TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

David Zingerman 179198IABB

WEB APPLICATION DEVELOPMENT TO SOLVE PROBLEMS OF DOCUMENTING MICROSERVICE ARCHITECTURE

Bachelor’s thesis

Supervisor: Inna Švartsman
MSc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

David Zingerman

VEEBIRAKENDUSTE VÄLJATÖÖTAMINE MIKROTEENUSTE ARHITEKTUURI DOKUMENTEERIMISE PROBLEEMIDE LAHENDAMISEKS

Bakalaureusetöö

Juhendaja: Inna Švartsman
MSc

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: David Zingerman

01.03.2021

Abstract

Web application development to solve problems of documenting microservice architecture

The aim of the thesis is to solve problems related to microservice documentation in a web application. The problems must be solved as a web application that gives the ability to manage microservice related information.

In this thesis it is described what methodology is in use and what techniques were used to validate the system. The result is a prototype-level system that is initially validated and is ready for further development

This thesis is written in English and is 35 pages long, including 4 chapters, 13 figures.

Annotatsioon

Veebirakenduste väljatöötamine mikroteenuste arhitektuuri dokumenteerimise probleemide lahendamiseks

Lõputöö eesmärk on lahendada mikroteenuste dokumenteerimise probleemid. Probleemid peavad olema lahendatud veebirakendusena kus saab mikroteenuste informatsiooni hallata.

Selles lõputöös kirjeldatakse, millist metoodikat kasutatakse ja milliseid tehnikaid kasutati süsteemi kinnitamiseks.
Tulemuseks on prototüübi taseme süsteem, mis on esialgu valideeritud ja on valmis edasiseks arenguks.
Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 4 peatükki, 13 joonist, 0 tabelit.

List of abbreviations and terms

JSON	JavaScript Object Notation
ITIL	Information Technology Infrastructure Library
ITSM	Information Technology Service Management
MVP	Minimum Viable Product
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
API	Application Programming Interface
UI	User Interface
JWT	JSON Web Token

Table of contents

List of Figures

1 Introduction

This thesis is solving the problem of microservices documentation. To solve it it's vital to understand the underlying problems and drawbacks of the status quo.

1.1 The Problem

In the current situation the microservice based architecture has a good momentum and is spread widely across the industry. The tooling is mostly internal or if it is a public tool then is usually outdated. The needs of such companies and teams is pretty specific and only a limited set of tools is capable of solving these problems. If you look at ITIL or ITSM methodologies for example, you will quickly understand that these methods cannot survive without proper tooling. But from the research it comes out that few such tools exist and are barely used.

1.2 The Goal

The goal is to create a web application dedicated to the needs of small to medium businesses to keep track of their microservices in a meaningful and comfortable way.

In order to solve the problem, it is required to:

1. Analyze the biggest problems with the existing tooling,
2. Pick the methodology to develop the product in a reasonable timeframe,
3. Choose the technology,
4. Develop the minimal viable product (MVP),
5. Test the result on the code and on the user levels.

The result is the MVP / private beta stage product that is solving the validated problems.

1.3 Work structure

In this thesis will be explained how the analysis of required features was done. The topic of user interface design and the topic of the methodologies that were used in the backend development will be also covered.

2 Methodologies

In this chapter it's described how the work was done. What tools were used to solve the problems? What practices did come in handy?

2.1 User Interface Design and Feature Analysis

The first step in the application development is to do the analysis of the requirements and lay it out. The lay out structure must fit some criteria.

It is important that the interface is:

- understandable,
- comfortable to use,
- solving the required problems,
- the workflows are several clicks long,
- the user is never lost in the interface and always knows what he is doing.

The central entity of a system that allows management the microservices is the microservice itself.

The ITIL describes a number of processes that are related here if we map the software to a microservice and model it a bit different. What is a service in our scope?

2.1.1 ITIL Perspective

A service is a microservice or a group of microservices. Let's see what processes apply here. From the list of processes, we see that the following processes apply [1]:

- Service support and monitoring
- Incident management

- Problem management
- Change management
- Release management
- Configuration management

Let’s see how the problems are solvable at the moment:

For the service support and monitor the services there is a market of tools. One example is Datadog [2]. They also recently added incident management tool [3] that covers another process of our list.

Problem and change management are also solved by a list of companies. For example, the Atlassian company [4].

Release and configuration management are also solved by a number of major companies.

The analysis shows that all of the processes are solved from the ITSM perspective.

2.1.2 Microservice perspective

Let’s now change our perspective. If a service is a microservice or a group of microservices then how would the processes map?

Service support process will be more complicated as there are now more services with more people involved and more parts to monitor. Incident and problem management must be split more granularly to be separated to a specific microservice as well as all the other processes.

So, to apply the ITIL processes to a microservice as an application, we need to focus on the layer between a service and a microservice. Each process has additional bit of information that needs to be present.

At this point it is required to transcribe the processes into workflows and map them to the user interface or our application.

There are a few major parts of a microservice based on the Netflix model as an example [5].

1. The service that it represents,
2. The people working on it,
3. Information in other systems about it,
4. Notes,
5. Keywords or tags

There is no point in isolation the application from the vast variety of the solutions that exist and are in use today. The best approach is to take advantage and become a part of the ecosystem.

2.2 Tools

For the interface design the tool is Figma, for the backend development the language of choice is Go due to its simplicity, suitability and performance.

Figma is a vector graphics editor and prototyping tool which is primarily web-based, with additional offline features enabled by desktop applications for macOS and Windows. The Figma Mirror companion apps for Android and iOS allow viewing Figma prototypes in real-time on mobile devices.

Go is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing, and CSP-style concurrency.

For the database technology it is chosen MariaDB as it’s easy to start with and provides all the necessary tooling to work with databases.

MariaDB is a community-developed, commercially supported fork of the MySQL relational database management system, intended to remain free and open-source software under the GNU General Public License.

2.3 Work process

The process consists of several parts. The first part is the analysis. The next is the user interface design and after that comes the development.

After analysing the ITIL processes, it was started to lay out the design taking into account the importance of comfort, easy workflows and other priorities described in the beginning of this chapter.

