Acquisition
Rating & Reviews
Manage

Total Acquisition

519 Last 90 Days | 975 Till Date

Conversion Funnel

100% (407) Page views

127.52% (519) Acquisition

Acquisition Trend

C.5 ANALYTICS.md

The following document is a snapshot of the `ANALYTICS.md` file from the Git repository of RxJS Debugging for vscode:

- https://github.com/swissmanu/rxjs-debugging-for-vscode/blob/1bb1c20cbf3633ef45cf0df16aacb3c3ea8a8c8c/ANALYTICS.md

Extension Usage Analytics

🥽 For Science!

The initial version of "RxJS Debugging for Visual Studio Code" resulted from a research project. Doing (serious) research relies on empirical data. Thus, "RxJS Debugging for Visual Studio Code" asks its users to opt-in to collecting user behavior analytics data on its first activation.

It is essential for us that our users understand what data we collect and why we do it. This document gives full disclosure on every event and data point we collect. We reveal further where and how information is stored and how you can access it for your research work or contribution to the extension itself.

Tracked Events

If analytics is enabled, the extension tracks user behavior at events detailed below. Each event consists of an anonymized machine identifier provided by Visual Studio Code and a set of event-specific data points.

All data points are carefully crafted to protect the users' privacy while providing empirical evidence for future research work. The machine identifier **DOES NOT** reveal the identity of users accordingly. Its sole purpose is the consolidation of events that belong together over time.

We discard IP addresses before storing any tracking event. This makes it impossible to reconstruct or estimate your geographical location later.

The following list documents all tracked analytic events. Feel free to review their implementation directly in the source code: packages/extension/src/analytics.

Extension Started

Data Point	Reason	Example Values
Visual Studio Code Version	The version of your Visual Studio Code installation. This data point helps us to understand, which versions of Visual Studio Code are relevant for our users. It allows us to decide on if we can stop supporting outdated versions of Visual Studio Code, or not.	`1.61.0`
Visual Studio Code Language	The preferred language of your Visual Studio Code installation. This data point allows us to prioritze the languages for which we might translate "RxJS Debugging for Visual Studio Code" next.	`en-US`, `de-CH`, `fr`, ...
Extension Version	Identifies the version of "RxJS Debugging for Visual Studio Code" currently installed on your machine. This data point helps us understand how our users install updates of our extension after release.	`1.0.0`

Debug Session Started

Data Point	Reason	Example Values
Runtime Type	The runtime type declares how the RxJS debugger connects to your application. This data point helps us to understand what kind of RxJS applications (e.g. backend or frontend) our users debug most.	`nodejs`, `webpack`

Debug Session Stopped

This event does not include any additional data points.

Operator Log Point Enabled/Disabled

Data Point	Reason	Example Values
Operator Name	Identifies **built-in operators** for which you enable/disable a log point. We will **NOT** track the name of a custom operator nor anything else related to your source code (line numbers, structure, etc.) This data point helps us to understand which operators are the most problematic ones for our users. Thus, it helps us to build better debugging tools in the future accordingly.	`map`, `flatMap`, ...

Data Transmission and Storage

All analytic events are securely transmitted over an HTTPS connection.

Usage analytics data is collected using Posthog. It runs on the premises of the Eastern Switzerland University of Applied Sciences (OST) where all data ist stored as well.

Open Source, Open Research and Open Data

Posthog does not allow the creation of read-only users at the time of writing this document. 🙏 Please create an issue using the appropriate template if you want to access to the collected analytics data for your own research project or contribution to the extension. We happily assist you either with an export of a data set or grant you access to Posthog itself if required.

C.6 CONTRIBUTING.md

The following document is a snapshot of the `CONTRIBUTING.md` file from the Git repository of RxJS Debugging for vscode:

- https://github.com/swissmanu/rxjs-debugging-for-vscode/blob/2da72e21a733c99522633d8477892f3b5b48113c/CONTRIBUTING.md

Contributing

This project welcomes any type of contribution! ❤️ Opening an issue to document a problem you encountered or suggesting a new feature is always a good start.

Before you submit a pull request, please discuss potential changes with the maintainer either in an issue, using GitHub Discussions or via email.

Development

Get Started

To get started with development, follow these four steps:

1. Clone the repo and run `yarn` to install all dependencies.
2. Open `workspace.code-workspace` with Visual Studio Code.
3. Run the "extension: Build and Watch" task, which will continuously (re-)build the extension.
4. Run the "Testbench: NodeJS" launch configuration to open a new Visual Studio Code window, which:
   - loads the RxJS debugging extension in development mode, so you can use the debugger in the original Visual Studio Code window.
   - uses `packages/testbench-nodejs` as workspace, so you can test the RxJS debugging extension with a real example.

Repository Structure

This repository is organized as monorepo. We use nx and lerna to streamline tasks.

