TU WIEN Informatics

Browser-exklusive Realisierung von GLSP als digitale Sandbox Umgebung

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Andreas Hell
Matrikelnummer 12123458

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr. Dominik Bork
Mitwirkung: Dr. Philip Langer

Wien, 21. Juli 2024

Andreas Hell	Dominik Bork

TU WIEN Informatics

Browser-only implementation of GLSP as a digital sandbox environment

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Andreas Hell
Registration Number 12123458

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Dominik Bork
Assistance: Dr. Philip Langer

Vienna, July 21, 2024

Andreas Hell | Dominik Bork

Erklärung zur Verfassung der Arbeit

Andreas Hell

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Juli 2024

Andreas Hell

Danksagung

Ich möchte meinem Betreuer Dominik Bork danken, der mich bei der Suche nach einem geeigneten Thema unterstützt hat und diese Projekt erst möglich gemacht hat. Weiter auch, dass er sich bei der Entwicklung des Projekts und dem Schreiben der Arbeit mit seinem Feedback eingesetzt hat.

Auch möchte ich Philip Langer danken, der sich die Zeit genommen hat, als Co-Betreuer zu agieren und seine Expertise zu GLSP und sein Feedback zum Projektfortschritt zu geben.

Feedback zum eigenen Ansatz zu erhalten, es während der Entwicklung einbinden zu können und Hilfe hinsichtlich den nächsten Schritten zu erahlten ist oftmals unbezahlbar und hat mir geholfen meine eigene Vision für das Projekt zu bilden.

Acknowledgements

I would like to thank my advisor Dominik Bork, who supported my search for a suitable topic and made this project possible in the first place. Furthermore, for aiding the development of the project and writing of the thesis with his feedback.

I would also like to thank Philip Langer for acting as co-advisor and taking the time to share his expertise of GLSP and his feedback regarding the project's progress.

Receiving feedback on your approach and incorporating it during implementation and guidance regarding the next steps to take are often invaluable and helped me to build my own vision concerning the project.

Kurzfassung

Das Language Server Protocol (LSP) hat es ermöglicht, Code-Editoren von spezifischen Programmiersprachen loszulösen. Infolgedessen, werden nun keine schwerwiegenden Entwicklungsumgebungen mehr benötigt, da einfache Text-Editoren über LSP mit einem Language Server kommunizieren können und so brauchbare Entwicklungswerkzeuge für jede Programmiersprache bereitstellen. Das G in GLSP steht für “graphical” und dessen Ziel ist es, dieselben Möglichkeiten auch für graphische Entwicklungsumgebungen, also in erster Linie Modelleditoren, zu schaffen. Somit ist es möglich, solche Editoren im Browser umzusetzen, welche mit einem entsprechenden Language Server kommunizieren. Das spezifische Ziel dieser Bachelorarbeit ist es, die GLSP-Implementierung von Eclipse vollständig im Browser zu realisieren, wodurch der Nutzer eine schnellere, und schneller verfügbare, Applikation bekommt und gleichzeitig Server-Kosten reduziert werden.

Abstract

The language server protocol (LSP) enabled us to separate code editors from specific programming languages. This lead to heavy-weight IDEs becoming optional, since simple text editors can communicate via LSP with a language server in order to provide viable development tools for every programming language. The G in GLSP means “graphical” and it aims to provide the same possibilities for graphical development environments, which primarily concerns model editors. Thus it becomes conceivable to implement such editors in browsers that communicate with a corresponding language server. The specific focus of this Bachelor’s Thesis is to fully realise the GLSP implementation of Eclipse in browser, in order to grant users access to a faster, and faster available, application and reduce costs incurred by servers.

Contents

CHAPTER 1

Introduction

In the course of the introduction, the progress of the central piece of software development tooling, the editor, followed by a short investigation why graphical modeling tools seem to lag behind and how this problem is engaged, will be shown. Afterwards, an outlook on the project and further chapters is given.

1.1 Progress of software editors

1.1.1 Text editors

At the beginning of digitized software development, which, in this context, means no external physical media are used for programming i.e., punch cards, there was little choice but to write directly executable programs in machine code. However, at the end of the 1940s into the beginning of the 1950s, developers began using simple instructions instead of unreadable (for a human) machine code [BB47], paving the way for assembly languages, which are written in text editors.

As such languages are still very basic and require much understanding of the underlying hardware, higher-level programming languages were created. They are easier to understand for humans, but need additional software, a compiler, to create executable programs. One such language is C [RJL+78]. However, at his point in time, only simple text editors were used.

1.1.2 Integrated Development Environments (IDEs)

At this point, various text editors, e.g. VIM, are used to write any given language as simple text, using further tools to create the program, debug it and more. This still necessitates a rather cumbersome workflow of writing the file, change to another program to compile, execute it and evaluate the development progress afterwards.

This would, however, change as it became clear around the 1980s that there is still much room to optimize the workflow in software development, namely by bundling the necessary tools and shorten the feedback loop [ABL+81]. This would ultimately cumulate in the Integrated Development Environment, or IDE for short, which, as the name suggests, integrates all the tools in a single environment for development purposes.

1.1.3 Text editors (again)

Over the years, various IDEs have been dominating the development landscape, with Visual Studio [Micc], Eclipse IDE [Foub] and, in recent years, IntelliJ IDEA [Jet] being main contenders [Oveb]. However, in 2016, Microsoft changed the game by introducing the Language Server Protocol (LSP) [BL23] in order to enrich its text editor Visual Studio Code [Micd] with language capabilities before only encountered in IDEs.

The technology will be shortly explained later on, but, to summarise, it allows the separation of the language smarts from its respective editor. This allows text editors to make a major comeback. Whereas in 2010 the market was dominated by various IDEs, with Eclipse being the most prominent and text editors being in the single digit percentages, now they account for almost 20% in the metric used [Oveb]. Furthermore, in the most recent Stack Overflow developer survey of 2023, Visual Studio Code is by far the most preferred editor [Ovea].

This serves to show how pivotal technologies can reignite interest and create further possibilities. For example, new programming languages require proper tooling to have any chance of adoption, which previously would have required much effort to create a functional IDE, whereas now a LSP server is sufficient.

1.2 The problem of digital software modeling

1.2.1 Whiteboards and no interoperability

While the previous account explains the impact of LSP on the creation of textual developer tooling i.e., text editors and IDEs, there is also a need to explain the issues with current graphical development tooling i.e., modeling applications.

These tools naturally developed from classic whiteboarding, that is, the act of drawing the software architecture, processes or anything that could be expressed in a formal (or sometimes informal) modeling language per hand [CVDK07]. As there was no base line graphical editor akin to a text editor, they necessitated the development of heavy IDEs. Consequently, these IDEs contain various formal modeling standards, in order to make the most use of the IDE development, which makes them bloated and overwhelming to use.

Furthermore, since many of these IDEs opt for implementing most common modeling notations, no obvious superior choice is available, due to a lack of distinction. However, the dependency on the IDEs and their specificities make interoperability difficult or

impossible. Even people using the same modeling standard, e.g. UML, need to also work in the same modeling tool in order to allow for cooperation.

1.2.2 More flexibility

While it is unlikely to create a standard modeling tool, it is very well possible, as was demonstrated using the example of LSP, to create a new standard that increases interoperability. This would not only allow the purpose-built implementation of singular modeling standards, but also allow for more flexibility to created domain-specific modeling languages with less overhead of platform development.

As will be introduced in the following chapter, the purpose of this Bachelor's thesis is to work with a potential solution regarding this problem.

1.3 Outlook

After discussing the general problem, an overview of the further chapters is in order. The next chapter elucidates the context and subject of this thesis. This is followed by a short exploration of related work.

Next, the concept, including the used technologies and architecture, will be explained while also giving a summary of the development steps. Then, the actual implementation, any pitfalls and surprises, is described and subsequently evaluated.

CHAPTER 2

Background

2.1 Context

This Bachelor's Thesis is the result of a cooperation between the Business Informatics Group of the Information Technology Faculty of the TU Wien and the Vienna office of EclipseSource, a company specialized in creating developer tooling. As part of their company policy, they support and sponsor Open Source development, in course of which, they also promote the continuing progress of the GLSP protocol, or Graphical Language Server Protocol.

The specifics of this protocol will be covered in length at a later point, but for now it is sufficient to understand its purpose as a framework to develop graphical modeling languages [MB23][GSB23] in a manner similar to the regular Language Server Protocol for programming languages.

The company is maintaining this Open Source project, which still has multiple avenues of extension available, on behalf of the Eclipse Foundation. Some of those avenues were discussed in an exchange with our company partner, Dr. Philip Langer. After a short analysis of the given prospects, the topic of creating a browser-only implementation of GLSP to serve as a light-weight digital sandbox environment without backend requirements was chosen.

2.2 Motivation

2.2.1 Sandbox environment

The value proposition of such a sandbox environment is to increase the conversion rate of potential users. As there may be many potential solutions to a user's given problem, they need to quickly assess whether their requirements are met. While documentation

is helpful in this aspect, a live sandbox environment serves to more clearly convey the purpose and potential.