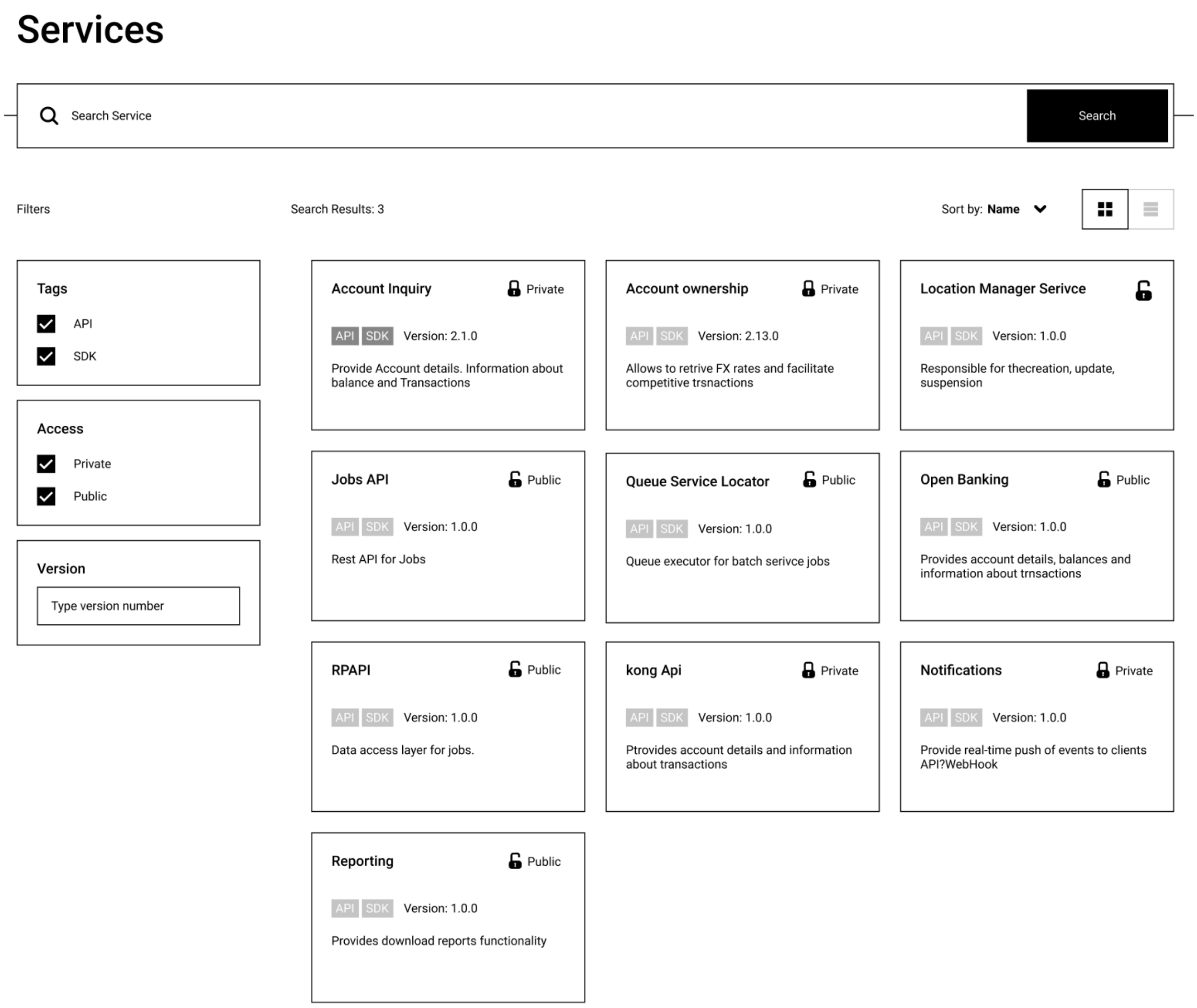
The ITIL analysis is described in the previous chapter.

2.3.1 List of microservices layout

The microservices view was laid out keeping in mind what the user might need to see when interacting with it. When the user looks at the list of microservices there are several attributes of the microservice he needs to see.

- Current or the latest stable version
- Name
- Description
- Type
- Tags

The list must be capable to do filtering and sorting by different attributes.


Services

Search Service | Search

Filters | Search Results: 3 | Sort by: **Name**

Tags
- ✓ API
- ✓ SDK

Access
- ✓ Private
- ✓ Public

Version
Type version number

Service	Access	Tags	Version	Description
Account Inquiry	Private	API SDK	Version: 2.1.0	Provide Account details. Information about balance and Transactions
Account ownership	Private	API SDK	Version: 2.13.0	Allows to retrive FX rates and facilitate competitive trsnactions
Location Manager Serivce		API SDK	Version: 1.0.0	Responsible for thecreation, update, suspension
Jobs API	Public	API SDK	Version: 1.0.0	Rest API for Jobs
Queue Service Locator	Public	API SDK	Version: 1.0.0	Queue executor for batch serivce jobs
Open Banking	Public	API SDK	Version: 1.0.0	Provides account details, balances and information about trnsactions
RPAPI	Public	API SDK	Version: 1.0.0	Data access layer for jobs.
kong Api	Private	API SDK	Version: 1.0.0	Ptrovides account details and information about transactions
Notifications	Private	API SDK	Version: 1.0.0	Provide real-time push of events to clients API?WebHook
Reporting	Public	API SDK	Version: 1.0.0	Provides download reports functionality

Figure 1. List of microservices view design

The Figure 1 shows how the services got laid out in the mock-up desing.

2.3.2 Microservice view layout

When you click on an item from the list, the user sees the classifiers that the service has. The user must be able to edit each classifier separately. The classifiers management should be basic as it is an MVP.

Here comes the requirement to Figure out what classifiers the microservice must have. To support the ecosystem of existing solutions it can have a list of links to other resources. In order to manage the versions and keep track of the releases it can have a releases management view. To manage the people dealing with a particular microservice it is required to add the Service Owners management view.

Microservices usually have communication with other microservices, therefore it might be handful to document the interactions between them. A Dependency management view could solve these problems.

As an additional option it could be possible to integrate a monitoring dashboard into the app, so adding a Dashboard management view.

I will start from the links view as it’s the most intuitive to sketch out.