Following packages can be found in the `packages` directory:

- `extension`: The main package containing the debugging extension for Visual Studio Code.
- `telemetry`: TypeScript types and helper functions used for communication between runtime and debugging extension.
- `runtime`: Contains rudimentary utilities to augment RxJS in an arbitrary runtime environment.
- `runtime-nodejs`: NodeJS specific augmentation functionalities.
- `runtime-webpack`: Webpack plugin, published as `@rxjs-debugging/runtime-webpack`, providing runtime augmentation for web applications built with Webpack.
- `extension-integrationtest`: An integration test suite verifying various aspects of the extension.
- `testbench-*`: Test environments simulating various scenarios to test the debugger.

Run Test Suites

Unit and integration tests are automatically executed once changes are pushed to Github. You can run them locally using the following commands:

- Unit tests:

  ```
  yarn nx run-many --target=test --all --parallel
  ```

- Integration tests:

  ```
  yarn nx run extension-integrationtest:integrationtest --configuration=test
  ```

Architecture Concepts

The ARCHITECTURE.md file gives an overview on the most important architectural concepts.

C.7 CODE_OF_CONDUCT.md

The following document is a snapshot of the `CODE_OF_CONDUCT.md` file from the Git repository of RxJS Debugging for vscode:

- https://github.com/swissmanu/rxjs-debugging-for-vscode/blob/e826882151e2767daa95c1f68b4ba4b1c0c2db8f/CODE_OF_CONDUCT.md

Code of Conduct

Rules

To keep it simple, we only have three basic rules:

1. Don't panic
2. Don't be evil
3. Don't feed the trolls

Examples

The following non-exhaustive list provides specific guidelines and examples:

- Be respectful, be responsible, be kind
- Avoid asking for deadlines
- Don't feel entitled to free support, advice, or features if you are not a contributor
- If you have a general question, don't use GitHub Issues
- If you are having a bad day and want to offend someone, please go somewhere else

Reporting

We encourage all community members to resolve problems on their own whenever possible. Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to us.

Enforcement

Any violation may be punished with a snarky comment and finally a "plonk", which means that we ignore you according to rule #3.

This Code of Conduct was adapted from PhotoPrism. Thank you for a great product 🙏

C.8 ARCHITECTURE.md

The following document is a snapshot of the `ARCHITECTURE.md` file from the Git repository of RxJS Debugging for vscode:

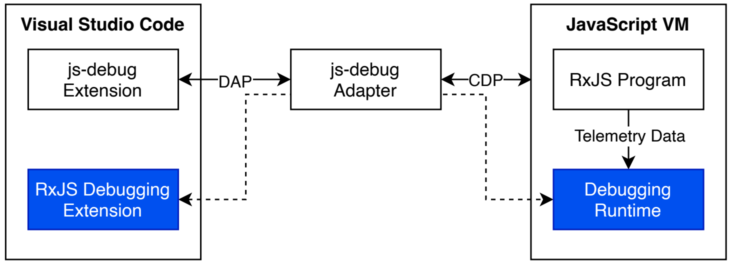
- https://github.com/swissmanu/rxjs-debugging-for-vscode/blob/50ca1b93427bd87e6af1a7466b46d1bf669cce6c/ARCHITECTURE.md

Architecture

Glossary

- *CDP*: Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/
- *DAP*: Debug Adapter Protocol. https://microsoft.github.io/debug-adapter-protocol/overview
- *VM*: Virtual Machine

Components



RxJS-specific debugging reuses debugging sessions started by *Visual Studio Codes* built-in JavaScript debugging extension (*js-debug*). The *RxJS Debugging Extension* communicates through *js-debug* using CDP with the *Debugging Runtime*. The *Debugging Runtime* interacts with the *RxJS Program*, running in the *JavaScript VM* (e.g., Node.JS or browsers like Google Chrome).

RxJS Debugging Extension

The *RxJS Debugging Extension* integrates with *Visual Studio Code* using its extension API and provides relevant user interfaces and functionalities. It allows developers to use RxJS debugging features like operator log points.

Furthermore, it ensures that, once a *js-debug* debugging session is started, essential hooks are registered in the *JavaScript VM* using CDP Bindings.

The communication protocol to exchange data with the *Debugging Runtime* is implemented in the extension's TelemetryBridge.

Debugging Runtime

A *Debugging Runtime* interfaces with the live *RxJS Program* and forwards relevant *Telemetry Data* (e.g. a value emitted by an Observable) to the *RxJS Debugging Extension*. A *Debugging Runtime* runs in the same process as the *RxJS Program*.

Specific *JavaScript VMs* require specific *Debugging Runtimes*. E.g., runtime-nodejs enables debugging of *RxJS Programs* executed in Node.JS. Web application bundled with Webpack require the runtime-webpack plugin likewise.

Independently from "how" a *Debugging Runtime* finds its way to the *JavaScript VM*, all of them fulfil following tasks:

- Use hooks registered using CDP Bindings to establish communication with the *RxJS Debugging Extension*
- Patch RxJS to provide required *Telemetry Data*
- Communicate with the *RxJS Debugging* Extension using the runtimes TelemetryBridge

CDP Bindings

A binding is a function available in a *JavaScript VM* global scope. It is created using the Runtime.addBinding function of a CDP client (i.e. the *RxJS Debugging Extension*). Once the *Binding* function is called, a callback in the CDP client is executed.

