Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Benjamin Loxley Wood
Martin Rui Andorsen
Sander Thorsen
Stian Tusvik

Empowering Data Analytics

A Data Documentation Tool for Optimizing Data Driven Decisions

Bachelor's thesis in Bachelor i programmering
Supervisor: Johanna Johansen
May 2023

Benjamin Loxley Wood
Martin Rui Andorsen
Sander Thorsen
Stian Tusvik

Empowering Data Analytics

A Data Documentation Tool for Optimizing Data Driven Decisions

Bachelor's thesis in Bachelor i programmering
Supervisor: Johanna Johansen
May 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

SUMMARY OF BACHELOR PROJECT

Title:	Empowering Data Analytics: A Data Documentation Tool for Optimizing Data Driven Decisions
Date:	May 22nd 2023
Participants:	
	Martin Rui Andorsen · Stian Tusvik · Sander Thorsen · Benjamin Loxley Wood
Supervisor(s):	Johanna Johansen
Employer:	Intility AS
Keywords:	Data science, data visualization, analytics, documentation, full-stack application
Pages:	131
Attachments:	8
Availability:	Open

Abstract:

Intility is a leading provider of IT operations services in Norway and would like to gain a competitive advantage by making more data-driven decisions. In their quest to use data more actively, they requested a tool which provides easy insight into their data pipelines, ultimately ending as reports used for making educated decisions.

Our group took on the challenge to create a web application for data documentation using .NET and React. The application consists of a REST API for data retrieval and manipulation, and a user-friendly web-based interface for managing and monitoring data pipelines.

The result is a comprehensive solution which gives an extensive overview of data processing pipelines and provides functionality to manipulate parts of the pipeline’s documentation. This report illustrates the development process and its challenges, and how we overcame each one.

SAMMENDRAG AV BACHELOROPPGAVE

Tittel:	Forbedre Dataanalyse: Et Data Dokumenteringsverktøy for Optimalisering av Datadrevne Beslutninger
Dato:	22. mai 2023
Deltakere:	
	Martin Rui Andorsen, Stian Tusvik, Sander Thorsen, Benjamin Loxley Wood
Veileder:	Johanna Johansen
Oppdragsgiver:	Intility AS
Nøkkelord:	Data science, datavisualisering, analyse, dokumentasjon, full-stack applikasjon.
Antall sider:	131
Antall vedlegg:	8
Tilgjengelighet:	Åpen

Sammendrag:

Intility er en ledende leverandør av IT-driftstjenester i Norge og ønsker å oppnå et konkurransefortrinn ved å ta mer datastyrte beslutninger. I deres ambisjon etter å bruke data mer aktivt, ønsket de et verktøy som gir enkel innsikt i deres datajobber, som til slutt resulterer i rapporter som brukes til å ta kunnskapsbaserte beslutninger. Vår gruppe tok utfordringen med å utvikle en nettapplikasjon for data-dokumentasjon ved hjelp av .NET og React. Applikasjonen består av en REST API for datahenting og manipulasjon, og et brukervennlig grensesnitt for administrasjon og overvåking av datajobbene. Resultatet er en omfattende løsning som gir en grundig oversikt over datajobber, og gir funksjonalitet for å manipulere deler av dokumentasjonen til jobbene. Denne rapporten illustrerer utviklingsprosessen og dens utfordringer, og hvordan vi håndterte hver enkelt hindring.

Preface

This report is the culmination of months of hard work and dedication by the authors. It is the result of a collaboration between the group and Intility, and represents our efforts to design and implement a user-friendly application that meets the needs of Intility's data scientists.

We would like to thank our supervisor Johanna Johansen for her guidance and support throughout the project. We are also grateful to the employees of Intility who provided us with valuable insights into their workflow and needs. An extra appreciation goes out to the product owner Hans Olav Vogt Myklebust, our assigned technical advisor Christian Richard Lie, and lastly our data science expert Kristian Senneset.

Finally, we extend our gratitude to our fellow students, and our friends and families for their constant support and motivation throughout our studies.

Oslo, 16.05.2023

Benjamin Loxley Wood, Martin Rui Andorsen, Sander Thorsen and Stian Tusvik.

Glossary

Bifrost - A design library in Figma and React complete with react components. Created and used by Intility's developers to create applications with similar designs and functionalities.

Business system – The starting point for a data pipeline. Can be internal or external to the company. Examples of business systems can be Microsoft's Power BI, an external API or other applications.

Data pipeline - A series of automated processes that move data from one system or application to another. Usually starting with complex and large datasets provided by a business system and resulting in a model or report usable and interpretable by humans.

Data platform manager – The name of our application, further described in chapter 1.1.4.

Exposures – A shareable resource which can be used by other Data Build Tool (DBT) projects or reports. The processed data after going through the data pipeline.

GDPR (The General Data Protection Regulation) – Is a law that aims to protect the privacy and personal data of EU citizens.

Intility standard – Intility has a standard when creating applications for their platform. This includes for instance design standards with specified colors, functionality standards with nav bars, authentication and much more. They also have standards in regards of programming languages where they use React with TypeScript for front-end and C#, with .NET for back-end development.

JSON (JavaScript Object Notation) – A lightweight data format which is easy for humans to read and write. Also easy for machines to parse and generate. It is the most used format for transferring information between computers, and is what we use to send data between back-end and front-end.

LINQ (Language Integrated Query) – Component in .NET that makes it easy for developers to query data from different sources by using a standard set of operations and syntax.

Mart – Short for Data Mart. A table that is designed to transform raw data into analytics ready data by applying SQL transformations. The analytics ready data is usually designed to support the reporting requirements in a business.

Regex (Regular Expression) – Pattern-matching language for finding and manipulating text patterns.

Reports – Human readable visualization of the data. Used for presenting the data in an easy-to-read manner to anyone and can be the basis of decision-making within a company.

Source – Data sources refer to the primary origin of the raw data. This can be a file, a database, or similar.

SSMS 18 (SQL Server Management Studio 18) - A software application developed by Microsoft that is used to manage and administer SQL Server databases.

Staging - A temporary table that holds data in its original state (raw data) after it is extracted from its source and before it is transformed in a data mart. Typically used for validating types and cleaning out unnecessary data before continuing in the data pipeline.

WAN – Short for Wide Area Network. It is a computer network that connects multiple Local Area Networks together, over a large distance. This allows different devices across the world to communicate with each other.

WCAG (Web Content Accessibility Guidelines) - Guidelines on how a website should be created to make the content accessible to everyone, including people with various disabilities.

Table of contents

Preface III

Glossary IV

Table of contents VI

List of Figures X

1 Introduction 1

1.1 Domain 1

1.2 Objective 4

1.3 Group background 4

1.4 Constraints 6

1.5 Delimitations 9

1.6 Project goals 9

1.7 Target audience 11

1.8 Structure of the report 12

2 Requirements 14

2.1 UML Domain Model 14

2.2 Functional requirements 17

2.3 Non-functional requirements 22

3 Design 26

3.1 GUI 26

3.2 Prototyping with Figma 30

3.3 System design 30

3.4 Easy-to-use 36

4 Technologies 38
4.1 Back-end 38
4.2 Front-end 41
4.3 Communication and helper tools 43
5 Implementation 45
5.1 Deployment 45
5.2 REST API 48
5.3 User authentication 52
5.4 Database 52
5.5 Implementation of GUI design 54
5.6 Business glossary page 54
5.7 Business system information page 55
5.8 Data Assets page 58
5.9 Search function Front-end 62
5.10 Individual Information Pages 69
5.11 Lineage graph 76
5.12 Changelog 84
5.13 Stats 88
5.14 Sustainability 90
6 Testing 91
6.1 Back-end Testing 91
6.2 Front-end Testing 98
7 Methodology – Usability Testing 104
7.1 User test preparation 104

7.2 User testing theory ... 104
7.3 Test questions ... 105
7.4 Test Scenario ... 107
7.5 Test results ... 107
7.6 Ethics ... 108
8 Development process and communication ... 110
8.1 Scrum ... 110
8.2 Jira ... 110
8.3 Training period ... 112
8.4 Meetings with Intility ... 115
8.5 Meetings with bachelor guidance counselor ... 115
8.6 Internal meetings ... 116
8.7 Work environment ... 116
9 Summary ... 118
9.1 Results ... 118
9.2 Process ... 119
9.3 Learning outcomes ... 120
9.4 Future development ... 122
9.5 Significance ... 124
10 References ... 125
11 Appendices ... 1
Appendix A Consent forms usability test ... 1
Appendix B Project Plan ... 13
Appendix C Meeting minutes ... 28

Appendix D Task description 35
Appendix E Pictures of Sprint boards... 39
Appendix F Timesheets... 42
Appendix G Data Science Expert on the *Change Log* feature.. 44
Appendix H Examples of Figma Sketches 45

List of Figures

Figure 2.1 Association types 15

Figure 2.1 Association types 15

Figure 2.2 Multiplicity examples 15

Figure 2.3 Domain Model 16

Figure 2.4 Use case diagram 18

Figure 3.1 Standard website layout 27

Figure 3.2 Example of DBT docs information page 28

Figure 3.3 Example of dropdown menu for smaller screens 29

Figure 3.4 System architecture 31

Figure 3.5 Back-end solution 31

Figure 3.6 Data flow through back-end API 32

Figure 3.7 API folder structure 32

Figure 3.8 C# Models folder 32

Figure 3.9 Controllers folder 33

Figure 3.10 Data folder 33

Figure 3.11 C# Constants 34

Figure 3.12 Custom error message popups when user makes mistakes adding glossary words 36

Figure 3.13 Custom error message if the input fields are left blank 37

Figure 3.14 Warnings updating live when user inputs information in 'Add business system' fields 37

Figure 4.1 Mocking “DbContext” with Moq 40

Figure 4.2 Performance test for React vs Vue.js. (Borozenets, 2022) 42

Figure 5.1 Gitlab example, pipeline failure and success 46

Figure 5.2 Microsoft Azure API access 47

Figure 5.3 Resource URIs from the Swagger tool 48

Figure 5.4 “Data Sources” resource from swagger API 49

Figure 5.5 Response example and status codes from Swagger API 51

Figure 5.6 Logical Model 53
Figure 5.7 Business glossary page. 55
Figure 5.8 First draft of information page 56
Figure 5.9 Reworked information page to be more like DBT docs 56
Figure 5.10 Functionality to add several sources at once to a single business system 57
Figure 5.11 Data asset search input 58
Figure 5.12 Illustration of the loading animation during a search 58
Figure 5.13 Folder selection field, with possibility to search for folders and select multiple at once 58
Figure 5.14 Filtered table based on the folders selected 59
Figure 5.15 Example of filtering based on type 59
Figure 5.16 Pagination options 60
Figure 5.17 Spinning search button while loading 63
Figure 5.18 Setting the page to one, in order to display the first results 65
Figure 5.19 "removeSpecialCharacters" function which removes all special characters using regex 66
Figure 5.20 Example of the data asset page with search input 68
Figure 5.21 Example of search functionality on DBT docs 68
Figure 5.22 Individual Source page 70
Figure 5.23 Individual Staging page 70
Figure 5.24 Individual Mart page 71
Figure 5.25 Individual Exposure page 71
Figure 5.26 Visual representation of the anchor scrolling tab 72
Figure 5.27 Code for the handling of tab click. 73
Figure 5.28 Code section where copy to clipboard is used 73
Figure 5.29 The button changes when clicked from "copy to clipboard" to "copied". 73
Figure 5.30 Image of the formatted code with line numbers. 74
Figure 5.31 Depends on section with links to the marts and stagings it depends on 75
Figure 5.32 Early first draft of how the lineage graph should be presented. 76

Figure 5.33 Intility Topology graph used for inspiration. 77
Figure 5.34 Lineage graph example showing the mart “power_bi_reports” as the targeted node. 77
Figure 5.35 Example of lineage graph from DBT docs. 78
Figure 5.36 Lineage graph with information drawer extended.. 79
Figure 5.37 Business system info page. 79
Figure 5.38 Changelog example. Changed nodes above, changed relations below. 87
Figure 5.39 Statistics UI.. 89
Figure 5.40 Code for one element of the statistics UI 89
Figure 8.1 Jira board from sprint 7 111
Figure 8.2 Jira issue types 112
Figure 8.3 Git branch naming example 113
Figure 8.4 example commit message 113
Figure 8.5 A sketch we designed in Figma after the Bifrost Figma workshop........................... 114
Figure 8.6 Naming conventions for CSS styling of Bifrost components 114

1 Introduction

In today's data-driven world, organizations rely heavily on data to make informed decisions. However, managing, and documenting data can be a complex and time-consuming process (Kelleher & Tierney, 2018). The importance of having good and updated data keeps increasing, and Exasol has found that 58% of organizations make decisions based on outdated data (Exasol, 2020). In response to this challenge, our team has developed a data documentation tool aimed at simplifying the documentation process for data pipelines.