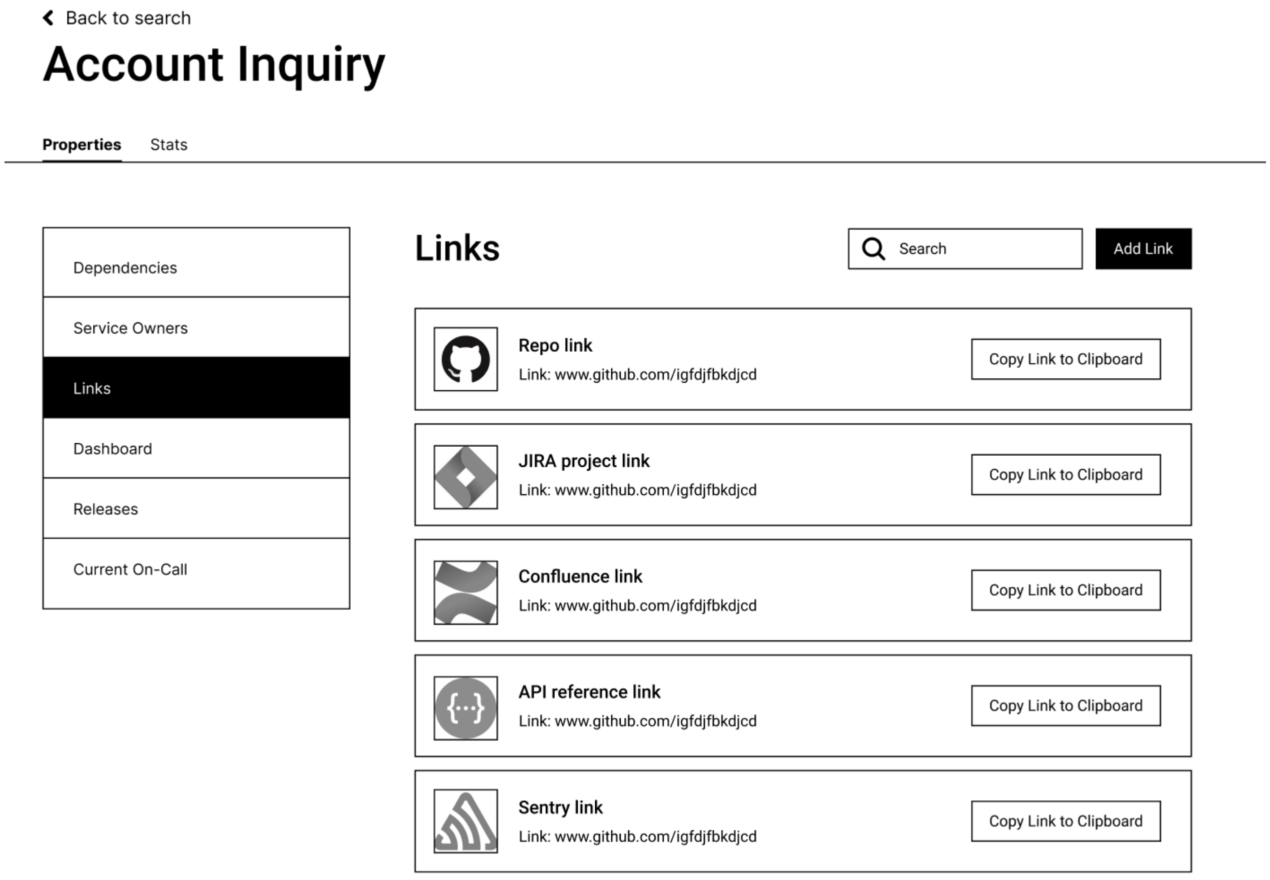


Figure 2. Links management view under the Microservice view

The Figure 2 shows how the links got laid out on the mock-up design.

3 The Process

3.1 Requirement analysis

It was required to build a comfortable MVP to manage the microservices. The primary requirements were taken from the ITIL processes mapping to the definition of a microservice. It was defined that the application needs to be able to manage a predefined set of entities, such as

1. Configuration files
2. Dependencies to other services
3. Links to other resources
4. Owners of the service
5. Release notes
6. Tags for easier categorization
7. Types for easier categorization
8. Services must be grouped for easier categorization

3.2 Architecture

The architecture consists of the backend and frontend applications.

The frontend is what a user sees and interacts with (user interface). The back end is part of the application that is hidden from the user. What some would call, under the hood. This part is responsible for data processing, storing the data, and mathematical operations.

Figure 3 shows the whole application structure (web application part):

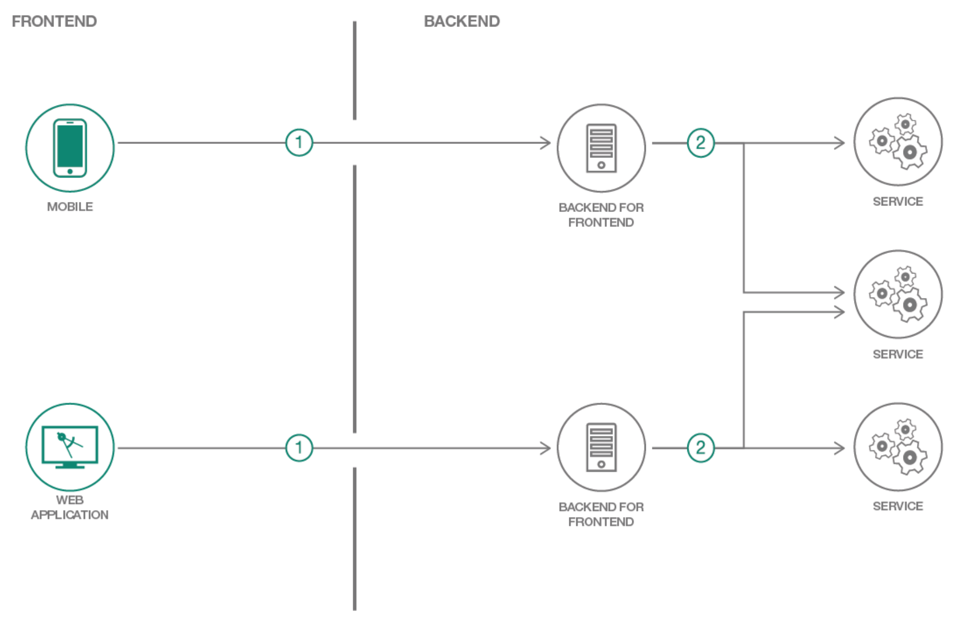
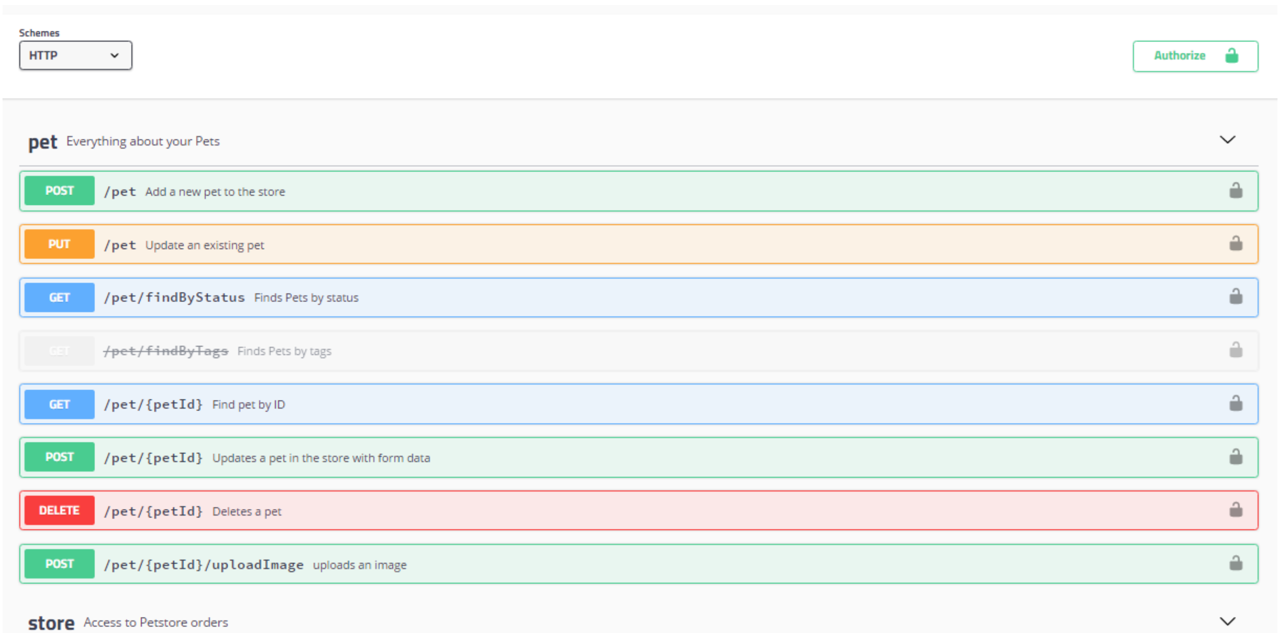


Figure 3. Backend for Frontend Application Architecture

I will explain the backend structure as this is the part where the author is involved into. The communication between the frontend and the backend is done via REST API.

To enable easy endpoint discovery and integration it was decided to use Swagger, example in Figure 4, to document the API automatically. There are libraries that can generate the documentation from the comments inside the code.


