HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

Code Panorama

Term Project

Department of Computer Science

University of Applied Science Rapperswil

Fall Term 2018

Authors:	Marc Etter, Patrick Bächli
Advisor:	Prof. Dr. Farhad D. Mehta
Project Partner:	IFS (Institute for Software, HSR)

Contents

Abstract

Objective

CodePanorama is a tool for software developers and reviewers. Its goal is to assist in identifying points of interest within a code-base to review. A software developer might join a new project and want to quickly find the most interesting parts of the code to get started. A supervisor must review the results of a project but does not have the time to look at the entire code-base. Instead, they look to CodePanorama to make an educated guess as to where their effort should be focused.

Procedure / Result

In contrast to other code metric tools, CodePanorama is designed to provide the user with a non-reductionist, "zoomed-out" overview of the entire code-base. It is up to the reviewer to find interesting patterns and curious anomalies based solely on indentation, spacing and line lengths, instead of the usual metrics.

After entering the URL to any git repository, CodePanorama will clone the repository in the background, and generate the panorama view. The result is basically a collage of all files in the repository, glued together.

Often, this new perspective on a code-base can find patterns such as duplicated code, excessive indentation, or any other feature the human eye might recognize.

Once such a feature has been identified, CodePanorama offers the functionality to simply click on a section of the panorama image. This allows the user to directly dive into the actual code at that location. From there, they can review the code in place, or just take a peek before switching to their tool of choice.

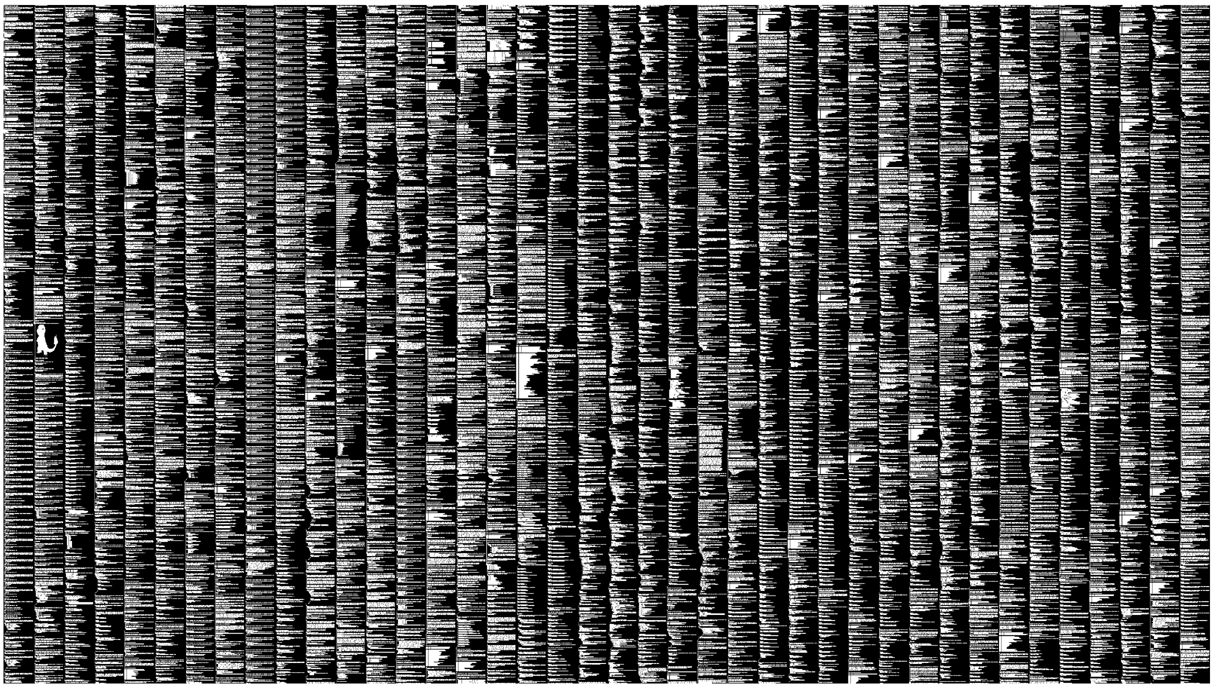

Figure 1 - Example of a CodePanorama (36'000 lines of code displayed)

Management Summary

Problem statement

Determining the quality of software code is hard. Several tools exist to assist in doing so, by providing mathematical formulas and metrics. This is useful to get an overall idea of the quality of a project but provides very limited help in getting a sense of exactly which parts of the code have problems. CodePanorama implements a new approach utilizing the human ability of pattern recognition in images to detect repetitions and anomalies inside code.

Target audience

CodePanorama is mainly targeted at two user groups:

- Software developers joining a new project, who would like to get a quick overview of the code.
- Project supervisors / examiners, who want to identify critical parts to review and grade in a time-efficient manner, without having to read through the entire project's code.