2.2.2 Replacement

The primary concern was replacing the existing sandbox environment built upon Theia Cloud, which is both cumbersome to use, as there is a significant start-up time, and generates costs, due to it running on a server and therefore according resource consumption. The goal was alleviating these issues by implementing an implementation of the protocol using the browser instead of a server, thereby being both readily available and generating no costs, as only static serving of files would be required. The detailed concept will be elaborated upon later on.

Secondary to this, was the potential to enhance the existing documentation, which mostly consisted of text, code and some images, by leveraging the much more nimble sandbox and adding prepared examples fitting for the respective chapters of the documentation to showcase the specific features of the protocol in source files and its resulting implementation.

Expanding upon this idea, making this sandbox not only interactive in respect to the implemented modeling language, but to also allow changing the GLSP implementation in the browser and "compiling" it live, was also suggested. This would represent a kind of meta programming of a graphical modeling language and grant the opportunity to not only showcase the end result, but also the workflow with the framework itself.

This is much more valuable for documentation as it concerns the creation of graphical modeling languages, as is the purpose of the protocol, and not only a potential resulting tool.

CHAPTER 3

Related Work

This chapter will expound upon some technologies that are similar to the sandbox to be created in this thesis, regardless of whether this concerns the graphical modeling or the editor.

3.1 Eclipse Epsilon Playground

This section will be largely derived from their documentation [Foua].

Similar to the GLSP project, Epsilon tries to enable more generic handling of models. However, it does not aim to completely abstract the server and client, but instead provides a limited set of tools to handle functionality from model transformation to validation and code generation. The model is accessed through a driver that creates a suitable meta model to allow handling of the actual model regardless of its idiosyncrasies.

This is achieved by providing many domain-specific languages to handle the functionalities. However, its functionality is limited, due to it being inextensible. Therefore, it is a set of tools to use, not a framework in which to develop new tools.

The project's authors do provide an online playground to test the tools under `https://eclipse.dev/epsilon/playground/`, where the model is editable directly and indirectly through the domain-specific languages. This capability to reprogram the model on the fly is similar to the editor planned for in this thesis' implementation, however not entirely the same in kind. Reason being that it concerns itself merely with changing the model within the limits of the specified DSLs, whereas this thesis aims for greater flexibility using the full capabilities of TypeScript.

3.2 Eclipse Theia

This section is based on the official website [Foud].

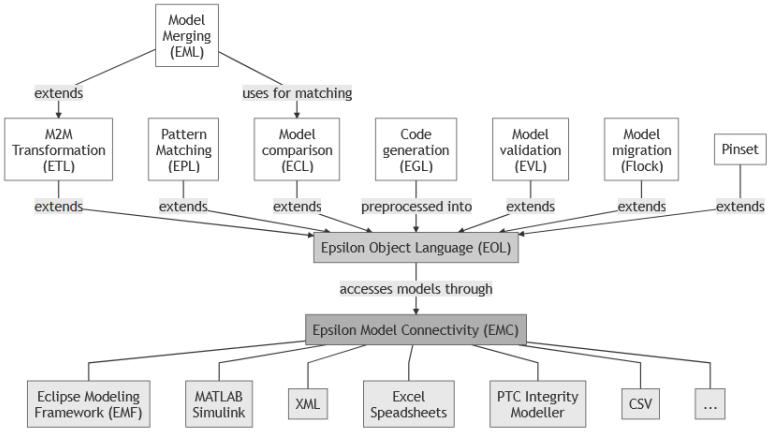

Figure 3.1: Epsilon architecture

Eclipse Theia provides an extensible platform to develop custom IDEs based on web technologies. Its layout, styling and functionality mirrors in large part Visual Studio Code. As it is described as a purpose-adjustable editor, it lends to be used for a sandbox environment with editing capabilities. Especially since it provides all the utilities one would expect from other IDEs.

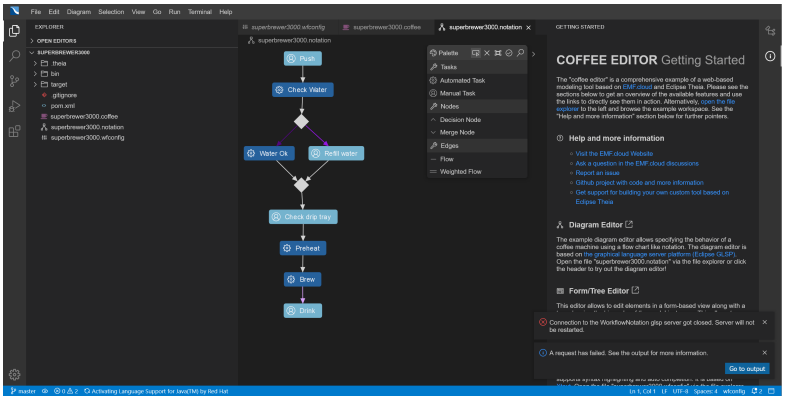

Figure 3.2: Eclipse Theia for GLSP

Theia has already been successfully used in the existing GLSP project's sandbox, as seen in the image, where the platform is, however, used in the same manner as the previously mentioned Epsilon Playground. That is, merely for developing in the given modeling language to adjust a given model. In contrast, this thesis aims to allow meta-editing of the modeling language.

As Eclipse Theia is the IDE platform, agnostic of its specific usage, it could be utilized to enhance the experience as it simply concerns a change of the editor technology and not the further application of the given user input.

CHAPTER 4

Concept

This chapter aims to give a concrete outline of the result that should be achieved in the course of this thesis. First of all, the used technologies will be expounded to ensure understanding of later sections. This is then followed by an architectural description of the initial and end situation. Finally, a short road map of the steps to take during implementation.

4.1 Technologies

First, there is a need to explore the technical background of GLSP, beginning with the underlying language server protocol. Additionally, since this project will necessarily make use of less common browser APIs, an examination of those is also in order.

4.1.1 Language Server Protocol (LSP) [Mica]

The language server protocol aims to separate the client and server when developing language editing support. Traditionally, language support features had to be repeated for each IDE, because of differing APIs. By providing a single API for communication, LSP allows for standardization by making development of language editing support independent from the used client and vice versa.

The protocol originated from Microsoft with focus on their Visual Studio Code editor, which has no integrated language support and needs to supplement this through language servers. It is a open source project available on GitHub `https://github.com/Microsoft/language-server-protocol`.

LSP works by separating concerns, as seen in Figure 4.1, with static highlighting i.e., syntax highlighting, being handled by the client, as it involves no deeper knowledge about the workings of a language. The contextual highlighting i.e., semantic highlighting (e.g.

intellisense), is instead handled by the language server that is invoked using the action performed and the code affected, which it then parses and generates the appropriate response for [BL23]. The client stays language agnostic and can be used for a multitude of languages, whereas the server does not care about the editor used, as it simply requires the communication to conform to the protocols standards. This communication is based on a versatile base-protocol called JSON-RPC.

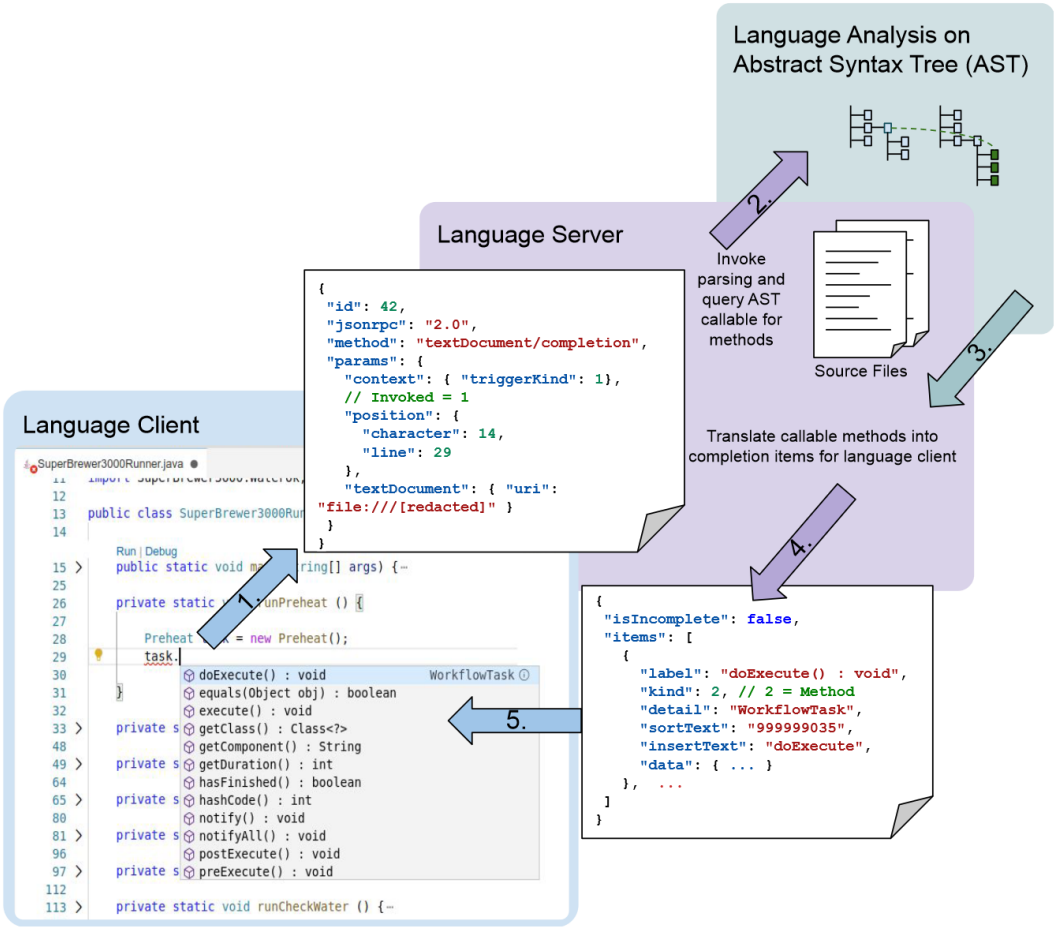

Figure 4.1: Example interactions between language client and language server for auto-completion (Figure 3, [BL23])

