Microservice Deveolopment Tool

Marc Bertelsen berte20@student.sdu.dk

Jesper Minor Melholt jems@tv2.dk

Bachelor Project in Software Engineering

June 2023

The Maersk Mc-Kinney Moeller Institute

University of Southern Denmark

Abstract

This project entails the process of creating a distributed software system, using microservice architecture. The system is designed with the main purpose of creating/generating new microservice that end-users have designed.

The report will go through the process of analysing, designing, and implementing each of the microservices in the core system. In the report there is also the details of the interface for the code generation microservices, as well as a single implementation of such a microservice.

The report concludes with a working core system that meets all the necessary requirement. Users can use a frontend website to design microservices and send requests to a code generation microservice that can generate new microservices written in Java, that the user can then download.

Contents

1 Introduction

Having previously made a few simple backends and microservices, it was found that getting an overview of how the system look as a whole became increasingly difficult, as the different teams made progress on the microservice they worked on. Some of these simple microservices, were also very similar, and so the work seemed somewhat repetitive. It would seem that there could be a better way to systematically design these microservices and have a system to generate the common part that made these microservice similar.

This project, nickname *Project-Sirup*, will try to tackle this concern, by creating a web platform where developers can collaborate on designing microservice. The platform will not only be for designing these microservice, but also for generating as much of the boilerplate and setup as possible.

Designing most of the system on the web platform, will also give the developers a place to get an overview of the different microservices, as they will be viewable from a single page.

2 Problem Description

2.1 Context

A very important decision to make when designing a complex software system, is the architecture of the system. These systems were commonly using the layered architecture [10][p. 137-138], however this is changing, and a new architecture pattern is gaining popularity [5], this is the microservice architecture [8][ch. 1].

2.2 Problem

The process of making a system that uses the microservice architecture, either from converting an existing system or creating an entirely new one, requires designing a varying amount of microservices, depending on the system. Deciding how and why these microservices are going to communicate, and how the entire system is going to look like at the end, can potentially become complex. When the design has been finalized and the different microservices are being implemented, there will be a lot of boilerplate for each, setting up the project, including the needed dependencies. This is a time-consuming process that might be repeated multiple times, if two or more services share similarities or if the developers are involved in developing multiple systems using microservice architecture.

2.3 Proposed Solution

An idea for creating a tool for developing/creating a new microservice architecture system. The tool will both be used for writing the documentation for the different microservices and for creating much of the boilerplate for the new microservices, from the given documentation. The tool should be able to create microservices that use different programming languages, databases, and communication protocols. The tool must also give the people involved an overview of the entire system, so it will be possible to see how the different microservice are expected to connect and interact with each other.

The tool should not limit the uses to any language for the generated service, and since a new language could some at any time, means that the system should be easily extended to feature new languages to offer.

The use of the tool can be accessed through a GUI to make it easy to see the options available, however some users might still want to make the generation files themselves, and so is will also be possible for users to provide these files.

The tool could help the users in the process of designing the different microservices, by given recommendations.

3 Analysis

3.1 Shareholders

Since the case does not directly make any mentions of other people being invested in the development of this software tool, they must be inferred from the desired outcome of the tool. The first shareholder would be the developer team behind of the tool itself, as the team will invest time into developing the tool and wish to see the project succeed. Second group of shareholders are the developers, who might use the tool in the future to develop their own projects, and so want a well made tool. This group is the intended end users of the tool. As the project aim to develop a tool with extendibility in mind, so future developers can add new functionality to the tool in the form of additional services for generating microservices in other languages. We can identify a third group of shareholders, the people how need the foundation of the tool to be well made, to make the extendibility as simple as possible. It is possible that these future developers are the same as the original, but the ability to easily extend the tool is still the same.

From the above the following shareholder are identified:

- The Developers
- Future Developers
- End Users

3.2 APIs

The final tool must provide the user with different choices of API for the microservices they design. Because of this, information needs to gather on the different types of API that are being used to develop microservices.

Request-Response

A common and more traditional approach for API is the *Request-Response*, where the client sends a request to a server and awaits a response. That means that the client is responsible for getting the information and staying updated. This can cause the client to potentially refresh without any information being updated, which waists resources on the server as it will have to process every request.

REST

Arguably the most simple type of API is *Representational State Transfer (REST or RESTful)* [8][p. 23]. *REST* are very useful for exposing resources. It uses HTTP with different endpoints corresponding to different resources. These endpoints are paired with a HTTP verb for different functionality.

- GET
- POST
- PUT
- PATCH
- DELETE

Given each resource is linked to a different endpoint, fetching data for multiple resources will cause the consumer to fetch multiple times, one for each resources type. This cost both time and computing resources for both provider and consumer, and so is not ideal for every microservice.

RPC

Remote Procedure Call or *RPC* [8][p. 20] is a technique that can be used to make APIs. There are many different implementations of RPC, each one working slightly different, but with the same concept. The network calling is abstracted away by a RPC framework. This makes it so that the usage of the API will look like a local dependency, which makes it easy to integrate into the code.

RPC is more focused on running some method/procedure, unlike REST that is focused on resources. This means that choosing one over the other is dependent on the responsibility of the service in question.

GraphQL

One problem that might be encountered when using a REST API is that multiple API call are required for a single page. This requires both time and resources to accomplish. Solving this issue is where *GraphQL* [9] comes in. Rather than having an endpoint for every resource, it only has one endpoint. This endpoint allows for the caller to send queries to the API, that the service then resolves. This way it is possible to get all the resources needed in a single API call. This makes GraphQL a better choice for service that are expected to send multiple resources based on a query.

Event-Driven

Taking an *Event-Driven* approach to an API flips the responsibility to make the server for updating the client when information is changed and so when an update is necessary. This means that the server is not processing unnecessary request and save resources by this.

Websocket

When there is a need to send data to and from both the client and the server, it is possible to send request every time. However, this is very wasteful, and can cause big delays in the code execution. A better way to do this is by using *Websockets* [6]. These allow for long-lived connections between the client and server, where both can transmit data over. So rather than establishing a new connection each time and closing it again, it will just keep it open for the entire duration when it is needed. This connection then allows for two-way communication.

ServerSentEvent

Server Sent Event or *SSE* [7], is a technique where the client can subscribe to a server and listen for incoming requests from the server. The server can send these requests based on different events in the server itself or from external actors. By using this technique, it is possible to make the client update itself without calling the server periodically, this saves both time and resources.

Webhook

Webhooks [4] are a way of providing a callback from the client to the server. The client is the one initiating the process, but rather than wait for the server to finish processing the request, it can continue running. When the server then finishes processing the request, it can call the callback address with the result of the request. The client can then update itself with the new data. This technique requires that both parties are capable of sending and receiving requests.

3.3 Databases

Each microservice in a microservice architecture system, will, if needed, have its own database. This should also be reflected in the design tool. There are different types of databases, each with their own use cases. This means choosing the best one is based on what and how the data will be stored and used.

SQL

Perhaps the most common type of database, is the relational database. This type uses a strict schema for the data, where the attributes and relationships between data is defined. Relational databases use Structured Query Language (SQL) [2] to create these schemas. SQL is also, as the name implies, used to query the database.

NoSQL

Not Only SQL or NoSQL [1] refers to any database with implementations different from relational databases. There are many different NoSQL databases, but for this section, we will only look at a few.

Document

It might not always be necessary to have a strict structure to the data we want to store. It might be different from entry to entry, or we might expect it to change over time. *Document* [1] databases solve this by storing entries as individual documents, these will typically be using some common format, like JSON or XML. A key aspect of this database type is that it is schemaless.

Key-Value

This type of database is very similar to using a Map data-structure in code, the main difference is that a *key-value* [1] database, will have persistence. So, this type is very useful for storing data with no relation, where we always want to make queries based on some unique identifier.

Graph

We can have times where we are more interested in how the data is related, than the actual data itself. In a *Graph* [1] database is the data structured into a graph where the data is represented as nodesa and the relationships between data is represented as edges in the graph. It is then possible to make queries based on these relationships.

3.4 Requirements

From the previous sections we found the requirement for the system, this includes both **functional** and **non-functional** requirement. All requirements are placed into one of the two categories and are prioritized based on the MoSCoW method. Where **must-have** requirement are part of the project *MVP*, this means that these requirements must be implemented to consider the project a success. The **should-have** requirements are next in line, when

the *MVP* is complete, and will be implemented if there is time. All **could-have** are requirements are 'nice-to-have', and would make the overall product more complete, but are not essential or important. The **would-have** or **wish-to-have** requirements are completely out of the scope of the project. To keep this section short and clear, only the **must** requirements are listed below, the full list of requirements can be found in appendix A.

Functional Requirements

ID	Name	Description
F-01	Generate Microservice	The system **must** be able to generate microservice
F-01.1	Specify Language	It **must** be possible to select the language of the microservice
F-01.2	Specify Database	It **must** be possible to select the database of the microservice
F-01.3	Specify API	It **must** be possible to select the type of API for the microservice
F-01.3.A	Specify Endpoints / Methods	It **must** be possible to define different endpoints/methods for the API chosen for the microservice
F-02	Designer	The service **must** provide a tool for designing microservices
F-03	Collaboration	Users **must** be able to collaborate on their work with other users
F-03.1	Organisations	Users **must** be able work with different organisations
F-03.1.A	Create Organisation	Users **must** be able to create organisations for their work
F-03.1.B	Invite User	Users **must** be able to invite other users to organisations
F-03.2	Projects	Organisations **must** be able to contain different projects
F-03.2.A	Create Project	Users **must** be able to create project under the different organisations they are assigned to
F-03.3	Services	Projects **must** be able to contain different services
F-03.3.A	Create Service	Users **must** be able to create services under the different projects that are assigned to
F-04	Admin access	These **must** be way for admin users to access the system
F-04.1	Create new admin users	This admin access **must** have a way to add new admin users to the system.
F-04.2	Create service credentials	This admin access **must** have a way for admins to create new credentials that extension services need

Non-Functional Requirements

ID	Name	Description
NF-01	Extendibility	It **must** be easy to add new generation services to the system
NF-01.1	Service Standard	Generation services **must** follow a specification standard
NF-02	Security	The system **must** implement measures to ensure that only authorized users can the resources that they are approved for
NF-05	Diagnostics	The system **must** contain come method of self-Diagnostics
NF-06	Logging	The system **must** keep a log of actions performed by users and services

3.5 Microservices

From the requirement we can see a number of concerns that needs to be addressed. Different microservice will be made to handle each major concern. The first microservice we can see we will need is an **AuthenticationService**, to handle the authentication of requirements from outside the system. We will need a microservice to handle the data generated by end-users, this service will be called **UserService**. A core part of the system will be adding new microservices that can generate code, we need a microservice to hanlde the registration of these new services, this service will be the **RegisterService**. To keep logs of user actions and system event, we will need a **LoggingService**. Users will need to be notified on some user actions and system event, this will be handles by a **NotificationService**. Lastly we have a need for a microservice to reflect on the system and check its "health", this job will be performed be a **DiagnosticsService**.

To get an overview of the system and the different microservices, we will analyses each of the microservice and define a contract for each. This contract will contain a short description of what the service will do, a list of the responsibilities, and a list of the other microservices that it is expected to communicate with.

AuthenticationService

We have a need for some parts of the system that is protected behind some Security, where only authorised user has access. For this purpose, we will need a microservice that the other microservices in the system can call to handle the authorisation.

MICROSERVICE CONTRACT		
Name	Service ID	Service Manager
AuthenticationService	S-0	Marc Bertelsen
Description This service will handle all system authentication, by generating and validating tokens, based on credentials.		
Responsibilities • Generate tokens • Validate tokens		
Connections • LoggingService (ID: S-3)		

UserService

There is a need for a microservice to store data provided by users. The data stored by this microservice will have different user roles associated with it. These roles must also be enforced by this service to limit the access of some resources.

MICROSERVICE CONTRACT		
Name	Service ID	Service Manager
UserService	S-1	Marc Bertelsen
Description This service will handle the storage of all data generated by users.		
Responsibilities • Provide CRUD operations for user created data • Enforce user roles		
Connections • AuthenticationService (ID: S-0) • LoggingService (ID: S-3)		

RegisterService

The system will need somewhere to have a register of the **GenerationServices** that have been added to the system. This should be its own microservice, as it could see a different level of use to the **UserService**, and so might be scaled at a different rate than it.

MICROSERVICE CONTRACT		
Name	Service ID	Service Manager
RegisterService	S-2	Marc Bertelsen
Description This service will handle the registrations of the extension **GenerationServices**.		
Responsibilities • Register **GenerationServices** • Provide registrations		
Connections • AuthenticationService (ID: S-0) • LoggingService (ID: S-3)		

LoggingService

To keep logs in the system in a single place, we would need a **LoggingService**, where the other microservice send their log messages to.

MICROSERVICE CONTRACT		
Name	Service ID	Service Manager
LoggingService	S-3	Marc Bertelsen
Description This service will handle all logging messages from the other services in the system.		
Responsibilities • Write log messages for other microservices • Read log messages for admin users		
Connections NONE		

NotificationService

Notifies users with messages from the system. Like getting an invite to an organisation by another user.

MICROSERVICE CONTRACT		
Name	Service ID	Service Manager
NotificationService	S-4	Marc Bertelsen
Description This service will handle notifications to users, based on triggers in the system.		
Responsibilities • Send notifications to users • Listen for notification triggers		
Connections NONE		

DiagnosticsService

Performs diagnostics on the system and find where a problem is.

This service should check the state of each of the other service as well as the response time. During this should give an indication to where a possible system failure is. Either if the microservice did not respond or the response time was exceedingly long.

MICROSERVICE CONTRACT		
Name	Service ID	Service Manager
DiagnosticsService	S-5	Marc Bertelsen

Description

This service will perform a system wide diagnostic, to verify that the system is fully operational.

Responsibilities

- Perform system diagnostics
- Interrogate every microservice in the system

Connections

- AuthenticationService (ID: S-0)
- UserService (ID: S-1)
- RegisterService (ID: S-2)
- LoggingService (ID: S-3)
- NotificationService (ID: S-4)

GenerationServices

This tool can potentially have a rather large scope, if we assume that users would require the tool to provide code generation in a large array of languages with each a different option for databases and APIs. For this reason, the system must be designed with a modular approach in mind, in such a way that multiple service can be added to handle the code generation. This way the rest of the system can remain the same while new generation services are added. In order to achieve this modularity, a well-defined interface for the system must be devised. The design of the interface should also keep the possibility for a better future interface in mind, such that a new interface affects as little as possible of the system.

3.6 Frontend

The system must have two separate frontends for users to interact with, the first will be for normal users to design and generate their own microservices, the second will only be accessible by admin users to perform actions that only they can perform.

Website

The primary frontend will be a website. This will be where users can design new microservices from and send requests to the generate these new microservices. The design page on the site, must be able to work with different **GenerationService** implementation.

AdminAccess

There need to be somewhere to get new tokens for the generations microservices. These new tokens should only be generated by admins. So, to make sure that admins can make these new tokens we will need a frontend that only admins can access. This frontend should also be able to notify all users. Additionally, this frontend should also be the only place to view the system logs.

4 Design

4.1 Architecture

Based on the service contracts from the chapter 3, we can get an idea of how the system will look. Some of the microservices in the system will only be accessible to the other microservices in the system, while some will be accessible from the frontend. The **AuthenticationService**, **LoggingService**, and **NotificationService** will only be accessible to the other microservices. This means that for both the **LoggingService** and **NotificationService**, can we assume that access to them is authenticated by the caller, for the **AuthenticationService**, it will be self-evident that access must be authenticated. The microservice that will need to be accessible to the frontend are the **UserService**, **RegisterService**, **DiagnosticsService**, and to some extend also the **GenerationServices**. For these last microservices we will assume that calls are untrusted, as they can come from the frontend/end-user. An early design of the system can be seen on figure 4.1.

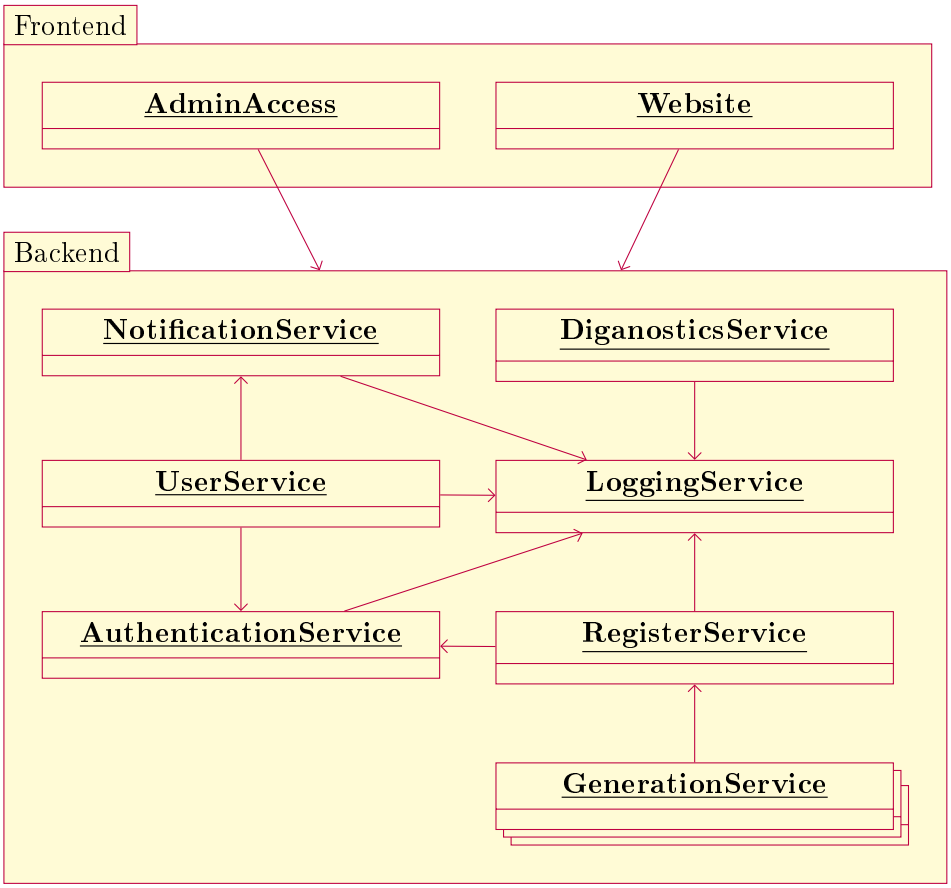

Figure 4.1: System diagram

Now that the overall system structure has been settled, we can design the internals of the individual microservices and the two frontends.

4.2 AuthenticationService

We need a solid authentication method for the system. It has to be flexible enough to be used for users and the **GenerationServices** that will extend the system, as both will have to prove their identity when accessing the system.

Firstly, we need to define how we will prove the identity of the entity trying to access the system. The idea here is to use tokens that are generated with some specific data related to the entity. This specific data could be a UUID as it will be easy make unique. The tokens should also expire after some time, this data will also be stored in the tokens. So users cant read the content of a token, in case a token gets intercepted, we will need to scramble the data contained

in the token. We will also need to read the data again when performing validation, so we need de use encryption and not hashing, to keep the data secret.

It might be necessary to change the encryption algorithm in the future, so we want to keep it behind an interface. This interface will have two methods: *encrypt* and *decrypt*. We can see the **Token** class and it relation with the **ICrypto** interface on figure 4.2.

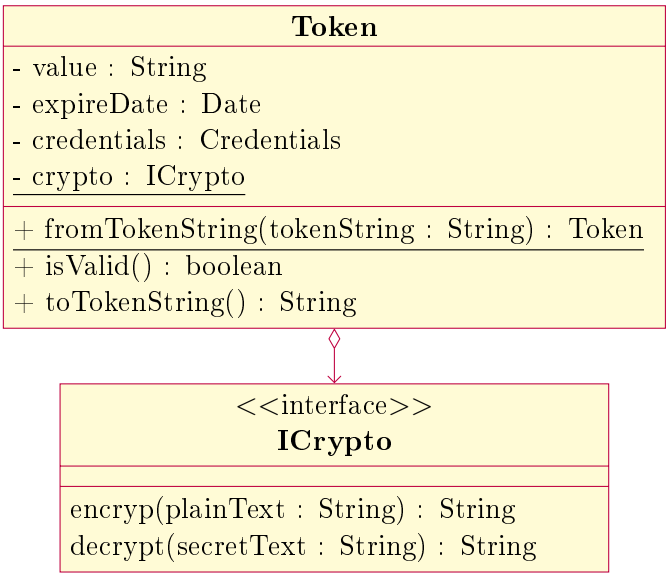

Figure 4.2: **Token** and **ICrypto** class diagram

Entities should also have different levels of access to the system, this access level should also be stored in the token data. There will need to be users with admin access to the system, they could potentially be paying users in the future, they will have *pro* access, but this will not be the focus. The **GenerationServices** should have their own access level, and only users with admin access will be able to generate tokens for these. Lastly there will be the normal users, and since the users access level will be stored with the user in the **UserService**, we do not want to expose users access level to other users, for this reason we will need a *hidden* access level. In total we end with the five access levels:

- ADMIN
- PRO
- SERVICE
- DEFAULT

- HIDDEN

We can see the design of **Credentials** on figure 4.3, this will be used to generate and validate tokens.