Approach

CodePanorama generates a panorama view of any project's code, by "gluing" together all files into a single large "poster". By "zooming out" of the code to a distance where only the silhouette of each file remains recognizable, new patterns start to emerge. With a bit of practice and programming experience, a user can very quickly identify unexpected patterns or unusual amounts of repetitions. By clicking on the parts in the panorama, the user can then dive into the code at that location to take a closer look.

Results

CodePanorama is publicly accessible[1] for anyone to analyze their project of choice. For more sensitive environments, we provide instructions on how to run CodePanorama on your own infrastructure for internal usage.

Currently, CodePanorama includes a small amount of filter options for which file-types should be included in the panorama view. Additionally, the size of the generated panorama can be configured, as well. Diving into a section of a panorama is fully functional.

Outlook

We are looking forward to using CodePanorama in business environments and receive user feedback. We are planning to extend the filter options, for users to include exactly the part of a project in their panorama they want. Furthermore, we intend to add color options to CodePanorama. With these color options, a user can highlight parts of the panorama according to various criteria, such as which code sections are changed the most. Finally, we would like to add a user management system. There, users could see all projects they previously analyzed in a user-specific dashboard view.

[1] https://codepanorama.io

1 Introduction[2]

1.1 Project Description

Several software quality metrics (lines of code per class/method, Cyclometric complexity, etc.) exist to estimate the quality and volume of large code bases. All metrics are reductionistic: They compute a number, or a set of numbers from a much larger volume of code. Although such metrics offer a quick overview of some aspects of the code, a more thorough code review is often needed in order to arrive at a more accurate estimation of code quality. For large code bases, such a code review can only be done on (often random) samples of the code.

The following idea is to be explored as part of this study project: In case it was possible to visualize the entire (or a large part of the) code-base on one large surface (screen, poster, etc.), it could be possible to identify patters or areas of the code as candidates for further inspection. In case version control is used, such a visualization could also provide historical and programmer-based information that could be relevant to such a further inspection.

1.2 Goals

The main aim of this project is to design and develop a software tool that visualizes the code contained in a Git repository for a developer to review it more effectively. The application must:

1. Be able to filter or highlight code based on file type, user, number of changes, age, or other yet to be discovered quality relevant properties.
2. Be easy and intuitive to use for a software developer or reviewer.
3. Have an attractive user interface.
4. If possible, require no prior installation (e.g. deployment as a single-page web-application).
5. If possible, require no application server.
6. Be maintainable and easily extensible (e.g. new types of filters or highlighting, etc.).
7. Use a CI / CD pipeline for development and deployment.
8. Use a Haskell-based toolchain as far as possible.
9. Be able to be effectively used for reviewing project code at the HSR.

1.3 Motivation

The original inspiration for this project came from a talk held at an ETH workshop on 13.10.2017. There a partial “panorama” of the Vampire Theorem Prover was shown. While the image already looked quite good, the process to generate it was rather tedious. It involved splitting the source-code into chunks small enough to not crash the LaTeX-compiler, which could only handle 180 pages at a time. Subsequently all the generated PDFs had to be merged together into a single page with yet another script.[3]
This project aims to automate these steps as much as possible and provide an easy-to-use application everyone can run to generate panoramas of their code-base(s).

[2] Topic presentation for term projects HS18 and personal communication with F. Mehta on 17.09.2018
[3] F. Mehta (personal communication, 26.09.2018)

2 Results

2.1 Evaluation

The sections below summarize our technical evaluations and decisions. A more detailed document can be found on the following page:
https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/wikis/documentation/design-decisions

2.1.1 Server Technology

As required by the original project description (see 1.1), the server is implemented with Haskell. The main challenges here were how to operate with git-repositories and how to generate the resulting panorama image.

For git, we decided to use the simplest, functional library we could find (aptly named “Git”). As this library only provides limited functionality, we use direct command line commands for missing features, such as cloning a repository.

At first, we used the image drawing library Gloss but later decided we wanted to generate our panoramas using the SVG format. The SVG format allows us to easily define sections and add metadata to the image. Additionally, this lets the client add their own styling through CSS on top of what the server already generates.

2.1.2 Web-Client Technology

After evaluating several frameworks and libraries providing functionality to build a web-client in a functional style, we decided on using Elm. Out of the compared possibilities it had the shortest setup time while also providing simple examples that worked out-of-the-box.