RxJS Debugging for Visual Studio Code uses this form of remote procedure calls (RPC) to communicate with the *Debugging Runtime* in a *JavaScript VM*.

Once the *RxJS Debugging Extension* detects a new *js-debug* debugging session, following bindings are registered:

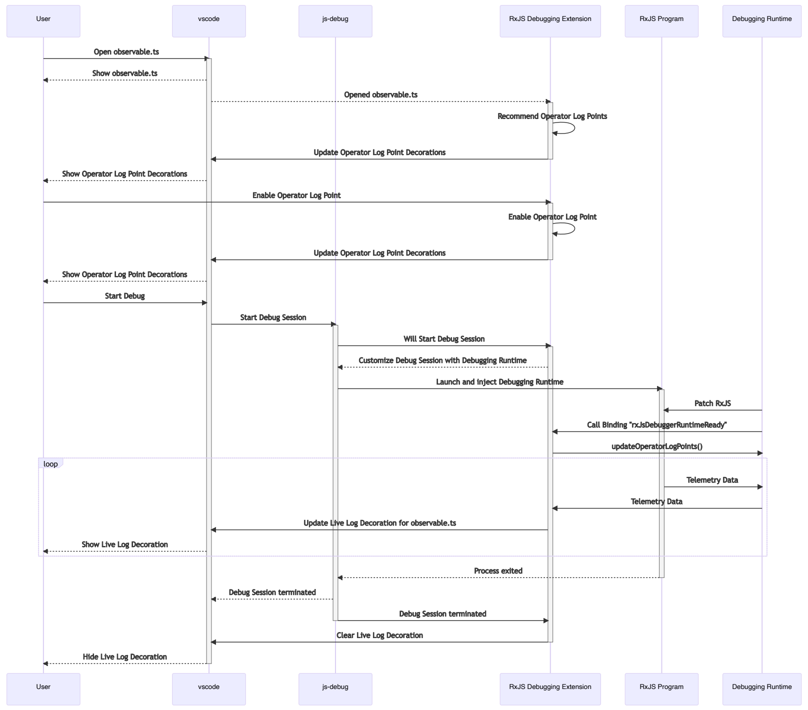
Name	Payload	Notes
`rxJsDebuggerRuntimeReady`	None	A *Debugging Runtime* is expected to call this binding once it is ready to debug an *RxJS Program*.
`sendRxJsDebuggerTelemetry`	`string`	Sends a JSON-encoded TelemetryEvent to the *RxJS Debugging Extension*.

Both the *RxJS Debugging Extension* as well as the *Debugging Runtime* use a well defined communication protocol implemented by their respective telemetry bridges.

Example System Interaction

Based on testbench-nodejs, the following sequence diagram shows typical interactions between the presented system components.

The JavaScript VM component is omitted for clarity.


User
vscode
js-debug
RxJS Debugging Extension
RxJS Program
Debugging Runtime
Open observable.ts
Show observable.ts
Opened observable.ts
Recommend Operator Log Points
Update Operator Log Point Decorations
Show Operator Log Point Decorations
Enable Operator Log Point
Enable Operator Log Point
Update Operator Log Point Decorations
Show Operator Log Point Decorations
Start Debug
Start Debug Session
Will Start Debug Session
Customize Debug Session with Debugging Runtime
Launch and Inject Debugging Runtime
Patch RxJS
Call Binding "rxJsDebuggerRuntimeReady"
updateOperatorLogPoints()
loop
Telemetry Data
Telemetry Data
Update Live Log Decoration for observable.ts
Show Live Log Decoration
Process exited
Debug Session terminated
Debug Session terminated
Clear Live Log Decoration
Hide Live Log Decoration

C.9 CHANGELOG.md

The following document is a snapshot of the `CHANGELOG.md` file from the Git repository of RxJS Debugging for vscode:

- https://github.com/swissmanu/rxjs-debugging-for-vscode/blob/1cdb1579b872243a10747c94d9c623759dfa83f0/CHANGELOG.md

Change Log

Install the latest version from the Visual Studio Marketplace.

1.1.1

- Bugfix: RxJS is not detected on Windows Systems #139

1.1.0

- Improvement: Support for Plain JavaScript #126

1.0.1

- Bugfix: Live Logs from Previous Debug Session shown again in a new Debug Session #123

1.0.0

- Feature: Support RxJS 7 #52
- Bugfix: Operator Log Point Decorations change Line Height #118

0.9.0

- Feature: Support Debugging of Browser-based Applications #43
- Feature: Collect Analytics Data on Opt-In #63
- Improvement: Add Integration Test for Operator Log Points #49
- Bugfix: Enabled Log Point stays where it was enabled once #102

0.1.2

- Fix: Log Point Events not displayed #54
- Improvement: Support for `pwa-node` launch configurations