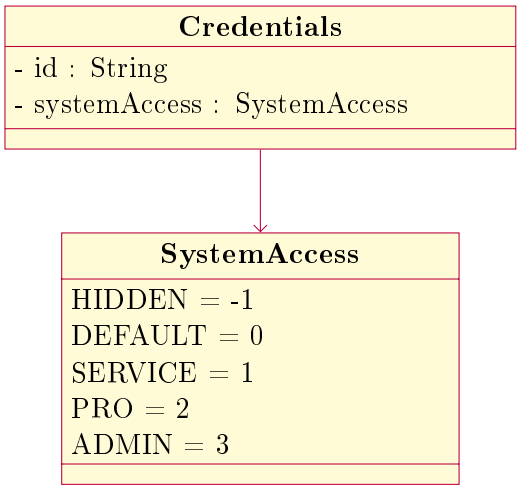

Figure 4.3: **Credentials** and **SystemAccess** class diagram

Now need to design the API for this service, it should feature a method for creating new tokens and validate them, it should also have a separate method for creating tokens for **GenerationServices**, that checks if the caller has the necessary access level. Lastly it should have a method for performing a 'health' check of the service, this is part of the process that the **DiagnosticsService** will perform, see section 4.7. The service interface can be seen on figure 4.4.

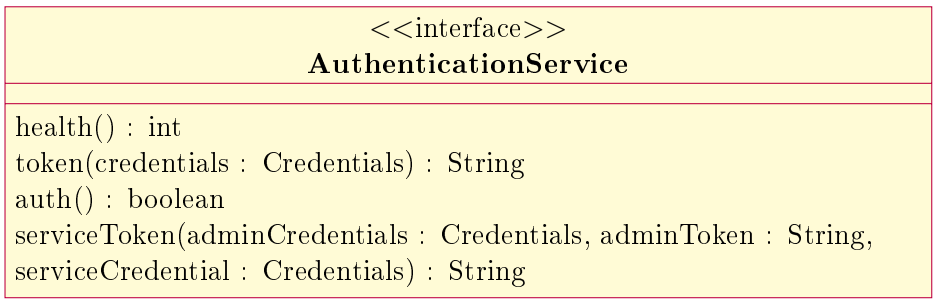

Figure 4.4: **AuthenticationService** interface

We keep the logic separate from the API, and so the service will have a **Authenticator** class, that will perform the actions of making tokens and validating them. Since this service will be responsible for generating new token,

it should be possible to set the how long the tokens valid duration is. Here we will have a **Policy** class, where the policy will determine the duration. The different available policies and their associated durations will be the following:

- DEFAULT · 1 Day
- SHORT · 12 Hours
- MEDUIM · 3 Days
- LONG · 1 Week
- VERY_LONG · 1 Month

To map the different policies to a duration, we also want a class to handle how long the durations are. To handle this we can use a **Duration** class in association with a **DurationUnit** class. The durations in the **Duration** class will each hold a value for how long their duration is in milliseconds. So the *SHORT* duration will have $12 \times HOUR$, which will translate to $12 \times 36E5$. For serviceToken, that does not expire, we will have a *MAX* duration option, which will be the highest possible value. On figure 4.5, we can see how the design of the **Authenticator**. The way it is designed now we can easily change the policy and, if needed, it is also possible to add or change durations easily.

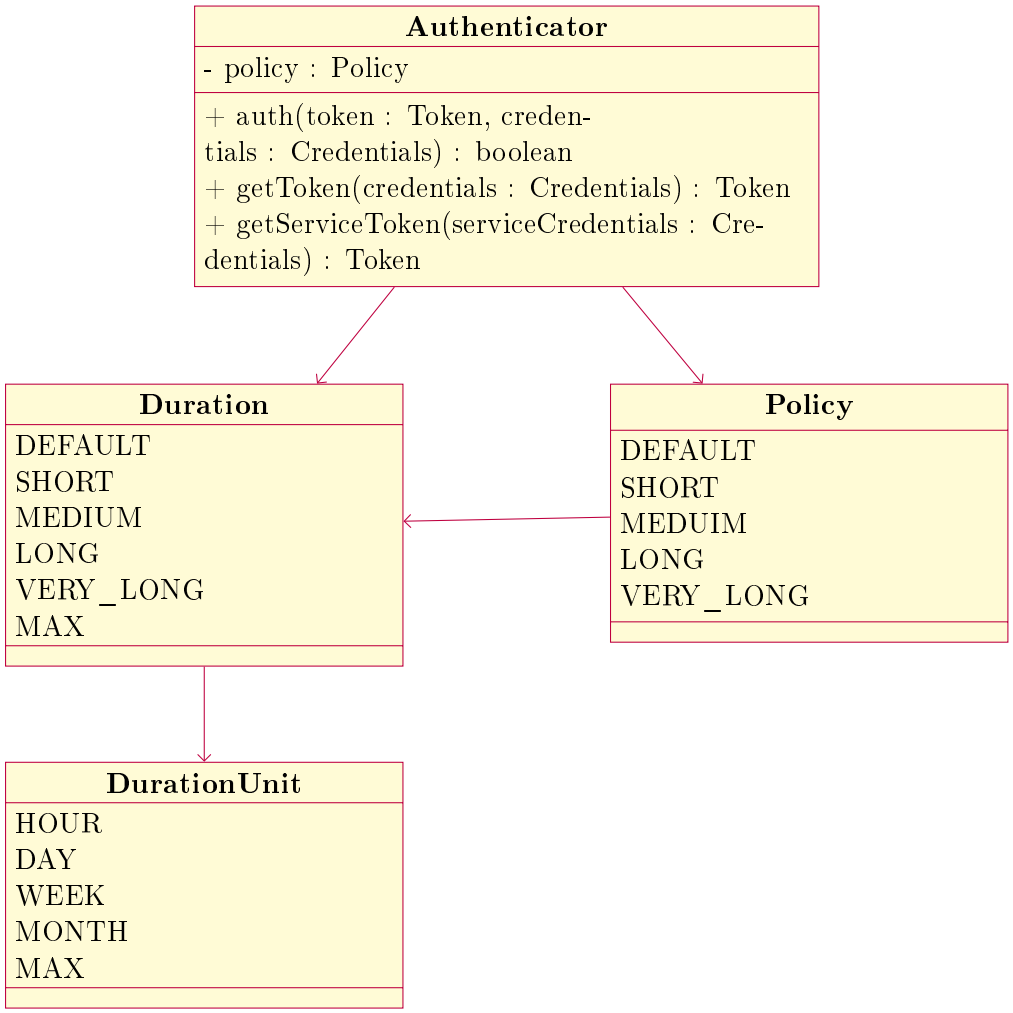

Figure 4.5: **Authenticator** class diagram

We now have the complete design for this service. Figure 4.6 show the entire class diagram for the **AuthenticationService**.

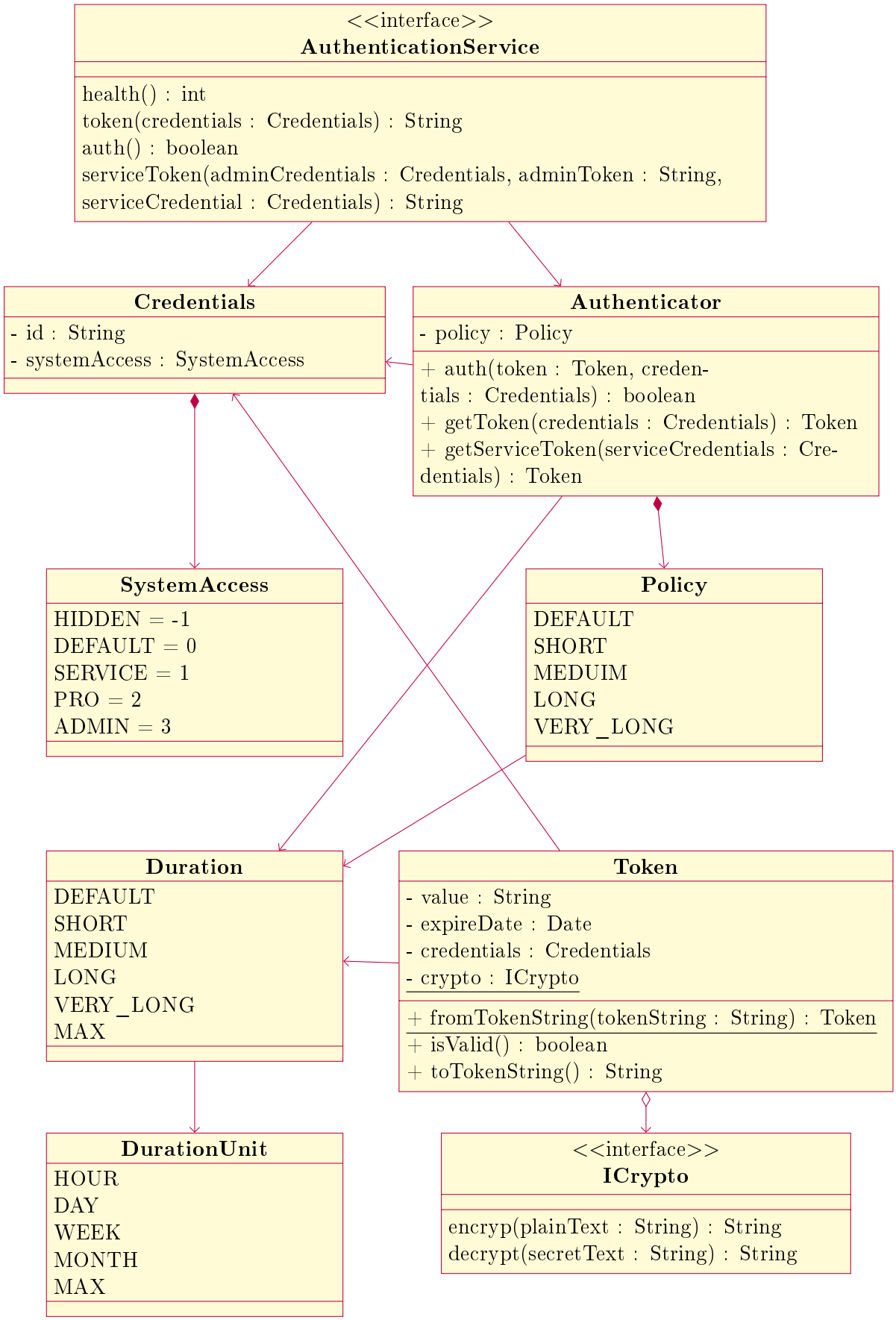

Figure 4.6: **AuthenticationService** class diagram

4.3 UserService

This service's only responsibility is to store and expose data created by users of the system. In order to design this service, we first need to know what data that needs to be stored and accessed. So to do this an **ER** diagram is created, it shows the different data entities, what data they contain and their relationships. The relations of the data we will need can be seen on figure 4.7.

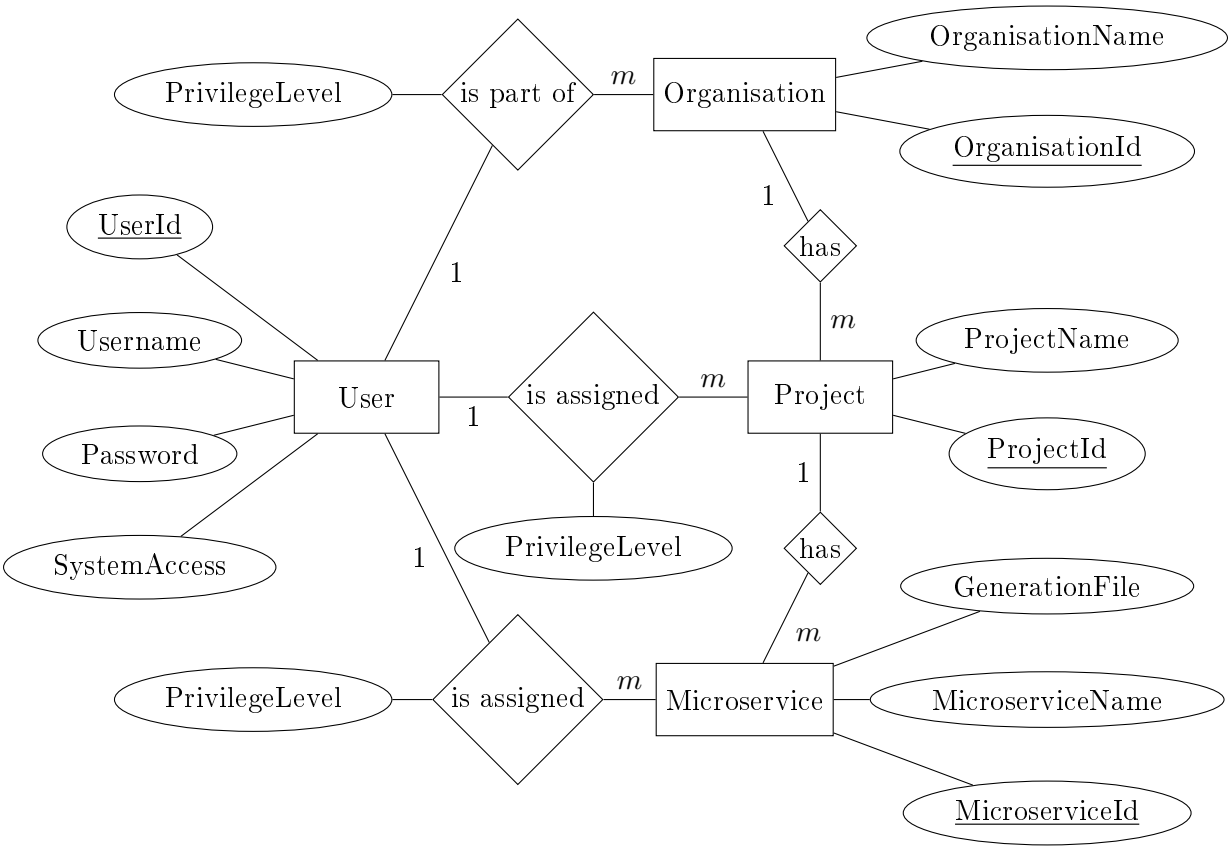

Figure 4.7: User data ER diagram

Because this service works by mainly serving resources on demand, it should either use a *REST* or *GraphQL* API, however the requests will be very simple, so a *REST* API makes the most sense in this case.

Now we can design the service. Most of the resources on this service has to be protected, this will be done with the **AuthenticationService**, see section 4.2 and by having permission in the data.

Permissions

We need to further protect resources from users who are logged in. This will be done with permissions associated with the resource and user in question. These permission can be seen in table 4.1. In order to simplify the permission, we can assume that every permission can perform the actions of every permission above it on the table, i.e., every permission with a lower id.

Permission	ID	Actions
NO_ACCESS	-1	None
VIEW	0	Can find a resource
EDIT	1	Can save file changes
MANAGER	2	Can invite and remove people
ADMIN	3	Can update properties
OWNER	4	Cannot lose permission

Table 4.1: Resource permissions

Endpoints

There will be two main groups of endpoints, the unprotected and the protected. Under the unprotected there needs to be an endpoint that can be called when a new user is being created, additionally there needs to be an endpoint for user login. The protected group will contain the remaining endpoints. All of the identified data entities will have endpoints for performing *CRUD* operations. Since some of the resources also require the user to have the necessary permission to access them, we also need to have endpoints for doing *CRUD* operations on these permissions.

To keep consistency to responses for this API, we need to make a standardised response. This response has to be informative about errors. Normal HTTP status codes are a good place to start, but the responses have to be more explicit about the error, and so it will contain a message as well as the status code.

Since users need the ability to work together, they need to have some way of adding other users to the organisations they have, and since we also want users to decide whether or not to accept this, we need to have an invitations that can be either accepted or declined. All of these endpoints are protected.

A table of all the endpoints can be seen in appendix B.

4.4 RegisterService

When adding new **GenerationServices** to the system, they need somewhere to register them self, this is where the **RegisterService** comes in. It will be a relatively simple service, as it will only deal with a single resource. Because of this need to store and expose a single resource we can use a *REST* API for this service.

Registrations

Figuring out what these registrations will look like will play a part in designing the interface for the **GenerationServices**(see section 4.9). The data needed from the **GenerationServices** when registering are an address we can call when we want to send a generation request to a **GenerationService**. We will also need to store the *manifest*, which is *JSON*. Lastly, we will need to store the id for the service, so we can access the registration based on a unique identifier. Seeing that the data we will need to store contains *JSON* and there is no relation between any of the data, we can design the service to use a *document* based database. This also allows the ability to add additional fields to the data, if needed, as it is schemeless.

Endpoints

We now know that this service will have a *REST* API, so we now need to know which endpoint it should have. Table 4.2 shows all the endpoints we will need for the **RegisterService**.

Method	Path	Description
GET	/health	Used by the **DiagnosticsService**, when performing a system check
POST	/register	Register a new **GenerationService**
GET	/register	Get all the registered **GenerationServices**
DELETE	/register/:serviceId	Unregister the **GenerationService** with the provided serviceId

Table 4.2: **RegisterService** endpoints

4.5 LoggingService

The **LoggingService** will have method for logging different level of log message. These levels follow a common convention of four levels: *debug*, *info*, *warn*, and *error*. The logs will be split into a collection for each microservice.

The service will additionally have a *health* method to comply with the need from the **DiagnosticsService**(see section 4.7), as well as a method for getting logs. The last method for getting logs will require the caller to specify which microservice to get the logs from. The interface for the **LoggingService** can be seen on figure 4.8.

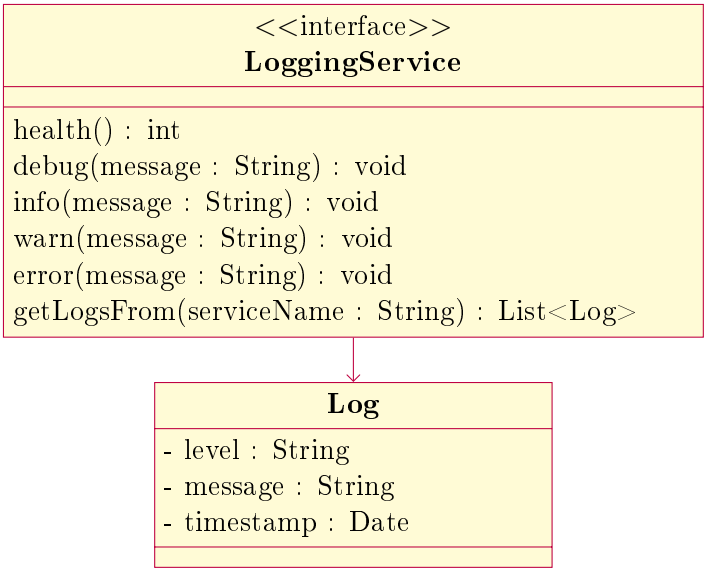

Figure 4.8: **LoggingService** class diagram

4.6 NotificationService

For this microservice we will need to think about it from an event-driven point of view, as it will send notification based on triggers in the system. To accomplish this, we will have endpoints that we can subscribe to and endpoints that we can call when an event has happened. Like all other service in the system we also need to have an */health* endpoint to comply with the **DiagnosticsService**(see section 4.7). On table 4.3 we can see an overview of the endpoints we will need for this service.

Method	Path	Description
GET	/health	Used be the **DiagnosticsService** when performing a system check.
POST	/subscribe/:id	Subscribe to the service and listen for events.
POST	/trigger	Send and event object and trigger notifications

Table 4.3: **NotificationService** endpoints

In this service we can have a single trigger endpoint where we in the request can specify the event. For now, there will be three types of events we will want to handle. The first is when a user gets invited to join an organisation by another user, here we want to update the GUI, so the invited user can respond to the invite without having to refresh the page. We can call this first type *invite*. Second event type is when a change happens in the system, like when a new **GenerationService** is added. This second type will display a message to all the subscribers, so we can call this type *global*. The last type will be for controls of the **NotificationService** is self, for when we want to clear the *global* message. We can call this last type for *other*, to encapsulate everything else.

We end up with these three event types:

- invite
- global
- other

Looking at what the entire **Event** request should contain, we know that it will inform about its type, for the *global* events we will want the request to contain a message that can be displayed. Lastly for the *invite* events, we need to have some data about the invite, we just have a *data* field that can hold an object of any type, to accommodate possible future event types. We can see the final design of the **Event** and **EventType** on figure 4.9.

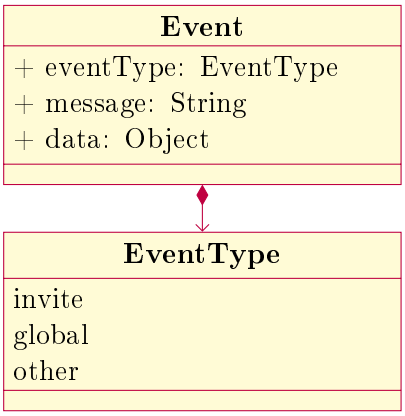

Figure 4.9: **Event** and **EventType** class diagram

4.7 DiagnosticsService

The **DiagnosticsService** will feature a simple system check, to ensure that all other service are running and how fast their response time is.

This service will only be usable for users with admin access, to ensure this restriction, it will need valid admin credentials. These credentials will be checked by the **AuthenticationService**(section 4.2). If these credentials are valid, it will then run a simple check and generate a report for the system.

The report will consist of a list of *vitals*. The *vitals* contain the name of a microservice, if it responded, and how long the response time was. The report will also include a total time the diagnostics check took.

To check the connection of the other service, we will need to make a **ConnectionChecker** class, that will contain the logic for generating the report. Looking at listing 4.10 we can see the full class diagram for this simple service.