2.2 Architecture

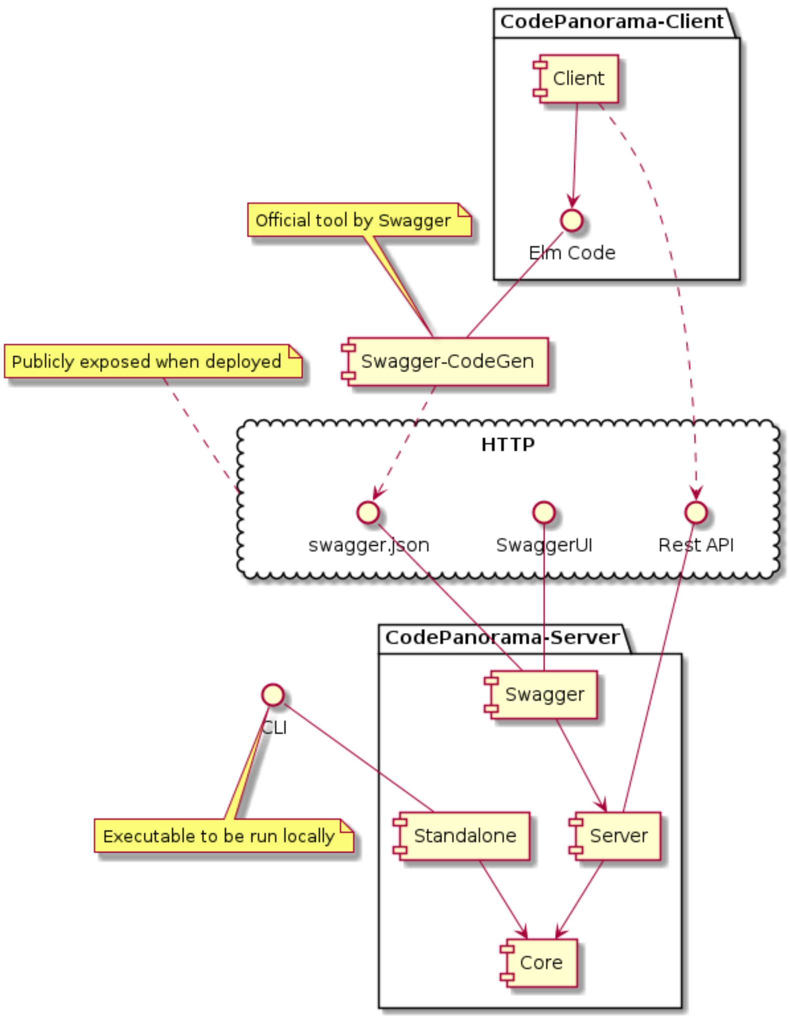

Figure 2 - Architecture of Code Panorama[4]

2.2.1 Layers

The application is divided into two layers. One being the presentation-layer and the other a combination of an application- and business-layer. So far, only trivial persistence was necessary, which we implemented with plain JSON files. Therefore, there is currently no “real” data-layer, rather the persistence is embedded into the business-layer. Communication between the presentation and application layers is based on the HTTP-protocol.

2.2.2 Modules

The application consists of multiple modules which are spread across the layers and are briefly described in the following sections.

2.2.2.1 Client

The client module contains all the code for the web-based user interface. It communicates with the server through a REST-API. Parts of the client-code are generated by the Swagger-CodeGen-module, based on the API specifications. The client is written in Elm[5], HTML, JavaScript, and LESS.

[4] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/tree/master#architecture, 11.12.2018
[5] https://elm-lang.org, 11.12.2018

2.2.2.2 Swagger-CodeGen

Reads an API-description file as defined by the OpenAPI specification[6] and generates the necessary data-types and request-functions as Elm-code. The generation is based on the official Swagger CodeGen[7]-process with minor modifications to accommodate for missing features in the default templates.

2.2.2.3 Swagger

Generates an API-description file as defined by the OpenAPI specification based on endpoint- and data-type-descriptions from the Server-module. Can also be used to run an instance of Swagger-UI[8] based on the API-description file.

2.2.2.4 Server

Defines and implements the API, and thus handles all the requests made by the client (or through Swagger-UI) and passes them on to the Core-module for further processing. It has built-in asynchronous functionality and can serve multiple clients at the same time.

2.2.2.5 Standalone

Originally used to test various libraries and their viability for the project. Could still be used as a CLI-tool for very simple tests but has none of the functionality provided by the client.

2.2.2.6 Core

The actual backend which does all the "heavy-lifting". Contains sub-modules to collect repository-data and generate panoramas.

2.3 Design Decisions

2.3.1 Client-Server Architecture

As outlined in chapters 2.1 and 2.2 we decided on a client-server-architecture. This decision was taken contrary to goals 4 and 5 of the project goals (see chapter 1.2) after consultation with our advisor. It is rooted in the fact that it would be disproportionally difficult and complex to implement a web-client in Haskell compared to other, more UI-specific functional languages (see the subsequent chapter for our final decision).

Since this meant that we had to work with two different technologies, we had to find a way to share type-definitions between those two. The solution came in form of Swagger. We found a way to generate a valid OpenAPI-specification from our Haskell-API-definitions. Thus, we were able to use the official code-generator provided by Swagger to generate the client-code necessary to work with the API.