JSON-RPC [Gro]

As is generally known, JSON [Cro] stands for JavaScript Object Notation and RPC [Thu] means Remote Procedure Call. This protocol provides a specification for data structures and rules of how to process them, with it being agnostic to the transport method. The name already tells of its use in client-server communication that aims to have the server do certain actions. LSP (and later GLSP) builds upon this framework to further define

instructions for the server to execute.

4.1.2 Graphical Language Server Platform (GLSP) [Fouf][MB23]

Analogue to LSP, GLSP uses strict client-server separation, communicating via a single coherent API, to enable greater flexibility when creating developer tooling for graphical editors, primarily diagram editors. Whereas the server is still handling the language smarts of a given language, here a graphical modeling language, the client is not simply text-based, but is concerned with the rendering of the given model.

More specifically, using the already implemented packages realizing the protocol, a new concrete implementation has the following areas of concern:

- **Source Model**: The format of data containing the graphical model. There exist modules for some, however, if a custom choice is made, implementing an interface for the server to work with becomes necessary.
- **Server**: The server handles the language smarts of the used modeling language, which implies the handling of the aforementioned source model. It communicates the model and corresponding changes via JSON-RPC, an extensible action protocol, to the client and vice versa. How those actions impact the model and translating the model in a generic graph structure is the given task when implementing a server for a new modeling language.
- **Client**: The client is responsible for the rendering based on the messages received from the server and sending changes to the model in the editor as actions to the server. It does so agnostic from its environment, which will be handled separately. The primary concern of a new implementation is the specific rendering of the received graph and mapping it to the client actions.
- **Tool-Integration**: While the client alone may be working fine, it is only a standalone tool so far. To make use of a greater ecosystem of tools and enhance the workflow, it needs to be integrated into a common editor, e.g. VS Code. The integration needs to map the client to the specific tools available in a given environment, the process of which is greatly eased using the provided packages.

Server

Source model storage [Fouk]

As mentioned before, server and source model are intrinsically linked, with the latter containing the data and the former being responsible for storing and loading that data. Once loaded, it is the central state of a given session and any changes are applied directly to it and then further propagated.

Graphical model factory [Foui]

The graphical model is a serialized and generalized representation of a given source model. It exists simply for rendering purposes on the client and is not supposed to contain modeling-language-specific information. Simple graph-elements like nodes and edges are used for this purpose, however, with extensibility in mind.

Action handlers [Foug]

Any client actions to be applied to the model are sent as actions, which are, upon receival, directly applied to the source model. This modified model is then once again transformed via the graphical model factory and sent to the client for re-rendering.

It is worth noting, that the server is capable of servicing multiple clients at once using separate sessions, which are themselves handled by the server. All actions occur in the context of such a session.

Client

Rendering [Fouj]

As mentioned before, the client concerns itself with the rendering of the serialized graphical model. It does so using SVG elements, called views, that represent an object of the chosen graphical modeling language. Those views need to be bound to their corresponding elements in the graphical model.

Adding a new model element is therefore as simple as creating a new view using SVG, derived from (abstract) base views, and binding it to its own type. This type is a string used in the graphical model denoting the particular language construct. Some default views already exist, making custom views optional.

Layouting [Fouh]

Since graphical representation is, by nature of the topic, of quintessential importance, methods of changing the layout are not to be disregarded. GLSP, as documented, differentiates between two kinds of layouting: macro and micro. While the macro layout is handled solely by the server, as it concerns the placement and size of the elements, the client is responsible for the micro layout, which covers the display of an element's internal structure, e.g. positioning of a label inside a node. The server necessarily persists this information, but it is solely of importance when rendering on the client.

Client-Server Communication [Foug]

It has been mentioned before, but the communication between server and client, and even more broadly, the entire event flow, is controlled with actions.

Both sides have an action dispatcher that decides whether a given event should be propagated in the internal event flow or sent to its counterpart. Such external actions are transported using JSON-RPC.

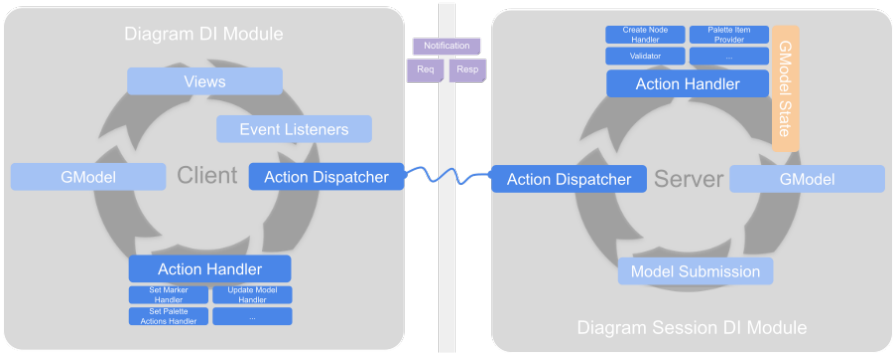

Figure 4.2: The action handler cycle between client and server. [Foug]

Since the server is the passive partner in this process, all event flows start from the client, triggered by events, whether they be startup of the client or dragging of an element. The subsequent action is either blocking or not, depending on the relevance for the client.

4.1.3 Browser technologies

Many Browser APIs are ubiquitous and do not warrant an examination, however some are less so and will be succinctly described here.

Web Worker API [WHA]

The purpose of Web Workers is to run operations in a background thread independent of the main execution, which is influenced by user interactions. As they are heavy-weight, they are supposed to be created for laborious tasks and long-running processes. They can't directly interact with the DOM or the main thread, but instead communicate via messages.

There are multiple types of workers:

- Dedicated workers are used by a single script, i.e. a different thread for the same window as the main execution.
- Shared workers can be utilized by many scripts running under the same domain as the worker, e.g. other windows, IFrames.
- Service workers are effectively proxies for network requests that are usually used to intercept network requests and enhance offline experiences, where the targeted server is not available.

4.2 Architecture

In the following, architectural diagrams and descriptions are used to both highlight the changes moving from the old to the new sandbox and showcase how the framework is used to accomplish this task.

4.2.1 Existing Architecture

The old sandbox is using Theia Cloud [Ecl] to provide the GLSP-client as an Eclipse Theia [Foud] based editor, while running a Java-GLSP-server. As this necessitates backend resources, especially since every user instantiates a new instance, there are costs associated with running this implementation.

Furthermore, this sandbox only provides the possibility to edit and work with a given graphical modeling language implemented via GLSP. There is no meta-editing of this language.

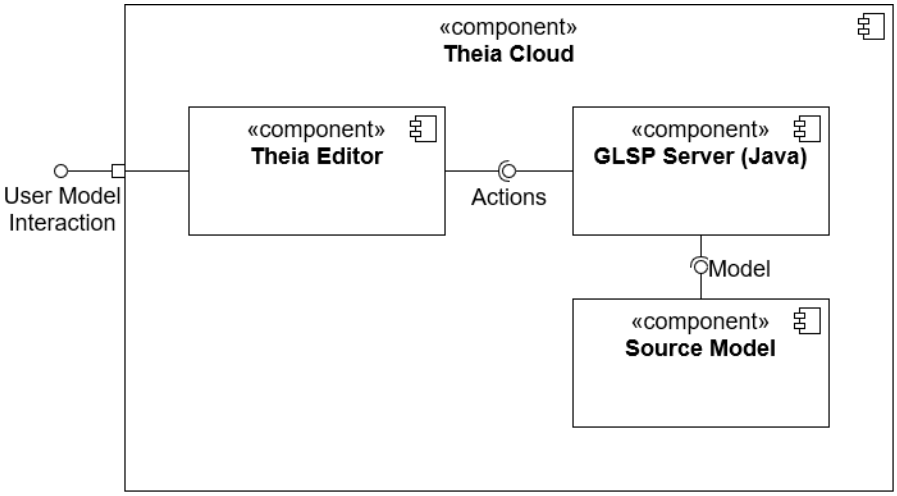

Figure 4.3: The GLSP sandbox hosted on Theia Cloud

4.2.2 New Architecture

In contrast, the new sandbox does not require any separate backend, since it is running solely in the browser. The GLSP-client is a standalone implementation using the browser, not based on any other editors like Eclipse Theia. The GLSP-server uses an alternative NodeJS implementation, which can be run in a Web Worker directly in the browser. The application, therefore, boils down to static files that can be hosted cheaply with no external computation resources required.