Schemes
HTTP
Authorize

pet Everything about your Pets

POST /pet Add a new pet to the store

PUT /pet Update an existing pet

GET /pet/findByStatus Finds Pets by status

GET ~~/pet/findByTags~~ Finds Pets by tags

GET /pet/{petId} Find pet by ID

POST /pet/{petId} Updates a pet in the store with form data

DELETE /pet/{petId} Deletes a pet

POST /pet/{petId}/uploadImage uploads an image

store Access to Petstore orders

Figure 4. Swagger UI

The language for the backend is Go. It is a service-oriented application from the inside. [6]

The dependencies are different from the example. To implement the HTTP service logic, it was decided to use another framework from the example, as well as for the tests. [7]

The package helps matching the requests based on the URL host, path, path prefix, schemes, header and query values, HTTP methods or custom matchers. It allows to reverse the URLs which helps maintaining references to resources. It helps using routers as subrouters to group the routes based on the path or the host.

3.3 Detailed design

The backend application has the controller, service, repository structure

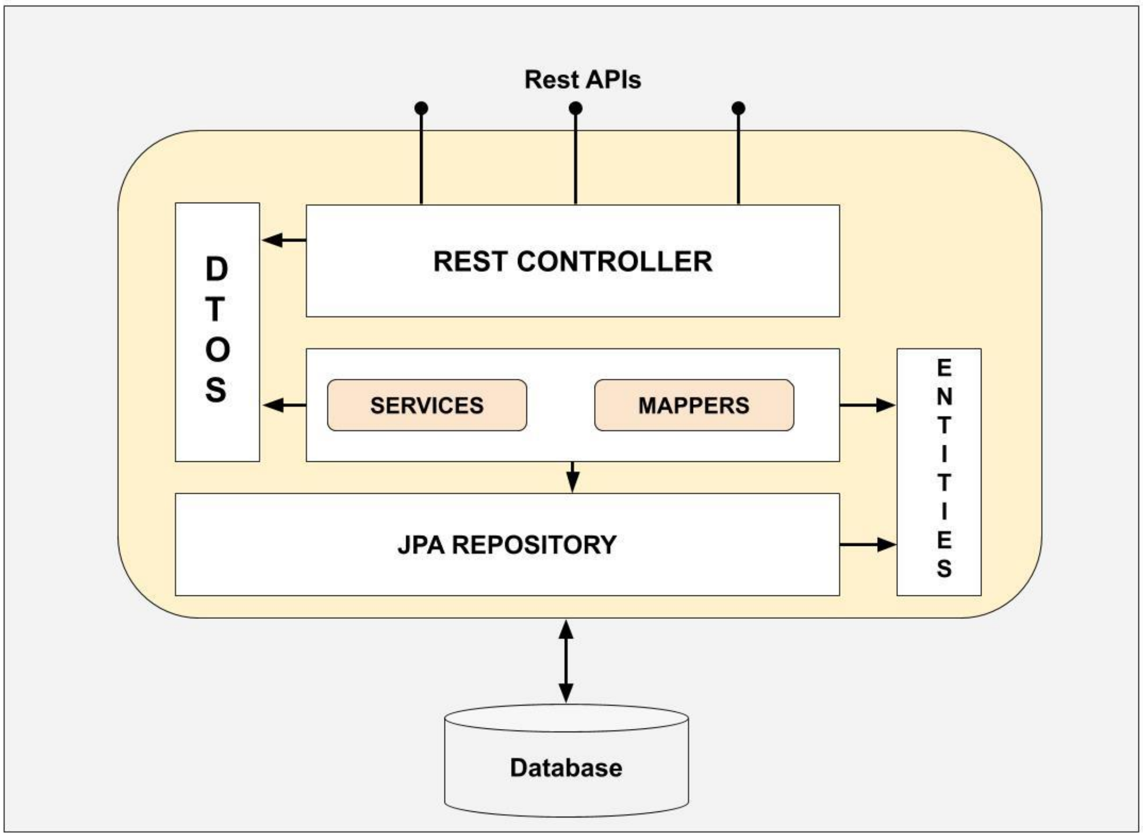


Figure 5. Controller, service, repository pattern

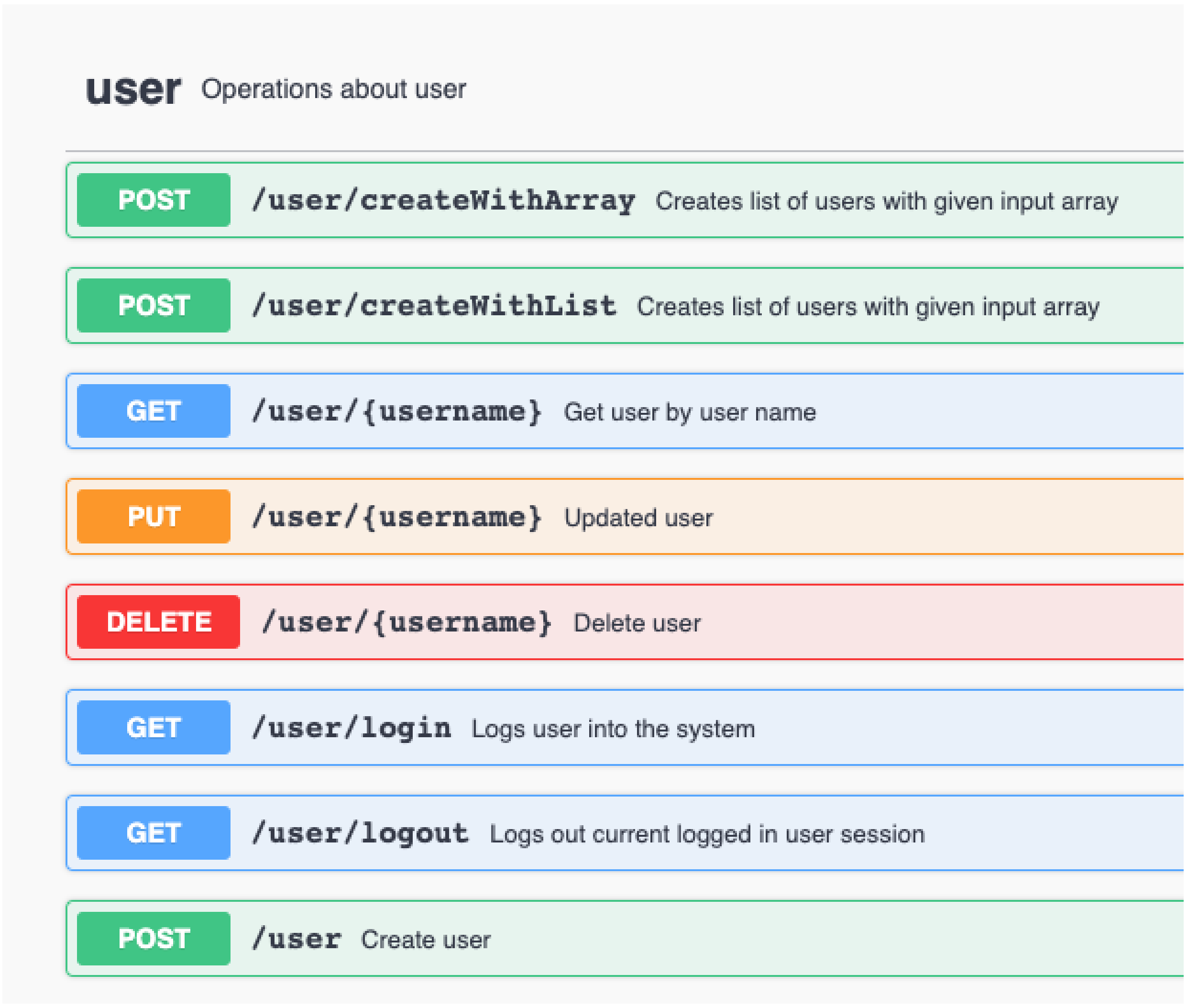
In the Figure 5 it’s shown how the components interact with each other. In my application there are services, REST controllers, repositories and the database.