2.3.2 Persistent Storage

As a pragmatic decision based on priority and time remaining, we decided to implement persistent storage in the cheapest way possible: We simply write information into JSON-files which are stored in the server's file system workspace. This does not support any sort of optimized querying like a database would. Furthermore, we are aware that this solution makes very weak concurrency guarantees. Since we currently do not store any critical information, we accept some amount of data loss in exchange for the

[6] https://swagger.io/specification, 11.12.2018
[7] https://swagger.io/tools/swagger-codegen, 11.12.2018
[8] https://swagger.io/tools/swagger-ui, 11.12.2018

quicker development time. If we later need to store data which we must not lose, we can still switch to a more robust solution.

2.3.3 Authentication

In order to properly support private repositories, we implemented a basic authentication method. Whenever a user requests a private (i.e. password-protected) repository, the server stores the user's credentials together with the repository URL and branch-name in a metadata file. Passwords are stored using the industrial standard algorithm BCrypt[9].

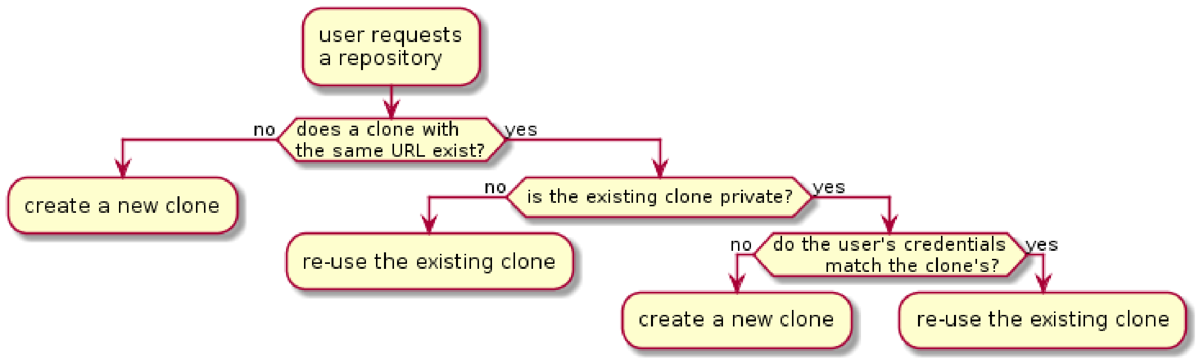
user requests a repository
does a clone with the same URL exist?
no
yes
create a new clone
is the existing clone private?
no
yes
re-use the existing clone
do the user's credentials match the clone's?
no
yes
create a new clone
re-use the existing clone

Figure 3 - Authentication decision tree

Due to an open issue[10], user credentials are also stored in cleartext inside the clone's `.git/config` file. Although this is unfortunate in comparison to our efforts to properly encrypt credentials, it is a known issue, with a researched solution. Additionally, the user interface warns the user of this issue and recommends running a local instance instead.

After the user has been granted access to a repository, the rest of the API relies solely on UUIDs to identify access to repositories. We plan to extend this with proper session management as part of the extended user and repository management.

2.4 Deployment

2.4.1 Docker[11]

For instructions on how to run your own instance of CodePanorama with docker, please refer to the docker part of the manual in chapter C.1.

2.4.2 Developers[12]

For instructions on how to run your own instance of CodePanorama without docker, please refer to the local part of the manual in chapter C.1.

2.5 Development

For instructions on starting development on CodePanorama, please refer to the manual in chapter C.2.

[9] https://en.wikipedia.org/wiki/Bcrypt 18.12.2018
[10] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/issues/70 18.12.2018
[11] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/blob/master/run-own-instance.md#docker, 13.12.2018
[12] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/blob/master/run-own-instance.md#local, 13.12.2018

3 Conclusion

3.1 Lessons Learned

3.1.1 Haskell

At first, the change in ecosystem from “main-stream” languages like Java was challenging. Tooling is drastically worse, well maintained libraries are rare, and documentation is either hard to read or non-existent. However, over time we grew accustomed to the mindset of the Haskell (and Elm) world.

We discovered that established practices for CI / CD using pipelines and docker can be used just as well with Haskell/Elm as with other languages. In a similar fashion, we would have known how to solve many problems in an OOP context. Transferring the solution to a functional programming language seemed hard at first. As it turned out, there is usually a known idiom that solves the problem elegantly.

3.1.2 Elm

Our initial decision to stay with Elm 0.18 instead of the recently released 0.19 has proven its worth. Many guides we consulted still referenced Elm 0.18. Furthermore, several libraries and tools we used were not even updated for Elm 0.19 yet.

3.1.3 Type system

Although it was not always easy to satisfy the compiler, we developed a strong confidence in the strict type system of these languages. Usually, we would be very certain that if the code compiles it will run correctly.

In order to understand more complex type signatures, it turned out to be helpful to just write down all types on a sheet of paper and do the type-operations by hand. This heightened our understanding of the type system in general and led to many “aha moments”.