The tool is designed to help data professionals track and document data transformation processes, provide an overview of data dependencies, and facilitate collaboration and knowledge sharing among team members leading to increased efficiency and productivity. In this report, we describe the design and implementation of our data documentation tool and provide an evaluation of its effectiveness in a real-world setting.

1.1 Domain

To ensure a shared understanding for this thesis, it would be helpful to clarify terminology and concepts, which we will do in the following subchapters. The assignment is based around the data science field, and the users are mostly experts within this subject. Currently, the data scientists are using DBT docs as the preferred documentation tool for data pipelines, which our application has an objective to replace. This will be achieved by using the JSON manifest file from DBT docs, and processing and displaying the information in a better and more efficient way.

1.1.1 Data science

Data science is a field which involves using statistical and computational methods to extract insights from large and complex datasets. It combines computer science, mathematics and statistics, machine learning and advanced analytics to identify patterns and relationships within the data (Kelleher & Tierney, 2018). The goal of data science is to use data-driven insights to make better predictions and decisions.

Data scientists are responsible for a range of tasks, such as data cleaning, preprocessing, analyzing, and visualizing, as well as building predictive models using various machine learning algorithms. To be successful, data scientists need good problem-solving, critical thinking, and communication skills to develop data-driven strategies and explain their findings to stakeholders (Campbell, 2021).

The importance of data science is increasing in many industries, including healthcare, finance, and marketing, due to the exponential growth of data collection (Campbell, 2021). This has led to a rapid expansion in the field of data science, which again calls for tools for documentation, like DBT docs.

1.1.2 The current solution – DBT and DBT docs

The employees in Intility's data science department use DBT (Data Build Tool) extensively in their daily tasks. DBT is a tool to help build, test, and maintain data pipelines. It provides features for testing, automatic documentation generation, modularity, and version control, making it easy to collaborate and manage changes (Barcheski & Collis, 2023).

In addition to building and maintaining data pipelines, DBT also provides a feature called DBT docs. This feature automatically generates a website containing documentation for your DBT project, including information on models, sources, and tests (See example in Figure 3.2). With DBT docs, you can easily keep your project's documentation up to date, collaborate with team members, and provide a source of truth for your data pipeline. Another feature within the website is a lineage graph, visualizing the data pipeline inside of DBT (Figure 5.35). By using DBT and DBT docs together, you can create a robust, maintainable, and well-documented data pipeline that is ready for analytics and reporting.

A major issue with DBT docs is the fact that it does not have a separate front-end and back-end. This means that the presentation and the business logic is all handled front-end, which makes the loading time for certain parts of the application very slow. The DBT docs front-end receives the entire manifest file (in some cases containing around 200,000 lines of code) from DBT and must do all the logic of finding relationships and create the lineage front-end, sometimes taking minutes to load.

Another issue with DBT docs is the search functionality. In the existing solution, whenever a user enters a search word, all the results appear in alphabetical order. This decreases efficiency whenever a data scientist is looking for one specific asset. The reason for this is that the asset they are targeting often begins with the search word, but will be sorted only based on alphabetical order, just like all other assets are sorted. The product owner requested an improved search algorithm, providing the results which are most relevant first.

1.1.3 JSON Manifest

The "Manifest.json" file can be exported from DBT docs and contains all the information about how the data is organized and transformed in the DBT project. This includes information about models, sources, and tests, as well as their relationships and dependencies. An example of such metadata can be the name, id, file path, type, description, and dependencies of an exposure. This information can be used to get a better understanding of what data the exposure contains and how it is affected by other data.

To address the issues mentioned above, our project will utilize the information in the manifest file to build a custom documentation tool which displays the data pipeline, where the data originates from, how it is transformed, and what the final output looks like.

1.1.4 Data platform manager

To provide clarity regarding the term "Data Platform Manager," it is important to discuss its usage within our project and report. The name of the initial task description given from Intility was Data Platform Manager, which also became the name of our application. It has been consistently referenced as such during the development process. While we acknowledge that a data platform manager should consist of several features our documentation tool does not include, Intility's vision is to keep developing the application for it to become a comprehensive data platform manager. Within this report, we consistently use the term "Data Platform Manager" to refer to our application.

1.2 Objective

The main objective of this project is to create an easy-to-use application for documenting data pipelines. What we mean with easy-to-use is described in chapter 3.4. The application should function as a substitute for the existing documentation solution (DBT docs) and have extended functionality to fit the needs of Intility's data science department. Our application should also include the functionality to display information about business systems, and the relationship between business systems and sources.

A key issue Intility has with the existing solution is that there are no opportunities to develop further functionality. Since the applications currently used are not developed by Intility, there is no option to modify the layout, alter existing functionality, or implement new features. Allowing for flexibility we can greatly enhance automation within Intility's data science department by expanding on the system with new features.

1.3 Group background

This report introduces a group of four bachelor students who are all enrolled in the "Programming" study program at NTNU in Gjøvik. Throughout the program, the students have acquired skills and knowledge about a variety of programming subjects. We have experience working in a group from previous projects throughout our education, which played a significant role in our decision to come together to write our bachelor thesis.

1.3.1 Competence

The table below showcases the relevant courses from the program, with a short explanation of their relevant learning outcomes.

Fundamental programming	Basic programming skills
User centered design	Sketching, prototyping, user testing
Software development	Scrum, project planning and execution
WWW-technologies	JavaScript, React
Advanced programming	Testing
Cloud technologies	APIs and cloud-based programming
Data modelling and database systems	Databases and SQL
Algorithmic methods	Recursive functions
Integration Project	Project experience

Table 1.1 Overview of relevant courses throughout our Bachelor in programming

In the courses mentioned in the table above, we learned the basics in order to start with a project like this. The reason why we say the basics, is that for this assignment we are using tools that were not taught during any of the courses. We learned similar frameworks and languages, but had to do some additional research for this project. Research is a category in our time sheets that encapsulates hours spent studying and implementing tools and frameworks for this application. Look at Appendix F in to get an understanding of how much time we used for necessary research.

As mentioned briefly in the table above, we used the knowledge from the courses in different parts of the project. We had to sketch some prototypes in Figma and have user tests with the users of this application. This was knowledge from the User Centered Design subject. The knowledge from Software Development came in handy for scrum meetings, sprint planning, sprint retrospective and sprint review. As learned in the WWW-technologies course, we used React for the front-end of the application. We also learned JavaScript in WWW-technologies, but the front-end for this assignment is written in Typescript which is a type-safe variant which is transpiled to JavaScript (Learn Typescript, n.d.). The course Cloud Technologies taught us

about cloud-based programming and how to create different APIs. Our application has a relational SQL database and for this, we used the knowledge acquired in the Data modelling and database systems course.

1.3.2 Choosing the task

In the initial phase of choosing a project, our main goal was to find a suitable task that would allow us to apply the knowledge and skills we have acquired throughout our studies. Among the suggested tasks from the university, we saw three tasks from Intility, which were the only ones where the students had to go through an interview process in order to get. To apply for these projects, we sent an application letter that included our CVs and grades from all previous courses. After two interviews, we were offered the opportunity to work with Intility.

We accepted the offer, as we found the task to be remarkably interesting and helpful to the company. We especially liked that it was an application which provided real value in form of better information flow to data scientists, would be frequently used by Intility, and that would be developed further after our time at Intility was over.
The agreement included that our group would develop the data platform manager this report is based on. Intility would provide the necessary equipment, transportation costs and commute housing every two to three weeks. We have described the work environment and collaboration with the company further in chapter 8.7 Work environment.

1.4 Constraints

The constraints of a project can play a critical role in shaping the direction of the work and its outcome. In the case of our bachelor thesis, we faced a range of constraints that influenced the development of our data documentation tool. These constraints were both internal and external and affected various aspects of the project. In this chapter, we will explore the constraints we faced and how we managed them to achieve our project goals.

1.4.1 Time

This assignment was large, and Intility did not expect us to complete the application within the given time. The application has limitless potential for further development, which meant that we had to set clear delimitations of what should be prioritized. The whole project, including writing the project report should take between 560 and 660 hours per student.

To manage this constraint, we decided not to create tests for every part of the code, but instead create good tests for the main functionality. This was to show that we have the knowledge and skills to create good tests and could have created tests for all functionality if time was no constraint.

1.4.2 Access to sensitive information

Access to Intility's sensitive internal information about customers and employees was another constraint we had to manage. Firstly, we did not have access to a database to retrieve employee names to have provide options when selecting owner and contacts for new business systems. After dialogue with Intility, we produced a solution which lets the users of the application choose from names which have been entered into the system previously, or alternatively add a new name. This will be changed when Intility take over the application development, where a list of all Intility employees will be available to use.

Secondly, we did not have access to the endpoint which let us access the manifest file as it is updated. This introduced a problem in the early stage of the development due to information from the updated manifest file which we did not have access to. The final product also did not have automatically updated information due to this constraint. The functionality to update the database based on a new manifest file is implemented, but for now the user must manually change the existing file and run an update. With access to the manifest endpoint, this can be automated to fetch and update at an interval.

1.4.3 Technical expertise

Technical expertise could pose a significant constraint in the development of the data documentation tool. Although we have obtained a strong foundation in computer science throughout our studies, the development of a complex tool like this requires advanced

technical knowledge and expertise. To achieve the objectives of this project, we first had to acquire the necessary knowledge and skills within these technologies.

The tool requires the integration of various data sources, such as business system information, reports, and automatically generated documentation from DBT. This required knowledge about DBT, DBT docs and the entire data science field, which we initially had little to no experience with.

Furthermore, the tool should be coded using Intility's preferred languages, namely TypeScript and C#, which are also completely new to everyone in our group. We have explained further how all these challenges have been tackled throughout the rest of this report.

1.4.4 Practical and physical

Effective communication within the team is essential to achieve a successful software development process. The physical distance between the team members and Intility was another hurdle we had to manage. With the use of several digital communication tools, we were able to keep a close collaboration with the company during the entire development process. As a group we were familiar with this method of communication due to the Covid-19 lockdown. More information about the digital communication tools we used can be found in chapter 4.3.3.