0.1.1

- Add Icon for Visual Studio Code Extension Marketplace. 🦉

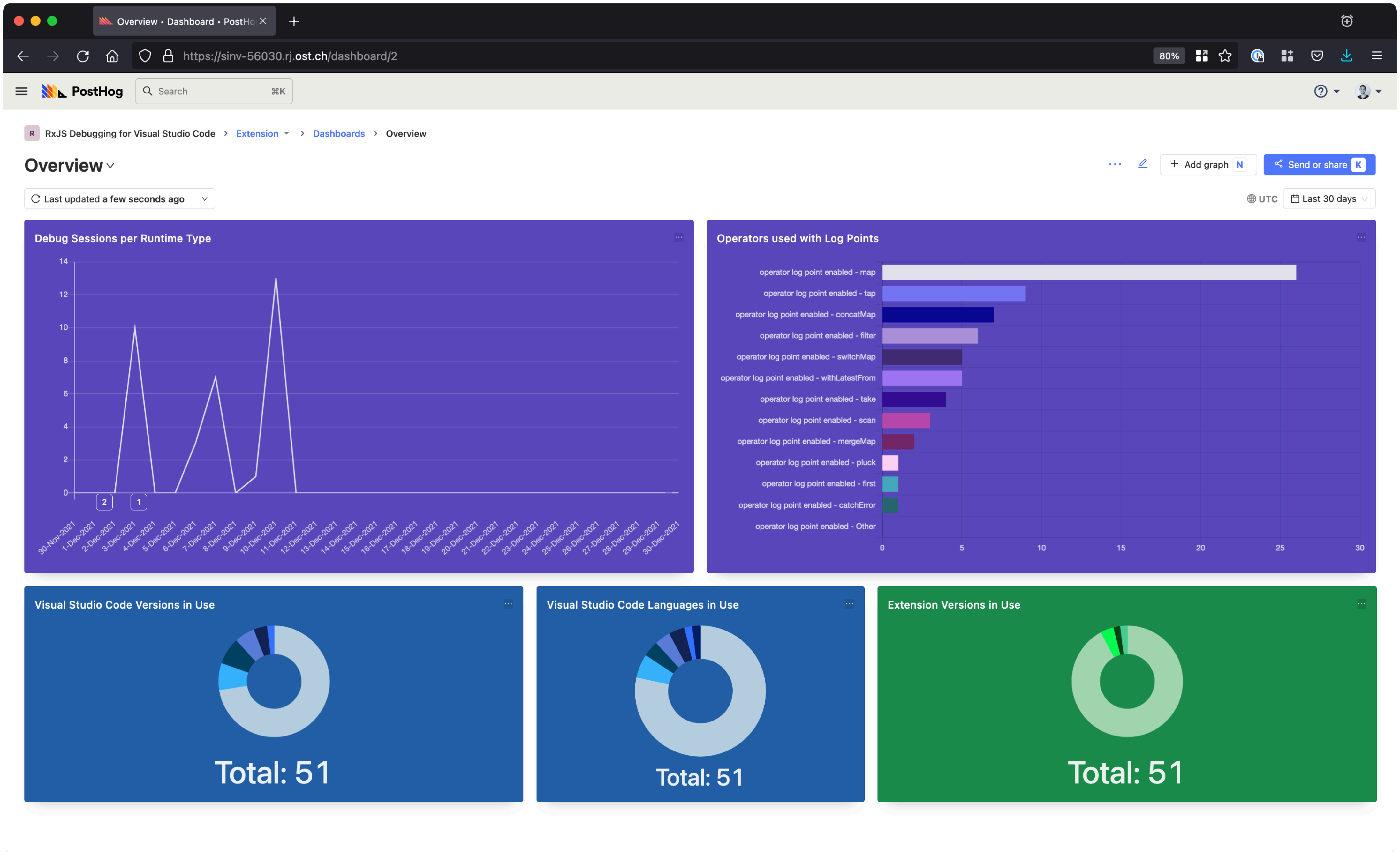
0.1.0

- Feature: Operator Log Points
- Feature: NodeJS Support

C.10 Analytics Dashboard

The following screenshot was taken at the 30th of December 2021, 19:00 CET.

Appendix C.5 describes relevant steps to get access to this data set.


Overview • Dashboard • PostHog
https://sinv-56030.rj.ost.ch/dashboard/2
80%
PostHog
Search ⌘K
RxJS Debugging for Visual Studio Code > Extension > Dashboards > Overview
Overview
Add graph N
Send or share K
Last updated a few seconds ago
UTC
Last 30 days
Debug Sessions per Runtime Type
Operators used with Log Points
operator log point enabled - map
operator log point enabled - tap
operator log point enabled - concatMap
operator log point enabled - filter
operator log point enabled - switchMap
operator log point enabled - withLatestFrom
operator log point enabled - take
operator log point enabled - scan
operator log point enabled - mergeMap
operator log point enabled - pluck
operator log point enabled - first
operator log point enabled - catchError
operator log point enabled - Other
Visual Studio Code Versions in Use
Total: 51
Visual Studio Code Languages in Use
Total: 51
Extension Versions in Use
Total: 51