Additionally, some new features were implemented that mainly concern meta-development of the protocol. In other words, the provided GLSP implementation can be edited and re-compiled in the browser to change the behavior of the graphical modeling language

live. This provides the opportunity to not only work with the end result of the framework, but also test out development using the framework.

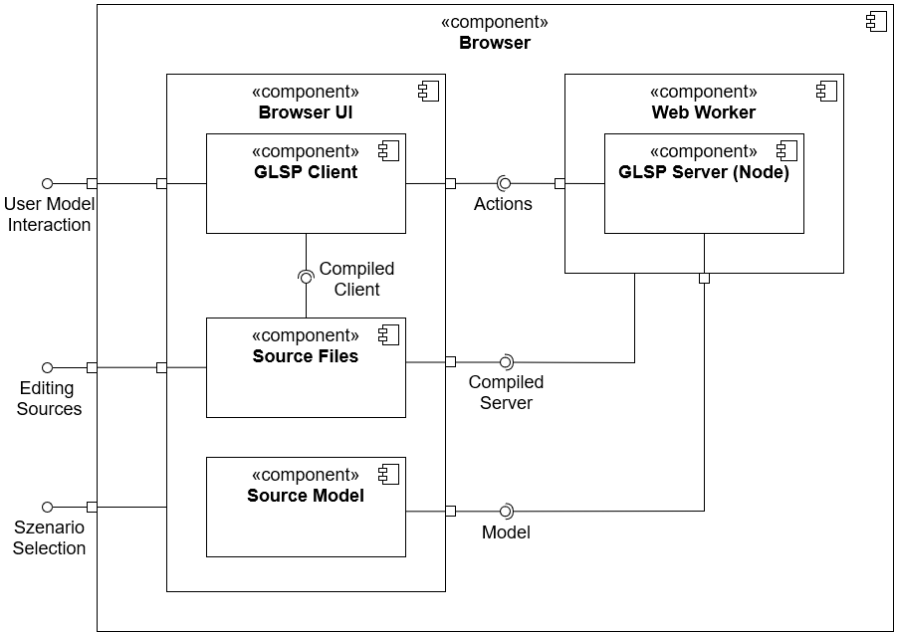

Figure 4.4: The new, browser-based GLSP sandbox

4.3 Plan

Now that the target has been specified, the steps to be taken during the implementation can be listed:

1. Running a GLSP-based modeling language solely in the browser.
   - The existing standalone GLSP-client expects to work with a GLSP-server using web sockets. Another connector needs to be provided that is able to work with web workers.
   - The TypeScript GLSP-server has to be adjusted to be able to run in a web worker.
   - Using the Workflow example, as it is already partly ready, a browser-only implementation of a GLSP-based modeling language should be showcased.
2. Adjust the Tasklist example for running in the browser, since it is easier to handle for further feature development.
3. Use the Monaco Editor [Micb] (the basis for VSCode's editor) to provide a read-only view on the implementation of the GLSP-based modeling language by displaying the source files.

- Integrate Monaco in the application.
- Prepare the used example to provide the source files as static content to use in Monaco.
- Display the resulting model editor in parallel as seen in Figure 4.5.

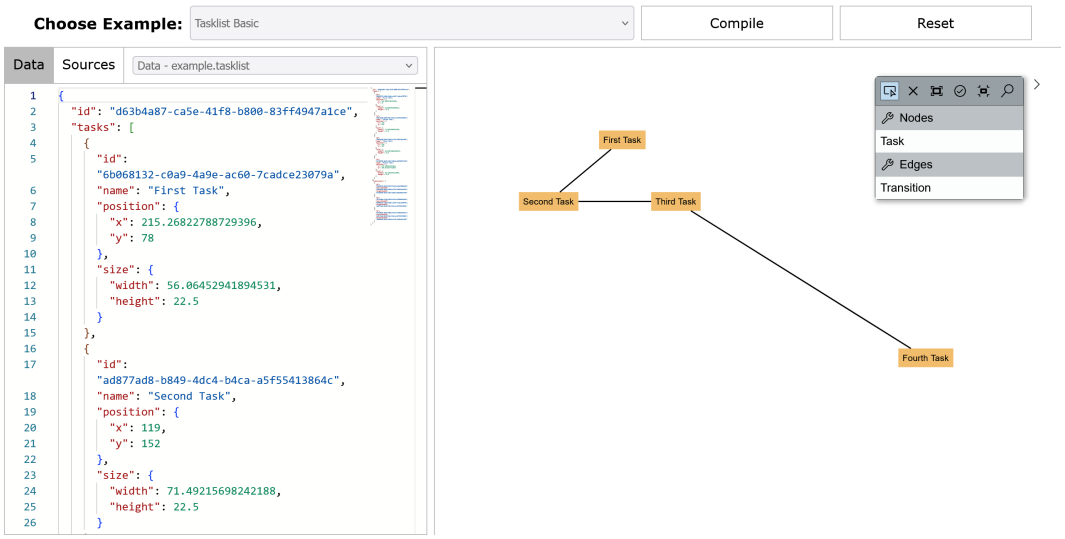

Figure 4.5: Finished GLSP editor

4. Reflect the changes in the model editor in a view of the source model.
5. Allow editing of the source files and subsequent compilation to change the implementation of the modeling language and, consequently, the resulting model editor. Therefore, enabling the user to develop a new GLSP-based modeling language (or rather, modify the existing one) and showcasing GLSP from a developer perspective.
6. Enable the selection of different examples, each being a different modeling language or, at least, a different set of files to allow editing of.

It should be noted, however, that this plan was revised multiple times during the course of the thesis, with this being the steps actually undertaken from a post-hoc perspective.

CHAPTER 5

Implementation

This chapter will, structurally based upon the outline given in the concept, describe the development process of the project, including any challenges and intermediate results. The underlying code is available at https://github.com/Sakrafux/glsp-browser-standalone-integration.

Running the GLSP example

The prerequisite step, before any implementation could begin, was to clone the existing GLSP repositories, as found in [Foue], and try it out locally and move step by step to the destination, that is, shift one component after another to the new architecture. However, this proved to be more difficult than expected, since the provided scripts were not fully correct in regard to the ports.

After resolving these issues by correcting the ports, this websocket-based implementation ran as expected. It was also reassuring that the client was working fine in the browser, despite usually being ran as a sub-component in other environments like Eclipse Theia.

5.1 Browser-only GLSP implementation

5.1.1 Readying the client

The `GLSPWebSocketProvider` used previously, expects an URL to connect with a running server instance. However, this needed to be replaced by a `GLSPWebWorkerProvider` that not simply connects to an already running server, but starts its own in a worker directly inside the browser. The main issue was to embed the communication between the server in the worker and the client in the GLSP connection architecture, but as it is built upon JSON-RPC, a fitting implementation was already provided, thus easily enabling the creation of the `GLSPWebWorkerProvider`.

An important feature to note, is that workers are created via an URL to an existing script that is supposed to be run inside the worker. However, URLs in the browser-available APIs of JavaScript are very versatile, allowing scripts to be written as strings and subsequently turned into blobs, which are file-like objects with their own URL. So workers can be started using scripts created at runtime.

5.1.2 Readying the server

There already existed a rather viable webworker-compatible version of the TypeScript GLSP-server that merely needed some adjustment in its configuration. Namely, it relied on a layouting engine called ELK [Fouc], whose JavaScript implementation had, as it turned out during development, issues running inside a worker that had to be resolved first.

It wanted to create workers itself and would, given certain circumstances like running inside a worker, use its own mock worker implementation that would be imported dynamically from the dependencies. However, since dynamic dependencies are not bundled and the given execution context inside the worker did, therefore, not know of this dependency, the server would abort. The solution was to manually provide ELK with a custom mock worker implementation.

5.1.3 Creating a standalone project

As both the necessary client and server changes were outside of the project's repository, but rather of the greater GLSP namespace, local compilation and copying of bundles and libraries was used to circumvent the inability to immediately import the given changes via the remote dependencies.

Specifically, the compiled (via Webpack) server and client bundle of the Workflow example were manually moved into the created project repository. The client was simply imported as a script in an html file and would start a `GLSPWebWorkerProvider` using the URL of the server bundle. Despite the browser being able to generate file-based URLs on the local PC, the worker implementation of all common browsers prohibits such URLs, because of security concerns and can not be manually overriden. Therefore, hosting the static files via a simple webserver i.e. local-web-server [Bro] becomes necessary.