Aside from the physical distance, there have not been any other practical constraints related to our work. Everyone in our group had the necessary equipment to work as efficiently as possible both at the office and at home. Intility provided us with all the needed software, hardware, our very own office spaces, and the access to meeting rooms. More about the physical working conditions can be found in chapter 8.7 Work environment.

Intility also covered lunch at the office and any travel costs related to the commute. The employees at Intility did everything they could to ensure that we had good work-conditions during this project.

1.5 Delimitations

As mentioned, the main objective of the project was to create a data pipeline documentation tool. However, we did not implement functionality which has a direct impact on the data pipelines. Rather, we documented information about each stage in the pipeline. The application does not have functionality to trigger or re-run pipelines. Nevertheless, we left this as an opportunity to expand for the company at a later stage.

The finished product features user authentication. However, individual user roles such as admins, users or developers are not part of this version of the application. This feature can be added at a later stage. We have not been provided with the privileges necessary to obtain information about individual employees when they are logged in to our application which restricts our options in this regard.

1.6 Project goals

During this project, we wanted to achieve a set of specific goals that helped us accomplish our overall task. Our goals were divided into three distinct categories: result goals, effect goals and learning goals. Result goals focused on the results we wished to achieve, while effect goals concentrate on the impact our project has on the intended audience. Learning goals are centered around the new knowledge and skills we acquired during the project. With these goals in mind, we were able to plan and work more efficiently throughout the duration of our project.

1.6.1 Result goals

The result goal of our project was to create an application capable of displaying all relevant information about a data pipeline. This was achieved by tying together automatically generated documentation from DBT with business system information and reports. In addition, we included a feature displaying a visual representation of the relationships between various parts of a data pipeline, while ensuring that the loading time of the visual representation was no more than 50% of DBT docs’ alternative. This was implemented as an additional feature to what was initially planned.

1.6.2 Effect goals

To achieve our project goals and improve the overall data management practices at Intility, we identified several key effect goals for our data documentation tool. Firstly, we aimed to increase adoption and usage of the data documentation tool outside of the Data Science department, which would result in increased overall understanding and collaboration across teams. This is accomplished by creating a user-friendly interface which is easy to navigate and intuitive to use.

Another important goal was to improve the accessibility of information and ease of information gathering by implementing more forgiving search parameters. Additionally, we wanted to ensure data accuracy and reliability within the system by tracking changes made to the documentation. This would allow for better accountability and traceability in the event of errors or data inconsistencies.

Furthermore, we aimed to reduce the loading time for large lineage graphs, which would improve the user experience and allow for quicker identification of data dependencies. Finally, we made sure to design the application in compliance with WCAG accessibility guidelines and Intility's branding to ensure a seamless integration with their existing systems.

Overall, by achieving these goals, we aimed to provide a valuable tool for data-driven organizations seeking to optimize their data management processes and improve collaboration across teams.

1.6.3 Learning goals

Another important aspect of our bachelor thesis is learning goals. The goals below are what we have learned through conducting and concluding this bachelor thesis. Our group acquired more experience and knowledge of the following:

- Learn to utilize agile development and Scrum to achieve an effective workflow throughout the project.
- Git best practices, version control and CI/CD (continuous integration & continuous deployment).
- Acquire knowledge around using Azure and cloud platforms to host an application.

- Managing a larger project with deadlines for delivery.
- Become confident in .NET and Typescript and SQL for data management and documentation tasks.
- Learn more about how to collaborate with a company to develop a useful application that meets their needs.
- Make an application within an unfamiliar domain like Data Science.

1.7 Target audience

1.7.1 Application

Our data platform manager tool is designed to simplify the data analysis process, enabling users to gain a better understanding. The primary target audience for our application is individuals who work in the data science department of Intility. This includes data scientists, data analysts, and data engineers who require an intuitive and user-friendly tool to analyze large and complex datasets efficiently. However, the application may also be used by other employees at the company who work with data but do not have a specialization in data science. Departments such as finance or marketing can also benefit from using the application to analyze data relevant to their areas. To summarize, the application is valuable for anyone working with data at Intility, regardless of their level of expertise.

1.7.2 Report

The target audience for our report is primarily educators and evaluators. The report provides an in-depth analysis of the development process and implementation of our data platform manager. It covers the relevant information needed to understand how the application is implemented, our development process and other information such as time management and the theory to back up our decisions. In addition to educators and evaluators, our report may also be of interest to other students. The report can be valuable for students wanting to learn more about creating data analysis tools. Overall, our report is a comprehensive and informative resource that offers insights into the development and implementation of a data platform manager, making it a valuable resource for both professionals in data science, as well as students learning topics related to this field, and academical staff teaching in this field.

1.8 Structure of the report

Chapter 1 - Introduction

Explains key terms and concepts, and introduces the group and their background. Also presents the main objective and goals of the project as well as constraints and delimitations.

Chapter 2 - Requirements

This chapter focuses on detailing the requirements of the project. It includes a UML diagram, as well as a comprehensive analysis of the functional and non-functional requirements, capturing the needs and expectations of the project stakeholders.

Chapter 3 - Design

In this chapter, the design phase of the project is discussed. It covers the overall system architecture, prototyping, the GUI and how we have followed numerous WCAG guidelines. The chapter explains the rationale behind design decisions and how they align with the project goals.

Chapter 4 - Technologies

This chapter provides an overview of the technologies and tools chosen for the project. It includes a discussion of the programming languages, frameworks, libraries, and software used, along with the reasoning behind the choices of technology.

Chapter 5 - Implementation

Here, the implementation phase of the project is described. It discusses in detail the process of translating the design into code. The chapter covers the different functionalities and features implemented in the application.

Chapter 6 - Testing

This chapter explores the testing implemented in the project. It includes unit testing and integration testing for both front-end and back-end. The chapter also discusses the frameworks utilized in making the tests.

Chapter 7 - Methodology – Usability Testing

This chapter focuses specifically on usability testing and the theory behind it. It explains the methodology used for conducting usability tests, including test preparation, test scenario and results. Lastly, it discusses the ethics behind user testing.

Chapter 8 - Development process and communication

In this chapter, the development process and communication within the project team are addressed. It covers aspects such as project management methodologies, training period, meetings, and our work environment.

Chapter 9 - Summary

The final chapter serves as a summary and conclusion of the project. It recaps the main objectives, achievements, and challenges encountered, and discusses weather we achieved the goals from our introduction. The chapter also includes suggestions for further improvement, and information about what significance our application will have for the company.

2 Requirements

The requirements chapter outlines the specific needs and expectations for the data documentation tool developed as part of our bachelor thesis project. By defining these requirements, we aim to ensure that the tool effectively meets the needs of its target users.

This chapter covers an overview of the domain, the functional and non-functional requirements of the tool, including its desired features, performance specifications, and usability guidelines. We have identified these requirements through extensive research, meeting with the target users, analysis of existing data management tools, and research around the industry best practices. The resulting set of requirements provides a clear roadmap for the development of the tool.

2.1 UML Domain Model

"The first step in analyzing the requirements is to construct a domain model" according to (Rumbaugh, 2004). The domain model aims to give an understanding of a system by organizing real-world entities into classes. We are not interested in software constructs such as databases, cloud computing which used within our program (Rumbaugh, 2004). Rather, we want to have terms which makes sense without having deep technological understanding.

2.1.1 Domain Classes

According to "Object-oriented Modeling and Design with UML" classes should be as accurate as possible, and only generic where applicable. This was the case with the "Business System" class shown in Figure 2.4 domain model, which first we thought was too generic. But according to the application we are designing the class can be a wide range of things, such as the accounting or the support department, or pertaining to cloud data. Therefore, the name "Business System" made sense in context of our application.

2.1.2 Associations and Multiplicity

Associations are indicated by lines between classes and give understanding to how classes are connected to each other (JavaTPoint, n.d.). In our domain model we use four different types of associations, examples are shown in Figure 2.2. The first type is a *simple association* which

means there is a direct association between two entities. The second is a *directed association* which means the connection is only going in one direction. An example from our domain model is the relation between DBT docs and the manifest. While the manifest is a direct creation from DBT docs, it does not have any association back to DBT docs after creation. The third type is *aggregation* and means that the class is created or is part of the class it aggregates towards. For example, the data models in the domain model aggregates from the manifest. The final type is *generalization* or *inheritance* which means that an entity is a child of another entity and inherits all attributes and functionality (Nishadha, 2022).

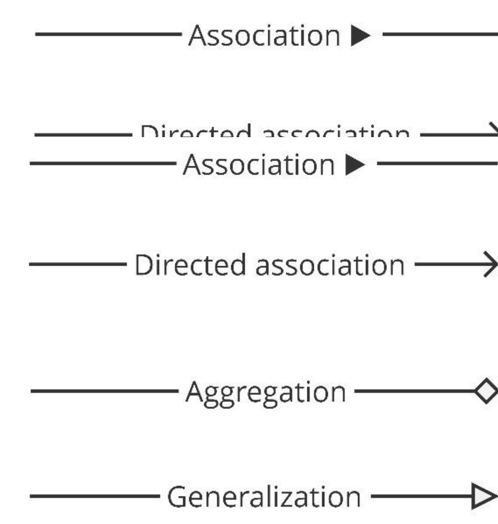


Figure 2.2 Association types

To indicate cardinality between classes we use multiplicity, which "describes how many instances of one class can be connected to another class through a given association" (UMLBoard, 2023). Figure 2.3 shows three examples of multiplicity to give an understanding of how it is used in our domain model. The first example is a simple one-to-one relation meaning each class A only has one class B and vice versa. The second is a one to many which indicates for each class A there may be many classes B, but for each class B there can only be one class A. The last example is many-to-many which indicates that for each class A there can be many classes B, and for each class B there can be many classes A.