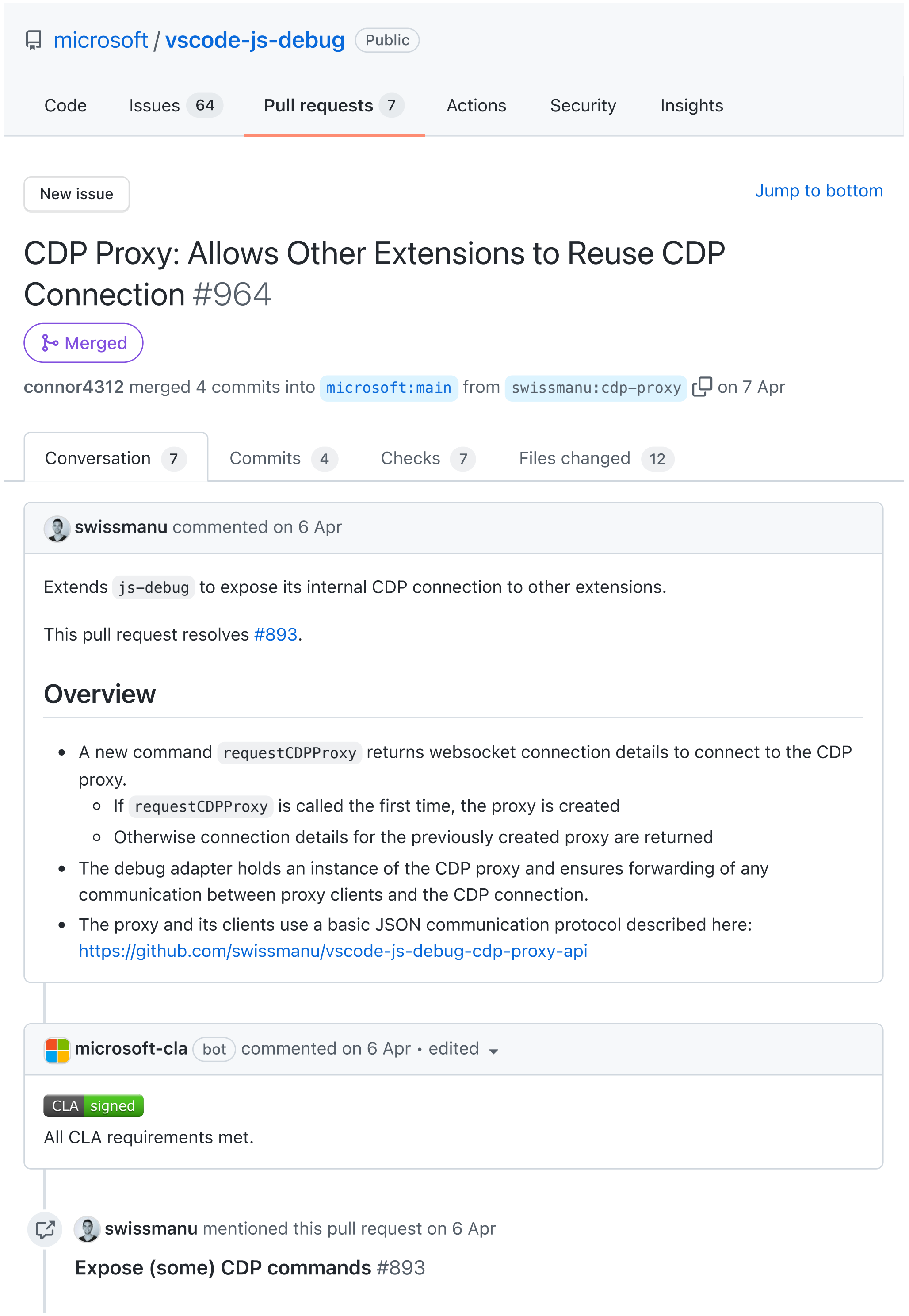
D vscode-js-debug Pull Request: Reuse CDP Connection

The following document was created at the 31th of December 2021, 10:00 CET and is publicly accessible:

- https://github.com/microsoft/vscode-js-debug/pull/964

Involved Parties

- Connor Peet *(connor4312)*
- Manuel Alabor *(swissmanu)*


microsoft / vscode-js-debug Public

Code | Issues 64 | Pull requests 7 | Actions | Security | Insights

New issue | Jump to bottom

CDP Proxy: Allows Other Extensions to Reuse CDP Connection #964

Merged

connor4312 merged 4 commits into `microsoft:main` from `swissmanu:cdp-proxy` on 7 Apr

Conversation 7 | Commits 4 | Checks 7 | Files changed 12

swissmanu commented on 6 Apr

Extends `js-debug` to expose its internal CDP connection to other extensions.

This pull request resolves #893.