3.2 Outlook

3.2.1 Continuation

We have already discussed with our advisor that we would like to continue work on CodePanorama. We have tentatively agreed upon continuing work as a bachelor thesis during the fall semester 2019.

3.2.2 Upcoming features

Two features that have not yet been implemented or are only available in a very crude version, are options and filters. Options allow the user to highlight parts of a panorama by certain criteria (e.g. change-frequency of a file, number of committers of a file, etc.). Filters allow the customization of what is included in the panorama at generation-time. As of now, the only possible filter is by file-type.

Another requested feature is a user management system. This would allow a user to see all their analyzed projects in a user-specific dashboard. There, the user would be able to delete previous analyses and download the generated images.

3.2.3 Bug fixing and optimization

During development a lot of bugs have been fixed and optimizations have been made. There are still some known quirks and issues that should be fixed in the future but have not been high enough of a priority.

3.3 Encountered Problems

3.3.1 GitLab

While GitLab is very good at most of the things it does, there are still some minor flaws we discovered, which caused some frustration over the course of the project. For a possible continuation of the project we want to evaluate other tools, which do the things in a way we would expect. Or at least a tool that allows for adjustments of existing workflows to suit our specific needs.

3.3.2 Time tracking and reporting

Without using third-party tools, it is not possible to generate reports based on tracked time. But still, those tools do not include data that would help to come up with fine-grained and detailed reports. From our experience during our respective engineering project, we know that better options exist.

Appendix A Project Plan

A.1 Phases / Iterations

#	Phase	Timespan
1	Inception	15.06.2018 – 25.09.2018
	Brainstorming and definition of problem statement	
2	Elaboration	26.09.2018 – 09-10.2018
	Creation of project plan, mock-ups, tech-stack evaluation, tech prototype, and simple documentation	
3	Construction	10.10.2018 – 02.12.2018
	Implementation of features and commence user-tests.	
4	Transition	03.12.2018 – 21.12.2018
	Completion of started features. Clean-up of source, code, UI and documentation. User-tests. Creation of final report.	

Table 1 - Phases / Iteration of project plan

A.2 Milestones

A.2.1 M0 – Problem statement

Problem statement is defined and accepted by all parties.

Planned: Beginning of semester; already completed before creation of this document.

A.2.2 M1 – Project Plan / End of Elaboration

Initial version of this document is completed. Basic UI mock-ups are drawn. Tech-stack is fixed, and relevant decisions are documented. Working prototype demonstrating bare technical essentials.

Planned: Completed by October 9.

Actual: Completed by October 9.

A.2.3 M2 – Standalone Application

Application is implemented as a standalone, locally executed program. Application displays meta-information about the selected git repository (e.g. Lines of Code, number of committers, etc.). The panorama view is generated, but without any highlighting or drill-down functionality.

Planned: Completed by October 21.
Actual: Completed by October 24.

A.2.4 M3 – Web-Client and Drill-down

All information is served via REST-API to the web-client; the UI resembles the mockups. User can “drill-down” into specific sections/files by clicking on the panorama view.
Planned: Completed by November 18.
Actual: Completed by November 22.

A.2.5 M4 – Customization

User is provided with various filters and options to customize the displayed panorama or file.
Planned: Completed by December 2.
Actual: Completed by December 6. Scope reduced to only include one basic filter and no options. Scope additionally included improved error handling, writing manuals, and handling private (password-protected) repositories.

A.2.6 M5 – Hand-in

Project is finished, and all necessary documents are handed-in. Potential further development as part of a bachelor thesis might be considered / out-lined.
Planned: Completed by December 21.

Appendix B Design Diagrams

B.1 Sequence diagram

The following diagram gives a quick overview of how the different pages in the client communicate with the server and in which situations redirects between the pages happen.