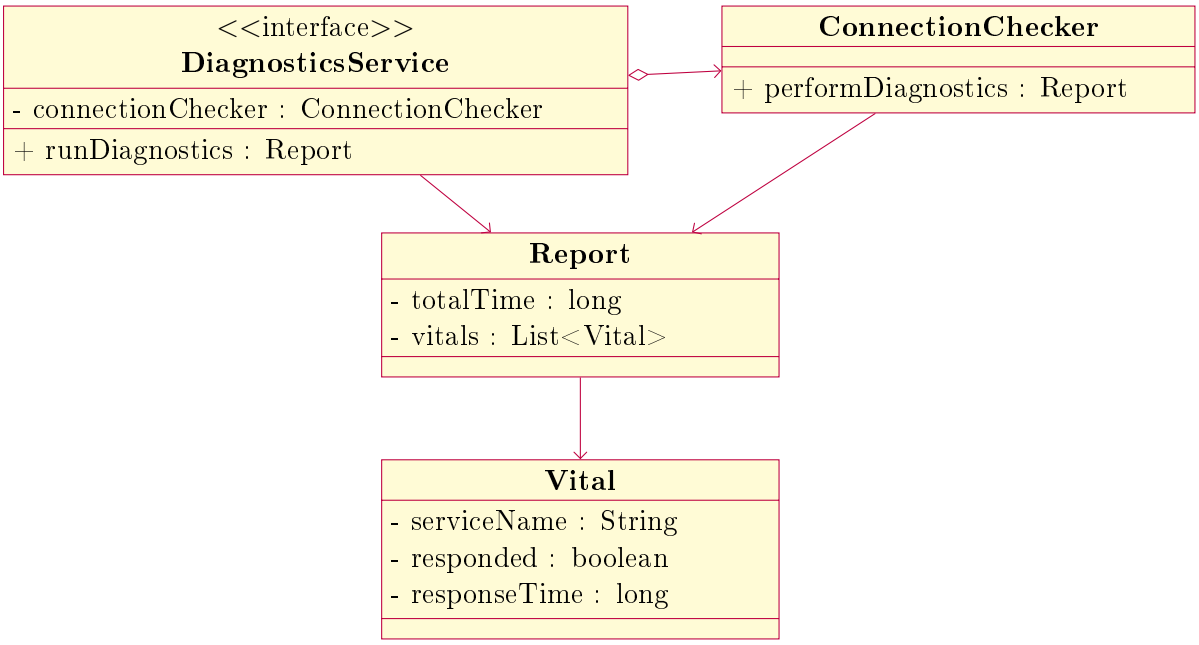

Figure 4.10: **DiagnosticsService** class diagram

To ensure that the **DiagnosticsService** can perform its job, we will need to keep in mind that all other services in the system must have an accessible way to check its "health". This problem will be solved by having a single endpoint/method we will call *health*. When this service is running a diagnostic of the system, it will check each of the other services *health* and generate the report.

4.8 GenerationServices

The core system itself will be rather useless as it will only store some data. The extensions of the system will provide the biggest part of the system, and for this reason, it need a good interface to base the extensions of.

4.9 Extension Interface

The interface will include how the extension must be deployed to the system and how it must interact with the existing parts of the system.

The system will also be open to possible new versions of the interface, but for now version 1 is the only one.

Interface V.1

Any GenerationService must fully implement the specified interface. The interface will try to be as accommodating for the greatest number of possible implementations. In order to do so we will make the individual **GenerationServices** specify its implementation details by providing a *manifest*. The full specification of **Interface V.1** can be seen on table 4.4.

Interface V.1

Requirement:

It must register it self with the **RegisterService**.

- It must register an address that can be called to generate code
- It must register a *manifest* object
- The registration must contain a UUID and in the request there must be a valid token

If possible, it should unregister itself when shutting down.
It must accept and parse generation request.
It must generate code from the requests.
After generating the code, it must return a link where the code can be downloaded from.
It must deliver the code as a *zip* file.

Details:

On register: HTTP POST <url>/register

```
body: {
    serviceToken: string,
    registration: {
        serviceName: string,
        serviceId: string,
        serviceAddress: string,
        manifest: object
    }
}
```

On unregister: HTTP DELETE <url>/register/:serviceId

Table 4.4: Interface V.1 specification

manifest.json

The provided *manifest.json* must have structure as seen on listing 4.1.

```
{
    "sirup_v": 1,
    "acceptedFormats": [],
    "templates": [],
    "languages": [],
    "apiTypes": [],
    "databases": []
}
```

Listing 4.1: manifest.json structure

The manifest specifies the version of the interface that is implemented under *sirup_v*. The GenerationServices can be implemented to accept the generation file in different formats, this is specified under *acceptedFormats*, an example can be seen on listing 4.2.

```
"acceptedFormats": ["json", "yaml"]
```

Listing 4.2: Accepted formats by implementation

GenerationServices can also provide templates for microservices that are frequently generated, like a *REST* API for performing CRUD operations on some data. These templates are specified under *templates*. An example can be seen on listing 4.3.

```
"templates": [
    {
        "name": "crud",
        "description": "Generates a microservice with a
   REST API for doing CRUD operations on the provided
   data collections"
    }
]
```

Listing 4.3: Provided templates

The three main section of the *manifest.json* is the *languages*, *apiTypes*, and *databases*. All three of these contain an array of objects. All these objects then contain a field with the name of this choice, a field with the description, as well as an *options* object. Listings 4.4, 4.5, and 4.6 each show examples for each of the three sections.

```
"languages": [
    {
        "language": "Java",
```

```
        "description": "Generates the microservice with Java 17",
        "options": []
    }
]
```

Listing 4.4: Provided languages

```
"apiTypes": [
    {
        "apiType": "REST",
        "description": "Generates a REST API using Apache Spark and controllers for endpoints",
        "options": []
    }
]
```

Listing 4.5: Provided API types

```
"databases": [
    {
        "database": "PostgreSQL",
        "description" : "Generates a PostgreSQL connections and classes capable of CRUD operations",
        "options": []
    }
]
```

Listing 4.6: Provided databases

Each of the three configuration options, contain an *options* array, that can be used to add additional configuration for the specific option. There are four different ways to add configuration in this *options* array.

The first and most simple addition is by *value*. On listing 4.7, we can see that the option for adding a port which will resolve to a number value. Currently only *string* and *number* is supported.

```
"options": [
    "port": {
        "value": "number"
    }
]
```

Listing 4.7: Value option

The second is for options with a predefined selection of values, this is done by *selection*. This can contain as many values as needed. Listing 4.8 shows a selection of build tool to choose from.

```
"options": [
    "buildTool": {
        "selection": ["maven", "gradle"]
    }
]
```

Listing 4.8: Selection option

The third is for more complex options, this is done by *object*. The *object* object must contain a *name* field with the name of the *object* and a *values* field with an array of values that the *object* contains. These values must also contain a *name* field with the name of that value and any of the option types (*value*,*sleection*,*object*,*repeated*). On listing 4.9 show how an *object* option might look like for an endpoint option.

```
"options": [
    "endpoint": {
        "object": {
            "name: "endpoint",
            "values": [
                {
                    "name": "method",
                    "selection": [
                        "GET",
                        "POST",
                        "PUT",
                        "DELETE"
                    ]
                },
                {
                    "name": "path",
                    "value": "string"
                }
            ]
        }
    }
]
```

Listing 4.9: Object option

The fourth and last option is for arrays of values, this is done by *repeated*. A *repeated* object can contain any option type (*value*,*sleection*,*object*,*repeated*). Listing 4.10 show an example of how *repeated* can be used to make groups of endpoints. This example also show how all the four option types can be used in combination.

```
"options": [
```

```
    "endpointGroups": {
        "repeated": {
            "object": {
                "name": "endpointGroup",
                "values" [
                    {
                        "name": "groupPath",
                        "value": "string"
                    },
                    {
                        "name": "endpoints",
                        "repeated": {
                            "object": {
                                "name": "endpoint",
                                "values": [
                                    {
                                        "name": "method",
                                        "selection": [
                                            "GET",
                                            "POST",
                                            "PUT",
                                            "DELETE"
                                        ]
                                    },
                                    {
                                        "name": "path",
                                        "value": "string"
                                    }
                                ]
                            }
                        }
                    }
                ]
            }
        }
    }
]
```

Listing 4.10: Repeated option

JavaGenerationService

To keep the extend of the system to a size where the core concept can be demonstrated, there will only be a single **GenerationService**, this service is going to generate *Java* code.

Since this service will generate code, we need to think about how this will be done, we need to specify that come parts of the code will be responsible for creating new code, to do this we can define an interface that we will call **Generatable**. We need to know the name of the generated class, a way to set its group id, it needs to know its own package, other **Generateable** classes need to import the new code, and for creating the actual file we need the directory and a method to insert the actual code in the file. The final interface can be seen on figure 4.11.

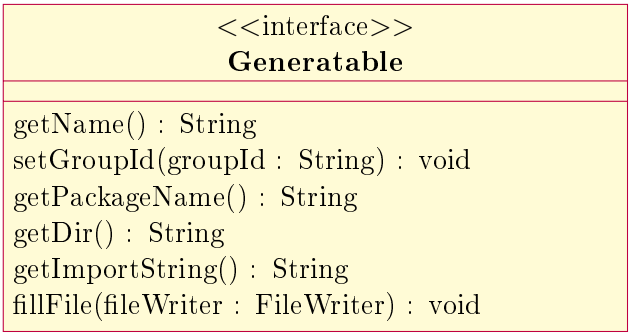

Figure 4.11: **Generatable** interface class diagram

This service will also generate the build tool for the generated code and for this reason we also need and interface for the parts of the generated code that will require dependencies so that can be added to the build tool configuration. This will be done with the interface **Dependency**. It will only require a single method to get the name of the required dependency, see figure 4.12. The build tool generation implementation will use this name to index in a *Map* to get the full dependency, this way we only need to update the build tool implementation when new dependencies are needed, and not the other way where the APIs and databases need to know about the different build tools.

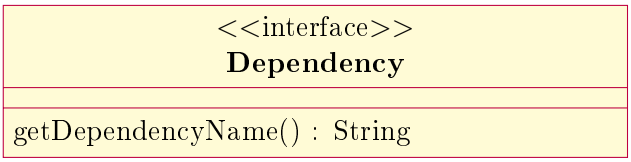

Figure 4.12: **Dependency** interface class diagram

Because this service must generate code that will feature different APIs and databases, we will need interfaces for both of these. For database we will make the interface **IDatabase**, see figure 4.13. This interface will need a method for getting the *services*, these are the classes that will interace with the database, a method for getting the data models, the actual data it will store, additionally we need a way to get the file that will initialise the database, such as a *SQL* file.

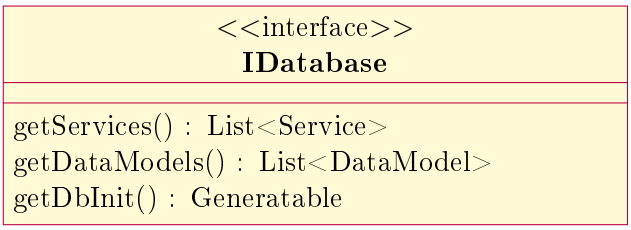

Figure 4.13: **IDatabase** interface class diagram

For the APIs we will make the interface **IApi**, see figure 4.14. For this interface we need a method for getting the controllers for the API, these will handle requests, we also need a method for getting the port that the API will be using, additionally the API will need to have a context of the generated system, this context will contain information about the generated database.

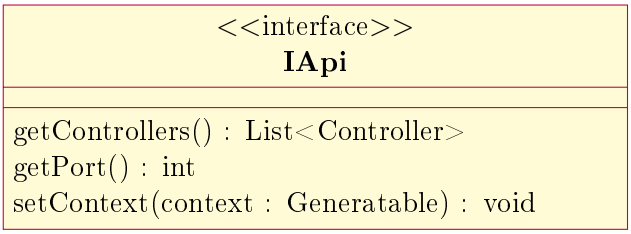

Figure 4.14: **IApi** interface class diagram

When building the microservice we need to make sure that all the dependencies are added to the build tool, to do this we make the interface **IBuild-**

Tool, see figure 4.15. We only need a single method that takes some implementations of the interface **Dependency**.

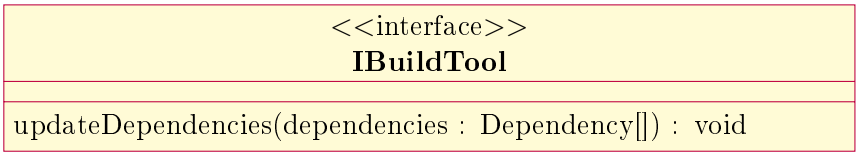

Figure 4.15: **IBuildTool** interface class diagram

Since this service will build microservices, we will use the *builder* pattern [3][p. 97] extensively, and for this reason we will make the interface called **Builder**, this interface need to be generic so it can be used for multiple different builder patterns. It will only feature a single method called *build*, see figure 4.16.

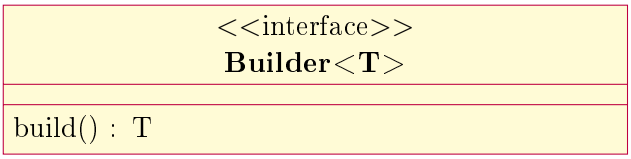

Figure 4.16: **Builder** interface class diagram

We want the the API and database-builders to handle the options provided. To do this we can use the *Strategy* pattern [3][p. 315], by making an interface we will call **OptionsStrategy**. It will have a method called *Options*, this method will act like builder method, so it should return the builder it will be part of. Additionally, we also want the specify which *options* object we want the method to handle. To handle both these concerns, we will need to make the interface generic. Looking at figure 4.17, we can see how the **OptionsStrategy** is finalised.

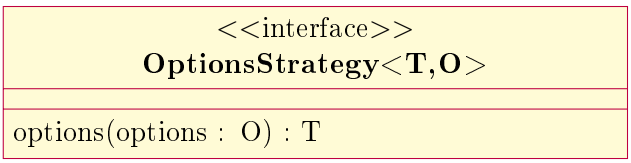

Figure 4.17: **OptionsStrategy** class diagram

From the **Builder** interface we can then create the builder interfaces for API and database. For the IApiBuilder, see figure 4.18, we want to keep it

generic, and to do so we add the entire *Options* object from the request and let the implementation handle how and what to do with it, so we will use the **OptionsStrategy**.

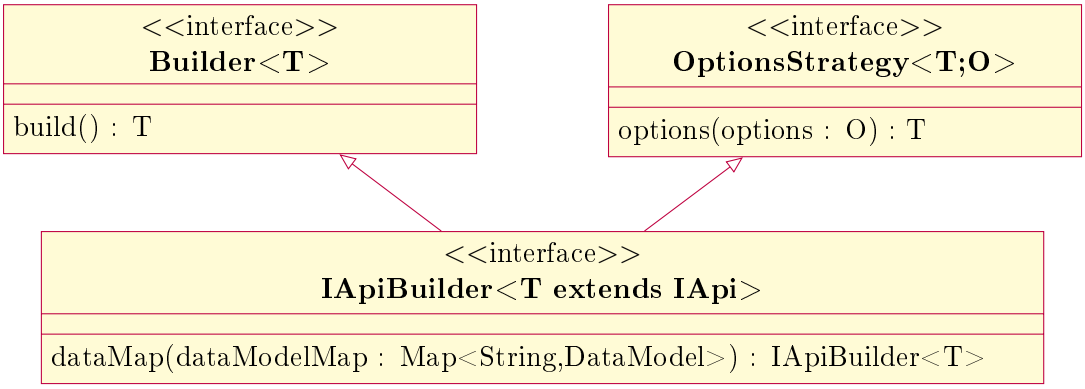

Figure 4.18: **IApiBuilder** interface class diagram

For the **IDatabaseBuilder**, see figure 4.19 we also will also send the entire *Options* object, and let the implementation handle it, but we also need to add any number of data models that the database will store.

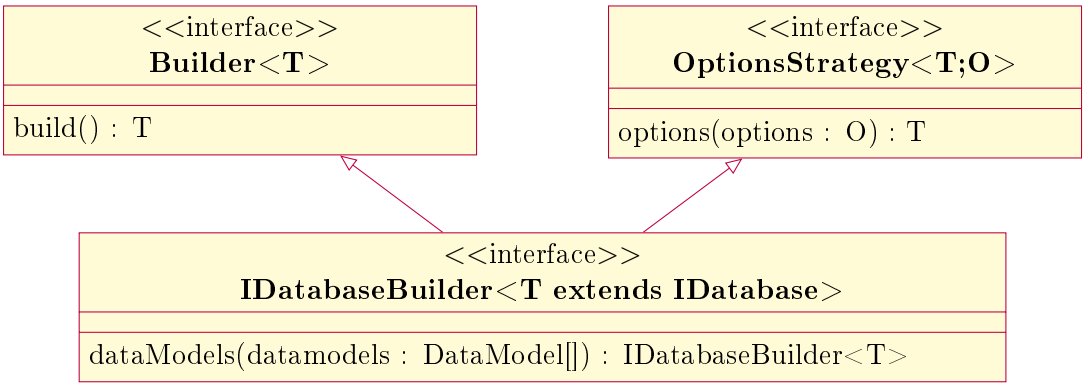

Figure 4.19: **Builder** interfaces class diagram

There is also a need for specifying parts of the microservice that will need to be containerized, i.e. have a *Dockerfile*. This includes the database and the and the API/service itself, but could expand to include more. In addition to generate the *Dockerfiles*, it will also want to generate a *Docker-Compose*, because of this we need to declare an interface that will allow us to get the necessary information needed to generate the *Docker-Compose*, this interface will be called **DockerService**. These **DockerServices** will need to have a

name, which internal and external port they will use, and where to build the service from. For the purpose of specifying which classes will need to be *containerized*, we will define a second interface that these classes will implement. It will be called **Contrainerizable**, and will have two methods. The first will be used to get a **Generatable** that will be used to generate the *Dockerfile* for that part of the service. The second method will return an implementation of the **DockerService** interface. These two interfaces can be seen on figure 4.20.

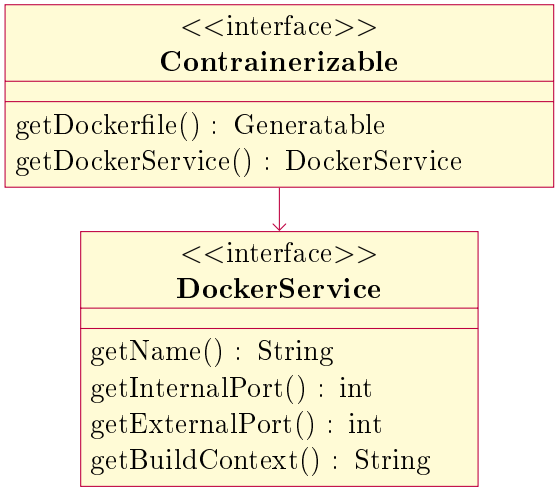

Figure 4.20: **Contrainerizable** and **DockerService** interfaces class diagram

4.10 Website

The website will be where the users can organise, create, and collaborate. So, we need to split the three different parts into different pages to give the users a structured overview. Additionally, we need a login and signup page. Perhaps the most important page on the website, will be the actual microservice design page. On this page we want users to have an easy view of what options they have available to choose from. The includes the different languages, APIs, and database, that the system can generate. So, we can add a section for each of these three, with a configuration section for each under. We can add a view of the current generation file, so users can verify the file, while working on it. This page must also have a button that will call a **GenerationService** to generate the microservice. After the code has been generated, we want a button to download it, this button should either be disabled or not show, before the code is ready to be downloaded. The style of the website will try to be minimal, so users the main focus can remain on creating new microservice. Simple design diagrams of all the different pages needed for the website can be seen in appendix C.

4.11 AdminAccess

This frontend will be designed as a CLI tool. Users will have to login when accessing this frontend, only users with admin level system access will be allowed to login. When users are logged in, they will have the following commands available:

- service_token · Generate a new service token and serviceId
- new_admin · Add a new admin user to the system
- logger · Get a list of all the logged services
- logs_from · Get a list of the logs for a specified service
- diagnostics · Run a full system diagnostic
- set_message · Set notification message
- remove_message · Clear the current notification message

The *service_token* command, will call the **AuthenticationService**(section 4.2) to get the token and id. Only admins are allowed to generate service tokens, this will be enforced by the **AuthenticationService**.

Using the *new_admin* command will prompt the user to enter a new username, password and confirm the password by entering it again. It will then call the **UserService**(section 4.3) and create the new user with admin level access.

It will be possible to view logs from the different services in the system, by using the *logs_from* command. The user will have to specify which service to get the logs from. To see which services that are logged, the user can use the *logger* command. When displaying the logs, they should not all be displayed at once, but in steps. The users should also be able to specify the step size. Both the *logger* and *logs_from* commands will be calling the **LoggingService**(section 4.5).

The command *diagnostics* will call the **DiagnosticsService**(section 4.7), to perform a full system diagnostic. From this it will display the full report of the diagnostics call.

The *set_message* and *remove_message* will both call the **NotificationService**(see section 4.6). The former will set a new message that will be displayed to all users. The later will clear any message.

5 Implementation

Even though microservice architecture lends itself to having services written in different languages, it has been decided that most of the services in the system will be written in *Java* as *Maven* projects. The user of APIs and database will differ from service to service, based on what best fits each service, but all *RPC* API will user the *gRPC* framework, *REST* APIs will use *Apache Spark*. The reason for these choices is to limit the time spend on learning new technologies and focus on implementation.