Overview

- A new command `requestCDPProxy` returns websocket connection details to connect to the CDP proxy.
  - If `requestCDPProxy` is called the first time, the proxy is created
  - Otherwise connection details for the previously created proxy are returned
- The debug adapter holds an instance of the CDP proxy and ensures forwarding of any communication between proxy clients and the CDP connection.
- The proxy and its clients use a basic JSON communication protocol described here: https://github.com/swissmanu/vscode-js-debug-cdp-proxy-api

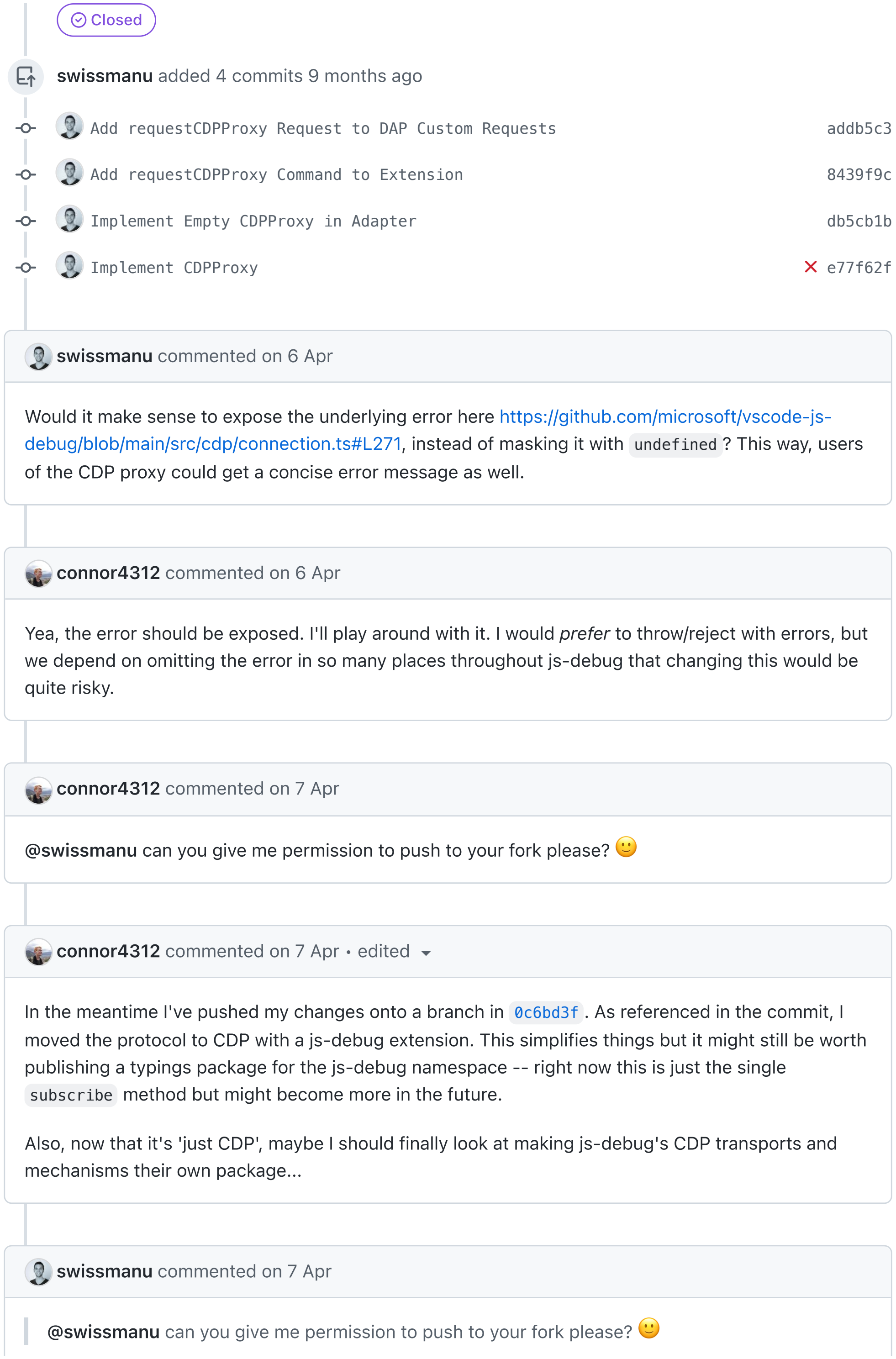
microsoft-cla bot commented on 6 Apr • edited

CLA signed

All CLA requirements met.

swissmanu mentioned this pull request on 6 Apr

Expose (some) CDP commands #893


Closed

swissmanu added 4 commits 9 months ago

```
Add requestCDPProxy Request to DAP Custom Requests    addb5c3
Add requestCDPProxy Command to Extension              8439f9c
Implement Empty CDPProxy in Adapter                   db5cb1b
Implement CDPProxy                                  ✕ e77f62f
```

swissmanu commented on 6 Apr

Would it make sense to expose the underlying error here https://github.com/microsoft/vscode-js-debug/blob/main/src/cdp/connection.ts#L271, instead of masking it with `undefined`? This way, users of the CDP proxy could get a concise error message as well.

connor4312 commented on 6 Apr

Yea, the error should be exposed. I'll play around with it. I would *prefer* to throw/reject with errors, but we depend on omitting the error in so many places throughout js-debug that changing this would be quite risky.

connor4312 commented on 7 Apr

@swissmanu can you give me permission to push to your fork please? 🙂

connor4312 commented on 7 Apr • edited

In the meantime I've pushed my changes onto a branch in `0c6bd3f`. As referenced in the commit, I moved the protocol to CDP with a js-debug extension. This simplifies things but it might still be worth publishing a typings package for the js-debug namespace -- right now this is just the single `subscribe` method but might become more in the future.