This artifact is still available as a sub-component of the editor as a static resource via https://sakrafux.github.io/glsp-browser-standalone-integration/workflow/diagram.html.

5.2 Adjusting the Tasklist example for the browser

Although this first result was already very promising, the Workflow example was too cumbersome to develop further features with, be it its comparatively longer bundling times or its vast array of additional dependencies. Thus, a more nimble implementation was needed and luckily the GLSP project already provided one in the form of the Tasklist

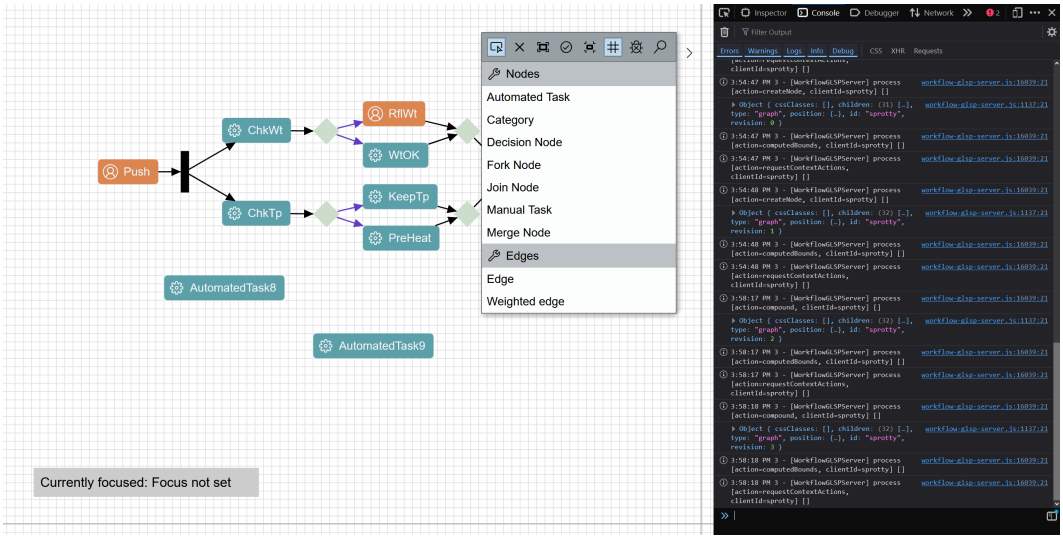

Figure 5.1: Workflow example running in a webworker

example (available in the official GLSP example repository https://github.com/eclipse-glsp/glsp-examples/tree/master/project-templates/node-json-vscode), which is a far more basic modeling language only allowing for simple nodes and edges.

However, while the adjustments necessary were rather small, mainly using the worker provider instead of websockets, some issues concerning the source model became apparent. Namely, the Tasklist example would directly read a JSON file as source, in contrast to the Workflow example, which already used external files dictated by the client. This functionality would have to be copied first.

Furthermore, the development workflow proved difficult at times, since the changed dependencies were not available yet and would therefore necessitate the manual copying of locally built dependencies. Furthermore, the debugging capabilities of scripts inside workers are lacking at best, which made the logger the debugging tool of choice, but would also require re-bundling of the application. Proving, why a lighter example project that could easier be iterated upon was necessary in the first place.

This artifact is also still available, same as Workflow, as a static resource via https://sakrafux.github.io/glsp-browser-standalone-integration/tasklist-basic/diagram.html.

5.3 Integrating Monaco

5.3.1 The editor

It has previously been shortly mentioned, but the Monaco editor [Micb] powers Visual Studio Code. While overarching file operations need to be handled separately, most text-based operations inside a file are already provided. This includes general features

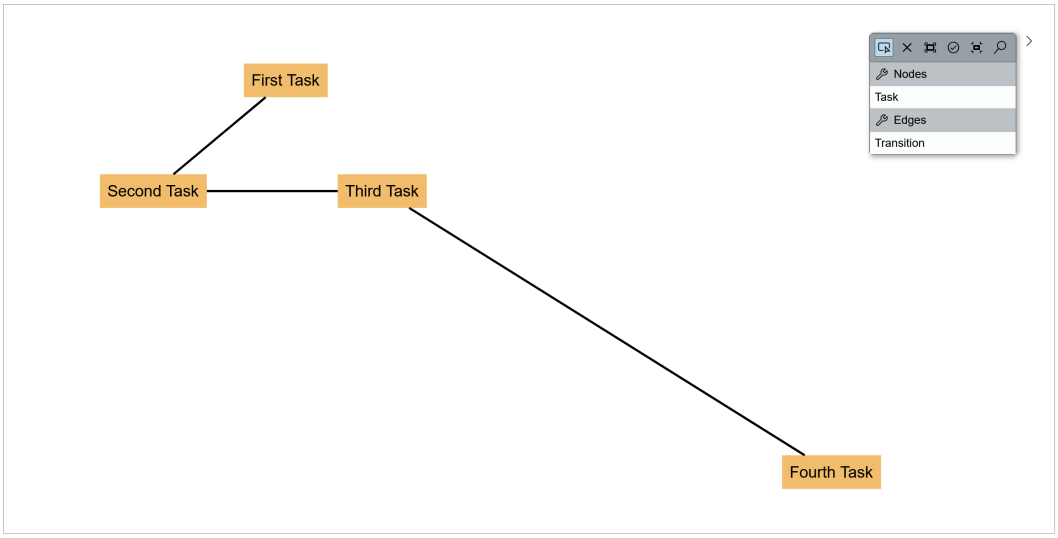

Figure 5.2: Tasklist example running in a webworker

like row numbering, text search or, depending on configuration, wrapping of rows, but also language specific ones, such as syntax highlighting, intellisense, and marking errors.

It is highly configurable and allows through proper configuration to know the context of files as needed, e.g. for linking imports. Furthermore, should ones preferred language not be available already, then the syntax for the purposes of highlighting can be defined.

The features provided by the Monaco editor make it very suitable to the use case in question i.e., a, eventually programmable, sandbox displaying source files.

The integration of which proved simple at first, but later on, more problems crept up. The options given and APIs available after creating a code editor instance are easy to use and extensive, allowing to update the content of the editor whenever necessary. Handling of TypeScript and other languages was no problem, since they are treated simply as text, however more complex formats, such as JSX, that is JavaScript XML, caused problems.

It is used in some source files for SVG icon generation, but in order to provide correct syntax highlighting, the text is not sufficient, since it can't be determinedly checked without understanding the semantics of the code. Monaco is also capable of this by keeping models of the files as temporary files, the URL resolution of which proved rather problematic.

Furthermore, changing the displayed files, especially with different languages to be used, created further issues that were solved by disposing and creating the models, or for different languages the entire editor, instead of simply changing the content.

5.3.2 The files

While the previous section explained the explicit integration of the editor, there needs to be content to fill the editor with.

As the goal was a completely static bundle that could be easily hosted, a static solution was needed to provide for consumption the source files that were bundled into the client and server bundle. Backwards engineering of the bundles proved fruitless, since even if the necessary sections were located, they constituted transpiled JavaScript, which differed from the TypeScript used in the source files, and were optimized by Webpack, making them barely legible.

The solution would be rather simple by adding into the root of the client and server projects a file enumerating all the files that should be available, which would then be copied using Webpack during the bundling. The overarching project bundler subsequently read from a configuration file defining the static bundles to provide, both the merged script files used in the client script and server worker, but also the copied sources.

5.3.3 Displaying editor and model

Now that both sections, the modeling language that should be usable in the browser and the editor to display the code behind the model, were implemented, they had to be integrated in one user interface. The Monaco editor could easily be created in a div-container marked with a specific id, the model, however, would require an additional iframe to ensure functionality and prevent interference.

This is where the aforementioned artifacts come into play as an iframe that can, similar to workers, be populated by an URL, which again, could be a blob, but at this point simply uses a static resource. At this point the editor is still read-only as well.

5.4 Reflect source model changes

The application is now served as a sandbox for a pre-generated model with a read-only display of the concerned source files. The next step is to also show the underlying data used in the model and to reflect changes at the moment of occurrence in this view.

Up to here, the save action in the model had no functionality, but would now be required, instead of persisting it to some storage, to communicate it back to the overarching application in order to be displayed via the Monaco editor. It is important to consider that the server is two degrees of separation removed from the application. One, as it is running inside a worker that needs to communicate with its surrounding context, and second, as this worker is situated inside an iframe that, again, needs to propagate the data to its parent.