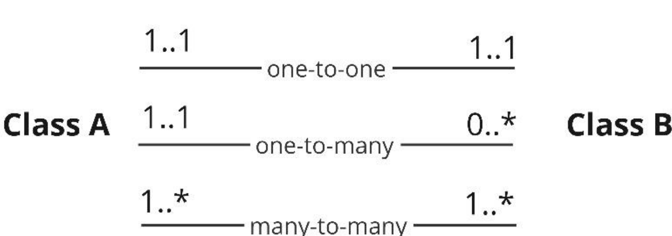


Figure 2.3 Multiplicity examples

2.1.3 Domain Model

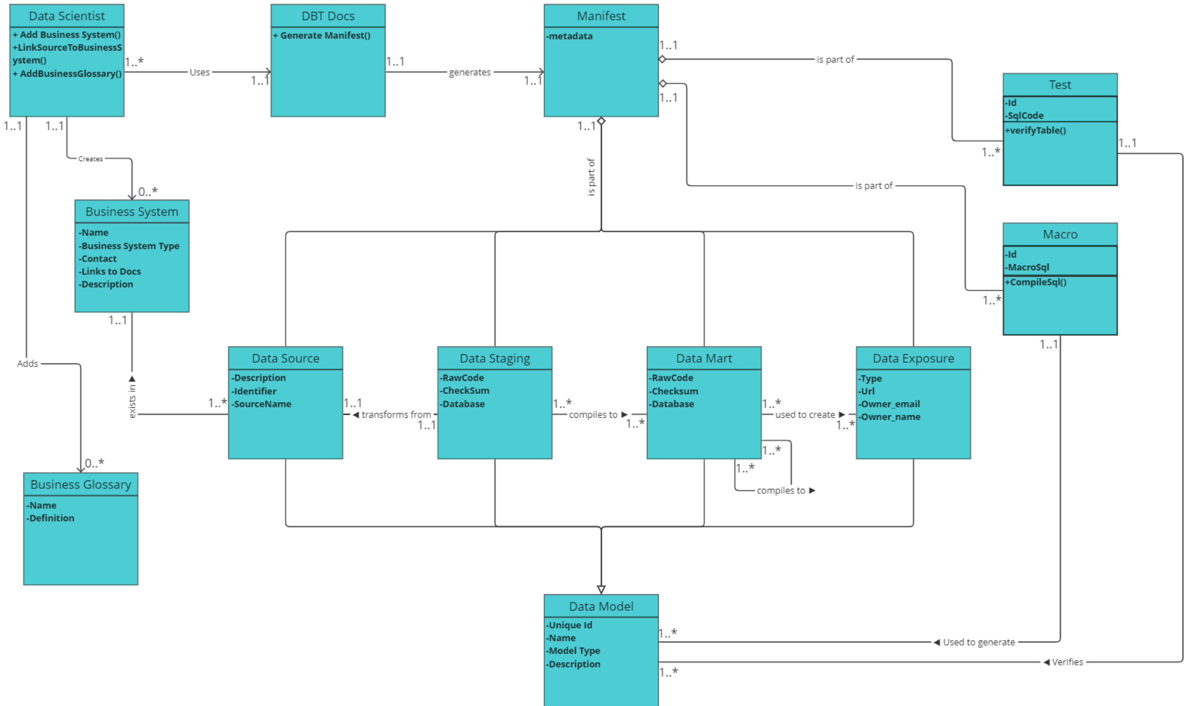


Figure 2.4 Domain Model

Data Models

Figure 2.4 describes all the objects and entities in our system and the relations between them. The “data scientist” is both a user of “DBT docs” and our system, which is built on the manifest from “DBT docs”. The manifest consists of many different data models, which is a generalization of the four models: “Data Source”, “Data Staging”, “Data Mart” and “Data Exposure”. These data models are reliant on each other, and every model’s data originates from a “data source”. The data source is compiled into data staging which is a one-to-one translation of raw data to a uniform format. Then staging tables are compiled into marts as many-to-many relations, meaning one staging can be used in many marts and one mart can use many different stagings and other marts. The marts are often “grouped by department or area of concern” (DBT docs, 2023) and similar data can be reused to create different contexts. Exposures describe the “downstream use of your DBT project” (DBT docs, 2023) and is the destination and final analysis of data. Exposures are also a many-to-many relationship from data marts. The manifest also consists of macros and tests, which are used to compile and test

the data models. In Figure 2.4 we aim to give a general description of the domain. Therefore, we did not include all attributes as it could lead to confusion and is not important to providing a general description of the domain.

Business System

The "data scientist" can add a "business system", which in turn are linked to different data sources. This is used to separate domains within the data catalogue and is helpful to see the data transformation in the given domain. An example of this could be linking all the data sources from the *billing department*. Then there would be a direct link from the billing department through data transformations into final data products, known as exposures.

The data scientist can also add business glossary definitions which helps give an understanding of terminology within the system. This is decoupled from the rest of the domain and is an important feature to help the user of the application to understand unclear or new glossary terms.

In conclusion, the domain model helped clarify the *data science* and *DBT* domain and its many concepts which were vague to us. This understanding enabled us to create functional requirements and effectively design our system. The domain model was also helpful to the creation of the logical model and data classes in our API.

2.2 Functional requirements

Functional requirements specify how the system should operate in terms of features, behaviors, and capabilities. In this chapter we present the desired functionality of the software, as well as what functionalities the different users have access to and detailed high-level use-cases.

2.2.1 Use-case diagram

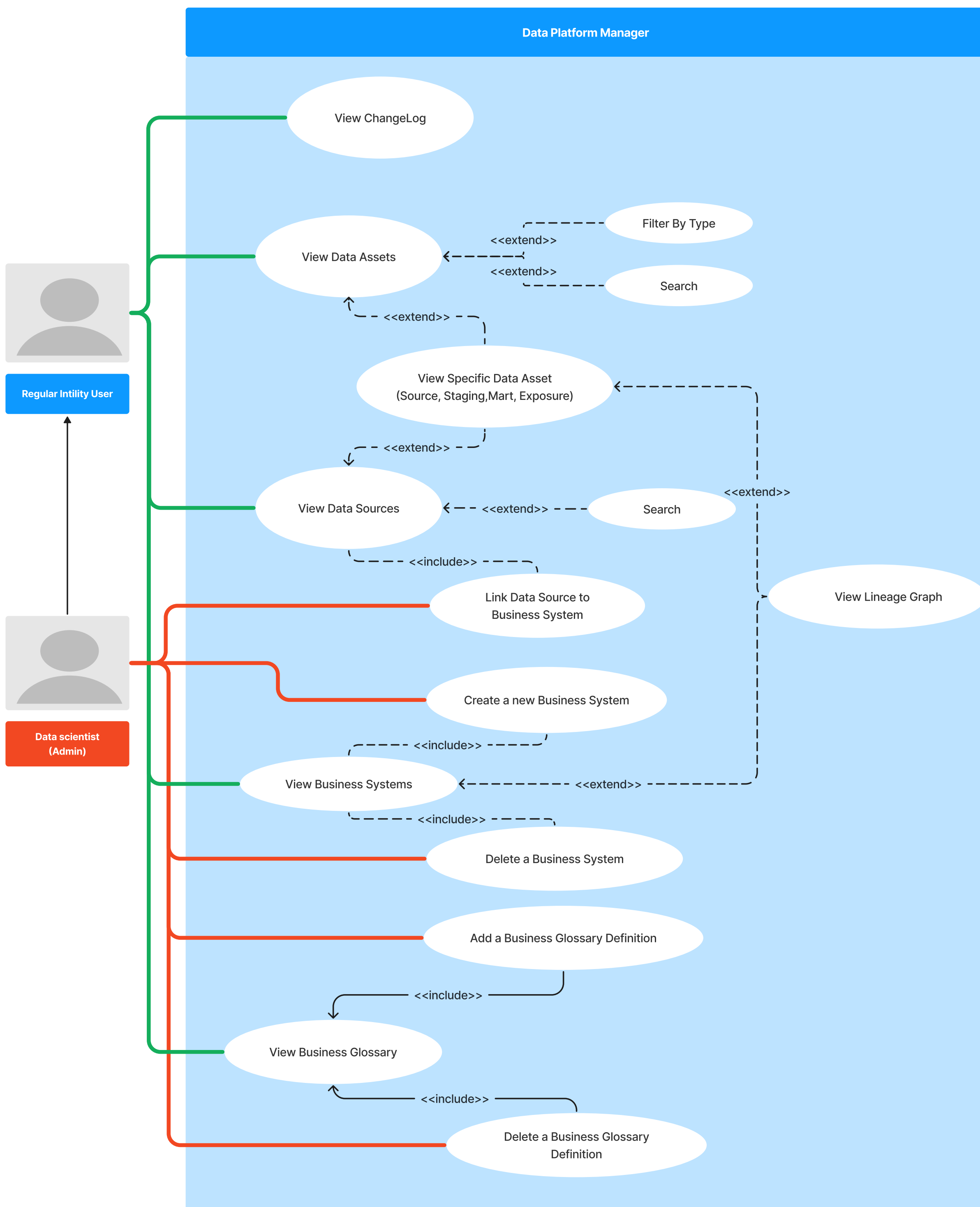


Figure 2.5 Use case diagram

The use case diagram shows how the different actors can interact with our system and the functionality it provides. This pertains to viewing the different pages in our system, as well as filtering and searching or adding new information. Some use-cases will be described in higher detail in chapter 2.2.3.

2.2.2 Actors

IBM DevOps documentation claims that “you should understand who the users of the system are before you begin to model it” (IBM Corp., 2023). Defining the system’s actors helps us understand who we are creating the system for and what their needs are.

All users in our system are employees working for Intility, or a part of their internal projects. Our system does not differentiate between regular users or data scientists, for reasons we explain in chapter 1.4.2 Access to sensitive information, and 9.4 Future development. Having two different users with two different goals was still beneficial to mapping the system’s requirements. This is because the need to view information and functionalities such as filtering is more important to regular users. Whilst adding and making changes to the application is more important to admin users.

Regular user: A regular user will have the possibility to inspect all data assets and metadata. The lineage graph will provide an overview and relations between business systems and data. The user will not be able to make any changes that could impact other users or the system outside of their own view.

Data Scientist: A data scientist will have all the functionality of a regular user. As well as the possibility of adding or deleting business systems and linking them to data sources. The data scientist will also be able to add or delete business glossary definitions.

2.2.3 High-level use-cases

Use case: View Data Assets and search. **Actor:** Regular User **Objective:** Find all data assets relevant to a specific keyword. **Description:** The user navigates to the data assets page showing a list of data assets. The user searches for the keyword in the search field. Capitalization should not impact the results which will be displayed in alphabetical order. However, it will filter based on ASCII (Loshin, 2021) (American Standard for Information Interchange) symbols used such as underscore or forward slash etc.

Use case: View specific data source. **Actor:** Regular User **Objective:** Find a specific data source and view its information. **Description:** **(main)** The user navigates to the data source page. **(alternative)** The user navigates to the data assets page and filters based on "*source".* **(continuation)** The user clicks on the source he wishes to see and enters the specific data source page which shows relevant information to a data source.

Use case: View changelog. **Actor:** Regular user **Objective:** View changelog for the last four days. **Description:** The user navigates to the application home page which displays recent changes to data formats such as deletions, additions, or changes to data assets. The user then selects a date four days prior, which filters to show the latest changes.

Use case: View the lineage graph for a business system.

Actor: Regular User

Objective: Navigate to a business system and display its lineage view.

Description:

The user navigates to the business system page and clicks on the business system they wish to view. Further the user clicks an icon or button that specifies the lineage graph and opens the lineage graph page. The page shows the business system's data sources and their lineage through data transformations into final data products.

Use case: Add business glossary.

Actor: Data Scientist (Admin)

Objective: Add a new business glossary

Description:

The user finds the help section of the navigation bar and selects "business glossary". The user searches the glossary and does not find the definition they are interested in. The user then clicks the add button or icon which allows the user to write a new word and description. If the word already exists, an error is issued, elsewise the word is added to the glossary.

2.2.4 Descriptive use-case

An example of a descriptive use case to show ideal flow of execution.

Use case: Create a new business system and link data sources then view the updated lineage graph.

Actor: Data Scientist (Admin)

Objective: Add a new business system and link data sources to the given business system

Description: The user navigates to the business system page and adds a business system. Then the user adds data sources which should be linked.

Program flow:

1. The user clicks the business system from the application menu.
2. In the business systems page the user clicks the *"add business system"* button or icon.
3. A form to create a new business system opens and the user fills input fields such as name, system type, description, owner and contact person.
4. The user then clicks submit, prompting any fields which do not pass validation that change is needed.
5. The page navigates to the newly created business system and the user clicks *"add new sources".*
6. A list of eligible sources is displayed and the user searches and checks off any source they wishes to link. Then the user clicks add.
7. The selected sources are displayed under the business system.
8. The user clicks the lineage graph icon or button and is navigated to the lineage graph.
9. The lineage graph shows the lineage to the added sources and their transformation through the system.