5.1 AuthenticationService

This service will work internally in the system, and will not have a public API, for this reason and since it will not store and expose resources, it will use a *RPC* API. Based on the interface that was designed for this service, we can create a *proto* file that will be used by *gRPC* to generate the classes needed for the API. Part of this file can be seen on listing 5.1.

```
service SirupAuthService {
    rpc Health(HealthRequest) returns (HealthResponse) {}
    rpc Token(TokenRequest) returns (TokenResponse) {}
    rpc Auth(AuthRequest) returns (AuthResponse) {}
    rpc ServiceToken(ServiceTokenRequest) returns (
    ServiceTokenResponse) {}
}
```

Listing 5.1: **AuthenticationService** service definition

Service Tokens

The *serviceToken* method, will need to be protected, such that only users with admin access can get these tokens. This means that we first need to verify that the request contains valid admin credentials. A simplified version of this method can be seen on listing 5.2.

```
@Override
public void serviceToken(ServiceTokenRequest request,
    StreamObserver<ServiceTokenResponse> responseObserver) {
    // ...
    boolean isValid = false;
    try { // Auth admin
        Optional<Token> optionalToken = Token.fromTokenString(
    adminTokenString);
        Credentials adminCredentials = new Credentials(adminId,
    systemAccess);
```

```
9           isValid = systemAccess == SystemAccess.ADMIN.id &&
10                    optionalToken.isPresent() &&
11                    auth.auth(optionalToken.get(), adminCredentials);
12      } catch (IllegalArgumentException iae) {
13          iae.printStackTrace();
14      }
15      if (!isValid) { // Not admin
16          serviceResponseBuilder.setError(ErrorRpc.newBuilder()
17                  .setStatus(498)
18                  .setErrorMessage("User is not an admin"));
19          responseObserver.onNext(serviceResponseBuilder);
20          responseObserver.onCompleted();
21          return;
22      }
23      // Generate new ServiceToken
24      // ...
25  }
```

Listing 5.2: Simplified *serviceToken* method

AuthenticationClient

In addition to the **AuthenticationService** there is also implemented a client version of the service. This is done to simplify the user of this service when used by the other microservices in the system. Having this client implementation means that the other microservice only need to include a single *Maven* dependency to have access to the **AuthenticationService**(as long as they are written in Java or a language with Java interoperability).

This client in implemented as a *singleton* [3][p. 127] class that holds the connection to the services, and has wrapper methods for all the service's procedures. The client only needs to know the address and port before it is possible to use. To do this it features a method called *init*, where the address and port is set. This method should be called as early as possible in any microservice that will access the **AuthenticationService**.

5.2 UserService

We can use the permission we defined in the design of this microservice to enforce the access to resources in the database, by adding constraints to the statements we perform. The interaction with the database is done through a class to abstract the access away from the controllers. Both the data-entities and data-relationships have a class for interacting with the database, to handle one table each. The methods in these classes will throw a **ResourceNotFoundException**, if it is unable the find the requested resource.

We can look at how the permissions are enforced, by looking at the SQL statement shown on listing 5.3, where we update the information for an organ-

isation. On line 6 can we see that we check if the oP(organisationPermission) is greater than or equal to 3, which is the permission of admin, the lowest permission that can perform this action.

```
1 UPDATE organisations o SET organisationName = ?
2     WHERE EXISTS(
3         SELECT * FROM organisationpermissions oP
4         INNER JOIN users u ON u.userid = oP.userId
5         WHERE oP.organisationid = o.organisationid
6         AND oP.permissionid >= 3
7         AND u.userId = ?
8         AND o.organisationid = ?);
```

Listing 5.3: Update organisation SQL statement

Because we will work with getting data from *resultsets*, from querying the database, we can simplify this by making a static method in the different data classes. These methods will need to handle the potential **SQLException**, and check that none of the data is missing. If we cannot make the requested resourece, we will need to throw an exception. Here we can make a new exception, that we will call **CouldNotMakeResourceException**. On listing 5.4, we can see an example of a method making a resource from a result-set.

```
1  public static Organisation fromResultSet(ResultSet resultSet) throws
       CouldNotMakeResourceException {
2      try {
3          String organisationId = resultSet.getString("organisationId"
       );
4          String organisationName = resultSet.getString("
       organisationName");
5          if (organisationId == null || organisationName == null) {
6              throw new CouldNotMakeResourceException("Could not make
       organisation from ResultSet");
7          }
8          return new Organisation(organisationId, organisationName);
9      } catch (SQLException e) {
10         throw new CouldNotMakeResourceException("Could not make
       organisation from ResultSet");
11     }
12 }
```

Listing 5.4: *fromResultSet* method from the **Organisation** class

Controllers

This API has many endpoints, to separate the responsibility of these different endpoints, we will use separate controller classes that will handle requests for endpoints for the same resource. So all the */user* endpoints will be handle by a single **UserController**, and so on. To keep the format of the responses the same, we can make a class to ensure that they all adhere to the same structure. Here we can create an abstract controller that all the other controller classes

inherit from. In this abstract class we can define a method that will take an object as input and format it to json. This method will be called *sendResponseAsJson*. This method will set the *Content-Type* header to *Application/json* and format the input object to json. In the responses we want to give good response messages. To ensure that there always wil be a message we can make a wrapper class for all responses called **ReturnObj**, this will have a message, status code and an object of any type. Now to ensure that this class will be used, we can make the *sendResponseAsJson* take a **ReturnObj** as input.

```
1 public Object find(Request request, Response response) {
2     try {
3         String organisationId = request.params("organisationId");
4         Organisation organisation = this.organisations.get(
      organisationId);
5         return this.sendResponseAsJson(response, new ReturnObj<>("
      Organisation found", organisation));
6     } catch (ResourceNotFoundException e) {
7         return this.returnDoesNotExist(response, e.getMessage());
8     }
9 }
```

Listing 5.5: *find* method from **OrganisationController**

On listing 5.5, we can see how the *sendResponseAsJson* and **ReturnObj** work together, the *organisations* object seen on line 4 interacts with the database. We can see that in the *catch*-block there is a *returnDoesNotExist* method, this is just a helper method that calls the *sendResponseAsJson*. Listing 5.6, show how the return methods are implemented, the last two were made as those responses are common.

```
1 protected String sendResponseAsJson(Response response, ReturnObj<?>
      returnObj) {
2     response.header("Content-Type","application/json");
3     response.status(returnObj.statusCode());
4     return this.gson.toJson(returnObj);
5 }
6 protected String returnDoesNotExist(Response response, String
      message) {
7     return this.sendResponseAsJson(response, new ReturnObj<>(Status.
      DOES_NOT_EXIST, message));
8 }
9 protected String returnBadRequest(Response response, String message)
       {
10     return this.sendResponseAsJson(response, new ReturnObj<>(Status.
      BAD_REQUEST, message));
11 }
```

Listing 5.6: *sendResponseAsJson*, *returnBadRequest*, and *returnDoesNotExist* methods from the **AbstractController** class

5.3 RegisterService

As part of the interface specified for the **GenerationServices**(see section 4.9), new registration must provide a valid token. This token can only be generated by admin users using the **AdminAccess**. We need to have an easy method of providing this token and its servidId partner. This is solved by having a *.env* file where these two values can be placed.

5.4 LoggingService

This service will we working internally in the system. Its primary function is to perform and action, so based on this it will be using a *RPC* API. Knowing this and having designed the interface for the service we can create a *proto* file. Part of this file can be seen on listing 5.7.

```
service SirupLogService {
    rpc Health(HealthRequest) returns (HealthResponse) {}
    rpc Debug(DebugRequest) returns (DebugResponse) {}
    rpc Info(InfoRequest) returns (InfoResponse) {}
    rpc Warn(WarnRequest) returns (WarnResponse) {}
    rpc Error(ErrorRequest) returns (ErrorResponse) {}
    rpc LogList(LogListRequest) returns (LogListResponse) {}
    rpc LogFrom(LogFromRequest) returns (LogFromResponse) {}
}
```

Listing 5.7: **LoggingService** service definition

Storing the logs

MongoDB is used to store the logs. The keep the log separate we use a collection for each of the services. This way we can specify which service to get the logs from later, and to have to get all logs and search through them. We can use a Map to keep the different collections and use the name of the individual services as the key. This way can use the name to get the collection and write the log. If there is no collection associated with a service, we can create a new collection and add it to the Map. The implementation of this can be seen on listing 5.8.

```
private final Map<String, MongoCollection<Document>> serviceLogs;
public void writeLog(Log log) {
    MongoCollection<Document> serviceLog;
    if (!serviceLogs.containsKey(log.serviceName())) {
        if (!this.database.listCollectionNames().into(new HashSet
    <>()).contains(log.serviceName())) {
            this.database.createCollection(log.serviceName());
        }
        serviceLog = this.database.getCollection(log.serviceName());
        this.serviceLogs.put(log.serviceName(), serviceLog);
    }
```

```
11      else {
12          serviceLog = this.serviceLogs.get(log.serviceName());
13      }
14      Document document = new Document();
15      // fill document
16      serviceLog.insertOne(document);
17  }
```

Listing 5.8: Collection Map and *writeLog* method

LoggingClient

For this service there is also implemented a standard client. This client can be added to any of the other services as a dependency. This way we do not have to rewrite the same client for every service.

This client implementation has a single extra method, that must be called before using the service, called *init*. The reason for this is that the client needs to know the address and port to call, and so will need to be initialised first. In addition to the address and port, it also needs to know the name of the service where the dependency is used. This has to do with how the logs are kept. They are separated by service, and so the name is used for this purpose.

5.5 NotificationService

To implement this service, we can take advantage of the server-sent-event (SSE) API, that is native to newer versions of all browsers. This allows us to keep a connection between the browser and the **NotificationService**, that we can use to send notifications. We can store the subscriptions in a Map, where we use the userId as the key and the connection as the value.

This microservice is written in *TypeScript* with *Node.js* as runtime. The reason for using *TypeScript* for this service and not *Java*, is that it was found that it worked better with the SSE API.

To notify a single user we will need the event to contain an receiverId in the data object. This id will then be used to lookup the connection in the Map. Sending messages with SSE we need to end every message with a double newline. On linsting 5.9, can we see how the service sends notifications.

```
1  private connections: Map<String,Response> = new Map();
2
3  private _notify = (id: String, message: String) => {
4      const res = this.connections.get(id);
5      if (res) {
6          res.write(`data: ${message}\n\n`);
```

```
    }
}
```

Listing 5.9: Notification method implementation

When subscribing to the service, we can register an *onmessage* handler. It get an EventMessage as input, where we can access the notification under *data*. We can see an example of this on listing 5.10.

```
const sse = new EventSource(`${url}/subscribe/${id}`);
sse.onmessage = (res) => {
    const data = res.data;
    //handle data
}
```

Listing 5.10: Subscribing to the service

5.6 DiagnosticsService

This service is more focused on running a procedure then exposing resources, so for this reason it be implemented with a *RPC* API. The *RPC* framework used to make the API will be *gRPC*. Having designed the interface for this service already, we just need to convert it into a *Protocol Buffer* file(.proto). Listing 5.11 shows the entire proto file for this service.

```
service SirupDiagnosticsService {
    rpc RunDiagnostics(DiagnosticsRequest) returns (
    DiagnosticsResponse) {}
}
message DiagnosticsRequest {
    string token = 1;
    string user_id = 2;
}
message DiagnosticsResponse {
    Report report = 1;
}
message Report {
    int64 total_time = 1;
    repeated Vital vitals = 2;
}
message Vital {
    string service_name = 1;
    bool running = 2;
    int64 response_time = 3;
}
```

Listing 5.11: **DiagnosticsService** proto file

5.7 GenerationServices

For demonstrating how a **GenerationService** implementation can look like, the **JavaGenerationService** will generate *Java* code and feature *REST* for the API and *PostgreSQL* for the database.

5.8 JavaGenerationService

This service will need to comply with the specifications stated in section 4.9, this includes that this service registers itself when starting. To do this it must call the */register* endpoint of the **RegistserService**. In this registration it must provide a *manifest* that contains the details of what it can generate. This initial implementation of this service will generate microservices with *REST* APIs and *PostgreSQL* as database, this generated code will be in *Java*. As part of the *options* for configuring the generated *REST* API, we provide the ability to add endpoints, these can be linked to some method and data from the database. The added option will generate the linked method in a controller class for the specified data and set the method to handle the requests for the endpoint. The full *manifest* can be seen in appendix D.

The specification for the **GenerationServices** also note that the service should, if possible, unregister when closing. We can do this by registering a shutdown hook in the service. This hook will call the endpoint that removes the registration. This hook will only work if the service closes gracefully, so there is no guarantee that it will trigger, which is fine.

MircoserviceBuilder

For building the microservice that this service will generate we need a class to handle the overall context and structure of this new microservice. Here we are using a class called **Microservice** that feature the builder pattern for its initialisation. The building process consists of setting the **IApi**, **IDatabase** and **IBuildTool**, as well as the name of the new microservice and its group id. To make this build process extendable for future additions to this generation service, we can use the *factory* pattern [3][p. 107] to create a *BuilderFactory* for both the **IApiBuilder** and **IDatabaseBuilder** implementations.

To handle bad input for the new microsrevice, the exceptions are added:

- **ApiNotSupportedException**
- **DatabaseNotSupportedException**
- **BuildToolNotSupported**
- **LangaugeNotSupportedException**

```
public final class Apis {
    public static IApiBuilder<? extends IApi> ofType(String api) {
        api = api.toLowerCase(); //Sanitise
        switch (api) {
            case "rest", "restful" -> {
                return Rest.builder();
            }
            default -> throw new ApiNotSupportedException("API [" +
    api + "] is not supported");
        }
    }
}
```

Listing 5.12: **Apis.java**, IApiBuilderFactory

To make sure that the *builder* methods that these factories are responsible for creating is only called from the factories, we can limit the access to these methods to only be accessible within the package. Listing 5.12 show how this is implemented for the APIs, a similar implementation exists for the databases.

ClassGenerator

To write the lines of code to the generated class files, a helper class called **ClassGenerator** is made. This class utilise the builder pattern, as the construction of a class require many inputs. The **ClassGenerator** takes a *FileWriter* and a *Generateable* as necessary inputs. After the two nesseary arguments, we can specify if we need it to generate the file as a *Class*, *Interface* or *Record*. As default it will generate the file as a *Class*. The *classType* method of the builder takes an implementation of the **ClassType** interface, all of these can be found the **ClassTypes** class, see listing 5.13.

```
public class ClassTypes {
    public static ClassType INTERFACE() {
        return ClassGenerator::generateInterface;
    }
    public static ClassType RECORD() {
        return ClassGenerator::generateRecord;
    }
    public static ClassType CLASS() {
        return ClassGenerator::generateClass;
    }
}
```

Listing 5.13: **ClassTypes**.java

The **ClassType** interface only contains a single method called *fill*, it takes a **ClassGenerator** as argument, as can be seen on listing 5.14. This way we provide the **ClassGenerator** with a callback function that will be called when calling the *make* function of the constructed **ClassGenerator** object.

```
public interface ClassType {
    void fill(ClassGenerator classGenerator) throws IOException;
```

```
}
```

Listing 5.14: **ClassType**.java

The next two parts of the configuration of the **ClassGenerator** is the imports and the actual body of the class. Both of these take a callback function which provides a **ImportGenerator** and **ClassGenerator** respectively. The **ImportGenerator** provides method for writing imports and static imports, this is done as a separated class to ensure that only these two methods can be used. The callback for the body of the class, provides the final **ClassGenerator** object that the builder is constructing, and provides methods for writing constructors, methods, attributes, annotations and even try-catch. An example of how to use a **ClassGenerator** can be seen on listing 5.15.

```
ClassGenerator.builder()
    .fileWriter(fileWriter)
    .generateable(this)
    .classType(ClassTypes.CLASS())
    .classImports(importGenerator -> {
        //All the imports
    })
    .classBody(classGenerator -> {
       //All the class contents
    })
    .build()
    .make();
```

Listing 5.15: Basic example of the **ClassGenerator** for generating a *Class*

When building the **ClassGenerator** it is also possible to specify if the class is generic, add interfaces that it implements, add a class that it extends, and specifically for *Record* classes we can provide a **DataModel**, data it then writes as a java *Record* class. Listing 5.16 show how to use the **ClassGenerator** to write a java *Record* class and listing 5.17 show the creation of a generic *Interface*.

```
ClassGenerator.builder()
    .fileWriter(fileWriter)
    .generateable(this)
    .classType(ClassTypes.RECORD())
    .dataModel(this)
    .implement("Model")
    .classImports(importGenerator -> {
        //All the imports
    })
    .build()
    .make();
```

Listing 5.16: Basic example of the **ClassGenerator** for generating a *Record*

```
ClassGenerator.builder()
    .fileWriter(fileWriter)
    .generateable(this)
```

```
    .classType(ClassTypes.INTERFACE())
    .generic("T extends Model")
    .classImports(importGenerator -> {
        //All the imports
    })
    .classBody(classGenerator -> {
        //All the class contents
    })
    .build()
    .make();
```

Listing 5.17: Basic example of the **ClassGenerator** for generating an *Interface*

Going through the final process of writing the code to file, we need to start with the *make* function. It calls the callback function provided when setting the **ClassType**, this then calls a private function called *generateJava*, that first writes the package for the class, this is provided by the **Generateable** object that was set during the building of this **ClassGenerator** object. Next it calls the callback that the *generateJava* provides, this will then call the *importsFiller* callback that was provided during the objects building phase, then the start of the actual class, starting with the class declaration followed by calling the *classBodyFille* that was set in the building phase, lastly the *generateJava* function ends the class. The code for this example can be seen on listing 5.18.

```
public class ClassGenerator {
    ...
    public void make() throws IOException {
        this.classType.fill(this);
    }
    ...
    private void generateJava(Filler filler) throws IOException {
        this.fileWriter.write(packageString(
            this.generateable.getPackageName()));
        filler.fill(this);
        this.fileWriter.write(endClassSting());
    }
    ...
    //ClassType implementation
    public void generateClass() throws IOException {
        this.generateJava((classGenerator) -> {
            //Provided when building
            this.importsFiller.fill(new ImportGenerator(this.
    fileWriter));
            this.fileWriter.write(classString());
            //Provided when building
            this.classBodyFiller.fill(classGenerator);
        });
    }
    ...
```

```
25 }
```

Listing 5.18: **ClassGenerator** *make* function code

Parsing the request

The incoming request are represented as *JSON*, we want to work with *Java* objects. To do this we can use a *JSON* library, like Google's *GSON*. For *GSON* to convert the request to an object it needs to know which object to convert to, so we need to create a class that is a one-to-one of the *manifest* for this service, this class can be seen in appendix E.

The first part, when parsing a request, is creating a Map of all the datatypes, this can be seen on listing 5.19. We need this Map later as the generated API and database will need to know which datatype are available. We use a Map so that we can index with the name of the datatype later.

```
1  private static Map<String,DataModel> buildDataModelMap(
       MicroserviceRequest m) {
2      Map<String, DataModel> dataModelMap = new HashMap<>();
3      m.microservice().database().data().collections().forEach(
       collection -> {
4          DataModel.DataModelBuilder dataModelBuilder =
5              DataModel.builder();
6          dataModelBuilder.name(collection.name());
7          if (collection.fields() != null) {
8              collection.fields().forEach(field -> {
9                  dataModelBuilder.dataField(field.type(),
10                     field.name(), field.ref());
11             });
12         }
13         dataModelMap.put(collection.name(), dataModelBuilder.build()
       );
14     });
15     return dataModelMap;
16 }
```

Listing 5.19: DataModels Map creation

The next part, is very easy since we have done all the planning. We can use the *builders* of the different parts that we created earlier. We just map the request object onto the *builders*. On listing 5.20 can we see the entire building process.

```
1  public static Microservice fromJsonRequest(MicroserviceRequest m) {
2      Map<String, DataModel> dataModelMap = buildDataModelMap(m);
3      String lang = m.microservice().language().name().toLowerCase();
4      if (!lang.equals("java")) {
5          throw new LanguageNotSupportedException(
6              "Language [" + lang + "] is not supported");
7      }
8      return Microservice.builder()
```

```
9           .id(m.microservice().microserviceId())
10          .docker(m.docker())
11          .name(m.microservice().microserviceName())
12          .groupId(m.microservice().language().options().groupId())
13          .api(Apis.ofType(
14            m.microservice().api().type())
15              .dataMap(dataModelMap)
16              .options(m.microservice().api().options())
17          )
18          .database(Databases.ofType(
19            m.microservice().database().name())
20              .dataModels(dataModelMap.values().stream().toList())
21              .options(m.microservice().database().options())
22          )
23          .buildTool(BuildTools.ofType(
24              m.microservice().language().options().buildTool()))
25          .build();
26  }
```

Listing 5.20: Microservice building process

After parsing the request and having built the microservice, we can now run the *make* method from the new microservice. The first thing it does it to create the directories. To keep the different generated code separated and so we later can access it, we name the base directory of the new code after the provided ID in the request. After this is done, are all the **Generateable** classes implementations of the *generateClassFile* or *generate* methods called. This will fist create the file and then call the *fillFile* method. After all the files have been generated and filled, we need to provide a link to were the code can be downloaded, we do this by returning a link to the base directory of the generated code, which is its ID, so it will look like: <api-url>/microservice/<id>.

Accessing this link will look for the directory if it finds it will recursively loop over the directory and zip it. It will then add this zip file as an attachment to the response. Having the generation and downloading separate means that we can keep the generated code, so we do only have to run the generation process once.