Most of the entities have a set of management calls, a controller that handles the requests and uses the entity service to perform the operations.

The services use the repositories or external services to fulfil the request.

The API has a list of error codes to easily troubleshoot and identify the error. The entities are the Restful Objects from the API perspective. [8]

The management calls include GET, PATCH, POST and DELETE HTTP methods. The GET method is used to get the object summary and the property values or links to them. The PATCH request enables updating the objects. The PATCH is much better than the PUT because it enables partial updates which are very handy from the frontend side. The DELETE call can delete one or more entities. The POST call creates an entity.



user Operations about user

POST /user/createWithArray Creates list of users with given input array

POST /user/createWithList Creates list of users with given input array

GET /user/{username} Get user by user name

PUT /user/{username} Updated user

DELETE /user/{username} Delete user

GET /user/login Logs user into the system

GET /user/logout Logs out current logged in user session

POST /user Create user

Figure 6. Sample set of management calls

The Figure 6 example illustrates how the set of calls looks from the Swagger User Interface. In this example though there are many calls. In the API for each classifier, it is preferred to have 4 calls. The methods are

1. Creating a single record,
2. Reading multiple or single record with the ability to sort and filter,
3. Updating (patrial updates supported),
4. Deleting (multiple records supported).

This set of calls is optimal for most of the cases.

3.3.1 Authentication

For the authentication it was decided to use the JWT. It is a compact self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a or a public/private key pair. [9]

When a user logs in or creates an account, the token is issued and can be used in the HTTP headers or cookies. The authentication lets the application know what user is accessing it and at the same time prevents unauthorized parties to access the information.

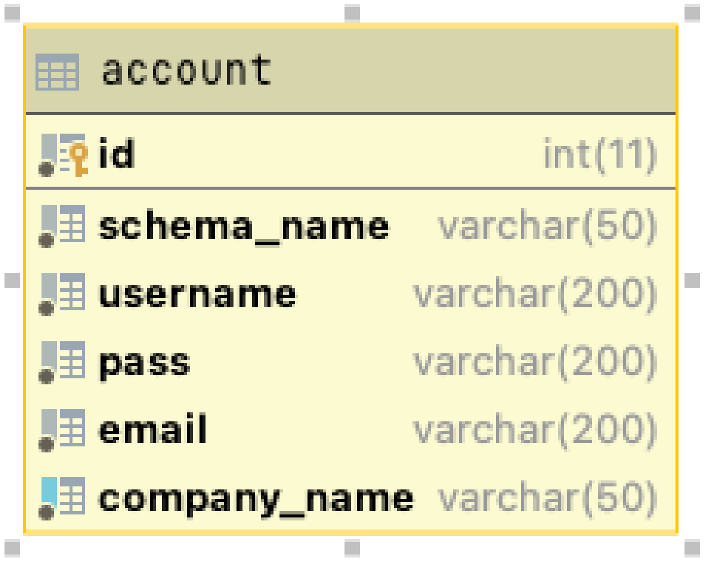
Putting the token in the cookies is more practical. That’s because storing a JWT in your frontend state will cause it to be lost any time the page is refreshed or closed and then opened again. This leads to a poor user experience—you don’t want your users to need to log back in every time they refresh the page.

Inside the application there is function that is called every time an endpoint is called that should be secured. The function then analyses the token and parses information of the caller’s identity and permissions.

The fields in the JWT token and called claims. Claims are simply pieces of information. It is very convenient to use the JWT tokens because the claims are stored in the JSON format which is easy to read and parse programmatically.

To enable the system of accounts a special schema naming or a separate storage is required. Another database was added to store the information about the accounts, their schema names and other necessary information. It can be useful when the information is required particularly at the step of identifying the user.

The database right now contains only one table to store the account records – Figure 7.



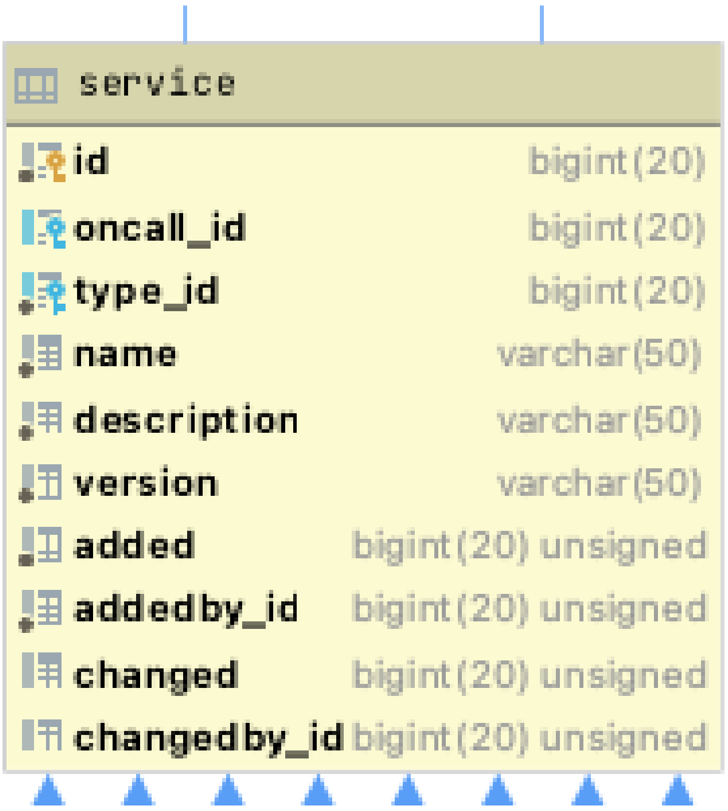
account	
id	int(11)
schema_name	varchar(50)
username	varchar(200)
pass	varchar(200)
email	varchar(200)
company_name	varchar(50)

Figure 7. Account table

3.4 Database design

For the database technology it was decided to use MariaDB. The database is account based, so each new account gets a new database created. This enables data security and integrity between accounts. For the field naming it was required to analyze and implement the common naming conventions. [10]

The central part is the service object. The majority of other tables are the service's dependencies.


service

id	bigint(20)
oncall_id	bigint(20)
type_id	bigint(20)
name	varchar(50)
description	varchar(50)
version	varchar(50)
added	bigint(20) unsigned
addedby_id	bigint(20) unsigned
changed	bigint(20) unsigned
changedby_id	bigint(20) unsigned

Figure 8. Service table

As you can see on the Figure 8, the service has dates when it was added or changed, the user's information that performed the action, name, description, and some classifier IDs.

Here are the tables of the classifiers and the relationships.