2.3 Non-functional requirements

"The non-functional requirements are those aspects of the IT system that, while not directly affecting the business functionality of the application but have a profound impact on the efficiency and effectiveness of business systems for end users, as well as the people responsible

for supporting the program" (Paradkar, 2017). These needs are crucial for longevity and usefulness of the system. A banking system which has all the specified functionality but does not meet its requirements in terms of security will not prove valuable long-term. These requirements are important to us as well and are divided into five categories: availability, reliability, resilience, scalability, and security.

By specifying these requirements, we know what we need to keep in mind and develop towards, but also what we do not need to prioritize. Our system has great need to be maintainable, as it will be developed further after our work is done. But if that was not the case, we could have spent less time on modularity and refactoring than we did.

2.3.1 Availability

After discussions with the product owner and the data science team it was clear that availability of the program would be especially important during business hours and preferably in the evenings. An agreement was made that nighttime would be the time to update our system with new metadata and changes made in the data catalogue. We therefore landed on an availability percentage of 99 which gives us roughly fifteen minutes each day of downtime for updating. This means that we needed to run a manifest update in less than fifteen minutes. In the end this was not needed as updating the application with a new manifest does not shut down our service.

2.3.2 Reliability

Since the system gives an overview of much of the company's data, an important factor is the reliability to trust that the system's information is correct and functional. Errors in data and dependencies between data can be an issue as it may give users wrong information resulting in bad business decisions and lost work hours. This is especially important as the system grows and expands functionality as described later in chapter 9.4 Future development. This meant that extra focus had to be put on testing the functions and logic which read and update metadata to ensure reliability. Therefore, we aim to perform integration and unit testing on the most crucial functionality as well as user testing. The most crucial functionality is related to the update of the manifest.json.

2.3.3 Maintainability

"Maintainability refers to the ease with which you can repair, improve and understand software code" (Sealights, n.d.). The software built during this project is going to be iterated on and added features to. As discussed later in 9.4 Future development, the ambition of this project long term to also be used for scheduling and monitoring data pipelines. Maintainability is not an easy thing to measure and can be quite subjective, but the objective is to write readable code without unnecessary complexity.

In terms of scalability, a bigger concern than the number of users is the capability of the application to handle more data. Each week since the start of the project, more metadata has been added. This could create problems which did not exist at the current development stage. Therefore, we needed to test for large data changes in our integration tests and monitor the speed of API closely. An API request with a response time of 2 seconds might be good enough at the current size, but not in the future.

2.3.4 Scalability

"Scalability is the ability to handle an increase in the workload without impacting the performance, or the ability to quickly expand the architecture" (Paradkar, 2017). In our project this pertains to how many users our application will need to support and how large manifest file we need to process. At the time of release the user-base is only going to be about five data scientists. As the system grows and other departments within Intility start using it we need to account for the system to handle two hundred users. Read more about how the system can handle this increase in load in chapter 5.1.2 OpenShift and Microsoft Azure.

The other aspect is running a large and potentially slow update as the manifest continues to grow. As we conclude our project the manifest size has passed two hundred thousand lines. After talks with the data science department they have specified it would be nice to be able to handle twice this size to ensure continued growth. How we ensured that our application could handle these scalability demands is covered in the end of chapter 6.1.2 Integration testing.

2.3.5 Security

This application's purpose is as an internal business tool for employees and potentially customers. Users outside of these constraints should not be able to use the application. To solve this issue, the system will use user-authentication for both the front-end and the API, described in chapter 5.1.2 OpenShift and Microsoft Azure.

The authentication in our application happens through bearer-tokens, which is light-weight in that it doesn't require cryptographic signing of each request, making responses faster.

The report will cover specific security measures in the relevant chapters.

3 Design

The design of any software system plays a critical role in its success. The graphical user interface (GUI) serves as the primary interaction point between the user and the system, and therefore plays an important role in the user experience. In this chapter, we present the design of our application, focusing on both the GUI and the system design. Our goal was to create a user-friendly and intuitive interface that allows users to easily navigate and interact with the application, as well as a clear system design for easier understanding when other developers continue the work.

3.1 GUI

The design of the Data Platform Manager utilizes Intility's React library Bifrost (Intility, n.d.), which offers a wide range of pre-built components and utilities that allow for the creation of responsive and scalable web applications. One key advantage of this library is its built-in coloring system, which additionally to all the components adhere to the WCAG guidelines. This makes Bifrost the ideal choice for Intility's developers, as it eliminates the need to follow multiple guidelines when developing an application, unlike other libraries like React Bootstrap. Moreover, since all applications and websites of Intility share the same styling foundation, it makes sense to use Bifrost in this case to ensure consistency with the existing applications and websites within the organization. By utilizing a more familiar and widely used library within the company, the development process of the Data Platform Manager becomes more streamlined and efficient.

The front-end design for the data platform manager follows a standard layout that is used to create a consistent look and feel for all Intility web apps (Intility, 2023). The layout consists of a top-bar, navigation-bar on the left and a main body with content. They are set up as follows:

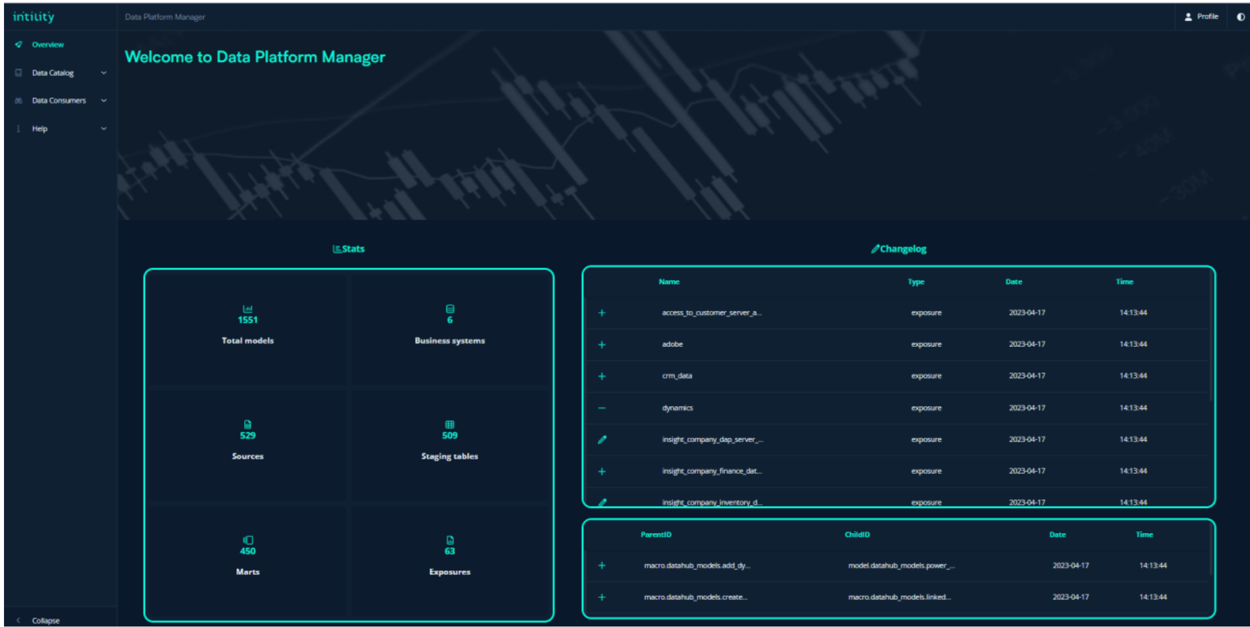


Figure 3.1 Standard website layout

Top bar: The top bar includes an Intility logo, name of the application, and on the right in the top bar, there are two buttons. One button is for the user's profile and the other to change between light and dark mode.

Left side navigation-bar: The left side navigation-bar has four main categories, overview, data catalog, data consumers and help. Overview links to the main page, the others have under-categories that link to the specific content.

Main body: The main body displays the content depending on which page that is navigated to.

The general look of our application's GUI is based on DBT docs. As described in chapter 1.1.2 The current solution – DBT and DBT docs, which is the current tool the data scientists are using today. Below you can see an image of how a regular information page looks like inside DBT docs.

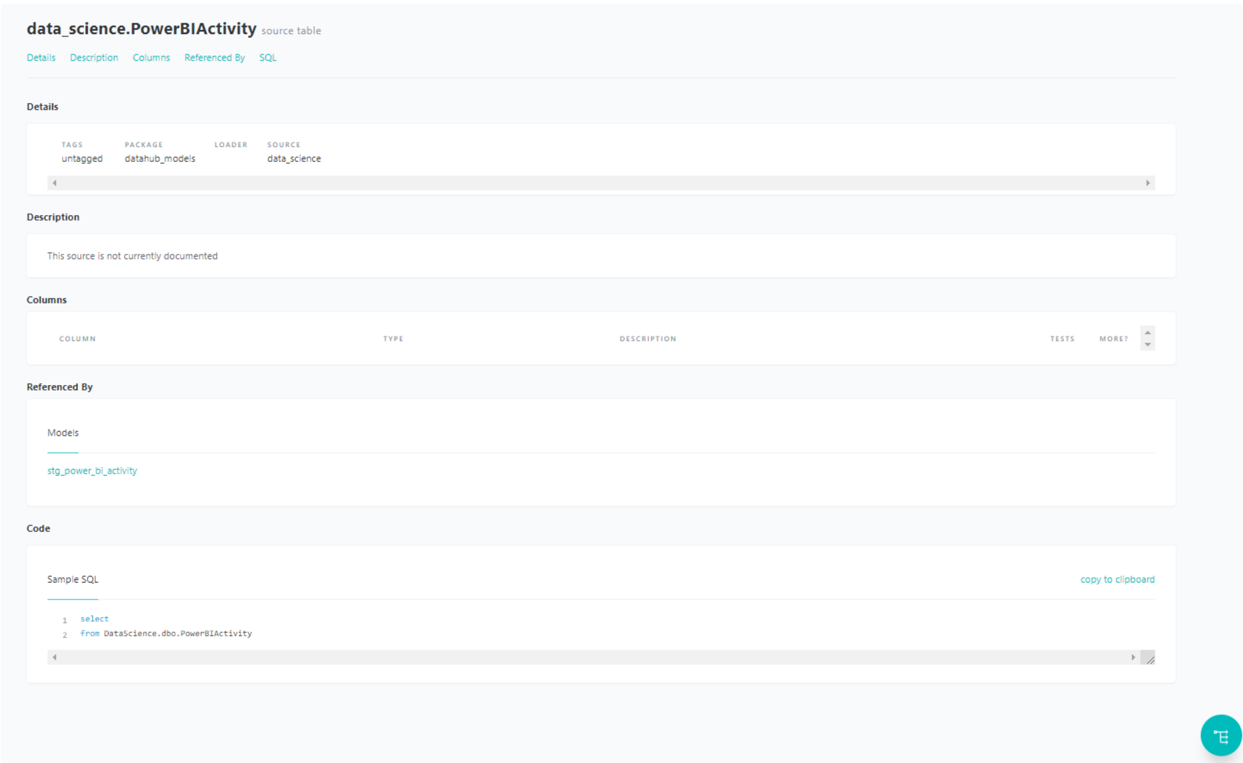


Figure 3.2 Example of DBT docs information page

To ensure a good user experience, the website is designed to be user-friendly and intuitive, utilizing font sizes and icons that follow WCAG guidelines. Icons are made and used to be easily distinguishable. Another example is when the website view is at a width below a certain threshold, the left side navigation bar will be hidden and replaced with a dropdown menu which can be seen in Figure 3.3. This will make it more appealing for users with smaller screens so that they will be able to see the main content just as good as those with a larger monitor.