Also, now that it's 'just CDP', maybe I should finally look at making js-debug's CDP transports and mechanisms their own package...

swissmanu commented on 7 Apr

> **@swissmanu** can you give me permission to push to your fork please? 🙂

I ticked the "Allow edits by maintainers" checkbox on this pull request only. Permissions to the fork should be work by now. Sorry for the delay.

> I moved the protocol to CDP with a js-debug extension.

I like like your removal of the additonal protocol layer. Makes things straight forward.

connor4312 merged commit **`e77f62f`** into `microsoft:main` on 7 Apr
3 of 8 checks passed

View details

connor4312 commented on 7 Apr

🚀 Thank you for your work on this!

👍 1

connor4312 added this to the **April 2021** milestone on 3 May

Reviewers

No reviews

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

April 2021

Linked issues

Successfully merging this pull request may close these issues.

✓ **Expose (some) CDP commands**

2 participants

E Marble Diagram Syntax

Observables

In a marble diagram, a horizontal arrow $\longrightarrow$ pointing from left to right represents the timeline of an observable. Such a timeline uses following depictions to show when the observable emitted an event:

- A marble ⓥ indicates a `next` event carrying the displayed value `v`
- A vertical bar | shows a `complete` event
- `next` and `complete` can take place at the same time, thus are shown as the combination of a marble and a vertical bar: ϕ
- Finally, the cross ✕ indicates that an `error` occurred

Operators

Operators are depicted using a rectangular box $\boxed{\text{operator}}$ labeled with the operators name.

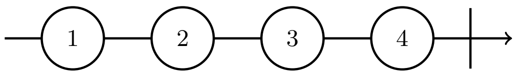
An event consumed by an operator is indicated with a vertical arrow ⇣ pointing from the source observable to the operator.

An event projected to the target observable is shown using a vertical arrow ⇣ pointing from the operator to the target observable.

Examples

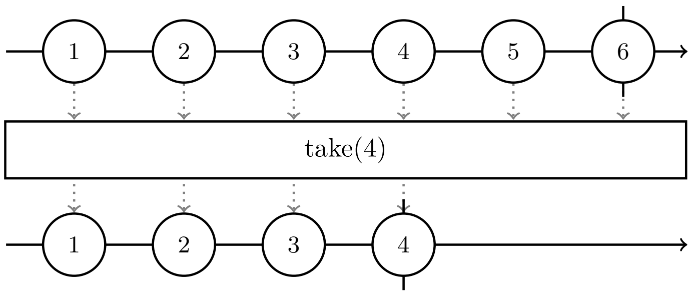
One Observable

A simple observable emitting integers from 1 through 4 that completes eventually.



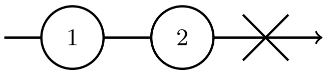
Operator

A marble diagram showing a source observable emitting the integers from 1 through 6. The source completes immediately after its last value was emitted. The `take`[11] operator projects the first four values and then completes the target observable.



Error

The following marble diagrams shows an observable that failed with an `error` event after it emitted the integers 1 and 2.



[11]https://rxjs.dev/api/operators/take

F Open Science

This Thesis

- https://github.com/swissmanu/mse-thesis
  (publicly accessible after May 2022)

Research Papers

- Debugging of RxJS-based Applications
  https://github.com/swissmanu/mse-paper-debugging-of-rxjs-based-applications
- Debugging Support for Reactive Programming: Feasibility of a Ready-to-hand Debugger for RxJS
  https://github.com/swissmanu/mse-paper-rxjs-debugger
  (publicly accessible after May 2022)

Studies

- Observational study
  https://github.com/swissmanu/mse-pa1-experiment
- Usability test
  https://github.com/swissmanu/mse-pa2-usability-test

User Behavior Data "RxJS Debugging for vscode"

As noted in Appendix C.5, user behavior data is available on request. Please follow the steps described there or on the linked page below to get access to the user behavior data set.