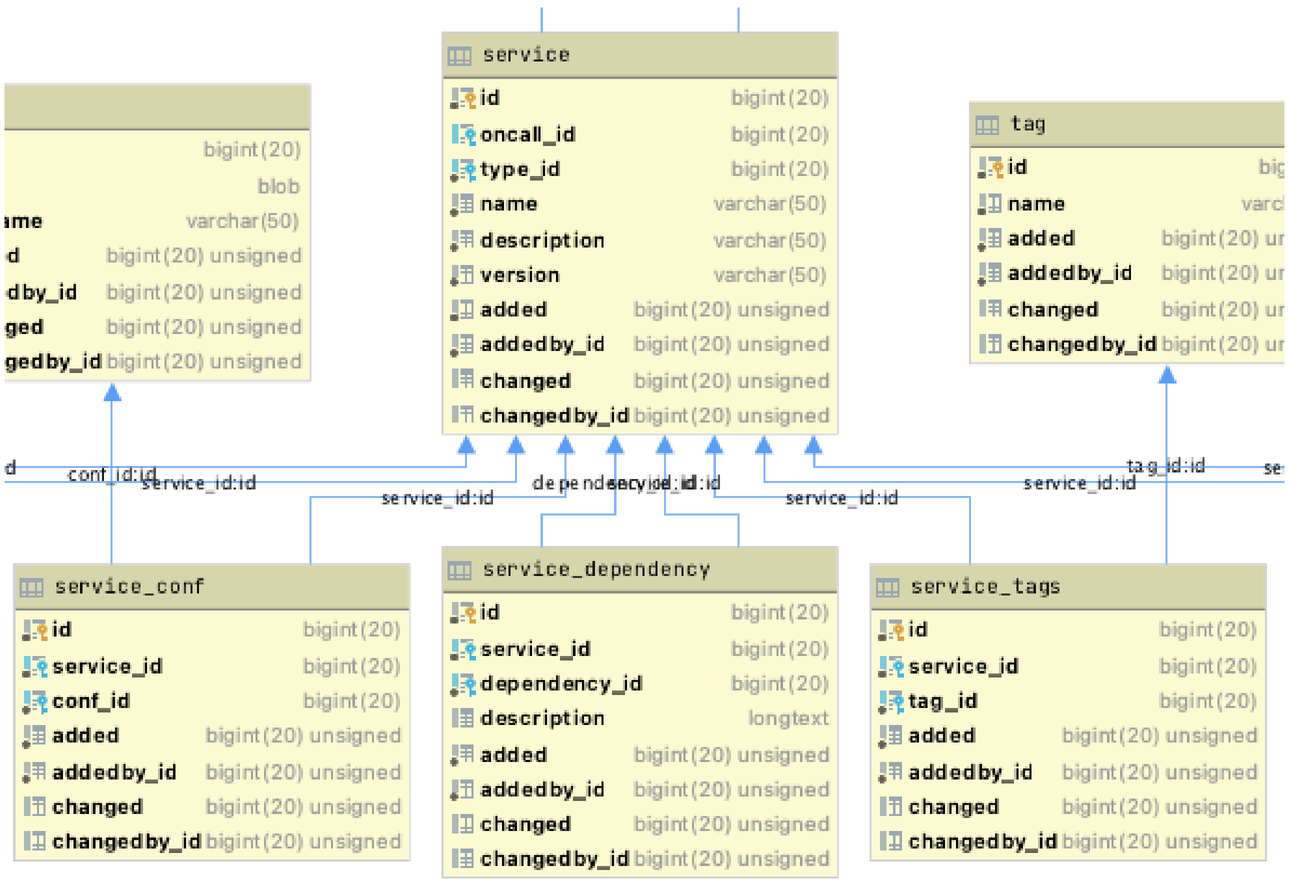

service
id bigint(20)
oncall_id bigint(20)
type_id bigint(20)
name varchar(50)
description varchar(50)
version varchar(50)
added bigint(20) unsigned
addedby_id bigint(20) unsigned
changed bigint(20) unsigned
changedby_id bigint(20) unsigned
tag
id
name
added
addedby_id
changed
changedby_id
service_conf
id bigint(20)
service_id bigint(20)
conf_id bigint(20)
added bigint(20) unsigned
addedby_id bigint(20) unsigned
changed bigint(20) unsigned
changedby_id bigint(20) unsigned
service_dependency
id bigint(20)
service_id bigint(20)
dependency_id bigint(20)
description longtext
added bigint(20) unsigned
addedby_id bigint(20) unsigned
changed bigint(20) unsigned
changedby_id bigint(20) unsigned
service_tags
id bigint(20)
service_id bigint(20)
tag_id bigint(20)
added bigint(20) unsigned
addedby_id bigint(20) unsigned
changed bigint(20) unsigned
changedby_id bigint(20) unsigned
conf_id:id
service_id:id
dependency_id:id
tag_id:id

Figure 9. Service classifier tables

On the Figure 9 you can see tables like “service_conf”, “service_tags” and “service_dependency” that are used to persist the classifiers of the services.

The “service_conf” table is used to store the links between the configuration files and the services. It is a many-to-many relationship. As you can see the link has added, changed dates as well as the user information that added or modified the link. On the API level the links are represented on the service Restful Object.

The “service_dependency” table is used to store the links between the service and its dependencies which are other services. . It is a many-to-many relationship. Effectively this table stores connections of records from the same table, but it’s still better from the design perspective to separate the concerns and use additional table. As you can see the link has added, changed dates as well as the user information that added or modified the link. On the API level the links are represented on the service Restful Object.

The “service_tags” table is used to store the links between the service and its tags which are stored in another table. It is a many-to-many relationship. As you can see the link has added, changed dates as well as the user information that added or modified the link. On the API level the links are represented on the service Restful Object.

The relationships are different. Some of them are one-to-many, some are many-to-many. In a one-to-many relationship (Figure 10), one record in a table can be associated with one or more records in another table. For example, each customer can have many sales orders. In a one-to-many relationship the object's id is attached to the record itself in the same table. For example, the "type_id" field. In another table the service type record exists and on the service object it's just an ID.