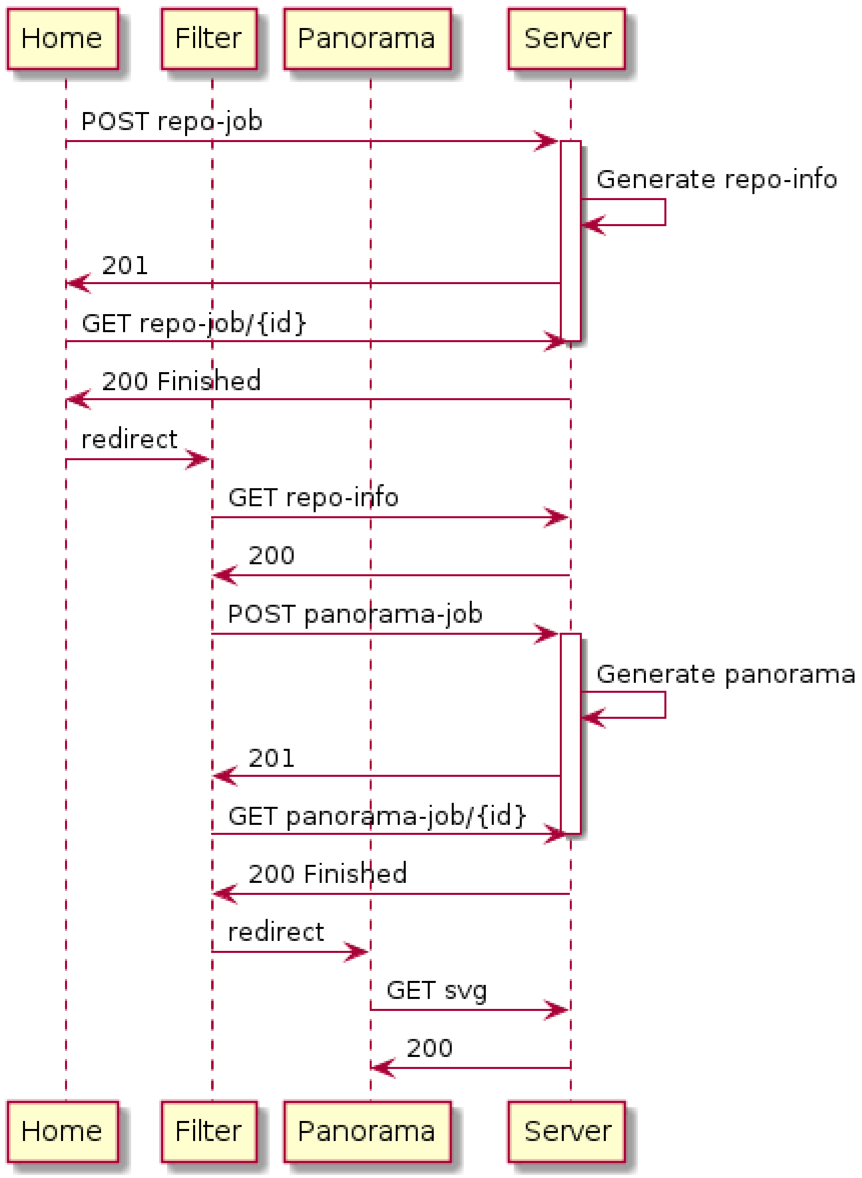

Figure 4 - Sequence Diagram[13]

The *GET repo-job* and *GET panorama-job* requests are sent periodically. The server responds to each request with the current status of the job running in the background. Once the server is done processing the original POST request, it will return a link to the produced result.

[13] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/wikis/documentation/design-decisions#sequence-diagram, 13.12.2018

Appendix C Manuals

C.1 Running your own instance of CodePanorama[14]

Running your own instance of CodePanorama

This manual describes how to run a local instance of CodePanorama, e.g. if you want to keep your private repositories truly private (which we can totally understand).

If you intend to make changes to the code, we recommend using the *Local* approach described below. If you simply want to run your own instance on your local machine or on your own server, we recommend using the *Docker* approach.

All command-line snippets are intended to be used with PowerShell or a similar shell. For usage under Linux, you might need to adapt some of the commands.

Local

This section describes how to build and run your own instance of CodePanorama without docker.

Prerequisites (Local)

The following software is required to be pre-installed on your machine:
All versions are the ones used at the time of writing - no guarantees are made for newer or older versions

- Windows 10 1803 (Building should be possible on other operating systems, but hasn't been tested outside of the docker build)
- Java JRE 11+28
- NPM 6.4.1
- stack 1.9.1 (GHC is not necessary, as stack will install it's own GHC anyway)

Installation of stack might take an hour or more, depending on your machine and internet connection!

Build

The simplest way to build CodePanorama is to use the following commands from the root of the cloned project:

Warning: The first build might take multiple minutes, as all dependencies are downloaded!

```
> npm install
> npm run build
```

Deploy (Local)

Under windows, the easiest way to start the locally built instance is:

```
> npm run all
```

[14] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/blob/master/run-own-instance.md, 13.12.2018

This will open multiple PowerShell windows to run the CodePanorama-Server, Swagger-UI, elm-live server for the CodePanorama-Client, as well as less-file-watch. This enables hot-reloading any changes to files in `CodePanorama-Client` and manual testing of the API through Swagger-UI.

A browser window/tab should open automatically, with CodePanorama running. If not, open http://localhost:8000. Swagger-UI will be running at http://localhost:6869 and CodePanorama-Server at http://localhost:6868.

If you do not wish to use any development features (such as extended logging, hot-reloading, etc.), use the following to start the server:

```
> cd CodePanorama-Server
> stack exec CodePanorama-Server-exe
```

Then, open `CodePanorama-Client/public/index.html` in your browser of choice.

Docker

This section describes how to build and run your own instance of CodePanorama using docker images rather than installing any build- or runtime-tools on your machine.