Lastly, the sidebar contains dropdown menus for each of the main categories that can easily be opened. With this feature, users can find the information they need without having to spend too much time searching for it, making their browsing experience more enjoyable. Overall, the application uses a minimalistic design that eliminates clutter and distraction making it user-friendly and simple to navigate.

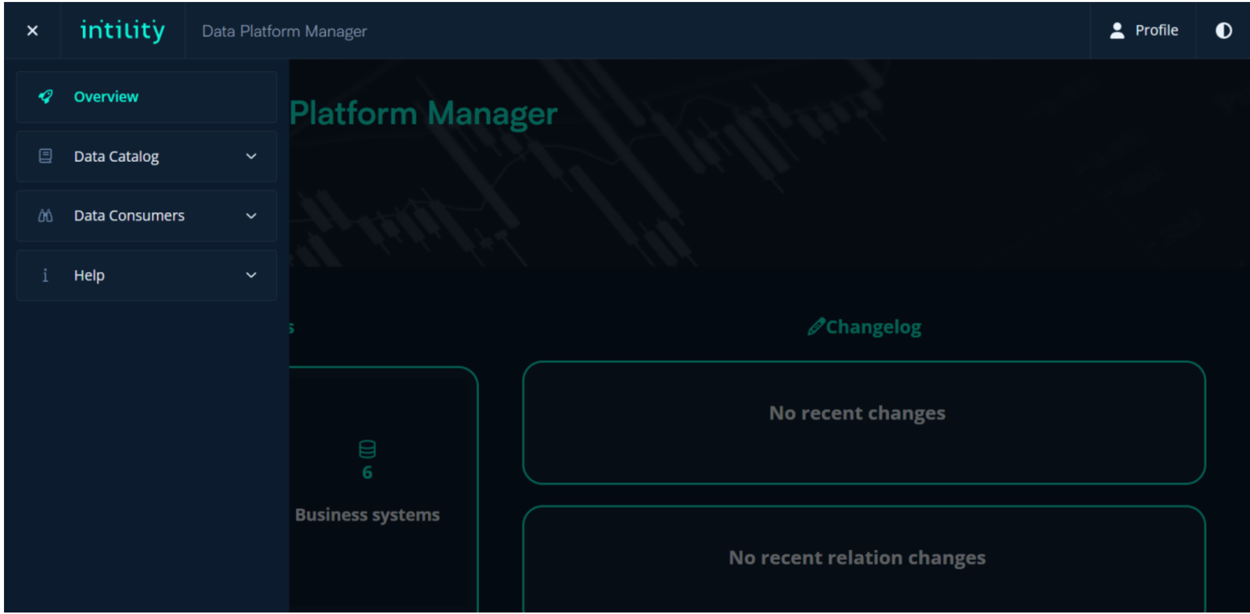


Figure 3.3 Example of dropdown menu for smaller screens

As mentioned earlier the data platform manager adheres to several WCAG guidelines to ensure that it is accessible to a wide range of users, including those with disabilities, and offers a good user experience. WCAG 2.2 Guideline 1.4 is met using font sizes and icons that adhere to the guidelines, ensuring that users can easily read and comprehend the content of the website. WCAG 2.2 Guideline 1.3 is met by providing alternatives for non-text content using easily distinguishable icons, making the website accessible to users with visual impairments. The built-in coloring system of Bifrost adheres to WCAG 2.2 Guideline 1.4.3, ensuring that the color contrast between text and background meets the minimum requirements, making it easy to read for everyone.

The website structure meets the WCAG 2.2 Guideline 1.3.1 by organizing the content into a top bar, left-side navigation bar, and a main body. This ensures that users can easily understand the structure of the website, find information they need, and navigate efficiently. WCAG 2.2 Guideline 1.4.11 is met using components with contrasting colors, ensuring that users with color blindness can easily distinguish between them. Finally, WCAG 2.2 Guideline 1.4.10 the left-side navigation bar is hidden into a dropdown menu on smaller screens making it so that users do not have to scroll or zoom in to view the content.

3.2 Prototyping with Figma

At the start of the development process, we created sketches for the application, these can be seen in Appendix H. We used the knowledge taught to us in the Figma Bifrost workshop mentioned in chapter 8.3.2. Our product owner and data science expert at Intility reviewed our sketches. They had some comments to our prototypes and some thoughts for improvements. They let us know they would implement some changes to the sketches before we started developing the application. After the user test mentioned in 7 Methodology – Usability Testing, we found out that the navigation bar in our suggestions (see Figure 8.5 for an example) would be preferred over the solution in the final sketches received from the data science team.

Changes of plans occur in development processes constantly, and we would have added the navigation feature to the application if we had more time.

3.3 System design

In the Gitlab repository, there are two folders, one for front-end and the other for back-end. Our group members had access to both folders to work on both back-end and front-end. It was important to have a good structure of both the front-end and back-end folder. The reason behind this is there is someone else at Intility that will take over the project after this bachelor's degree.

Looking at the structure of other Intility applications was important to make this process as easy as possible for the next development team. The Figure 3.4 System architecture below shows how the architecture of the system from the user to the deployment and database.

3.3.1 Back-end system design

The back-end is designed as one solution containing three projects as shown in Figure 3.4. Two of the projects are: “UnitTests” and “IntegrationTests” which contain all test functionality of our application. We created separate projects for testing as it is more organized and isolates the tests from our application. This in turn creates smaller and more efficient executables. The project “dpm-backend” is our main project, and the rest of the chapter will be describing how it is organized and designed.

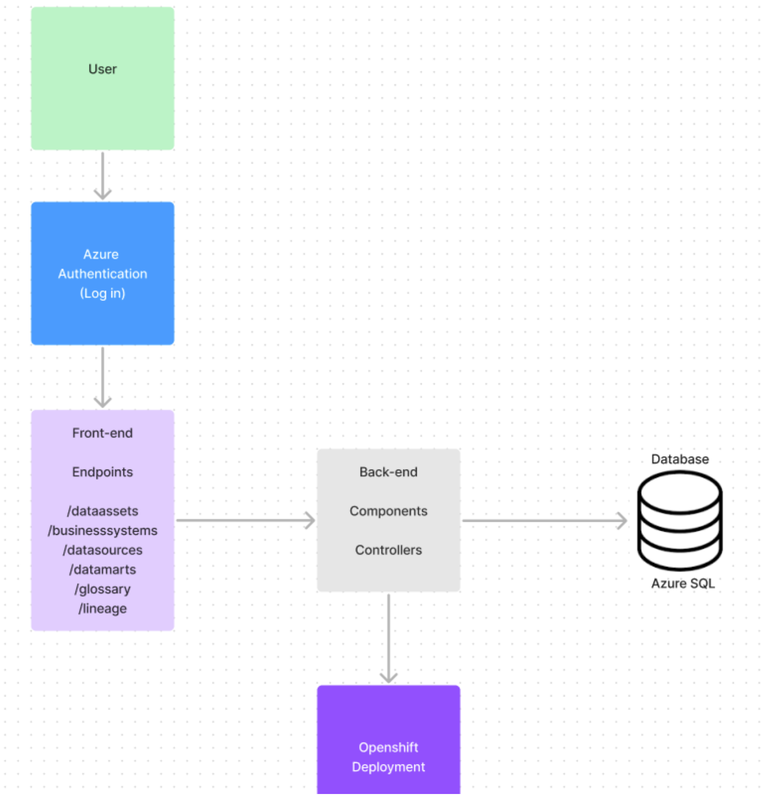


Figure 3.4 System architecture

The back-end is designed as a .NET API aiming to meet the standards described in chapter 5.2 REST API. The API is organized in two main layers: Controllers which handles incoming requests and responses, and Data which contain program logic and database functionality. The controllers are only responsible for handling HTTP requests and using dependencies from the data layer to fetch and update data. Figure 3.6 shows how each layer works in unison.

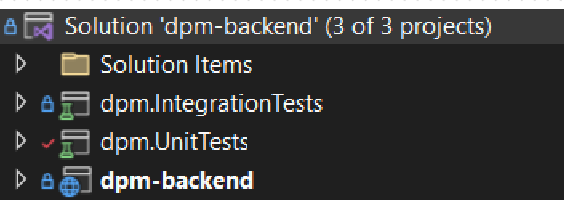


Figure 3.5 Back-end solution

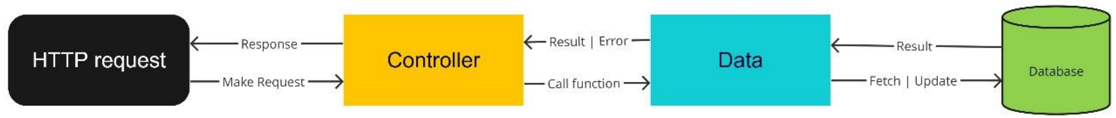


Figure 3.6 Data flow through back-end API

To achieve the workflow described in Figure 3.6 we used dependency injection and the architectural design patterns described by Microsoft for architectural principles (Microsoft, 2022). These patterns are described as "*separation of concerns* and *dependency injection".* Separation of concerns is achieved through separating domains as discussed earlier in this chapter. Dependency injection is done within the controller which injects the data layer, and the data layer which injects the database context. The injected dependencies are scoped to have the same lifetime within each layer, being destroyed when the outer object is no longer useful and destroyed.

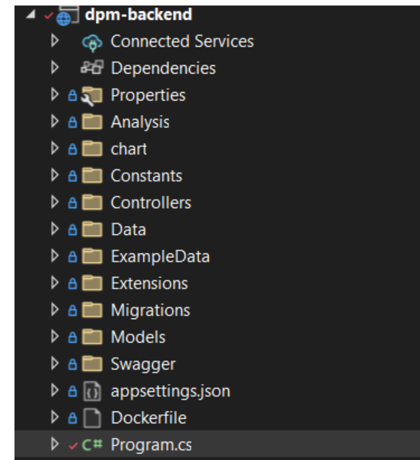


Figure 3.7 API folder structure

As Figure 3.7 shows, the Data folder is separate from Controller layer. Another folder to make note of is the analysis folder which is responsible for reading the manifest into C# classes.

The "models" folder contains all our C# models and interfaces and are inside subfolders to make the 36 objects organized. These classes are used by our database and data layer to send or translate information. See Figure 3.8.

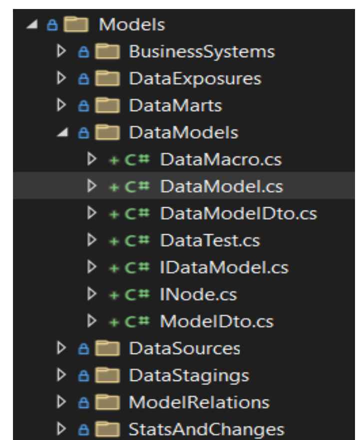


Figure 3.8 C# Models folder

Controllers

Controllers
V1
BusinessGlossaryController.cs
BusinessSystemsController.cs
DataAssetsController.cs
DataExposuresController.cs
DataMartsController.cs
DataSourcesController.cs
DataStagingsController.cs
ManifestController.cs
RelationsController.cs
StatsAndChangesController.cs

Figure 3.9 Controllers folder

The controllers are responsible for separate domains and aim to be quite small with few lines of code. This makes the API modularized and scalable. As the API grows, new controllers can be added, or new endpoints can be added to controllers where they fit. This also helped during development as we avoided working on the same files and the typical difficulties that brings with Gitlab.