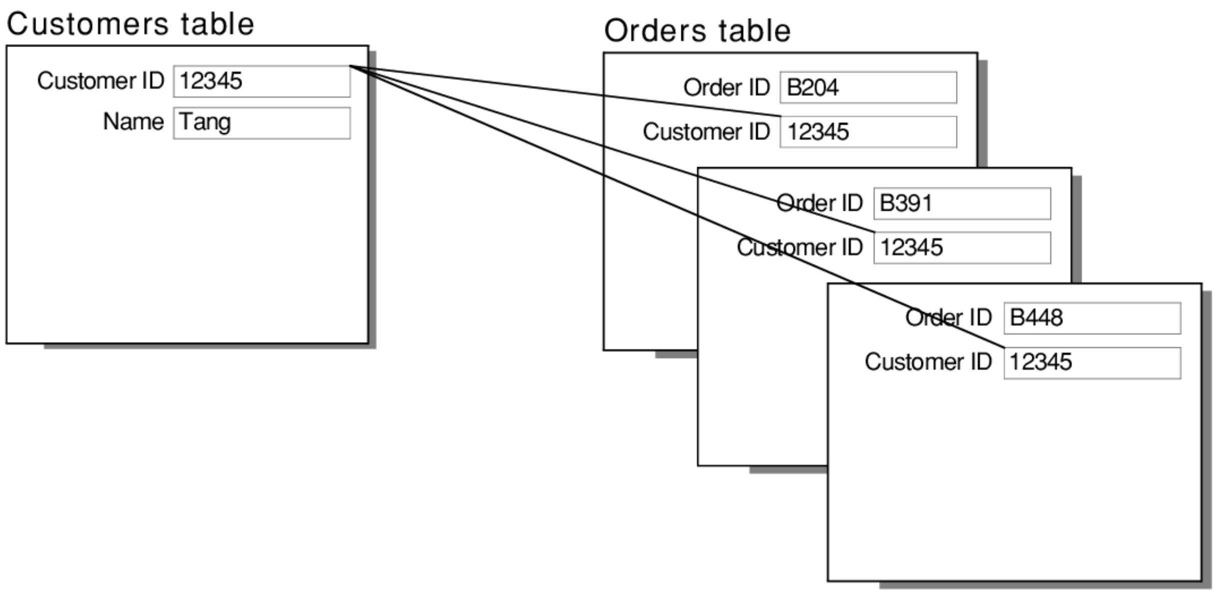


Figure 10. One-to-many relationships

In the many-to-many relationships (Figure 11), a separate table must exist to store the links between the records.

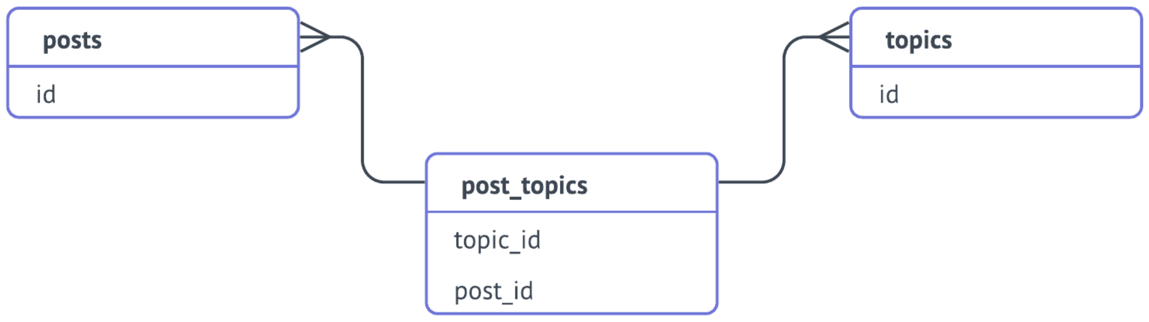


Figure 11. Many-to-many relationships

There are also tables not related to the services. The information related to the users, their permissions, user groups and other general information is required in every application. It was decided to leave the implementation outside of the MVP scope.

3.5 Testing

During the project development it was decided to use software testing techniques to validate the quality and use cases.

There are tools that allow automation of the process. For example, the static code analysis tools.

One type of such tools is linters. Go language has its own Go linter - a linter for Go source code. The linter is used to flag programming errors, bugs, stylistic errors and suspicious constructs. The term originates from a Unix utility that examined C language source code. The second type of such tools is formatters. Go language has its own Go formatter – a formatter for Go source code. The formatter is required because the language forces its own formatting style to ease the analysis of the code that others have written.

3.5.1 Unit testing

Unit testing is the type of testing where tested objects are units of components of a software piece. The testing is done during the phase of development of the application.

Go language provides a minimal framework for unit tests, but it does not provide packages that allow assertion. Assertion is a popular method of unit testing that is used in many other programing languages.
An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program. A test assertion is defined as an expression, which encapsulates some testable logic specified about a target under test.

There are several metrics that allow assessing if the testing is done right. Or done enough. The metrics include

- code coverage percentage

- Code metrics (Cyclomatic Complexity and Maintainability Index)

Maintainability Index calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. Color coded ratings can be used to quickly identify trouble spots in your code. A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 and indicates low maintainability.

Cyclomatic Complexity measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program. A program that has complex control flow requires more tests to achieve good code coverage and is less maintainable.

In the application it was decided to skip the code coverage metric as it can be wasteful to keep the codebase at a maximum coverage percentage. Instead, it was decided to cover the most important components with the unit tests.

The most important components of a service-oriented architecture are probably the services themselves.
The services are covered with unit tests which partially test the repository code as well.

3.5.2 Integration testing

Integration testing is a level of software testing where units or components are combined and tested as a group. The purpose of such testing is to expose faults in the interaction between the units. Mocking is used here to replicate the components.

Mocking means creating a fake version of an external or internal service that can stand in for the real one, helping your tests run more quickly and more reliably. When your implementation interacts with an object's properties, rather than its function or behavior, a mock can be used.

The metrics include number of errors that happen between layer during a given period of time. If there are a lot of errors that happen it means that the integration testing techniques are applied wrongly of not applied at all.

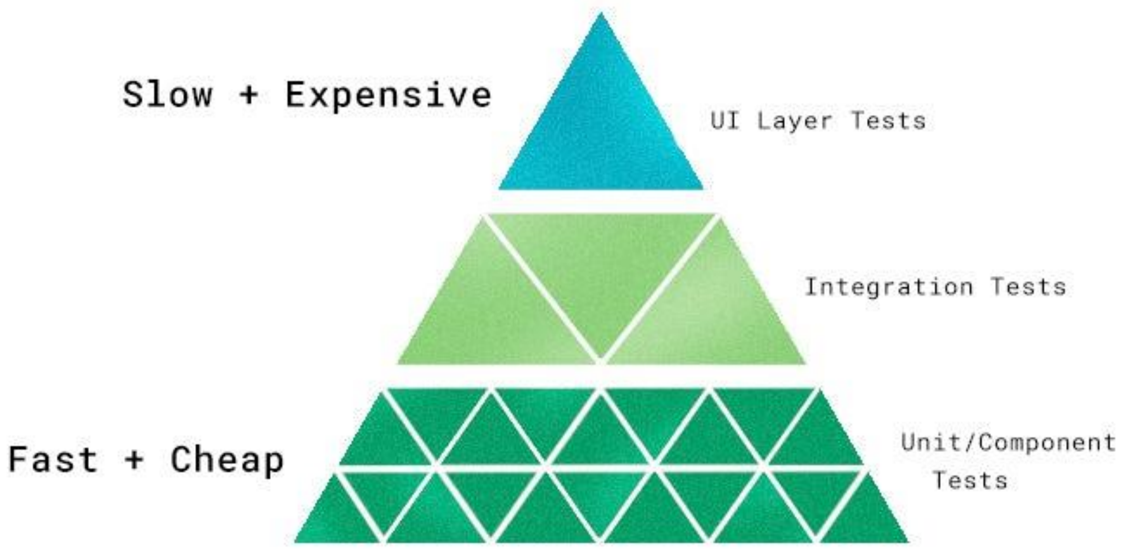


Figure 12. Software testing pyramid

In the tests that are used in the project, the mocks are not used as they require more time to be created and bring less value compared to the unit tests. It was decided to combine the unit tests and integration tests to get the most value with the minimal amount of time spent as shown in the Figure 12.