The server implementation required only a slight extension in the storage, where the formerly stubbed function now used the message API to `postMessage` the source model

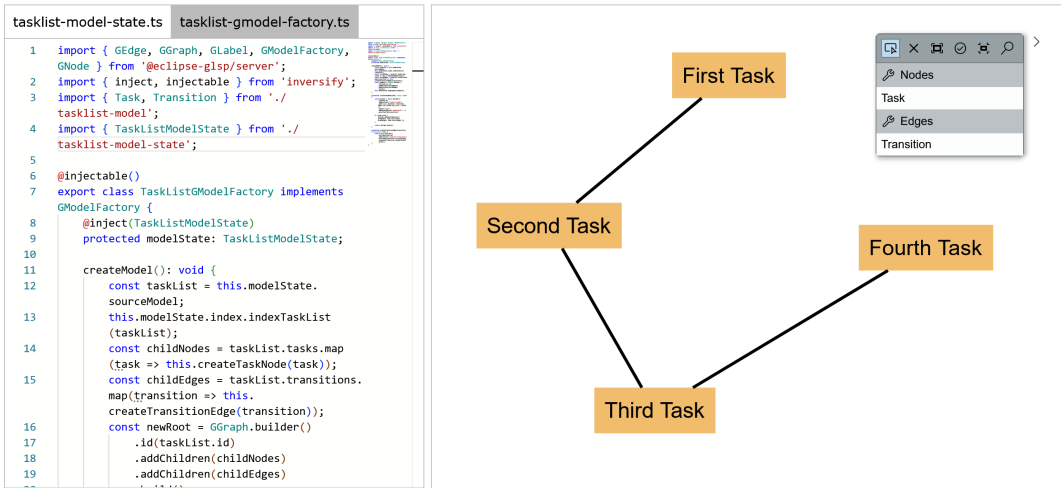

Figure 5.3: Read-only editor next to model

representation to its creator on both save and initial creation. The client would, however, be much busier as it not only needed to listen on the created worker for potential messages and errors, but also further propagate them to its parent via the aforementioned message API.

In the overarching application, a listener needed to be attached to the iframe housing the model and listening to the propagated source model, which it subsequently displayed in the editor.

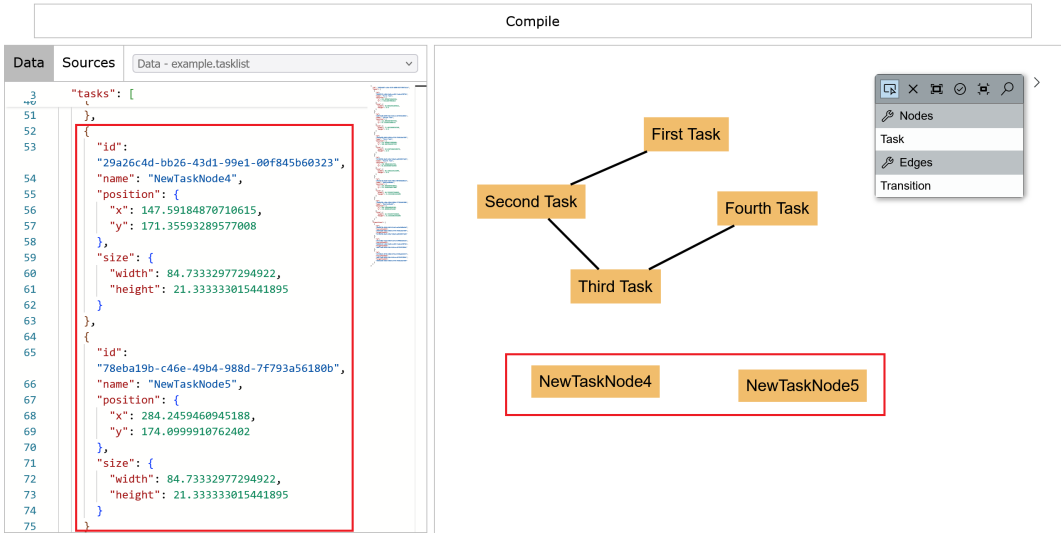

Figure 5.4: Changed model reflected in the editor

5.5 Source file compilation

Allowing for compilation of source files is the next, big step to take the application finally from a sandbox to try models implemented using GLSP to a sandbox for actually trying to implement a model in GLSP.

It has previously been mentioned that JavaScript is able to create script files at runtime, that such files have an URL exposing them and that both workers as well as iframes are utilizing URLs. The bundled JavaScript files used for the client and server are available and could carry the changes onwards. Lastly, the TypeScript files that are the source of said bundles are available.

5.5.1 Transpilation

Adjusting Monaco to allow editing was the easy part, but still left the problem of how the aforementioned facts could be leveraged in order to generate new client and server JavaScript bundles from the available TypeScript sources. The first step of which would be the transpilation of the source files.

As those files are written in TypeScript, they are, regardless of how they are injected, not immediately ready for use and need to be transpiled into plain JavaScript first. However, TypeScript is simply a npm package as any other and can be imported at runtime, providing, among all its features, the option to transpile files. The only issue, discovered through various incompatibility errors at runtime in later steps, was that the resulting files could vary immensely based on the TypeScript configuration, which needed to mirror the official GLSP TypeScript rules.

5.5.2 Bundling

While other vectors would probably be possible in order to inject the adjusted source files, now transpiled to JavaScript, the one chosen here, as alluded to previously, was the adjustment of the given bundles. The Webpack bundling process flattens all the dependencies into a single file, including node module dependencies, and marks them as a single source of truth in regard to the import of any given dependency. It also does so with all imported JavaScript modules, which subsequently occupy a fixed place in the large bundle.

This fact was leveraged by searching the relevant positions of the originally bundled source files and replacing them with the newly transpiled files.

5.5.3 Injection

The edited bundle strings could now be used to create script blobs, whose blob URL would now constitute the “compiled” model. This requires, again, a multi-layered approach, by modifying the html file representing the model to provide overrideable variables for the URLs that would either be imported via a script tag or started as a worker. This

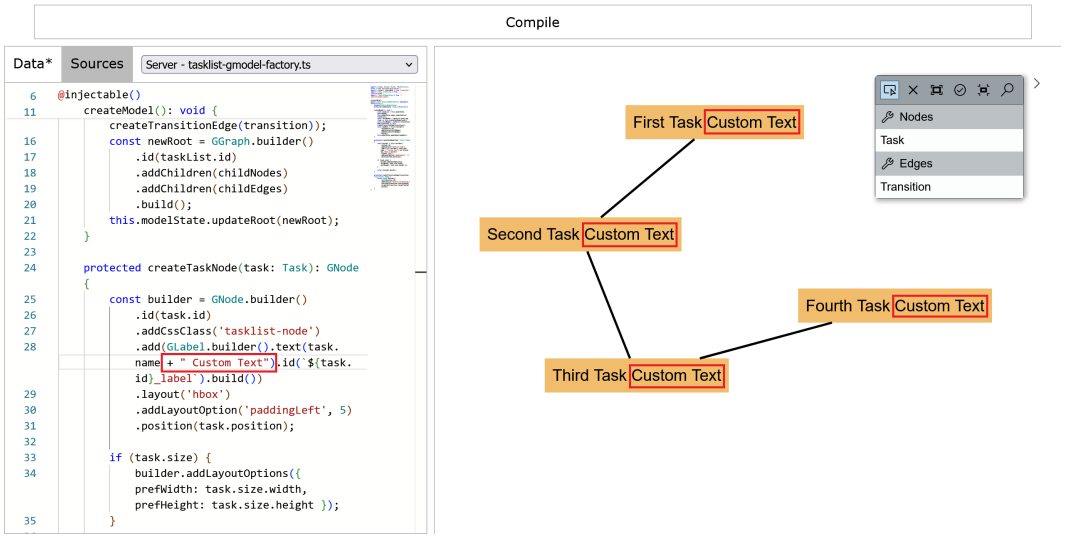

Figure 5.5: Changed sources reflected in model

adjusted html string would then also need to be used in blob creation, which would finally yield a new URL for the iframe to use.

5.6 Creating more examples

At this point, the application was almost feature complete, the only remaining tasks being the creation of more examples and making the examples accessible.

5.6.1 Creating the examples

As has been mentioned previously, but the display of source files necessitated the creation of a framework to select the files to copy and how they should be provided. This would now be extended to not only allow for configurability regarding source model type, source file language and more, but also to enable multiple examples to be taken from the same base implementation. In this way, one could use separate subsets of files to highlight different features of the underlying GLSP framework for documentation or tutorial purposes.

In the first step, the Workflow example was used for a first webworker prototype, but then discarded in favor of the more lightweight Tasklist example. Now it is brought back in the guise of another example for the sandbox.