Generate REST

To generate the code needed for the REST API of the new microservice, we need to get the endpoints from the *options* object that is passed when building the microservice. We need get all the endpoints from the endpoint groups that are in the *options*, these can contain other endpoint groups, so we need to recursively iterate over these endpoint groups. If an endpoint group has been linked with some of the data from the data, we need to send the request for that endpoint to the controller class for that data model. Whenever a new endpoint group is added we also add the controller that the group is linked to. Listing 5.21 shows how these endpoints and controllers are added.

```
1    private void addEndpointGroup(EndpointGroup endpointGroup) {
2        this.endpointGroups.add(endpointGroup);
3        updateControllers(endpointGroup);
4    }
5    private void updateControllers(EndpointGroup endpointGroup) {
6        if (endpointGroup.getController() != null) {
7            this.controllers.add(endpointGroup.getController());
8        }
9        for (EndpointGroup innerGroup : endpointGroup.getInnerGroups
     ()) {
10           updateControllers(innerGroup);
11       }
12   }
```

Listing 5.21: *addEndpointGroup* and *updateControllers* methods from **Rest** class

Getting the data model is done by indexing into the *dataMap* that also is passed in the building process. The *dataMap* is also used when generation the code that will interact with the database, so we know that if we can find the data model in the Map, then it will also have a controller and service class implemented. This process is run when the *build* method on the builder is called. The *build* method can be seen on listing 5.22.

```
1    @Override
2    public Rest build() {
3        if (options.endpointGroups() != null &&
4            this.rest.dataMap != null) {
5            this.rest.addEndpointGroup(
6                this.rest.iterateEndpointGroups(
7                    options.endpointGroups().get(0),0).build());
8        }
9        return this.rest;
10   }
```

Listing 5.22: *build* implementation on the **RestBuilder** class

Generate PostgreSQL

To generate the code needed for PostgreSQL, we first use the list of data model that we want the database to store. When building the microservice we pass a list of data models, we use this to add setup service classes, these will be the once that interact with the database, there will be one for every data model. On listing 5.23, can we see the implementation for the *dataModels* method on the **PostgreSQLBuilder**.

```
1    @Override
2    public IDatabaseBuilder<PostgreSQL> dataModels(List<DataModel>
     dataModels) {
3        this.postgreSQL.addDataModels(dataModels);
4        for (DataModel dataModel : dataModels) {
5            this.postgreSQL.addService(
```

```
6                   PostgreSqlService.of(dataModel, this.postgreSQL));
7           }
8           return this;
9       }
```

Listing 5.23: *dataModels* method on the **PostgreSQLBuilder** class

Generating the Microservice

When the building of the microservice is done, we sort the different **Generateables** into either the list of all classes that will be generated, of the list of the other files. Then we can call the *make* method on the **Microservice** class, this then creates a **FileGenerator** class, that we then pass each of the **Generateables** to. This can be seen on listing 5.24. The individual **Generatables** are then each generating the code for the class that they are an abstraction for.

```
1       public String make() {
2           long start = System.currentTimeMillis();
3           if (this.id == null || this.id.isEmpty()) {
4               this.id = UUID.randomUUID().toString();
5           }
6           FileGenerator fileGenerator = new FileGenerator(this.id,
    this.getName(), this.groupId);
7           fileGenerator.generateFileStructure();
8           for (Generateable clazz : this.classes) {
9               fileGenerator.generateClassFile(clazz);
10          }
11          for (Generateable file : this.otherFiles) {
12              fileGenerator.generate(file);
13          }
14          logger.info("Microservice " + id(this.id) + " created in: "
    + (System.currentTimeMillis() - start) + "ms" );
15          return this.id;
16      }
```

Listing 5.24: *make* method from **Microservice** class

The method on listing 5.24 return the id of the service, we need this to create the link where the code can be downloaded from. Given the *fromJsonRequest*(listing 5.20) method, that can build the microservice from the request and the *make*(listing 5.24) method, that creates the files and code, makes the actual method for handling the generation request very simple. We need to catch any of the -NotSupportedExceptions that might be thrown. On listing 5.25 is the entire method that handles generation request.

```
1       public Object generate(Request request, Response response) {
2           try {
3               Microservice microservice =
4                   Microservice.fromJsonRequest(request.body());
5               String microserviceId = microservice.make();
6               return this.gson.toJson(
```

```
7                 new ReturnObj<>(201, "Microservice created",
8             Env.API_ADDRESS + ":" + Env.API_PORT + Env.API_BASE_URL
    + "/microservice/" + microserviceId));
9         } catch (ApiNotSupportedException |
    DatabaseNotSupportedException | NoSuchBuildToolException |
    LanguageNotSupportedException e) {
10            e.printStackTrace();
11            halt(405, e.getMessage());
12        }
13        return this.gson.toJson(new ReturnObj<>(500, "Something went
     wrong :/", null));
14    }
15    private record ReturnObj<T>(int statusCode, String message, T
    data) {}
```

Listing 5.25: Generation request handler *generate* method

Extending the extension

If we look at what it will require to add the ability for the service to generate with a new type of API. We would need to do three things:

- The actual implementation that generates the code (the hard part).
- Add the new type to the builder factory.
- Update the *manifest.json*.

The same process is also required when adding a new database or build tool.

5.9 Website

The frontend for the service will be made as website, many different framework and libraries are available to choose from. As *Svelte* is very simple to use, it was chosen, and to minimise implementation error it will be written with *TypeScript*. The frontend has to implement parts of the *Extendsion interface*(see section 4.9) in reverse to the actual *GenerationServices*, as to has to translate the *manifest.json* that the *GenerationServices* provide.

To keep most of the logic separated from the UI elements, it will use MVVM architecture. This mean that we have a viewmodel for each of the different data type (User, Organisation, Project, and Microservice), the viewmodels will handle all the logic related to its datatype.

For implementing the website, we have two main points of interest, the first is how we will interact with the different microservices in the backend, the

second is how we will translate the *manifests* from the **GenerationServices** into HTML.

Fetching data from the backend

The frontend will have to interact with the **UserService**, **RegisterService**, and the different **GenerationServices**. Since the responses for these services are standardised, we also need to handle the responses the way they are formatted. For this reason, we need to make a type for the responses. The responses will always contain a status code and a message, if the request was successful, we also have the data in the response, this can be a number of different things, so we need to type to be generic. The final type is called **ApiResponse** and can be seen on listing 5.26.

```
export type ApiResponse<T> = {
    statusCode: number;
    message: string;
    data: T;
}
```

Listing 5.26: **ApiResponse** type

To make it easy to access these services, a wrapper for the *fetch* API was used, called **AJ**(AppleJack). This class is implemented as a *singleton* [3][p. 127] to keep the user credentials that are nesseary for request authentication. This class is only accessed through another class that allows for specifying which of the services that will be accessed. Listing 5.27 show how this class is used.

First *aj()* is called from this there are three options: *user()*(**UserService**), *register()*(**RegisterService**), and *generator(url)*(**GenerationService**). This then allows for performing normal **HTTP** request, with the added bonus of specifying the types for both the request and response body. On the example we can see a *POST* request that sends a **User** object in it body and expects both a **User** and **Token** as the response. Then the endpoint for the **UserService** is specified ("/user/login") and the **User** object is passed.

```
login = async (_user: User): Promise<boolean> => {
    const res = await aj().user().POST<User, {user: User, token:
    string}>("/user/login", _user);
    return UserViewModel.handleUser(res);
}
```

Listing 5.27: **AJ** use example

AJ will some of the error handling but the consumer of this class should still check the status code or message before working with the data. Listing 5.28 shows how the wrapper for *POST* request are handled. The URL is set when selecting the service.

```
POST = async <S,R>(endpoint: string, body: S): Promise<ApiResponse<R | null>> => {
    try {
        const res = await fetch(`${this.url}${endpoint}`, {
            method: "POST",
            headers: {
                "Content-Type": "Application/json"
            },
            body: JSON.stringify(body),
        });
        if (res.status === 200 || res.status === 201) {
            return await res.json();
        }
        return { statusCode: res.status, message: res.statusText, data: null } as ApiResponse<null>;
    } catch (err) {
        console.log(err);
        error.set({ code: 503, type: "Service", message: "Service Unavailable" } as Error);
        return { statusCode: 503, message: "Service Unavailable", data: null } as ApiResponse<null>;
    }
}
```

Listing 5.28: **AJ** POST wrapper

Microservice Design Page

The page that allows users to design the microservices that they want the system to generate, we need to take the **GenerationServices** provided *manifests* and translate it into *HTML*, so users know what options they have available to choose from. *Svelte* allows for adding conditional logic, with *if-blocks*, and for looping over arrays of values, with *each-blocks*, in the *HTML*. This enables us to take a *manifest* as input and check for the different allowed types, as specified in section 4.9, and convert them into *HTML*.

Listing 5.29 show a code snippet from the microservice design page, where the option from a *manifest* gets translated. We can notice that in the first *each-block* that *language* is prefixed with a '$', this is how reactive states can be accessed in *Svlete*. In this case the *language* refers to the language that is selected, and so if a different language get selected instead we can render the page with the new *manifest* that corresponds to the selected language.

```
{#each $language.options as option, i}
    {#each Object.entries(option) as [optionName, optionValue]}
        {#each Object.entries(optionValue) as [key, values]}
            {#if key === 'value'}
                <!-- HTML -->
            {/if}
            {#if key === 'selection'}
```

```
                <!-- HTML -->
            {/if}
            {#if key === 'repeated'}
                <!-- HTML -->
            {/if}
            {#if key === 'object'}
                <!-- HTML -->
            {/if}
        {/each}
    {/each}
{/each}
```

Listing 5.29: Translation of manifest language options into HTML

The *HTML* for the first two types (*value* and *selection*) are the easiest to convert, as they are more limited in what they can contain.

Value

For *value* we can place an *input* element of the same type as declared in the *manifest* for that option. An example of this can be seen on listing 5.30, where we can see that the value of the *input* is bound to a *languageConfigs* array. This array stores the values before they are added to the mircoservice generation file. In addition to the *input* where the user can input the value there is also an *input* of type submit, this listens for *on:click* events and will call the *addLanguageOption* function when this event happens.

```
{#if key === 'value'}
    <h3 class="option-name">{optionName}</h3>
    <div class="add">
        <input
            type={value}
            name={optionName}
            id="i_{optionName}"
            bind:value={$languageConfigs[i]}
            placeholder="Enter {optionName}"
        />
        <input
            type="submit"
            value="Submit"
            on:click={() => addLanguageOption(optionName, i)}
        />
    </div>
{/if}
```

Listing 5.30: *Value* option from *manifest* to *HTML*

Selection

For the *selection* object we can have the header for the option act as a toggle to show or hide the selection list. When displaying the selection list, we can

use an unordered list element(<ul>), and use a list element(<li>) for the individual selection. An example of how this is implemented can be seen on listing 5.29.

```
{#if key === 'selection'}
    <h3
        class="option-name selection-title"
        on:click={() =>
        ($languageConfigsShow[i] = !$languageConfigsShow[i])}
    >
        {optionName}: {$genJson.microservice.language.options[
    optionName]}
    </h3>
    {#if $languageConfigsShow[i]}
        <ul>
            {#each values as value}
                <li on:click={() =>
                addLanguageOptionValue(optionName, value, i)}>
                    <h4 class="selection-title">{value}</h4>
                </li>
            {/each}
        </ul>
    {/if}
{/if}
```

Listing 5.31: *Selection* option from *manifest* to *HTML*

Object

The *object* option is the most complicated, as it can contain multiple options. We know that it will contain a *values* object that we can iterate over. The inner values in the *object* option can then contain inner options of any of the option types. The example on listing 5.32 show a simplified implementation of the *object* option.

```
{#if key === 'object'}
    {#each value.values as objectValues}
        {#if objectValues.value}
            <!-- HTML -->
        {/if}
        {#if objectValues.selection}
            <!-- HTML -->
        {/if}
        {#if objectValues.repeated}
            <!-- HTML -->
        {/if}
        {#if objectValues.object}
            <!-- HTML -->
        {/if}
    {/each}
{/if}
```

Listing 5.32: *Objcet* option from *manifest* to *HTML*

Repeated

The *repeated* option is more complicated in its implementation, it will need to need to check the inner objects which again can contain all the other option types. Listing 5.33 show a simplified implementation of the *repeated* option. We can refer to the previous subsections to see how the inner options are handled.

```
{#if key === 'repeated'}
    <h3 class="option-name">{optionName}</h3>
    {#each Object.entries(value) as [innerKey, innerValue]}
        {#if innerKey === 'value'}
            <!-- HTML -->
        {/if}
        {#if innerKey === 'selection'}
            <!-- HTML -->
        {/if}
        {#if innerKey === 'object'}
            <!-- HTML -->
        {/if}
    {/each}
{/if}
```

Listing 5.33: *Repeated* option from *manifest* to *HTML*

Checking that the implementation was correct, during implementation, there was created a simple testing *manifest*(see appendix F), this would validate that the option type worked as expected.

5.10 AdminAccess

The admin access is one part that is likely to see new features added. So, to keep the admin CLI extendable, we want to make the process of adding new commands as easy as possible. To do this, we can use the *decorator* pattern [3][p. 175] by using *Reflections* with *Annotations*. We need to be able to mark classes and methods that will handle commands. For the classes, we create the annotation, **CliActionsClass**, as can be seen on listing 5.34, it will mark any class that will contain methods used for handling commands.

```
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    public @interface CliActionsClass {}
```

Listing 5.34: **CliActionsClass** annotations

To mark the command methods, we now create the annotation, **CliAction**, it will have a field for specifying the commands name, called *command*. Additionally it will also have to optional fields: alias, for a short name for the command, and description, for giving the command a description that can be seen when using the *help* command. The **CliAction** annotation can be seen on listing 5.35.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CliAction {
    String command();
    String alias() default "";
    String description() default "";
}
```

Listing 5.35: **CliAction** annotaion

Some commands will have optional arguments, and so we also want to mark these for the different commands. Here we create the annotations **CliArgs** and **CliArg**, see listing 5.36. The **CliArgs** annotation holds an array of the **CliArg** annotaion, so we can add multiple arguments for the different commands. The **CliArg** annotation, has a field for specifying the argument letter, called *flag*, and two optional fields: arg, for when the argument needs a value, and description, for giving the argument a description that will be displayed when using the *help* command.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CliArgs {
    CliArg[] value();
        @Retention(RetentionPolicy.RUNTIME)
        @Target(ElementType.METHOD)
        @Repeatable(CliArgs.class)
        @interface CliArg {
            String flag();
            String arg() default "";
            String description() default "";
        }
}
```

Listing 5.36: **CliArgs** annotation and **CliArg** annotation

Looking at an example of how these annotations are used, we can look at listing 5.37.

```java
@CliActionsClass
public class AuthActions {
    @CliSecureAction(command = "log_from", alias = "lf", description
     = "Get all the logs for a service")
    @CliArgs(value = {
            @CliArgs.Arg(flag = "s", arg = "serviceName",
    description = "Specify the service to get logs from"),
            @CliArgs.Arg(flag = "n", arg = "number", description = "
    Specify the amount of logs printed at a time")
    })
    public static void getLogsFrom(Console console, Map<String,
    String> argMap) {
        ...
    }
}
```

Listing 5.37: Example of **CliAction** annotation usage

The example from listing 5.37 will result in the printout seen on listing 5.38.

```
Enter command: help
...
log_from, lf -> Get all the logs for a service
    commands options:
    -s <serviceName> -> Specify the service to get logs from
    -n <number> -> Specify the number of logs printed at a time
...
```

Listing 5.38: Example of help printout

In the design of the admin frontend, it was stated that the users will have to login before accessing the CLI. To keep the CLI-kit more versatile, we will want to have an option to have the login optional. This is done by having an interface called **LoginHanlder**, it will have a single method called *login*, this method will return a Boolean. It will return true if the login was successful and false otherwise. When running the CLI it will check if an implementation of the **LoginHandler** is provide, if there is, it will first call the login before running a secure loop. If no **LoginHandler** is provided it will skip and run the unsecure loop. When starting in secure mode, the CLI-kit will parse all the classes annotated with **CLiSecureActionsClass** and add all the methods with the **CliSecureAction** to the existing map containing commands and methods.

In addition to the **LoginHandler**, it is possible to provide a custom dynamic welcome message. This is again done with an interface **PrintCallBack**, with a *print* method. This method will be called on successful user login.

```
1    public static void main(String[] args) {
2        String packageName = "sirup.admin" //The base package
3        new SirupCli(packageName)
4        .addLoginHandler(Security::loginHandler) //optional
5        .addWelcomeMessage(Security::printWelcomeMessage) //optional
6        .start();
7    }
```

Listing 5.39: **SirupCli** - CLI-kit usage

On listing 5.39, we can see how the CLI-kit is used. On line 4, we can see that a **LoginHandler** implementation is added, and line 5 shows the addition of a dynamic welcome message. Both of these implementations are located in a separate class that handles all the custom secure methods. The added **LoginHandler** can be seen on listing 5.40, the *login* method seen on line 6, calls the **UserService**(see section 5.2), that will handle the validation of the provided credentials. From the login request we also we the users system access, and so before the login can be a success it will need to be an admin.

```
1    public static boolean loginHandler(Console console) {
2        System.out.print("username: ");
3        String username = console.readLine();
```

```
4           System.out.print("password: ");
5           char[] pass = console.readPassword();
6           return login(username, pass);
7       }
```

Listing 5.40: **LoginHandler** implementation

6 Evaluation

6.1 Handling requirements

Looking back the work done, we can now see if the project was a sucess, in terms of handling the most important requirements from section 3.4.

F-01 and its sub-requirements are all being handled by the **JavaGenerationService** (see sections 5.8 and 4.9).

F-02, **F-03**, **F-04**, and their sub-requirements are handled by the website (see sections 3.6, 4.10, and 5.9) in combination with the **UserService** (see sections 3.5, 4.3, and 5.2).

F-04 and its sub-requirements are handled by the **AdminAccess** (see sections 3.6, 4.11, and 5.10).

NF-01 and its sub-requirement is handled by the interface for the system (see sections 3.5, 4.8, and 5.7).

NF-02 is handled by the **AuthenticationService** (see sections 3.5, 4.2, and 5.1) with its use by the other microservices.

NF-05 is handled by the **DiagnosticsService** (see sections 3.5, 4.7, and 5.6).

NF-06 is handled by the **LoggingService** (see sections 3.5, 4.5, and 5.4) and its use by the other microservices.

6.2 Working with microservices

Making a system consisting of microservice that, at the end, could create other microservices, was a good way of getting a deeper understanding of how to implement both the microservices that would be part of the core system, and the way to design the **GenerationServices** and how they would interface with the system.

6.3 Generation file

During the project, it was considered that a new file format could have been used as the generation files, that the **GenerationServices** use to generate the code from. However, it was decided that using JSON, and potentially YAML, was a better choice, as most developers are familiar with both formats. Making a new format could result in files that would be more concise, but it would also require that new developers would have to learn this format. Designing this new format would also take time, while using existing format would not. So, using an existing format meant that more time could be spent on designing the rest of the system and how it could be modular.

7 Discussion

7.1 Where to go from here?

Having to design and implement the entire system in a relatively short time, leaves much to be decried in terms of wanting to continue the work. There are so many features that could be added to the system, which make it feel like the work has only just begun. The most obvious place to continue is with the **JavaGenerationService** and adding other **GenerationServices**. Both options will require some time to implement, but the underlying system is made to incorporate both.

There are also a lot of things that would be nice to add to the website. The whole permissions aspect of it is far from finished, as they are not enforced by the website. A crucial part of the system was that it could be used to as documentation while designing and generating the different microservice. This part could be better realised, possibly by having three stages to the creation of a new microservice. The first would be about analysing and result in making a contract for the microservice that would detail the responsibility of the microservice. The second stage would be much like the current design page, where users could design the service by making the generation file. The last stage could then be after the users had generated the base microservice, from a **GenerationService**, and then focus on implementing the more complicated parts them self. Splitting the process up like this could help to give users a more structured approach to using the system and could give the other users a better understanding of how far in the process the different microservices are.

The scope of this project is virtually limitless, as there are a lot that could be added to the current **JavaGenerationService** and so many other **GenerationServices** that could be added. This large scope was also felt during the project, as the focus had to set on implementing the core system and limiting the work on the **JavaGenerationService**, as it could demonstrate the idea.

7.2 Monetisation

So how could the system stay operational, where would the money come from to pay for the upkeep?

Currently there is no monetisation scheme in the system, but it could be added, by setting a limit to the amount of microservices each user could generate before paying for an upgraded account. The rest of the system would still be useable to normal users, and only the generation feature would be limited. This is partially integrated with the PRO user level access in the **UserService**. This role was planned to be used for paying users.

7.3 Could the JavaGenerationService be done better?

In the process of making the **AdminAccess**, the *decorator* [3][p. 175] pattern was used. This allowed for adding new features without having to update the core of the code. A similar approach could be used, where annotation would be used to mark the different classes. The system would then make the *manifest.json* and update the **BuilderFactories** by itself. This would make the process of adding new options for API types and databases better, as the development would only be concerned with implementing the generation code. The reason for not having done this in the first place, is that the knowledge of how to implement such a mechanism was first learned while implementing the **AdminAccess**, where the **JavaGenerationService** already was far in its development.

8 Conclusion

This project set out to create a system that could generate microservice from a design file, this goal was achieved. Users can create organisations where they can invite other users to collaborate with. In these organisations can they create projects where they can then create multiple different microservices. They can then use the design page to design and configure a file that can be used by a part of the system to create a microservice, that the user then can download. The current **GenerationService**, is only able to generate relatively simple microservices, but it demonstrates the intend of the project. A core part of the system was also that it should be modular in the way **GenerationServices** could be added. This part was also realised, seen by how the **RegisterService** was implemented in the system.