3.5.3 End-to-end testing

End-to-end testing is a technique that tests the entire software product from beginning to end to ensure the application flow behaves as expected. It defines the product's system dependencies and ensures all integrated pieces work together as expected.

In order to implement the end-to-end testing, it is required to do

- Study of an end to end testing requirements,
- Test Environment setup and hardware/software requirements
- Describe all the systems and its subsystems processes.
- Description of roles and responsibilities for all the systems
- Testing methodology and standards
- End to end requirements tracking and designing of test cases
- Input and output data for each system

While end-to-end testing is a very important methodology it was decided not to implement it at such an early stage as the MVP development phase. In the future it is planned to implement and use it.

3.6 Project management

Another key factor in the development process is the project management. There are frameworks and methodologies used to achieve goals and follow the deadlines that are set in the beginning.

One of such frameworks is Scrum.

Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems. Scrum co-creators Ken Schwaber and Jeff Sutherland have written The Scrum Guide to explain Scrum clearly and succinctly. This definition consists of Scrum’s accountabilities, events, artifacts, and the rules that bind them together.

In a nutshell, Scrum requires a Scrum Master to foster an environment where:

- A Product Owner orders the work for a complex problem into a Product Backlog.
- The Scrum Team turns a selection of the work into an Increment of value during a Sprint.
- The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint.
- Repeat

The flow of our work was somewhat similar to what is shown on the Figure 13.

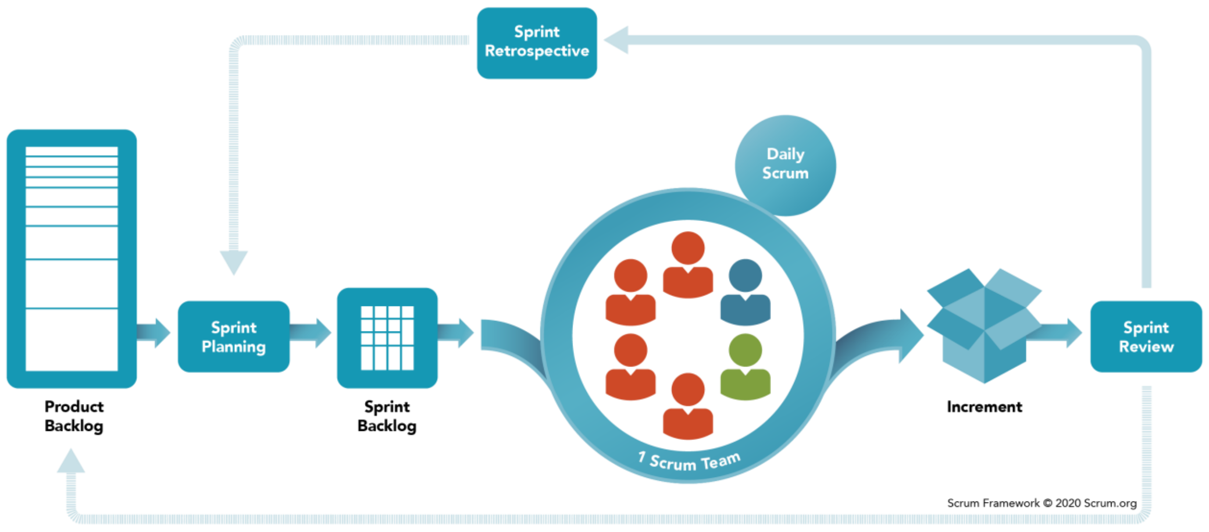


Figure 13. Scrum

In our case the team consisted of two people acting as the Product Owners and the developers at the same time. We did weekly meetings to sum up work that was done during the week and set priorities for the next week.

4 Analysis and validation

After developing the application, it is important to reflect and do the analysis. To understand if the goals are met it is needed to validate the results. For the technical validation unit tests and integration tests can be used as well as the end-to-end tests.

4.1 User level testing

In the development process it was decided to cover the codebase with unit tests where is necessary This step is described in the previous chapter. On the stage of validation, it was important to test the actual user flows from the UI. Several scenarios were tested on the API level and then from the UI level as well.

The API level testing included creation of different entities, linking them together, updating and deleting them.

UI level tests included simplified user flows that the users could have used in the process of the MVP demo.

4.2 Questionnaire

In order to validate the result, it was decided to create a questionnaire. The questionnaire consisted of 11 questions and was answered by 30 respondents.

The questions were open-ended, and the aim was to get an idea if this solution would satisfy them and what are the steps for them to adopt this technology and use on a daily basis.

The questions were like

- How do you think this product will help you?
- What are some tools you’ve tried and why have they not stuck for you?
- What are your thoughts or questions regarding these feature options?

The answers were very different, but it was good to get interesting perspectives on what the system was missing and what parts of the system users do not understand. Part of the solution came out to explain the users how to use the features to solve their needs and link them with the right feature to build the flow of solving their problems.

4.3 Demo

During the survey there were many e-mails laying around, so it was a good chance to organize some demos with the MVP.

The respondents could submit their own data and reflect to the flow and the system usage. While it was clear that it’s just an MVP some problems were spotted during the process and will be fixed in later versions.

The demo was laid out to a specific script. It is important to compose the script before doing the demos. The respondents must understand the incoming information quicky and be able to react and respond.

It was important to

- Determine the objective and the length of the script
- Determine the data that will be in use – here the choices are to use the vendor's data or the data that can be prepopulated. Both ways have their pros and cons.

It was decided to use prepopulated data as it is quicker to populate and manage for other demos as well.

The length was agreed to be at most 90 minutes including the questions and answers part.

5 Conclusion

It was an interesting project to work on. The goal is accomplished because the system was tested and can be developed further. During the study there was a lot of new knowledge gained about the ITIL and information systems in general. The system design and system architecture topics were learned a bit also. For the future work it is good that the validation was done on this early stage, the system development has more clear goals to accomplish.

6 References

[1] ITIL processes, 2020. [Online material] [Accessed 01.03.2021]
Available: https://www.edureka.co/blog/itil-processes/

[2] Datadog, 2021. [Online material] [Accessed 02.03.2021]
Available: https://www.datadoghq.com

[3] Datadog incident management, 2021. [Online material] [Accessed 02.03.2021]
Available: https://www.datadoghq.com/blog/incident-response-with-datadog/

[4] Atlassian company, 2021. [Online material] [Accessed 03.03.2021]
Available: https://www.atlassian.com/

[5] Slacable microservices at Netflix, 2014. [Online material] [Accessed 04.03.2021]
Available: https://www.infoq.com/presentations/netflix-ipc/

[6] Service pattern for Go, 2019. [Online material] [Accessed 06.03.2021]
Available: https://github.com/irahardianto/service-pattern-go

[7] Gorilla Mux HTTP Framework, 2021. [Online material] [Accessed 06.03.2021]
Available: https://github.com/gorilla/mux

[8] Restful Objects, 2017. [Online material] [Accessed 08.03.2021]
Available: https://en.wikipedia.org/wiki/Restful_Objects

[9] What is JWT, 2021. [Online material] [Accessed 10.03.2021]
Available: https://jwt.io/introduction

[10] SQL naming conventions, 2014.. [Online material] [Accessed 11.03.2021]
Available: https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I David Zingerman

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Web application development to solve problems of documenting microservice architecture”, supervised by Inna Švartsman
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

17.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.