5.6.2 Accessing the examples

The final issue, as the result of this thesis should be used flexibly to enhance existing documentation, was the development of a sensible resource schema to access all the

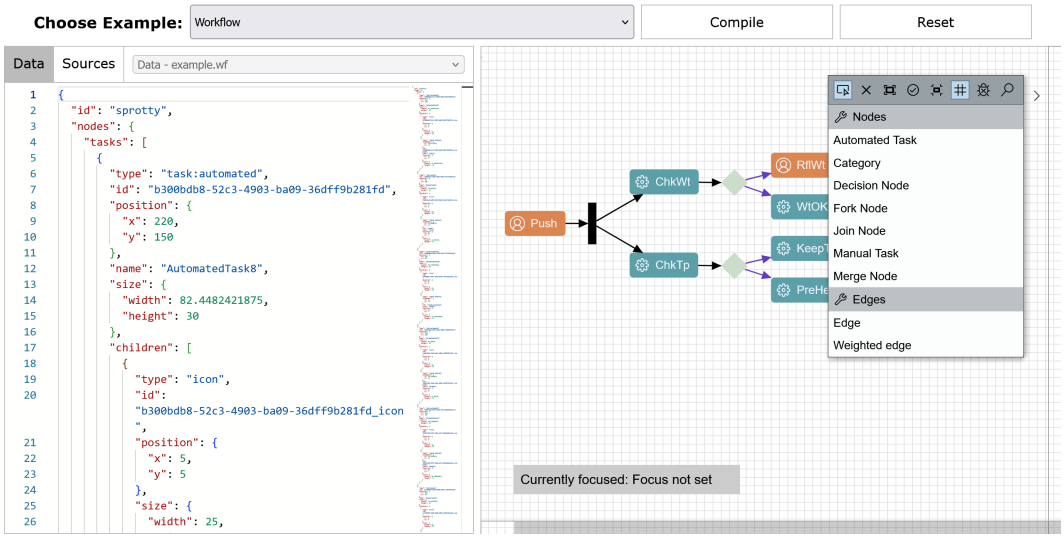

Figure 5.6: Workflow available

examples either together in a full sandbox experience or individually for specific documentation. Only the implemented model without editor should also be available to serve as interactive showcases.

The result would be the following:

- `/` - The complete sandbox with all examples
- `/«example-id»` - A singular sandbox
- `/«example-id»/«diagram-name».html` - The implemented model without editor

Deployment necessities

In addition to the actual implementation of the project, some further utilities were required as well, namely, the hosting of the resulting static files and a way to deploy them to that host.

As the project already used a GitHub repository at https://github.com/Sakrafux/glsp-browser-standalone-integration, it was natural to use GitHub Actions for the actual deployment process and, finally, GitHub Pages to directly host the files, available at https://sakrafux.github.io/glsp-browser-standalone-integration/.

CHAPTER 6

Evaluation

After establishing both the target to reach and the steps to reaching it, the merits of the completed application can be examined.

6.1 Light-weight Solution

First off, the primary goal was to substitute and replace the existing playground in Theia Cloud, because of its cumbersome and expensive nature. Instead, a more lightweight and statically hosted solution was sought out.

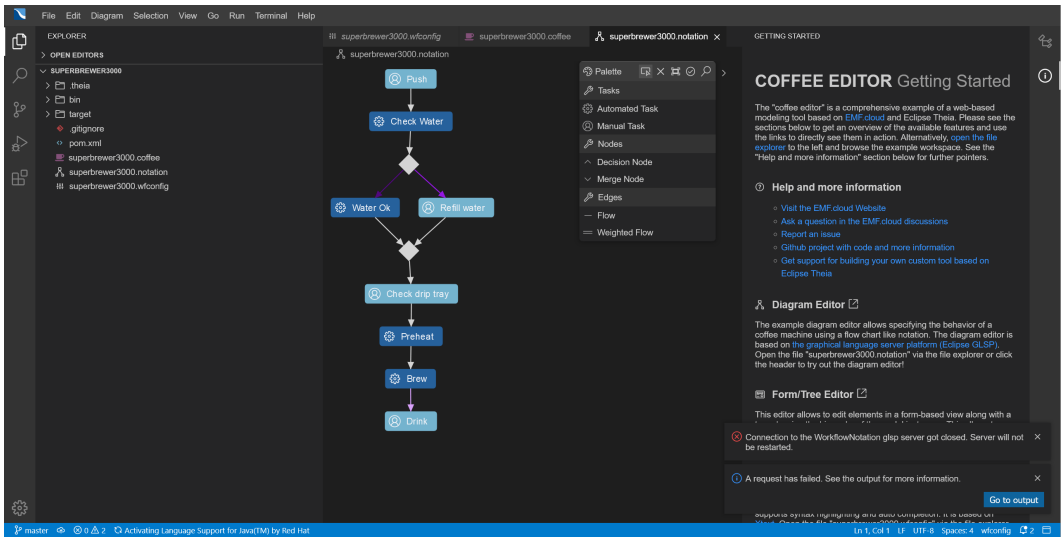

Figure 6.1: Theia Cloud sandbox

This impression is very fitting, as the Theia Cloud playground takes about half a minute until a model can even be seen, which includes the time of the startup. As there are

background processes running and potentially failing, sometimes the model can not even be displayed, because of untraceable errors. Both aspects being detrimental to actually convincing a potential user of this solution.

The new sandbox implementation, in contrast to this, needs no instances and just serves static content. The time it takes to load the application is merely defined by the network requests loading the bundles. Despite the main bundle being slightly larger than of the Theia Cloud solution, this is only a matter of milliseconds and more than compensated for by the lack of initial startup time.

The application itself runs extremely smoothly in the browser and the resource usage is barely noticeable. In the same manner, even the "compilation" of an example with changes to the source files, which would lead to manual changes in the megabyte-sized bundles, is handled almost instantaneously.

This performance makes it possible to replace even images or other non-interactive depictions with an iframe of the model, as is planned with some gifs on the documentation website displaying an implemented model.

6.2 Programmable modeling language

The secondary, and arguably more interesting, goal of the project was to not only replicate the existing solution, but to enhance it by means of enabling the user to edit the code comprising the model itself. This allows users to experience the developer workflow of working with GLSP to implement a modeling language editor.

6.2.1 Editing

For clarities sake, the editing workflow will be explained using an image of the user interface (Figure 6.2) with the three component sections being coloured green, blue, and red, respectively.

At first, an example has to be chosen, provided the complete sandbox with multiple examples is used. This is done using a simple dropdown in the green "Example and Compile"-section of the interface. The names are defined in the example configuration previously mentioned.

The current model, displayed in the red "Model"-section, is always available for use and shows the implemented modeling language that is the end result of any development effort using GLSP. Interactions with the model are reflected in the "Data"-tab of the blue "Editor"-section of the interface.

The user experience in the model section is driven by the underlying GLSP client implementation and independent of the sandbox. As the protocol already proposes methods to enhance usability, e.g. keyboard shortcuts, this would merely have to be applied in the specific client implementation.

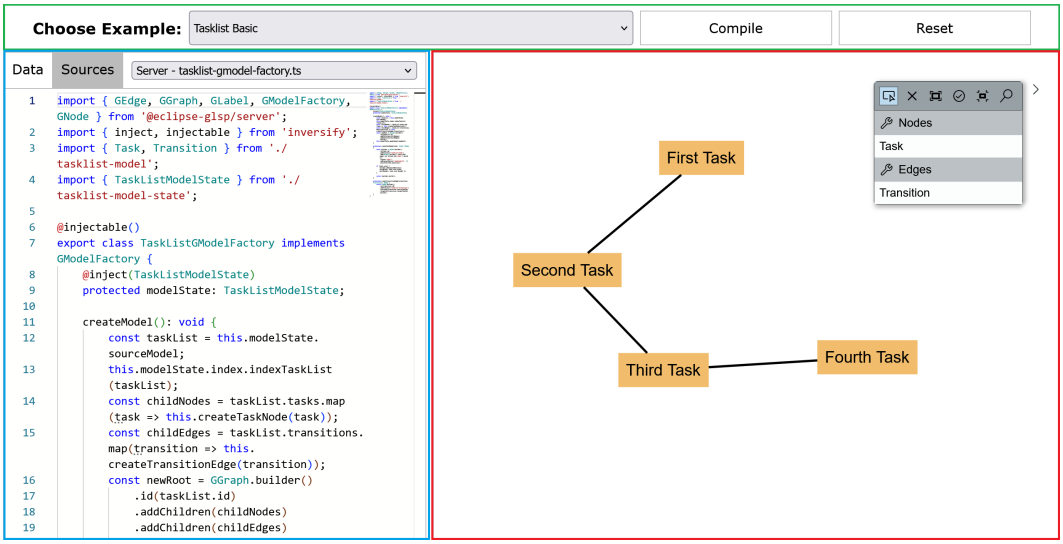

Figure 6.2: Sections of the user interface

The “Data”-tab has already been mentioned to reflect the *current* state of the source model, but the “Sources”-tab, found next to other tab in the “Editor”-section, allows for the manipulation of the *initial* state of the model language. In this context, “initial” means at the moment of compilation.

In order to provide a better user experience, the editor is configured to properly work with TypeScript files, including JSX, and provide some basic intellisense and syntax highlighting. It should, however, be noted that this could be further enhanced, since the imports have, for simplicity’s sake, been ignored and deactivated.

To apply these changes, the “Compile”-button in the green section has to be used, which will subsequently reload the iframe on the right with the new model, including the source model source file. This editable source model is not to be mistaken for the current source model found in “Data”.