Even though the **JavaGenerationService** was designed to work with a microservice architecture system in mind. It could easily work as a standalone program. This speaks to a strong aspect of microservice architecture system, that each service can be deployed and work separately. This works well with the current system, where we can deploy new **GenerationServices** separately while the system is running.

Many different microservices were created, each with their own responsibility to handle and solve part of the system requirements. This led to these microservice being quite different, and so had been approached in different ways. This would lead to finding which type of API and database would be the best fit for each microservice.

Though the main goal has been achieved and the system works, but since only a single API type and database can be chosen when designing the new microservices, makes it hard to say that the system is complete. It is fairer to say that it is in an early stage of development, and with time could expand into something useful. In its current state, it is more of a tech demonstration.

Bibliography

[1] Benjamin Anderson and Brad Nicholson. *SQL vs. NoSQL Databases: What's the Difference?* https://www.ibm.com/cloud/blog/sql-vs-nosql. 2022.

[2] Hector Carcia-Molina, Jeffrye D. Ullman, and Jennifer Widom. *Database Systems*. Pearson Prentice Hall, 2009.

[3] Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[4] Red Hat. *What is a webhook*. https://www.redhat.com/en/topics/automation/what-is-a-webhook. 2022.

[5] JetBrains. *Microservices*. https://www.jetbrains.com/lp/devecosystem-2021/microservices/. 2021.

[6] Mozilla. *The WebSocket API (WebSockets)*. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. 2023.

[7] Mozilla. *Using server-sent events*. https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events. 2023.

[8] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[9] Apollo Team. *What is GraphQL?* https://graphql.com/learn/what-is-graphql/. 2023.

[10] Frank Tsui, Orlando Karam, and Barbara Bernal. *Software Engineering*. Jones & Bartlett Learning, 2018.

9 Appendix

A Requirements table

Functional

ID	Name	Description
F-01	Generate Microservice	The system **must** be able to generate microservice
F-01.1	Specify Language	It **must** be possible to select the language of the microservice
F-01.1.A	Different Languages	It **should** provide the user with different language options
F-01.2	Specify Database	It **must** be possible to select the database of the microservice
F-01.2.A	Different Databases	It **should** provide the user with different database options
F-01.3	Specify API	It **must** be possible to select the type of API for the microsrevice
F-01.3.A	Different API types	It **should** provide the user with different types for API
F-01.3.B	Secifiy Endpoints / Methods	It **must** be possible to define different endpoints/methods for the API chosen for the microservice
F-02	Designer	The service **must** provide a tool for designing mircoservices
F-02.1	Documentation	The design of the microservices **should** also be usable as documentationd
F-03	Collaboration	Users **must** be able to collaborate on their work with other users
F-03.1	Organsiations	Users **must** be able work with different organisations
F-03.1.A	Create Organsiation	Users **must** be able to create organisations for their work
F-03.1.B	Invite User	Users **must** be able to invite other users to organisations
F-03.1.C	User Permissions	Users **should** have different permissions
F-03.1.D	Change Permissiosn	It **should** be possible for users with a high permission level to change the permission of other users

F-03.2	Projects	Organisations **must** be able to contain different projects
F-03.2.A	Create Project	Users **must** be able to create project under the different organisations they are assigned to
F-03.3	Services	Projects **must** be able to contain different services
F-03.3.A	Create Servce	Users **must** be able to create services under the different projects that are assigned to
F-04	Admin access	These **must** be way for admin users to access the system
F-04.1	Create new admin users	This admin access **must** have a way to add new admin users to the system.
F-04.2	Create service credentials	This admin access **must** have a way for admins to create new credentials that extension services need
F-04.3	View logs	This admin access **should** have a way for to display logs

Non-Functional

ID	Name	Description
NF-01	Extendability	It **must** be easy to add new generation services to the system
NF-01.1	Service Standard	Generation services **must** follow a specification standard
NF-02	Security	The system **must** implement measures to ensure that only authorized users can the resources that they are approved for
NF-03	Minimal User Data	The system **should** contain as little user specific data as possible
NF-04	Deployment	The system **should** be easely deployable
NF-04.1	Docker	All microservices **should** be deployable with Docker
NF-05	Diagnostics	The system **must** contain come method of self Diagnostics
NF-06	Logging	The system **must** keep a log of actions performed by users and services
NF-07	Notification	The system **should** be able to notify users when event happen in the system
NF-08	Scalable	The system **should** be horizontally scalable

B UserService endpoints

Method	Path	Description
GET	/health	Used by the **DiagnosticsService**, when performing a system check
POST	/user	Logs in a user. Will generate a new token from the AuthenticationService and return it and the user if login is successful.
POST	/user/login	Creates a new user in the system. Will generate and return a new token and user if successful.
*	/protected/*	Every endpoint efter this are protected.
PUT	/user/:userId	Updates a user with the given userId.
DELETE	/user/:userId	Delete a user with the given userId.
GET	/user/?userName&amount	Finds a specifed amount of users by searching for their userName. Will only return the public user information.
GET	/invite	Finds all the invitations for the user with the given userId from the request header.
POST	/invite	Creates a new invitation.
POST	/invite/response	Accepts/Declines an invitation.
DELETE	/invite	Deletes the invitation with the given senderId, receiverId and organisationId.
GET	/organisation	Finds all the organisations for the user with the given userId from the header.
GET	/organisation/:organisationId	Finds an organisation with the given organisationId.
POST	/organisation	Creates a new organisation and adds the user as owner of the organisation.

PUT	/organisation/:organisationId	Updates the organisation with the given organisationId.
DELETE	/organisation/:organisationId	Deletes the organisation with the given organisationId.
GET	/project/organisation/:organisationId	Finds all the projects for the organisation with the given organisationId.
GET	/project/:projectId	Finds a project with the given projectId.
POST	/project	Creates a new project and adds the user as owner of the project.
PUT	/project/:projectId	Updates the project with the given projectId.
DELETE	/project/:projectId	Deletes the project with the given projectId.
GET	/microservice/project/:projectId	Finds all microservices for the project with the given projectId.
GET	/microservice/:microserviceId	Finds a microservice with the given microserviceId.
POST	/microservice	Creates a new microservice and adds the user as owner of the project.
PUT	/microservice/:microserviceId	Updates the microservice with the given microserviceId.
DELETE	/microservice/:microserviceId	Deletes the microservice with the given microserviceId.