Prerequisites (Docker)

The following software is required to be pre-installed on your machine:
All versions are the ones used at the time of writing - no guarantees are made for newer or older versions

- Docker 18.06.1-ce

If you have access to the `gitlab.dev.ifs.hsr` docker registry, you can use the pre-built docker images (and skip the *Build from scratch* steps below):

```
> docker login gitlab.dev.ifs.hsr.ch:45023
```

Build from scratch

Currently, we do not provide any publicly accessible docker images. This might change in the future. However, with the following steps you can build all required docker images yourself (but it might take a while...):

```
> docker build --tag gitlab.dev.ifs.hsr.ch:45023/fmehta/codepanorama/server-build --
file .\CodePanorama-Server\build.dockerfile .
> docker build --tag gitlab.dev.ifs.hsr.ch:45023/fmehta/codepanorama/client-build --
file .\CodePanorama-Client\build.dockerfile .
```

You can now follow all steps for the *pre-built images*, skipping the `docker login`-part.

Build with pre-built images

If you only want to run your own instance, without actually building the entire application from source, you can skip the build steps below, if you have access to the `gitlab.dev.ifs.hsr` docker registry.

```
> $CodePanoramaDir = "C:/dev/CodePanorama"
```

Replace the value of `$CodePanoramaDir` with the absolute path on your machine to the root of the cloned repository. This variable will be used in most steps that follow, and must be absolute.

First, build the server code:

```
> docker run -it --volume ${CodePanoramaDir}:/opt/ws
gitlab.dev.ifs.hsr.ch:45023/fmehta/codepanorama/server-build /bin/bash -c "cd
CodePanorama-Server && stack build"
```

If you get an error message something like this:

```
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'C:\\dev\\CodePanorama\\CodePanorama-Server\\standalone\\.stack-work\\dist\\x86_64-
linux\\Cabal-2.2.0.1\\build\\CodePanorama-Standalone-exe\\CodePanorama-Standalone-exe-
tmp\\.stack-work\\dist\\x86_64-linux\\Cabal-2.2.0.1\\build\\CodePanorama-Standalone-
exe\\autogen\\Paths_CodePanorama_Standalone.dump-hi'
```

This can usually be resolved by deleting the `.stack-work` directory in the subfolders of `CodePanorama-Server` (not the `.stack-work` directly inside `CodePanorama-Server` !).

Then, generate the swagger.json specification from the server code:

```
> docker run -it --volume ${CodePanoramaDir}:/opt/ws
gitlab.dev.ifs.hsr.ch:45023/fmehta/codepanorama/server-build /bin/bash -c "cd
CodePanorama-Server && stack exec CodePanorama-SwaggerGen-exe"
```

Now you can generate the client-code from the swagger.json specification:

```
> docker run -it --volume ${CodePanoramaDir}:/opt/ws openjdk:11-jre /bin/bash -c "cd
/opt/ws/CodePanorama-Server/swagger && java -jar swagger-codegen-cli.jar generate -l
elm -i ../swagger.json -t swagger-templates -o ../../CodePanorama-Client"
```

Finally, build the client:

```
> docker run -it --volume ${CodePanoramaDir}:/opt/ws
gitlab.dev.ifs.hsr.ch:45023/fmehta/codepanorama/client-build /bin/bash -c "npm run
build-client"
```

Deploy (Docker)

If you don't have access to the `gitlab.dev.ifs.hsr.ch` docker registry:

```
> docker-compose --file CodePanorama-Server\docker-compose-local.yml build
> docker-compose --file CodePanorama-Server\docker-compose-local.yml up -d
```

If you do have access, it is quicker to use the following:

```
> docker-compose --file CodePanorama-Server\docker-compose.yml up -d
```

Now, the CodePanorama-Server will be running at http://localhost:6868 and Swagger-UI at http://localhost:6869.

Persisting the workspace

If you want to keep the server's workspace (i.e. repo clones, panorama SVGs, and repository infos) persistent, you can add the following to the docker-compose file at the end of the `code-panorama-server` section (replacing `C:/dev/CodePanorama/ws` with any path on your machine where you would like to save the server's workspace):

```
volumes:
  - "C:/dev/CodePanorama/ws:/opt/ws/workspace"
```

Deploying to your own server

If you wish to deploy CodePanorama to your own server (i.e. not on localhost), the following change needs to be made, so that the client will find your backend:

1. Edit the `apiBasePath` in `CodePanorama-Client/env/prd.js` to point to your own server/domain, where CodePanorama-Server will be running.
2. Build the client code (using either the *docker* or *local* steps above).

Alternatively, you can edit the `apiBasePath` in `CodePanorama-Client/public/out/env.js` instead, but this will be overwritten, if you build the client code anew - but may be quicker, if you simply want to change the `apiBasePath` in a pre-built distribution.

No changes are necessary to deploy CodePanorama-Server to your own server.