Data

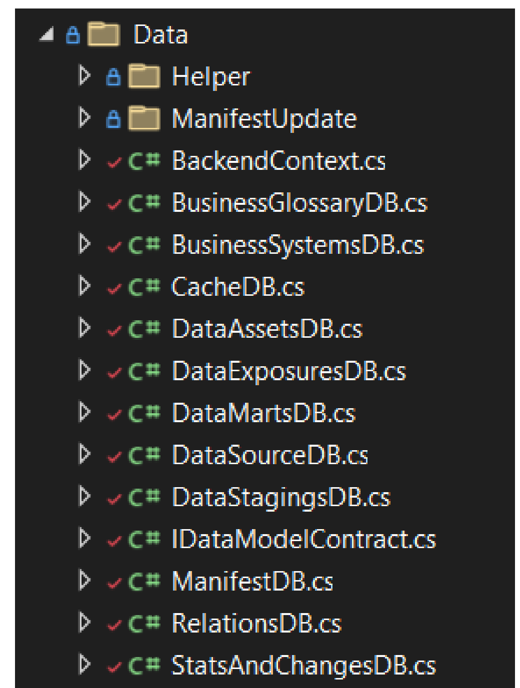


Figure 3.10 Data folder

The data folder is where all functions that are used in the controllers are located. There are functions to handle functionality related to updating and fetching information from the database. These functions handle the logic when a request is sent from front-end. There are also some functions which check the input from front-end before the database is populated with latest information. It is important to check for invalid input before the database is populated, even though there are checks for valid information front-end. This is to prevent SQL-injection if the user is not sending a request directly from front-end, but with the use of tool like Swagger or Postman.

Figure 3.10 also includes a “ManifestUpdate” folder which is responsible for updating the database with manifest information. Within this folder there is also a static class with help-functions commonly used to filter and manage data.

```
47 references
public static class Constants
{
    // Constants used in DPM system
    // Data Models:
    4 references
    public static string Source => "source";
    4 references
    public static string Staging => "staging";
    5 references
    public static string Mart => "mart";
    4 references
    public static string Test => "test";
    4 references
    public static string Exposure => "exposure";
    4 references
    public static string Macro => "macro";

    1 reference
    public static string BusinessSystem => "businessSystem";

    2 references
    public static string Node => "node";
    1 reference
    public static List<string> NodeTypes => new() { Staging, Mart, Test };

    // Time configuration:
    public const string TimeDisplayChangeLog = "TimeDisplayChangeLog";
    public const string TimeDisplayChangeLogDescription = "Configures the timespan to display changes in changelog";

    public const string TimeDeleteChangeLog = "TimeDeleteChangeLog";
    public const string TimeDeleteChangeLogDescription = "Configures at which age changes should be deleted";


    // Constants used in DBT manifest
    public const string ManifestStaging = "view";
    public const string ManifestMart = "table";
    public const string ManifestMartIncremental = "incremental";
    public const string ManifestTest = "test";
}
```

Figure 3.11 C# Constants

Figure 3.7 from earlier shows a constants folder which includes configurations used across the program. Figure 3.11 above shows the main Constants class and on the top left you can see it is referenced 47 places, meaning it is widely used within the application. Having this file to configure reoccurring values lets us avoid hard coding values and helps maintainability as the program changes.

3.3.2 Front-end system design

Front-end is organized using a modular folder structure which includes several folders organizing the different types of files used in the application.

One of these directories is the “Assets” folder, which houses static files such as images and logos used throughout the application.

Another folder is the “Auth” folder, which contains all files related to authentication, such as login and authorization of users. It also includes a function for fetching data from the API with extra security authentication.

One of the most important folders in the front-end application is the “Components” folder. This folder stores reusable UI components which are used across multiple pages and features. The use of reusable components helps to ensure consistency in the user interface and saves development time by avoiding the need to build the same UI elements from scratch for each new feature or page.

The components in the “Components” folder are organized according to how the sidebar of the website is listed. For example, the “Data_Catalog” folder holds all components related to that category such as “Data_Assets”. This organization is done to improve code organization and maintainability.

In addition to the component directory, there is a “Pages” folder that consists of the actual pages of the application, which may be composed of multiple components. This folder helps to keep the pages organized and easy to find.

Furthermore, there are three more folders: “Tests”, “Types” and “Utils”. The "Tests" folder contains files related to automated testing of the application. Tests are important for ensuring that the application works correctly and meets the desired requirements. The "Types" folder contains files that define and standardize the types and interfaces used throughout the application. One example is types for the API responses. The "Utils" folder contains utility related functions and configuration files.

Finally, there are files related to the setup and initialization of certain things, such as routing and test setup. These files are crucial for ensuring that the application is properly configured and set up before it is run.

3.4 Easy-to-use

This chapter focuses on the principles we have implemented to make our data platform manager application easy-to-use for a wider audience. In order to make our application easy-to-use, we have followed certain principles. In this chapter, we will talk about some well-known principles we have implemented during the process of developing an easy-to-use application. The principles we have followed are listed below.

The first principle we followed for this application is to keep it simple. We have aimed to create an application with a simple design and without too much different layers for the functionalities. This is to have a simple user interface and to avoid unnecessary complexity. See chapter 3.1 about the design of the application.

When developing the application, we wanted to create it as intuitive and accessible as possible. Buttons have been placed in corners or in a suitable position. The buttons also have a text, saying for example "click for more details" in order to make it accessible (Mozilla, n.d.). The user test discovered some aspects of the application, this is discussed further in chapter 7 about the user test.

The user of the application is provided with clear instructions if their actions are prohibited. At the Business Glossary page, the user gets an error message in the bottom right corner if the word already exists in the application. See Figure 3.12 for the screenshot. The user also gets an error message if the input fields are left blank and is then asked to fill in a word and a description. See Figure 3.13 for a screenshot of this.

If the user fails to fill in input fields when adding a new business system, an error message will appear indicating the mistake. The user is then asked to fill in with acceptable characters to submit the newly create Business System. See Figure 3.14 for the screenshot of this.

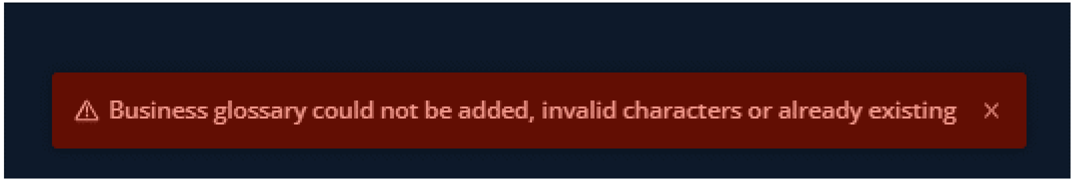


Figure 3.12 Custom error message popups when user makes mistakes adding glossary words

Figure 3.13 Custom error message if the input fields are left blank

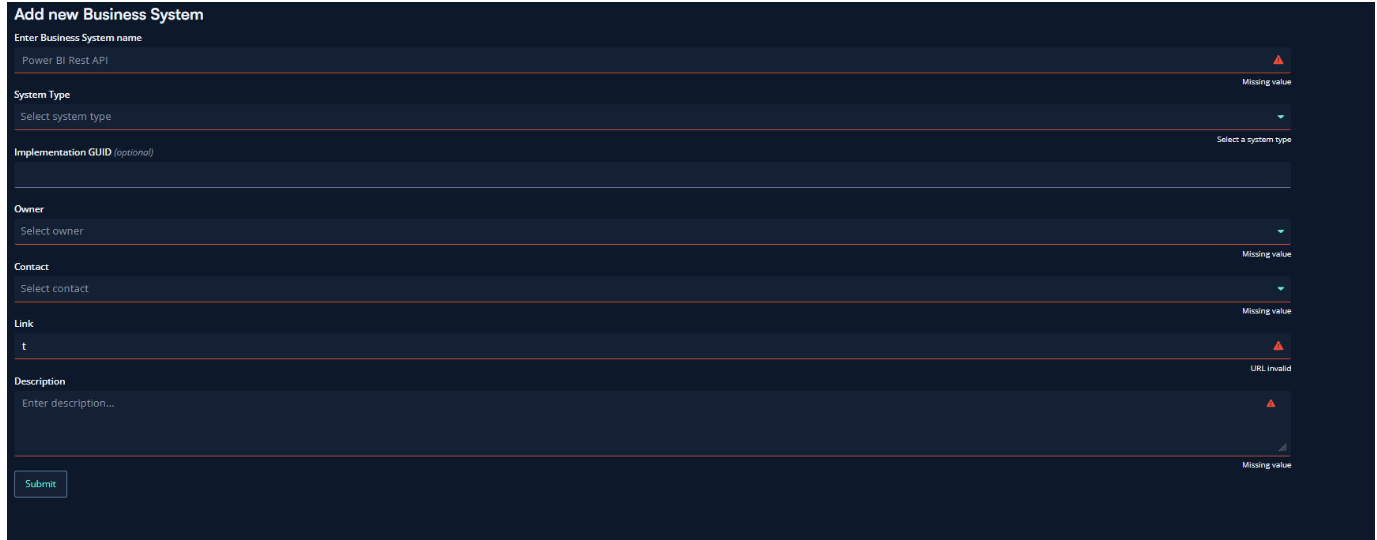


Figure 3.14 Warnings updating live when user inputs information in 'Add business system' fields

The last principle we have followed for this application, is not only related to this application. Our users of the application are going to be the employees at Intility; therefore, it is important that the application follow a consistent design and layout. This is also referred to as “The Intility Standard”, which can be read more about in the glossary list.

4 Technologies

The development team chose to use the standard Intility technologies and tools. There are several reasons for this which will be explained in this chapter. The main reason is that we wanted to create a product that will be useful and maintainable for a company like Intility. It was also important to ensure that further development of the product is possible with the knowledge of their in-house developers.

4.1 Back-end

At the start of this project, the development team chose to use .NET for the back-end development which is a collection of technologies created by Microsoft (Microsoft, n.d.). .NET provides great support to create web APIs with "Asp Net Core" (Daniel Roth, 2022).
There were no problems connecting the back-end to the front-end and the database in Azure. In one of our previous courses called "Integration Project", we struggled to connect the front-end and back-end, with HTTPS protocol. This was not an issue that we encountered during this project due to better choice of back-end technologies and the integration with Azure.

Another reason to use .NET, is how easily it can be integrated with Swagger which we have used to manually test and debug our endpoints. Figure 5.4 from chapter 5 shows an example of how a controller for the endpoints look like. To use swagger, we had to authenticate with our Intility account. The back-end swagger was running at all times with the code pushed to Gitlab, but we could also run the back-end locally and check if the code worked as intended with localhost swagger. This was to ensure that the code pushed to Gitlab could run without any problems (Razvalinov, u.d.).

The current solution DBT docs sometimes uses a long time to load because of the large amounts of unorganized information in the front-end. Therefore, it is important to choose a programming language which runs fast and efficiently (Bhadwal, 2023). This is the reason we chose to use C#, a statically compiled language. As mentioned in chapter 1.6.1 and 1.6.2, our goals was to decrease the loading time for the data documentation pages and for the lineage

graph. A possible solution to this was a fast back-end which could cache large amounts of data ensuring that the load time will be faster after the first time opening the website.