C Website Design

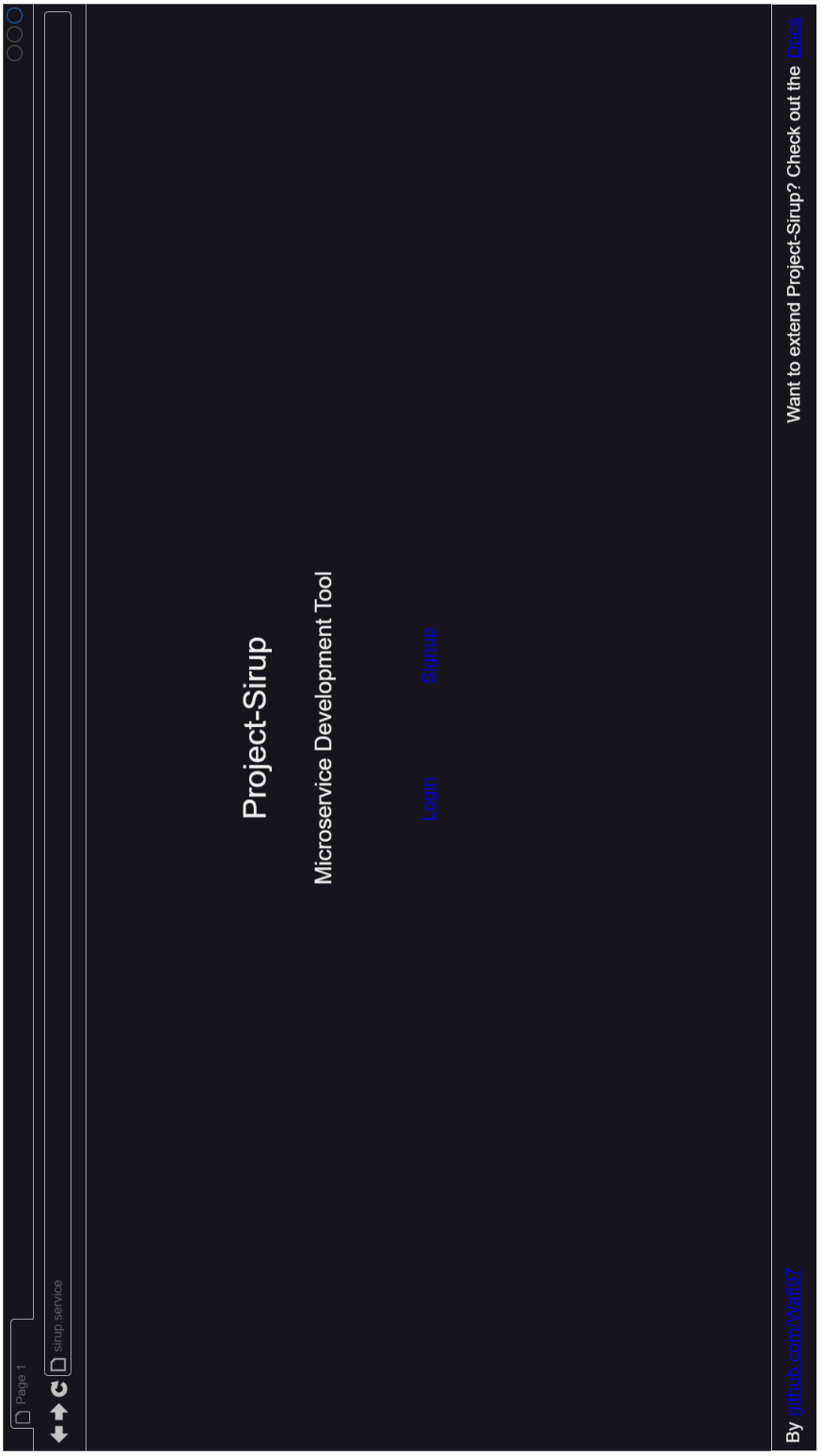

Figure C.1: Website Frontpage

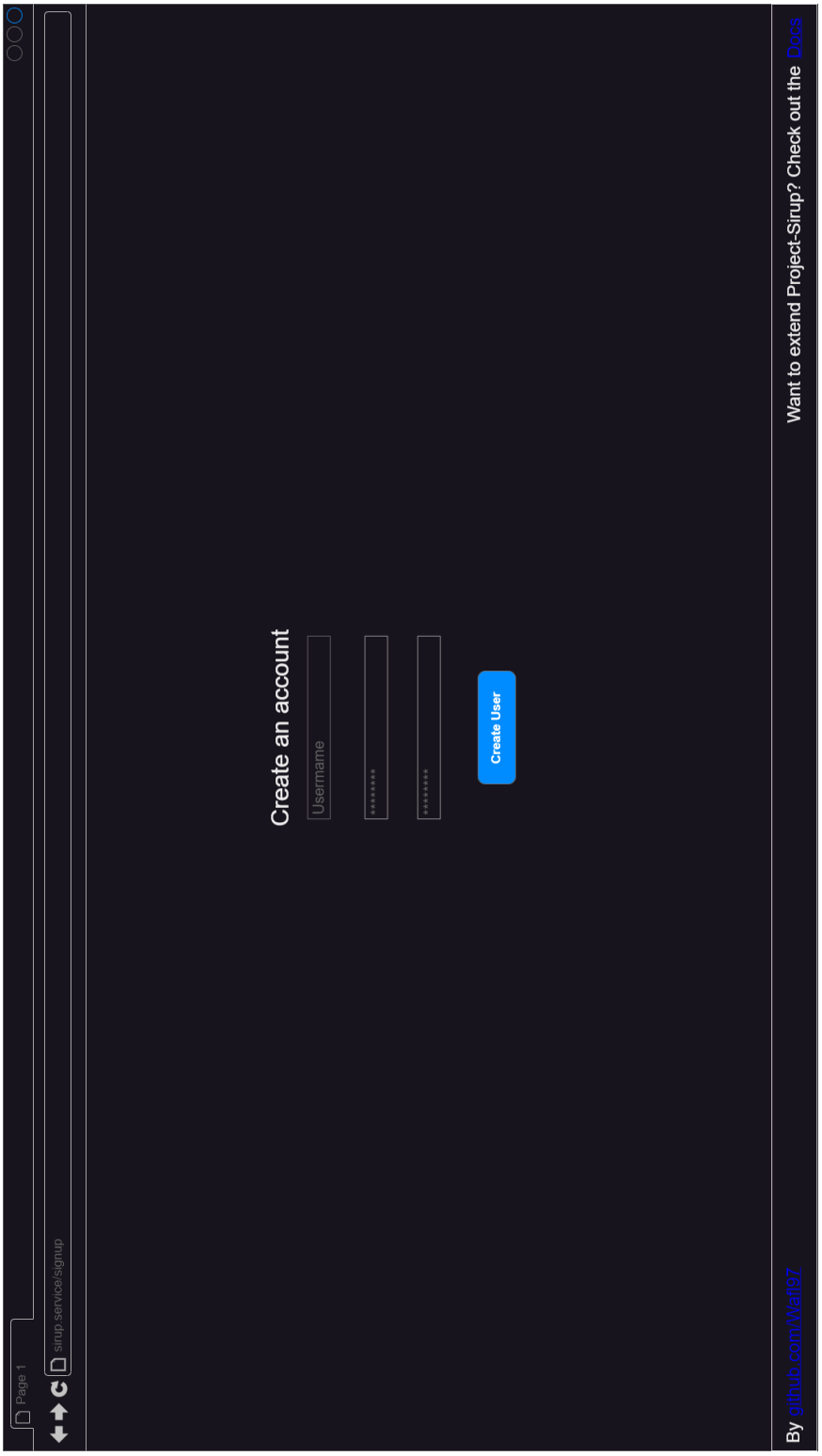

Figure C.2: Website Login

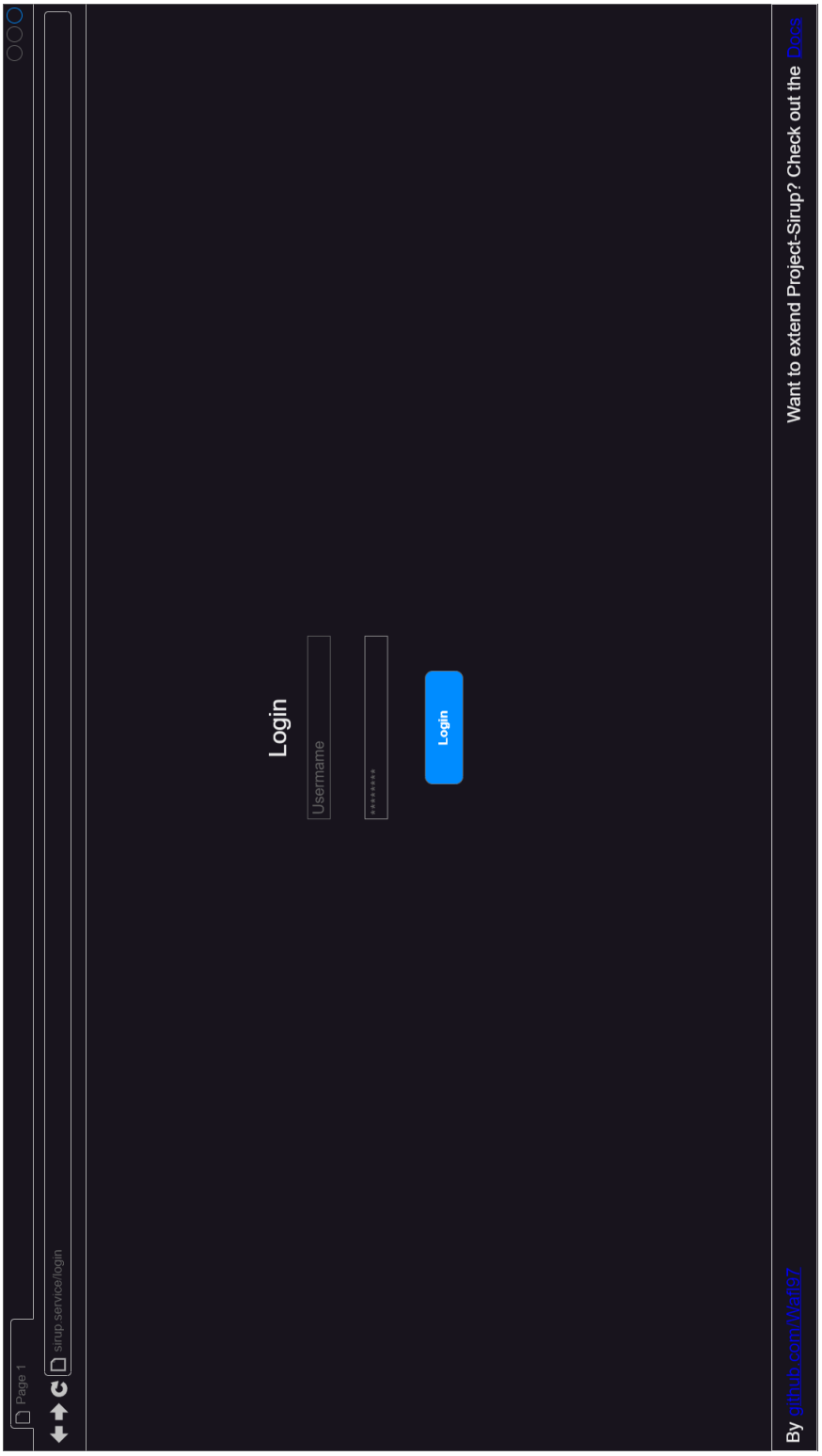

Figure C.3: Website Signup

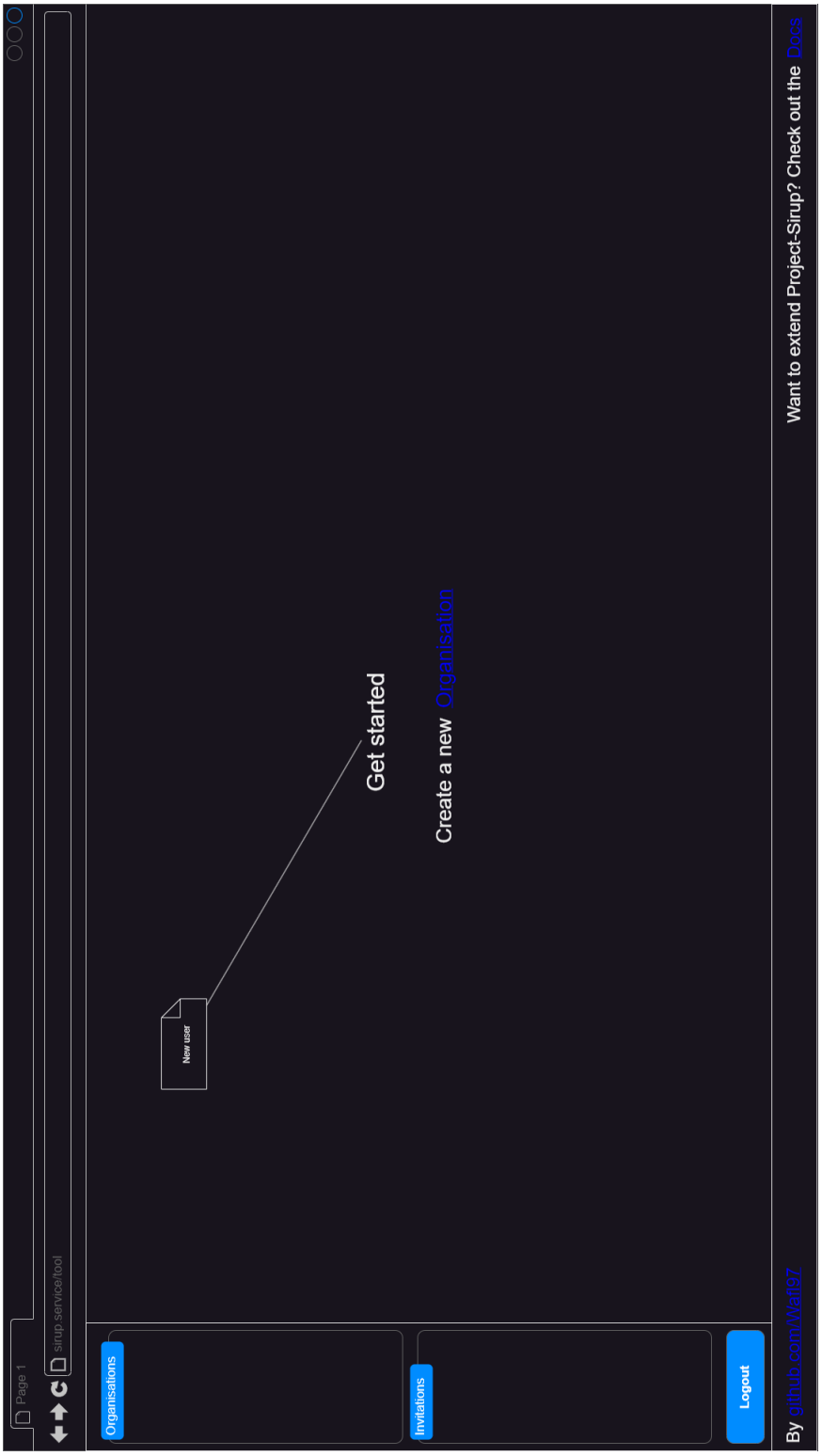

Figure C.4: Website New User Homepage

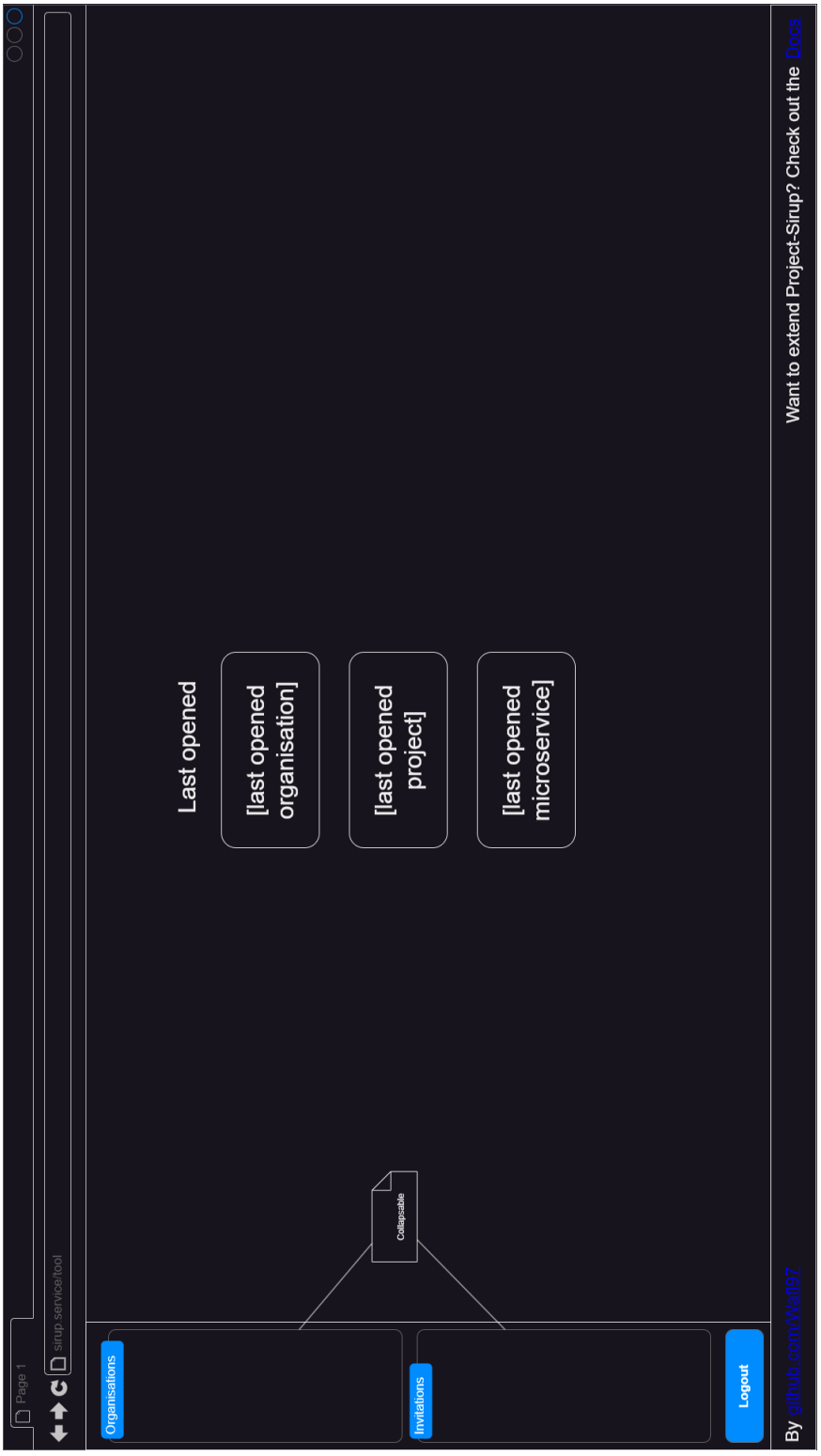

Figure C.5: Website User Homepage

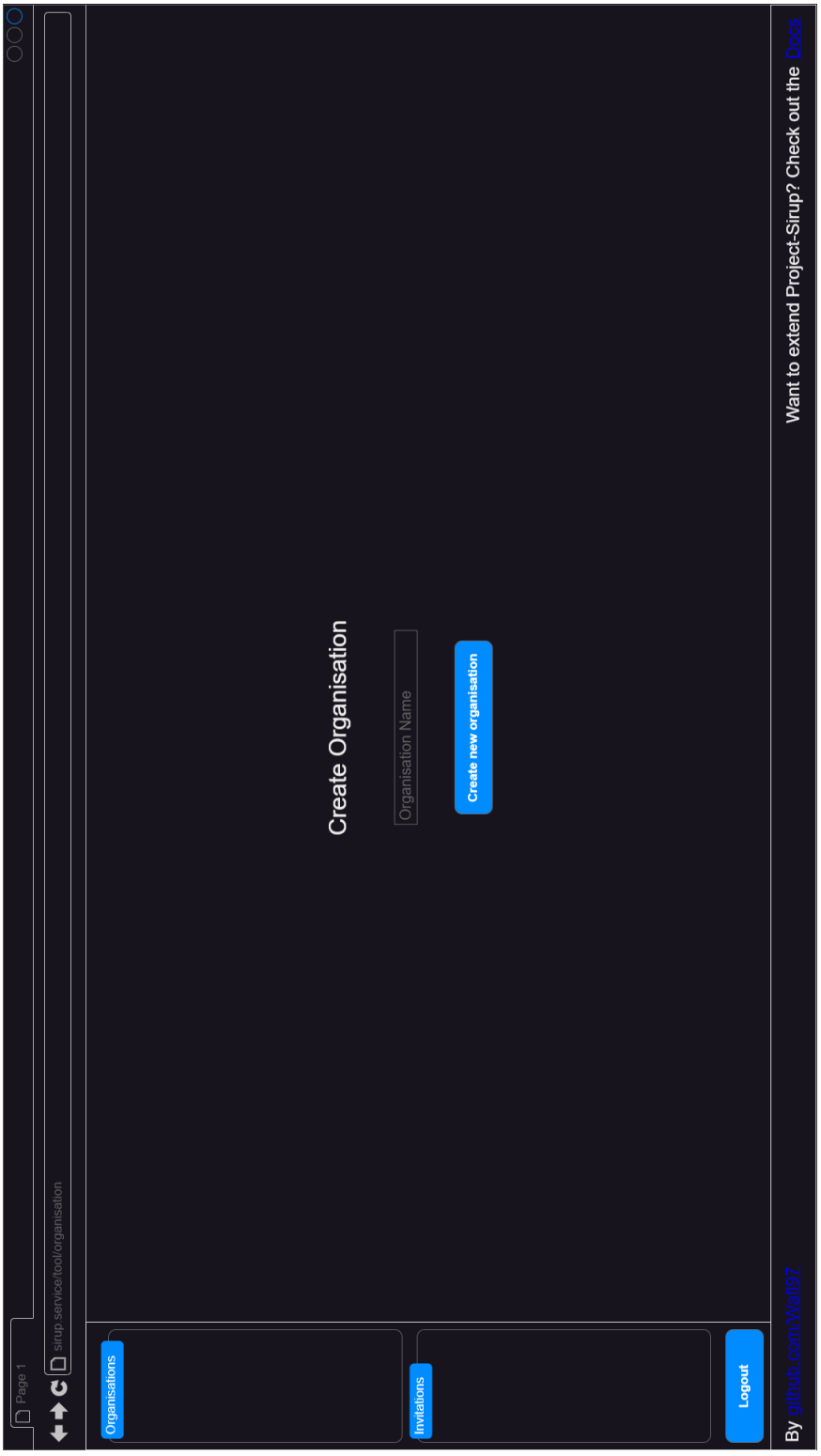

Figure C.6: Website Organisation Creation Form

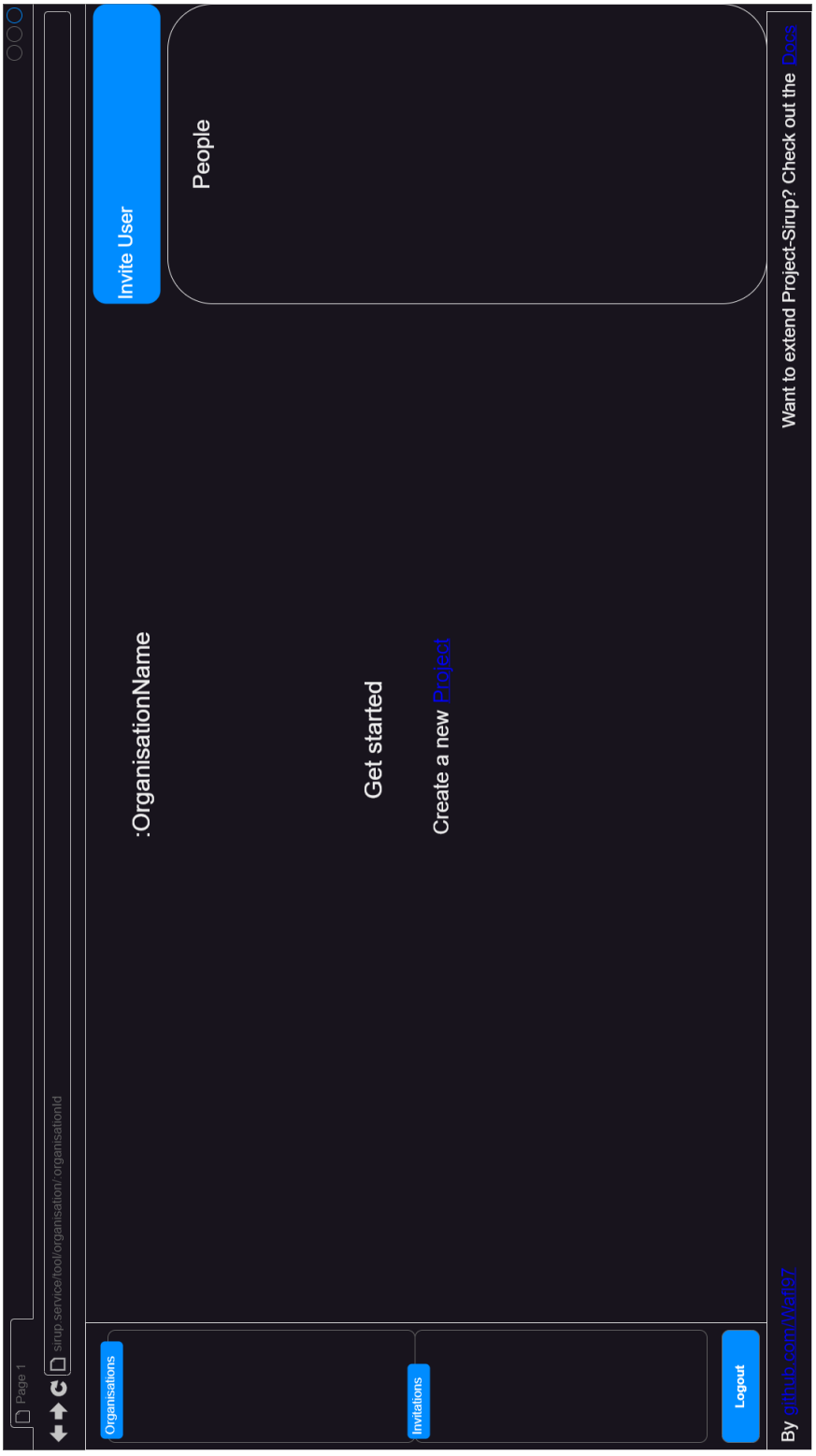

Figure C.7: Website New Oragnisation View

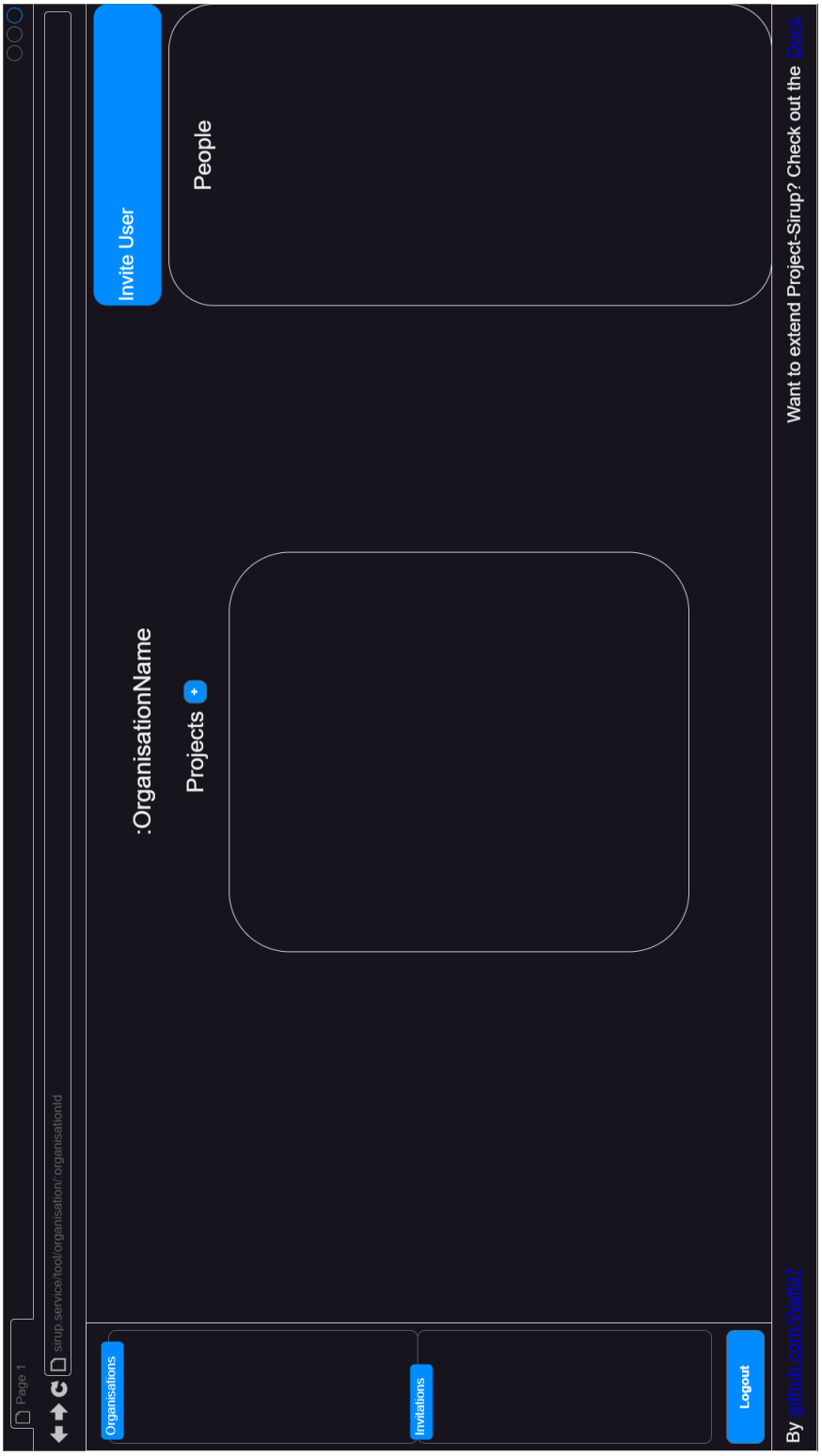

Figure C.8: Website Oragnisation View

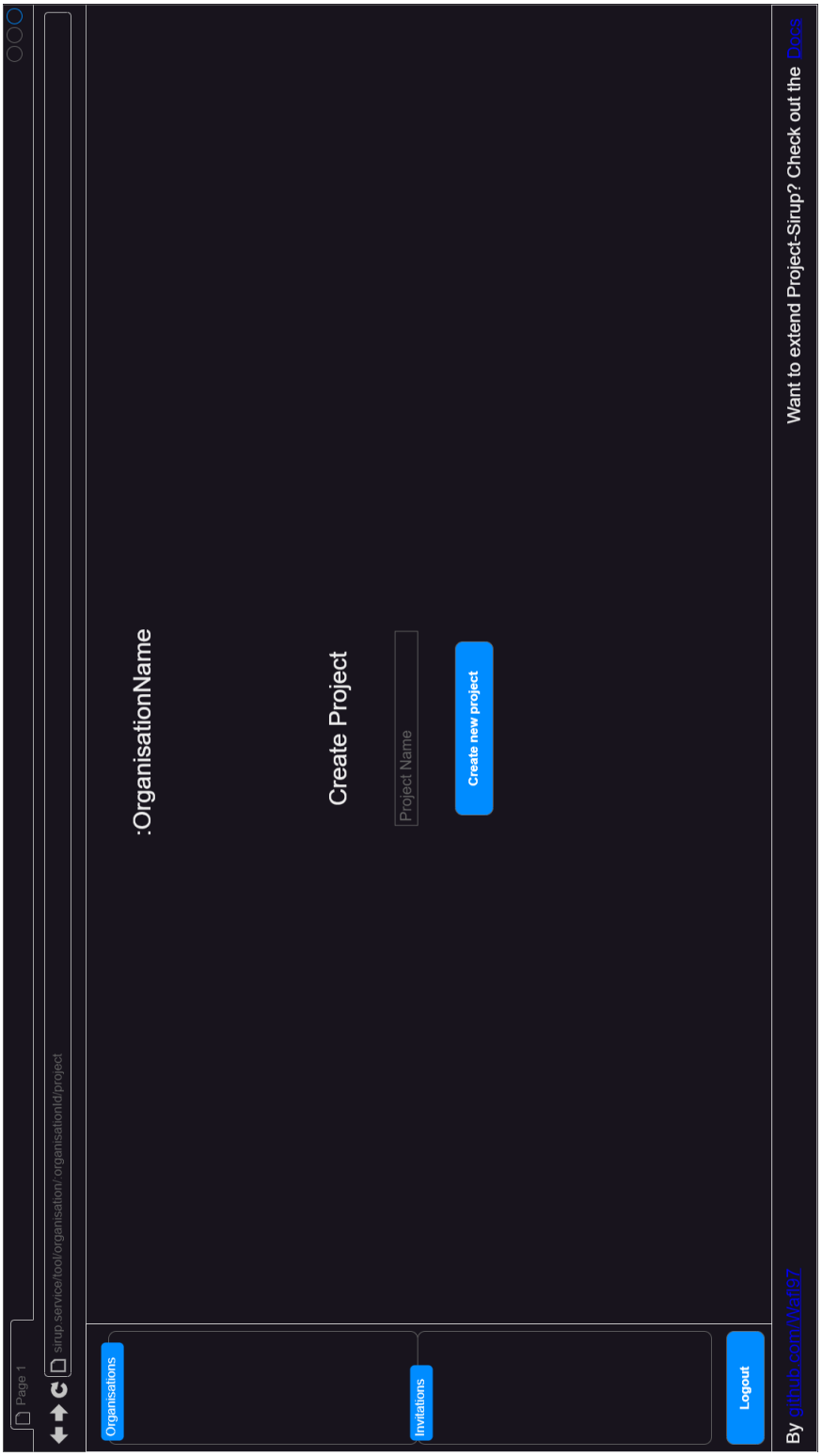

Figure C.9: Website Project Creation Form

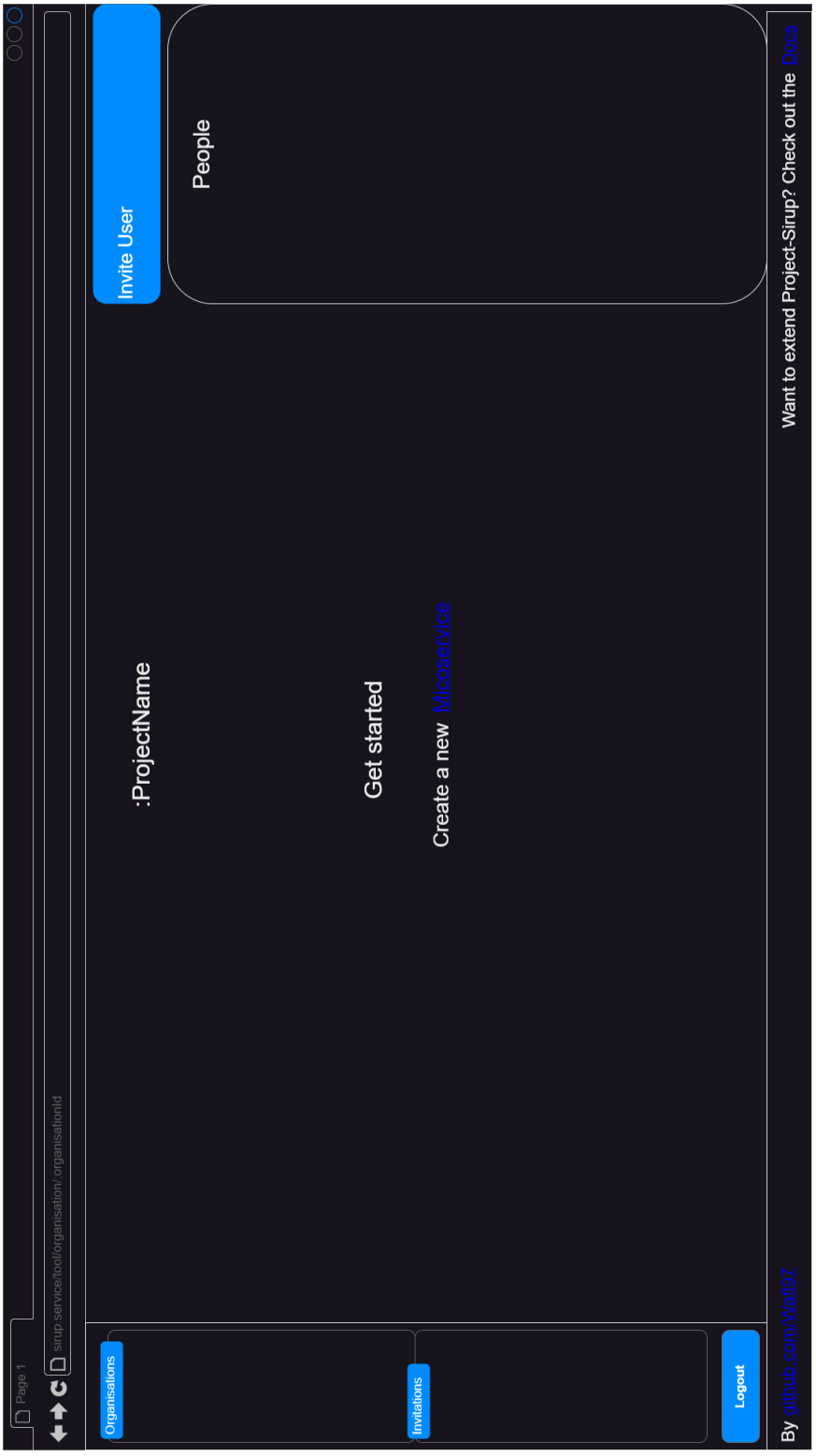

Figure C.10: Website New Project View

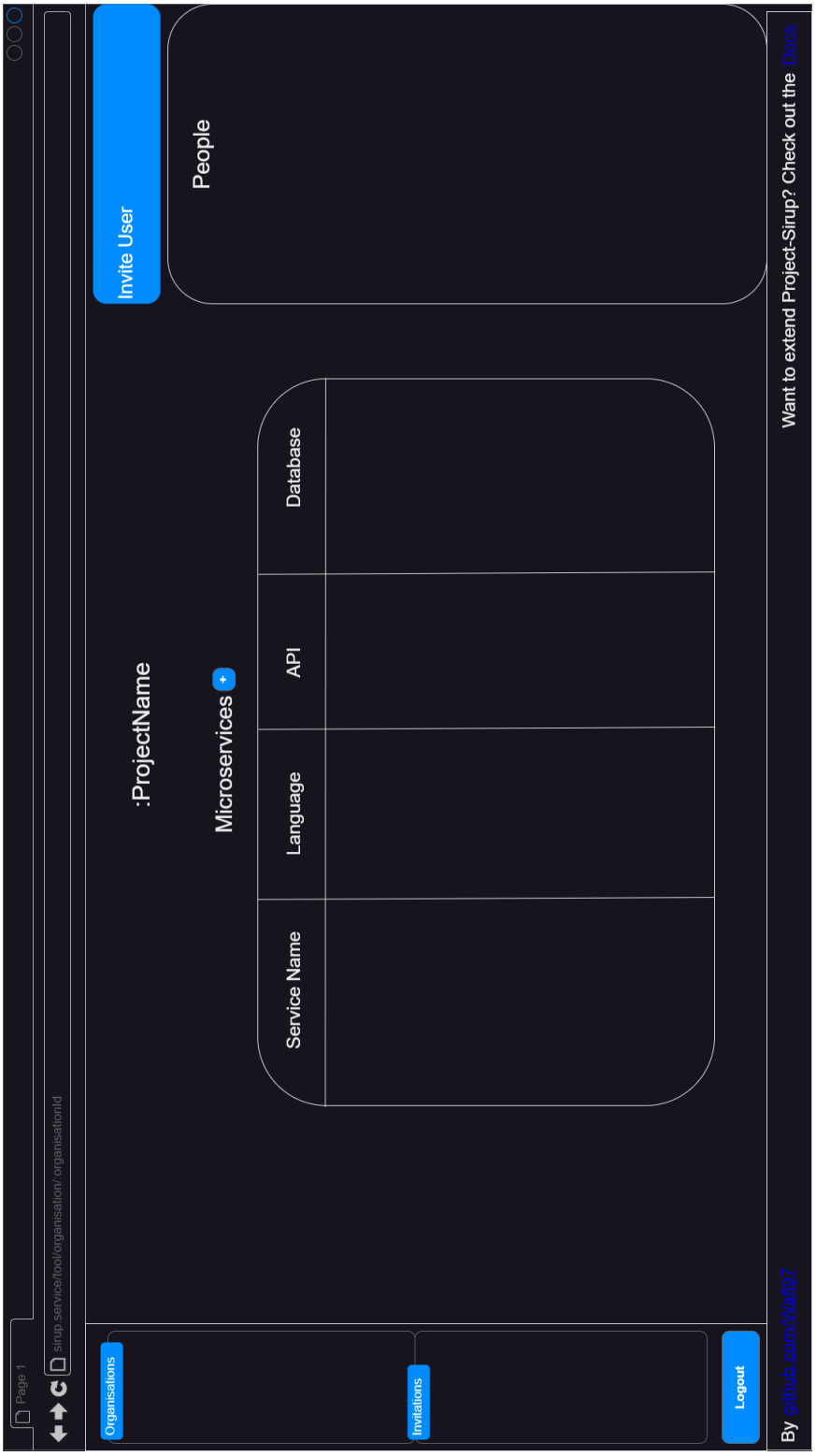

Figure C.11: Website Project View

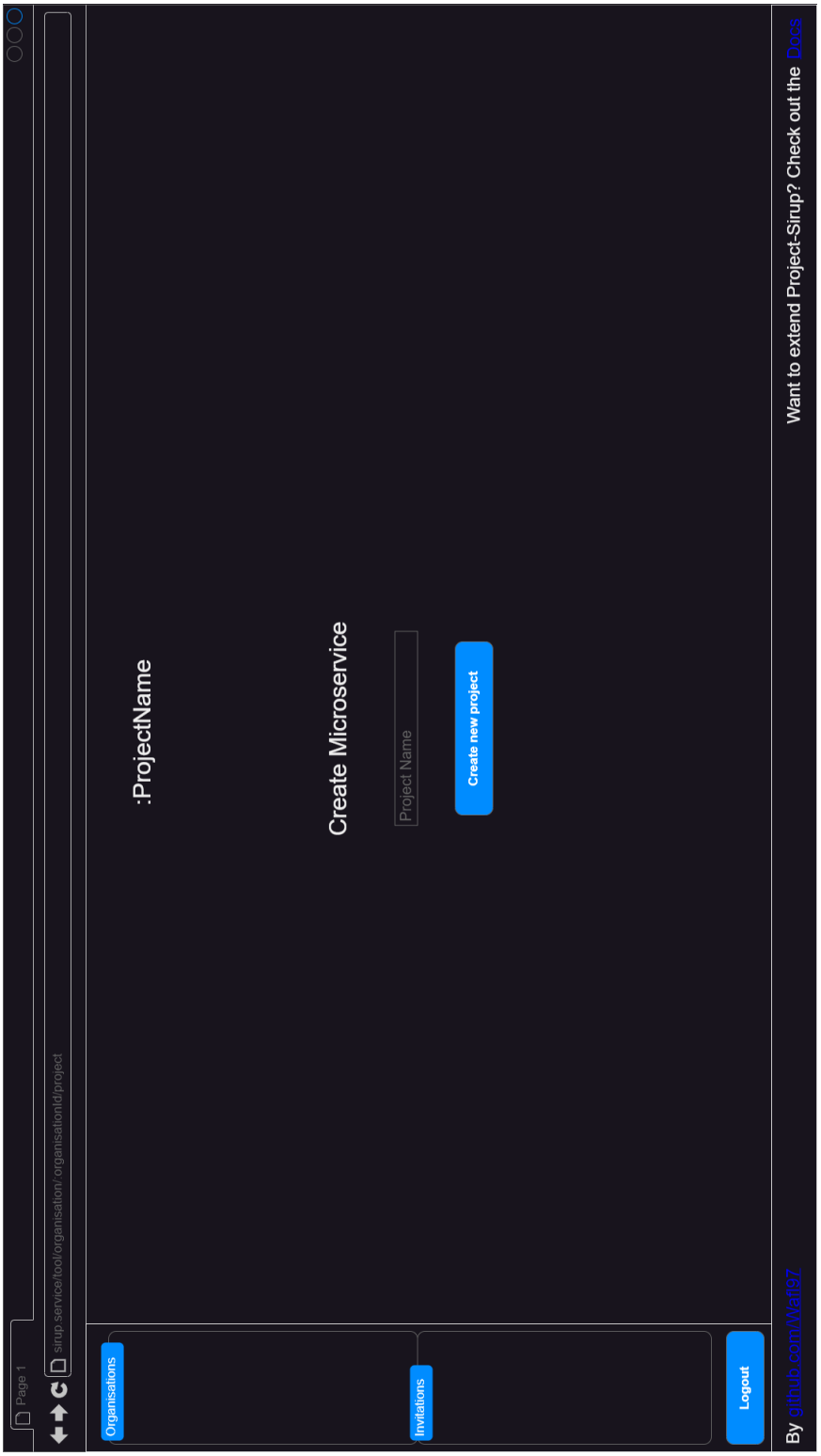

Figure C.12: Website Microservice Creation Form

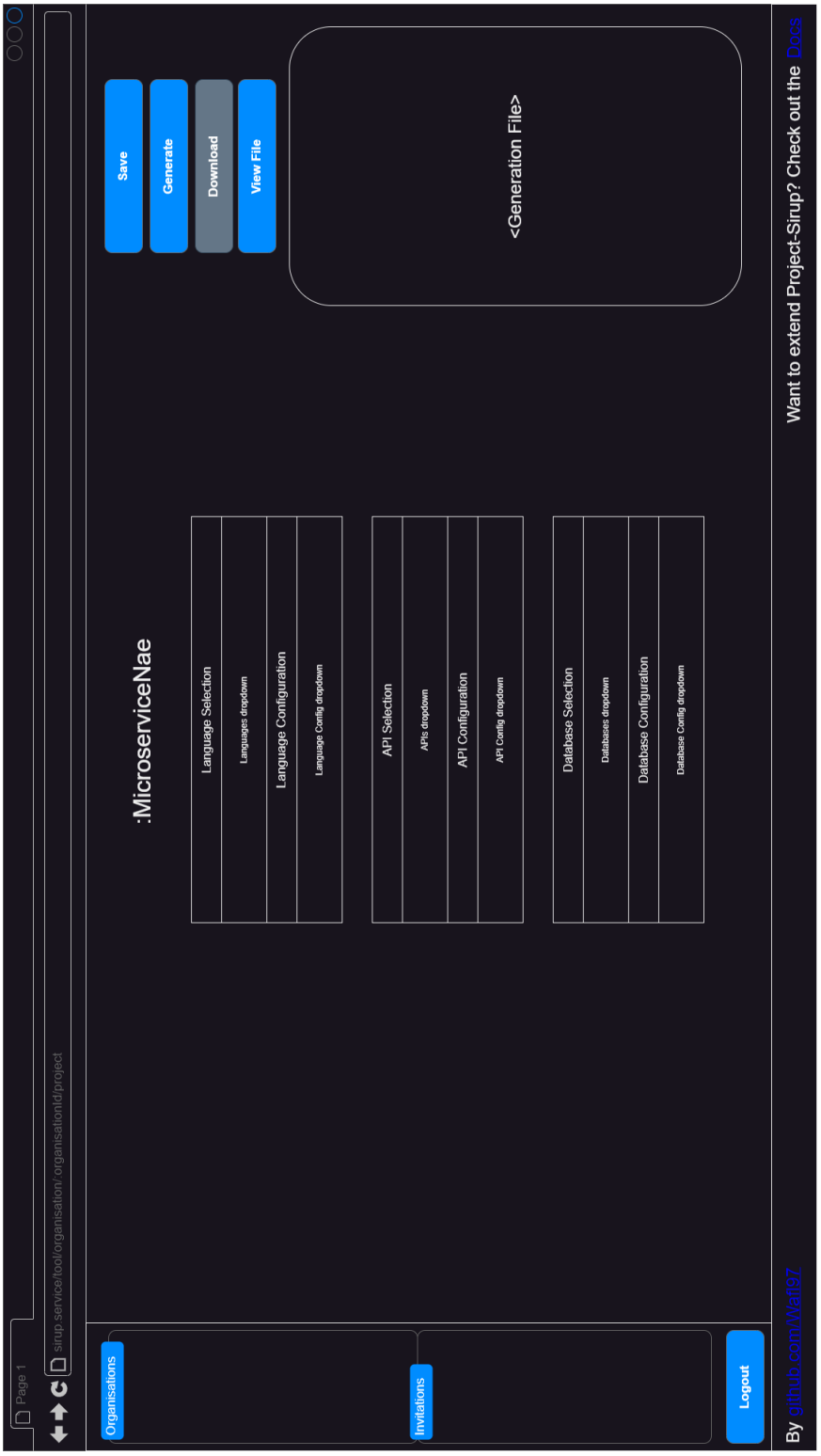

Figure C.13: Website Microservice Designpage

D JavaGenerationService manifest.json

Listing D.1: Java Generator Service manifest

```
{
  "sirup_v": 1,
  "acceptedFormats": [
    "json"
  ],
  "templates": [],
  "languages": [
    {
      "language": "Java",
      "description": "Generate a microservice written in Java 17",
      "options": [
        {
          "buildTool": {
            "selection": [
              "Maven",
              "Gradle"
            ]
          }
        },
        {
          "groupId": {
            "value": "string"
          }
        }
      ]
    }
  ],
  "apiTypes": [
    {
      "apiType": "REST",
      "description": "Generates a microservice with a REST API, using Apache Spark",
      "options": [
```

```
        {
          "port": {
            "value": "number"
          }
        },
        {
          "endpoints": {
            "repeated": {
              "object": {
                "name": "endpoint",
                "values": [
                  {
                    "name": "method",
                    "selection": [
                      "GET",
                      "POST",
                      "PUT",
                      "DELETE"
                    ]
                  },
                  {
                    "name": "path",
                    "value": "string"
                  },
                  {
                    "name": "linkedMethod",
                    "value": "string"
                  }
                ]
              }
            }
          }
        },
        {
          "endpointGroups": {
            "repeated": {
              "object": {
                "name": "endpointGroup",
                "values": [
                  {
                    "name": "groupName",
                    "value": "string"
                  },
                  {
```

```
                    "name": "endpoints",
                    "repeated": {
                      "object": {
                        "name": "endpoint",
                        "values": [
                          {
                            "name": "method",
                            "selection": [
                              "GET",
                              "POST",
                              "PUT",
                              "DELETE"
                            ]
                          },
                          {
                            "name": "path",
                            "value": "string"
                          },
                          {
                            "name": "linkedMethod",
                            "value": "string"
                          }
                        ]
                      }
                    }
                  },
                  {
                    "name": "linkedData",
                    "value": "string"
                  }
                ]
              }
            }
          }
        }
      ]
    }
  ],
  "databases": [
    {
      "database": "PostgreSQL",
      "description": "Generates the microservice with a
 PostgreSQL connection and classes for accessing data
 in the database",
```

```
      "supportedTypes": [
        "string",
        "int32",
        "int64",
        "float",
        "boolean"
      ],
      "options": []
    }
  ]
}
```

Listing D.1: Java Generator Service manifest

E JavaGenerationService MircoserviceRequest class

```
1  public record MicroserviceRequest(int sirup_v, boolean docker,
       Microservice microservice) {
2      public record Microservice(String microserviceId, String
       microserviceName, Language language, Database database, Api api,
        External external) {
3          public record Language(String name, Options options) {
4              public record Options(String buildTool, String groupId)
       {}
5          }
6          public record Database(String name, Options options, Data
       data) {
7              public record Options() {}
8              public record Data(List<Collection> collections) {
9                  public record Collection(String name, List<Field>
       fields) {
10                     public record Field(String name, String type,
       String ref) {}
11                 }
12             }
13         }
14         public record Api(String type, Options options) {
15             public record Options(int port, List<EndpointGroup>
       endpointGroups, List<Endpoint> endpoints, List<Procedure>
       procedures, List<Message> messages) {
16                 public record Endpoint(String method, String path,
       String linkedMethod) {}
17                 public record EndpointGroup(String groupName, List<
       EndpointGroup> innerGroups, List<Endpoint> endpoints, String
       linkedData) {}
18                 public record Procedure(String procedureName, String
        requestMessage, String responseMessage) {}
19                 public record Message(String messageName, List<Field
       > fields) {
20                     public record Field(String fieldName, String
       fieldType) {}
21                 }
22             }
23         }
24         public record External(String name) {}
25     }
26 }
```

Listing E.1: MicroserviceRequest class

F Test manifest.json

Listing F.1: Test manifest

```
{
    "languages": [
        {
            "language": "TypeScript",
            "description": "Generates a microservice written in TypeScript with Node.js as runtime",
            "options": [
                {
                    "test1": {
                        "value": "string"
                    }
                },
                {
                    "test2": {
                        "value": "number"
                    }
                },
                {
                    "test3": {
                        "selection": ["a","b","c"]
                    }
                },
                {
                    "test4": {
                        "object": {
                            "name": "test4Object",
                            "values": [
                                {
                                    "name": "test4Value1",
                                    "value": "string"
                                },
                                {
                                    "name": "test4Value2",
                                    "value": "number"
```

```
                                    },
                                    {
                                        "name": "test4Value3
",
                                        "selection": [1,2,3]
                                    }
                                ]
                            }
                        }
                    },
                    {
                        "test5": {
                            "repeated": {
                                "value": "string"
                            }
                        }
                    },
                    {