C.2 Developer Guide[15]

Developer Guide

This guide contains instructions, recommendations, and best-practices we gathered during the course of this project for working on CodePanorama and with Haskell/Elm in general.

Although it is possible to work with basically any IDE, we found Visual Studio Code to work well. This section describes our recommended setup, which we used ourselves.

Prerequisites

See Prerequisites (Local).

Visual Studio Code Extensions

The following Visual Studio Code extensions worked well for us:

- elm 0.24.0
- Haskell Syntax Highlighting 2.5.0
- haskell-linter 0.0.6
- Haskero 1.3.1
- hoogle-vscode 0.0.7
- indent-rainbow 7.2.4
- Rainbow Brackets 0.0.6
- stylish-haskell 0.0.10

Additionally, some of these extensions require external software:

```
> stack install hlint
> stack install intero
> stack install stylish-haskell
```

Workspace Setup

For the code extensions to work properly, we found it best to use two Visual Studio Code windows, where the `CodePanorama-Server` and `CodePanorama-Client` are used as root workspaces, respectively.

Working with the project

For convenience, we have setup `npm` shortcuts for all commonly used tasks during development. Take a look at `package.json` to see a list of all commands, but the most commonly used ones are:

`npm run build` - builds server, generates client code and builds client
`npm run test` - tests server and client
`npm run swagger` - generates swagger.json specification and client code
`npm run build-server` - only builds the server
`npm run build-client` - only builds the client
`npm run server` - starts CodePanorama-Server
`npm run client` - starts CodePanorama-Client (hot-reloading live-server) and opens a browser window with the home page
`npm run swagger-ui` - starts Swagger-UI
`npm run all` - builds and starts everything

[15] https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/blob/master/dev-guide.md, 13.12.2018

Appendix D Self-Reflection

D.1 Report by Marc Etter

For a while now, I have been intrigued by the ideas of functional programming. Since I predominantly use object-oriented languages at work, I only had the chance to dabble in functional languages on the side. Additionally, I am a big fan of measures, tools, and metrics to analyze code and provide feedback on issues and quality. Combining both fields together in this term project therefore was an easy pick for me.

Although I struggled at first, I quickly came to appreciate all the new paradigms and mindsets proponed by the Haskell language and community.

Working on the term project felt much more relaxed than on the Engineering Project last semester. Partly because the regulations for formal documentation and regular hand-ins are more relaxed. While I often found myself in the mentor role during the Engineering Project, it felt more comfortable to be working with someone who was learning the language and eco-system from scratch with me. We had many laughs about quirks and absurdities deemed regular in this community and faced the same challenges.

I had two personal goals coming into this term project: Learning and using a functional language in a real project; and, obviously, implementing a useful tool for generating CodePanoramas. Although the true usefulness of the tool in its current state might be debatable, it fulfills the basic requirements we set for ourselves. As to whether I achieved my goal of learning a functional language, I feel like the following quote best summarizes my experience, as to why I think it was worth learning:

«A language that doesn't affect the way you think about programming, is not worth knowing.» - Alan Perlis, 1982

D.2 Report by Patrick Bächli

Just like every other software project I have worked on so far, this one had its pitfalls and challenges. But this one was a bit different from the beginning, since one of the requirements was to use a functional programming language. Most of my experience stems from working on business applications used in the service sector, which so far is not a place where that paradigm has gained a considerable foothold. So, I saw – and still see – this as a very good opportunity to gather some first-hand experience in it.

After a rather bumpy start, I eventually managed to leave enough of the imperative mindset behind and became more and more comfortable using my limited functional knowledge and expand on it. This certainly would not have been as easy or possible at all without the help and patience of Marc. To me it almost seemed like he was "playing a home match" at times, but he never hesitated to pause and explain a piece of code or a consideration he had been making.

Looking back upon the weeks since the start of this semester, I had a really interesting and educational time. From the beginning, I had to leave my paradigm-comfort zone and get accustomed with something mostly new and unfamiliar. But despite of all the difficulties that this has brought with it, I pulled through and tried to learn as much as I can. The first fruitful successes are already emerging, as I have started my first private project based on the functional paradigm during this semester.

Appendix E Meeting Minutes

Please refer to the following page: https://gitlab.dev.ifs.hsr.ch/fmehta/CodePanorama/wikis/meeting-protocols/index.

Appendix F List of figures

Appendix G Glossary

A

API	*Application Programming Interface*

C

CI / CD	*Continous Integration / Continous Delivery*
CLI	*Command-line Interface*
CSS	*Cascading Style Sheets*

H

HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*

J

JSON	*JavaScript Object Notation*

L

LESS	*Leaner Style Sheets*

O

OOP	*Object Oriented Programming*

R

REST	*Representational State Transfer*

S

SVG	*Scalable Vector Graphics*

U

UI	*User Interface*
URL	*Uniform Resource Locator*
UUID	*Universally Unique Identifier*