- https://github.com/swissmanu/rxjs-debugging-for-vscode/blob/main/ANALYTICS.md#open-source-open-research-and-open-data

List of Figures

1 Taxonomy of relevant programming paradigms 2
2 Marble diagram visualizing an observable 4
3 RxJS stack trace 5
4 Research process in four phases 7
5 Debugging techniques 8
6 Screenshot RxJS Debugging for vscode Prototype 9
7 System architecture 10
8 Empirical validation as the next step in a future follow-up. . . . 12

List of Listings

1 An observable emitting integers 1...8. Two operators process the integers before they are handed to the subscriber, which prints them to the console. 3
2 JavaScript program replicating Listing 1 using an imperative programming style. 4
3 RxJS-based program from Listing 1 manually augmented with print statements. 5

List of Tables

1 Overview of all artifacts delivered per process phase. 7

References

[1] 1990. *IEEE standard glossary of software engineering terminology.* IEEE. DOI:https://doi.org/10.1109/IEEESTD.1990.101064

[2] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN international workshop on reactive and event-based languages and systems* (REBLS 2020), Association for Computing Machinery, 15–24. DOI:https://doi.org/10.1145/3427763.3428313

[3] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys* 45, 4 (August 2013), 1–34. DOI:https://doi.org/10.1145/2501654.2501666

[4] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging data flows in reactive programs. In *Proceedings of the 40th international conference on software engineering*, ACM, 752–763. DOI:https://doi.org/10.1145/3180155.3180156

[5] T. Boren and J. Ramey. 2000. Thinking aloud: Reconciling theory and practice. *IEEE Transactions on Professional Communication* 43, 3 (September 2000), 261–278. DOI:https://doi.org/10.1109/47.867942

[6] Yaofei Chen, R. Dios, A. Mili, Lan Wu, and Kefei Wang. 2005. An empirical study of programming language trends. *IEEE Software* 22, 3 (May 2005), 72–78. DOI:https://doi.org/10.1109/MS.2005.55

[7] Conal M. Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on haskell - haskell '09*, ACM Press, 25. DOI:https://doi.org/10.1145/1596638.1596643

[8] Ralph Johnson Erich Gamma Richard Helm and John Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software.* Pearson Education India.

[9] Fabian Fagerholm and Jurgen Munch. 2012. Developer experience: Concept and definition. In *2012 international conference on software and system process (ICSSP)*, IEEE, 73–77. DOI:https://doi.org/10.1109/ICSSP.2012.6225984

[10] Kim Goodwin. 2009. *Designing for the digital age: How to create human-centered products and services.* Wiley Pub.

[11] Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* 21, 3 (September 1989), 359–411. DOI:https://doi.org/10.1145/72551.72554

[12] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *ACM Computing Surveys* 36, 1 (March 2004), 1–34. DOI:https://doi.org/10.1145/1013208.1013209

[13] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE international symposium on empirical software engineering and measurement*, IEEE, 383–392. DOI:https://doi.org/10.1109/ESEM.2013.43

[14] Jonathan Lazar, Jinjuan Feng, and Harry Hochheiser. 2017. *Research methods in human-computer interaction* (2nd ed.). Elsevier: Morgan Kaufmann Publishers, an imprint of Elsevier.

[15] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software - onward! '13*, ACM Press, 53–62. DOI:https://doi.org/10.1145/2509578.2509585

[16] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices* 48, 10 (November 2013), 1–18. DOI:https://doi.org/10.1145/2544173.2509515

[17] Ayman Nadeem. 2021. Human-centered approach to static-analysis-driven developer tools: The future depends on good HCI. *Queue* 19, 4 (2021), 68–95.

[18] Jakob Nielsen. 1994. Estimating the number of subjects needed for a thinking aloud test. *International Journal of Human-Computer Studies* 41, 3 (September 1994), 385–397. DOI:https://doi.org/10.1006/ijhc.1994.1065

[19] Mie Nørgaard and Kasper Hornbæk. 2006. What do usability evaluators do in practice?: An explorative study of think-aloud testing. In *Proceedings of the 6th ACM conference on designing interactive systems - DIS '06*, ACM Press, 209. DOI:https://doi.org/10.1145/1142405.1142439

[20] ReactiveX. 2021. ReactiveX. Retrieved from http://reactivex.io/

[21] Adam Richardson. 2010. Using customer journey maps to improve customer experience. *Harvard business review* 15, 1 (2010), 2–5.

[22] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on modularity - MODULARITY '14*, ACM Press, 25–36. DOI:https://doi.org/10.1145/2577080.2577083

[23] Guido Salvaneschi and Mira Mezini. 2016. Debugging for reactive programming. In *Proceedings of the 38th international conference on software engineering - ICSE '16*, ACM Press, 796–807. DOI:https://doi.org/10.1145/2884781.2884815

[24] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering* 43, 12 (December 2017), 1125–1143. DOI:https://doi.org/10.1109/TSE.2017.2655524

[25] International Organisation for Standardization (ISO). 2010. Ergonomics of human-system interaction - part 210: Human-centered design for interactive systems. ISO 9241-210:2010. (2010).

[26] David A. Watt, William Findlay, and John Hughes. 1990. *Programming language concepts and paradigms.* Prentice Hall.

[27] Cathleen Wharton, John Rieman, Lewis Clayton, and Peter Polson. 1994. *The cognitive walkthrough: A practitioner's guide.* Institute of Cognitive Science, University of Colorado.

[28] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.