                        "test6": {
                            "repeated": {
                                "value": "number"
                            }
                        }
                    },
                    {
                        "test7": {
                            "repeated": {
                                "selection": ["a","b","c"]
                            }
                        }
                    },
                    {
                        "test8": {
                            "repeated": {
                                "object": {
                                    "name": "test8Object",
                                    "values": [
                                        {
                                            "name": "test8
Value1",
                                            "value": "string
"
                                        },
                                        {
```

```
                                        "name": "test8Value2",
                                        "value": "number"
                                    },
                                    {
                                        "name": "test8Value3",
                                        "selection": [1,2,3]
                                    }
                                ]
                            }
                        }
                    }
                ]
            }
        ],
        "apiTypes": [
            {
                "apiType": "REST",
                "description": "Generates a REST API using express.js",
                "options": [
                    {
                        "test9": {
                            "value": "string"
                        }
                    },
                    {
                        "test10": {
                            "value": "number"
                        }
                    },
                    {
                        "test11": {
                            "selection": ["a","b","c"]
                        }
                    },
                    {
                        "test12": {
                            "object": {
                                "name": "test12Object",
```

```
                                "values": [
                                    {
                                        "name": "test12Value
    1",
                                        "value": "string"
                                    },
                                    {
                                        "name": "test12Value
    2",
                                        "value": "number"
                                    },
                                    {
                                        "name": "test12Value
    3",
                                        "selection": [1,2,3]
                                    }
                                ]
                            }
                        }
                    },
                    {
                        "test13": {
                            "repeated": {
                                "value": "string"
                            }
                        }
                    },
                    {
                        "test14": {
                            "repeated": {
                                "value": "number"
                            }
                        }
                    },
                    {
                        "test15": {
                            "repeated": {
                                "selection": ["a","b","c"]
                            }
                        }
                    },
                    {
                        "test16": {
                            "repeated": {
```

```
                                "object": {
                                    "name": "test16Object",
                                    "values": [
                                        {
                                            "name": "test16
Value1",
                                            "value": "string
"
                                        },
                                        {
                                            "name": "test16
Value2",
                                            "value": "number
"
                                        },
                                        {
                                            "name": "test16
Value3",
                                            "selection": [1,
2,3]
                                        }
                                    ]
                                }
                            }
                        }
                    ]
                }
    ],
    "databases": [
        {
            "database": "MongoDB",
            "description": "Generates the microservice
with a MongoDB connection",
            "options": [
                {
                    "test17": {
                        "value": "string"
                    }
                },
                {
                    "test18": {
                        "value": "number"
                    }
```

```
192                 },
193                 {
194                     "test19": {
195                         "selection": ["a","b","c"]
196                     }
197                 },
198                 {
199                     "test20": {
200                         "object": {
201                             "name": "test20Object",
202                             "values": [
203                                 {
204                                     "name": "test20Value
    1",
205                                     "value": "string"
206                                 },
207                                 {
208                                     "name": "test20Value
    2",
209                                     "value": "number"
210                                 },
211                                 {
212                                     "name": "test20Value
    3",
213                                     "selection": [1,2,3]
214                                 }
215                             ]
216                         }
217                     }
218                 },
219                 {
220                     "test21": {
221                         "repeated": {
222                             "value": "string"
223                         }
224                     }
225                 },
226                 {
227                     "test22": {
228                         "repeated": {
229                             "value": "number"
230                         }
231                     }
232                 },
```

```
                {
                    "test23": {
                        "repeated": {
                            "selection": ["a","b","c"]
                        }
                    }
                },
                {
                    "test24": {
                        "repeated": {
                            "object": {
                                "name": "test24Object",
                                "values": [
                                    {
                                        "name": "test24Value1",
                                        "value": "string"
                                    },
                                    {
                                        "name": "test24Value2",
                                        "value": "number"
                                    },
                                    {
                                        "name": "test24Value3",
                                        "selection": [1,2,3]
                                    }
                                ]
                            }
                        }
                    }
                ]
            }
        ]
}
```