Should a user want to reset the current model to its initial state, the aptly named “Reset”-button in the green section has to be used.

6.2.2 Defining examples

The ability and requirement of defining examples for the sandbox to use warrants some explanation. As the relevant `examples.json` file is properly described by the types defined in the `types.ts` (found at https://github.com/Sakrafux/glsp-browser-standalone-integration/blob/main/src/ts/ types.ts), those will be further expounded upon:

- `id` - A string identifying the example that needs to be unique.

- `name` - The string to be displayed in the example-dropdown. It doesn't necessarily need to be unique, but should serve to concisely inform users of its purpose.
- `diagram` - The name of the html-file to be loaded in the "Model"-section, including the file ending e.g. `"diagram.html"`.
- `sourceModelType` - Identifying the language used in the source model for purposes of properly configuring the Monaco editor.
- `path` - The path under which an example should be found, supposing that only a single example is required instead of the complete sandbox.
- `buildPath` - The build time path, which directory in `/examples` contains the sources for the example. Does not need to be unique as multiple examples could be based upon the same source code. This string is missing in the TypeScript definition, as it is merely important in the build step and not at runtime and therefore not needed in the code.
- `serverBundle` - The name of the bundled GLSP server implementation.
- `clientBundle` - The name of the bundled GLSP client implementation.
- `sources` - An array of the actual source files to be available in the "Editor" section for manipulation. Different examples based upon the same source may use a different set of sources for the sake of not overwhelming a potential user.
  - `name` - The name of the source file to provide including file ending.
  - `path` - The relative path in the resulting static file structure with the example directory as root.
  - `language` - The programming language of the source file for the sake of properly configuring the Monaco editor.
  - `side` - One of three possible strings "data", "client" and "server" that primarily serves to better differentiate files for an improved user experience.
  - `ignoreForCompilation` - An optional flag to skip a source file for compilation, which is usually the case for pure TypeScript files containing type information. As this is completely irrelevant for the compilation of the model and, on the contrary, detrimental due to an empty JavaScript module being emitted, it is sensible to preemptively exclude them.

Creating another example would consist of possibly writing a new GLSP implementation under `/examples` and then creating a fitting entry in the previously referenced `examples.json` according to the here described parameters.

CHAPTER 7

Conclusion

The target to be achieved, set at the beginning, was a replacement of an already existing sandbox environment with a cheaper, statically hosted alternative. Furthermore, faster invocation times due to leveraging the lack of network communication was expected. Lastly, the adding of additional features, namely the creation of a meta-sandbox enabling the editing of the implemented model was a very important, though not essential, target.

The value provided by such a sandbox is an increase in user interest and to help them evaluate whether the GLSP framework is suitable for their specific use case. The new solution further strengthens this value propositions by both, reducing the obstacles via a faster environment and enriching the sandbox with meta-editing capabilities, approximating a real workflow using the framework.

After evaluation, it can be concluded that the set target has been achieved to complete satisfaction, as corroborated by the external project partners.

Future Work

As this thesis was concerned with the creation of this implementation sandbox and its inherently time-limited nature, there are still many further avenues to explore and expand upon.

The application allows to already start developing one's own modeling language using GLSP. It does, however, not allow one to either save or download this development. Therefore, the application could be extended to provide a download containing a complete project setup including the changes made, simultaneously acting as a bootstrapping tool.

Another objective firstly rescheduled and later abandoned, due to infeasibility inside the time frame, is the extension of the Workflow example spanning all of the GLSP repositories. The problem to be rectified being the reliance on a source model corresponding completely

to the GLSP internal JSON graph model. It should be replaced with a different structure and the examples adjusted accordingly to properly showcase the power of GLSP without relying on artificial shortcuts.

Considering the functionality of the editor itself, it has already been mentioned that Eclipse Theia can be used to create custom IDE experiences. This may be leveraged to replace the currently implemented editor with a Theia IDE, thus enhancing the developer experience, but carrying over the compilation capabilities. On a related note, the creation of new files and providing a proper development experience may be desirable.

In regard to enhancing the model section instead of the editor, an interesting feature could be adding detailed traceability for the model and its source files. This would not only reflect the data changes but guide the user through the source files impacted by actions inside the model section.

However, this list is not exhaustive as they are simply some avenues of exploration that may warrant further examination.

List of Figures

Bibliography

[ABL+81] C. N. Alberga, A. L. Brown, G. B. Leeman, M. Mikelsons, and M. N. Wegman. A program development tool. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 92–104, New York, NY, USA, 1981. Association for Computing Machinery.

[BB47] Andrew Donald Booth and Kathleen H. V. Britten. *Coding for A.R.C.* Sep 1947.

[BL23] Dominik Bork and Philip Langer. Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.*, 18:9:1–16, 2023.

[Bro] Lloyd Brookes. Local-web-server. `https://github.com/lwsjs/local-web-server` (last accessed: 10.06.2024).

[Cro] D. Crockford. Json rfc. `https://www.ietf.org/rfc/rfc4627.txt` (last accessed: 03.06.2024).

[CVDK07] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, page 557–566, New York, NY, USA, 2007. Association for Computing Machinery.

[Ecl] EclipseSource. Theia cloud. `https://theia-cloud.io/` (last accessed: 07.06.2024).

[Foua] Eclipse Foundation. Eclipse epsilon. `https://eclipse.dev/epsilon/` (last accessed: 07.06.2024).

[Foub] Eclipse Foundation. Eclipse ide. `https://eclipseide.org/` (last accessed: 20.07.2024).

[Fouc] Eclipse Foundation. Eclipse layout kernel (elk). `https://eclipse.dev/elk/` (last accessed: 10.06.2024).

[Foud] Eclipse Foundation. Eclipse theia. https://theia-ide.org/ (last accessed: 07.06.2024).

[Foue] Eclipse Foundation. Github - glsp. https://github.com/eclipse-glsp/glsp (last accessed: 10.06.2024).

[Fouf] Eclipse Foundation. Glsp documentation. https://eclipse.dev/glsp/documentation/ (last accessed: 03.06.2024).

[Foug] Eclipse Foundation. Glsp documentation - action handler. https://eclipse.dev/glsp/documentation/actionhandler/ (last accessed: 03.06.2024).

[Fouh] Eclipse Foundation. Glsp documentation - client-side layouting. https://eclipse.dev/glsp/documentation/clientlayouting/ (last accessed: 03.06.2024).

[Foui] Eclipse Foundation. Glsp documentation - graphical model. https://eclipse.dev/glsp/documentation/gmodel/ (last accessed: 03.06.2024).

[Fouj] Eclipse Foundation. Glsp documentation - rendering. https://eclipse.dev/glsp/documentation/rendering/ (last accessed: 03.06.2024).

[Fouk] Eclipse Foundation. Glsp documentation - source model. https://eclipse.dev/glsp/documentation/sourcemodel/ (last accessed: 03.06.2024).

[Gro] JSON-RPC Working Group. Json-rpc 2.0 specification. https://www.jsonrpc.org/specification (last accessed: 03.06.2024).

[GSB23] Philipp-Lorenz Glaser, Emanuel Sallinger, and Dominik Bork. EA modelset - A FAIR dataset for machine learning in enterprise modeling. In João Paulo A. Almeida, Monika Kaczmarek-Heß, Agnes Koschmider, and Henderik A. Proper, editors, *The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings*, volume 497 of *Lecture Notes in Business Information Processing*, pages 19–36. Springer, 2023.

[Jet] JetBrains. Intellij idea. https://www.jetbrains.com/idea/ (last accessed: 20.07.2024).

[MB23] Haydar Metin and Dominik Bork. On developing and operating glsp-based web modeling tools: Lessons learned from BIGUML. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*, pages 129–139. IEEE, 2023.

[Mica] Microsoft. Lsp specification. https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ (last accessed: 03.06.2024).

[Micb] Microsoft. Monaco editor. https://microsoft.github.io/monaco-editor/ (last accessed: 11.06.2024).

[Micc] Microsoft. Visual studio. https://visualstudio.microsoft.com/vs/ (last accessed: 20.07.2024).

[Micd] Microsoft. Visual studio code. https://code.visualstudio.com/ (last accessed: 20.07.2024).

[Ovea] Stack Overflow. Developer survey 2023. https://survey.stackoverflow.co/2023/#technology-most-popular-technologies (last accessed: 06.06.2024).

[Oveb] Stack Overflow. Top ide index. https://pypl.github.io/IDE.html (last accessed: 06.06.2024).

[RJL+78] Dennis M Ritchie, Stephen C Johnson, ME Lesk, BW Kernighan, et al. The c programming language. *Bell Sys. Tech. J*, 57(6):1991–2019, 1978.

[Thu] R. Thurlow. Rpc rfc. https://www.ietf.org/rfc/rfc5531.txt (last accessed: 03.06.2024).

[WHA] WHATWG. Web worker specification. https://html.spec.whatwg.org/multipage/workers.html (last accessed: 03.06.2024).