4.1.1 Alternatives to .NET and C#:

Python is an alternative to C# that we could have used. Some teams within Intility are using python, but compared to C#, python is as good at handling high-performance computing, and it is not the best language for processing substantial amounts of data (Invedus, 2022).

Golang is a familiar language to C#, but it is a simpler language with fewer in-built features compared to .NET (cshark, 2022). Features like LINQ and pattern matching are not supported in Golang. LINQ is a fantastic addition to a language which supplies a consistent query experience to relational databases (TutorialsTeacher, u.d.).

Java is one of the most widely used programming languages for development, ranking first within statically, type-strict languages (Berkely edu., n.d.). There are a lot of reasons why Java is well suited for the software industry. For instance, Java can run on any operating system with its appropriate compilers and Java Runtime Environment. That also applies to .NET which can also be cross-compiled. However, to integrate with Microsoft Azure and Intility as a Microsoft based business, we decided to use .NET.

A reason to use on .NET over Java is because the development team is more familiar with Visual Studio as an IDE compared to IntelliJ or other IDEs. This is not the main reason, because most IDEs are similar, but we found it to be advantageous to go with a familiar IDE. This decision will be discussed further in chapter 4.1.3.

All the three languages are exceptional for web and cross-platform development, but we did not find any other reason to why we should deviate from the standard Intility back-end language (Vats, 2022).

4.1.2 Manually testing the endpoints

In order to test and develop the back-end endpoints the development team used swagger. The advantage of Swagger is that you can quickly test the API as it creates documentation, example requests and data from the code. In the development phase, this is a useful tool to keep the

cost of change low. The development team could also have used Postman for the manual testing of the endpoints, but Swagger is well integrated with the verification tokens (bearer tokens) which are needed to access the API.

4.1.3 Back-end IDE

The IDEs used for this project are both Visual Studio Code and Visual Studio 2022. The reason behind this is different preferences within our group. There are several purposes with using Visual Studio 2022 and Visual Studio Code instead of examples like IntelliJ and JetBrains Rider. The main reason is that the development team is more familiar with Visual Studio.

4.1.4 Back-end testing tool

XUnit

We decided to use XUnit as our testing tool because it is lightweight and easy to get started with compared to other tools such as NUnit and MSTest (Sheth, 2021). The differences between them are negligible, but "Unit's popularity and the fact that it is open source ensures higher probability of long-term support and documentation. We utilized XUnit for both our unit tests and our integration tests.

Moq

Since many of the components and classes are interdependent it is hard to truly isolate and unit test functionality. To help with this we utilized Moq, a library which helps mock classes and entities. "The purpose of mocking is to isolate and focus on the code being tested and not the behavior or state of external dependencies" (Telerik, n.d.). Moq's lets us achieve this by simulating functionality of dependencies and returning valid results. It does not guarantee or test any of the dependencies. However, it allows us to test functions in isolation and test smaller units of code.

Moq also provides the functionality to mock "DbContexts" from EntityFramework which is the ORM system we're using. This allows us to test functions working directly with a database using simple in-memory lists as shown in Code snippet 6.6.

```
[Fact]
0 references
public void DataMartDBTest_GetDataMarts_Empty()
{
    // Arrange
    // Arrange DataModel and DataMart

    var mockContext = new Mock<BackendContext>();
    mockContext.Setup(c => c.Models).ReturnsDbSet(new List<DataModel>());
    mockContext.Setup(c => c.Marts).ReturnsDbSet(new List<DataMart>());
    var dataMartsDB = new DataMartsDB(mockContext.Object);

    // Act
    var result = dataMartsDB.GetModels();

    // Assert
    Assert.Empty(result);
    Assert.IsAssignableFrom<IEnumerable<MartMiniDto>>(result);
}
```

Figure 4.1 Mocking "DbContext" with Moq

4.1.5 Database tool

The database tool used for this assignment is Microsoft SQL Server Management Studio 18, which is a graphical integrated development environment used to administrate and manage a Microsoft SQL Server. By using this tool, we could create tables within the code and use the query editor in SSMS 18 to write and execute SQL queries to see that the information in the database was correct. The tool was also used in the early stages of the development process, to find sources, for example, mart, in order to check if our endpoint for finding all information about a unique mart worked.

4.2 Front-end

The front-end tools used for this project were Typescript with React. The main reason behind this combination was Intility's preference for these tools. In the two following chapters, we will discuss why the chosen technologies did not deviate from the standard Intility tools and compare them with alternative options.

4.2.1 Typescript vs JavaScript

In the development of our project, we utilized TypeScript, a superset of JavaScript, which is strongly typed and adds additional features to the language. Although we could have used JavaScript for the same purpose, Intility recommended TypeScript due to its numerous benefits. TypeScript improves code readability and maintainability, enhances error-checking and debugging capabilities, and increases code scalability. With TypeScript, the code is easier to read and understand as the type definitions make it clear what data types are being used. Furthermore, TypeScript catches errors before the code is run, simplifying debugging, and reducing the time spent fixing issues. TypeScript also scales better for larger projects, as it allows for improved code organization and modularization. Overall, the use of TypeScript was a wise decision as it enabled us to develop a more robust and maintainable codebase.

4.2.2 React vs Vue

During a previous course "Integration Project", we developed an application where Vue.js was chosen as the framework for the application. Both Vue.js and React are excellent frameworks for building web applications and have a many similarities, but we ultimately decided to use

React for this project. They both support large-scale application, have typescript support and are both very similar in performance. Vue does however have slightly better performance as seen in Figure 4.2 (Borozenets, 2022). As they are so similar in performance, we could not choose based on it.

Name	Slowdown Geometric Mean
vue-v2.5.16-keyed	1.41
react-v16.4.1-keyed	1.54

Figure 4.2 Performance test for React vs Vue.js. (Borozenets, 2022)

However, some differences between the two led us to choose React for this project. One factor we considered was React's popularity and large developer community both within Intility and in general, and gaining more knowledge within such a popular framework could prove beneficial later in our careers. Intility provided us with a React course and assistance when needed. Therefore, since it is the standard framework within the company, we found no reason to deviate from the standard. Another reason why we chose React is its flexibility and modularity, which allows us to easily integrate it with other libraries and tools as needed. The pre-built Bifrost library available in React also drove us in this direction. Overall, the decision to use React for this project was based on a careful evaluation of its strengths and our projects requirements.

4.2.3 Front-end testing tool

We used the front-end testing tool Vitest. It is a newer, open-source testing library for React components and functions, which includes the Jest framework and simplifies testing. We mainly used it for unit testing, but it also provides multiple other types of tests such as integration testing and end-to-end testing. Overall, it was easy to learn, and helped us find bugs, and improve some of our functions.

4.2.4 Bifrost

Bifrost is an Intility design system which is a useful tool for developers to quickly develop new projects. All Intility websites use this design system in order to make the interface of the sites reflect the brand of Intility. The Bifrost library consists of different models and examples for front-end development. The front-end team can refer to the Bifrost documentation and use various already created components in order to develop their own website. However, the models from Bifrost are not tailored to each project, so in some cases, modifications to the existing models are needed.

4.3 Communication and helper tools

In the following chapters, we will list the various tools employed in this project. While not utilized directly in the programming aspect of the project, these tools were leveraged for cloud storage, version control, and intra-group communication, in addition to communication with Intility.

4.3.1 Azure

Our application is hosted in Azure. The cloud service is owned by Microsoft, and it is a public cloud computing platform. Even though Azure provides a lot of services, we have only used the storage and networking services.

Alternatives to using a cloud service would be to host the application on a local server, which was very popular in the development industry some years ago. There are several reasons to use a cloud service instead of a local server. To meet our requirements in chapter 2.3, it is important that we use a cloud service which is scalable, reliable and accessible. Having a local server is costly. With the cloud services, it is possible to only pay for the resources needed, rather than having to have the resources in-house for worst-case scenarios. This may result in lower server infrastructure and maintenance costs.

4.3.2 Gitlab

Gitlab has been the version control tool for this project. We have also worked with similar tools like GitHub before. GitHub is also a great tool for version control, but our group decide to use Gitlab for this project. At the beginning of the project, we set up a pipeline for both the front-

end and back-end projects and preset some rules on Gitlab so that it is not possible to commit to the main branch.

At the start of the project, we had a Gitlab workshop with an employee in Intility, which taught us to set up some necessary configurations like CI/CD for the pipelines. It was a useful workshop where we learned more about best practices with naming conventions, usage of different branches and merging. More information on the training period can be found in chapter 8.3 Training period.

4.3.3 Communication and documentation tools

Teams

Teams have been the main platform for meetings and communication with both Intility and the bachelor guidance counselor. There are several other options like Google meet, but Teams is what the development team is the most familiar with and it provides a lot of different functionality for scheduling meetings and asking questions to other employees in Intility. Slack is also a tool used to contact other employees about questions related to the project.

Discord

Discord has been the main platform for communication within the group. In Discord, it is possible to create a Discord server, with different text channels for various categories. This allows for storage of links, meeting minutes and other important material related to the bachelor project.

Jira

The development team had a biweekly sprint period where they had several issues in the sprint. Initially, our group attempted to use issue tracking in GitLab, which is the standard issue tracking tool within Intility. However, we found it easier to keep track of their issues using Jira instead of GitLab. The backlog is another feature provided in Jira, which is a list of issues where the developer could select additional issues if there was any time left. Jira is an exceptional tool for this purpose.

5 Implementation

The implementation chapter of our thesis details how we created the different parts of our application. Here, we will describe the process we followed to develop and integrate the front-end, back-end, and database components. We will explain how we implemented each aspect of the application, as well as the challenges we faced and how we overcame them. By presenting our implementation approach and decisions, we aim to provide insights into our development process and demonstrate the feasibility of our solution.

We have created a front-end to handle the presentation, and a back-end to handle the business logic. This is different to DBT Docs' solution, as mentioned in "The current solution – DBT and DBT docs". The reason for this is to separate the two layers, which allows us to develop and maintain the layers independently and balance the workload of each layer.

5.1 Deployment

Continuous deployment of the application is important to ensure collaboration within the team and to ensure accountability that our front-end and back-end API communicates and works successfully. Therefore, we wanted to deploy our application continuously as we made changes.

Deployment options are complex and numerous with choices such as self-hosting, cloud hosting and which container provider to use. We wanted to use tools which supported our application, but without too much upfront cost in terms of development time. Therefore, the choices have been made based on our previous experience and the experience of our technical contact point.

The pipeline to deploy the application was created early in the development process. Since the company is Microsoft based, we decided to host on OpenShift through Microsoft Azure. This was advantageous to handle user authentication, and CI/CD which will be discussed in the following chapter.

5.1.1 Continuous Integration and Continuous Deployment

Gitlab offers integrated CI/CD which we can use to automate building, testing and the deployment process. This is done through a "*gitlab-ci.yml*" file which defines the stages in the pipeline (Gitlab, n.d.). For our back-end these stages were: build, test, image and deploy. The *Build* stage compiles our application with a .NET SDK to a runnable .exe file. This stage will fail if the build fails and the pipeline will be cancelled, allowing us extra safety that the deployed application is working. The *testing* stage runs all tests in the solution, which includes tests from the unit test and integration projects as discussed in chapter 6 Testing.

This is valuable since it assures us that the changes made hasn't broken functionality from the tests. The *image* stage creates a docker image based on the "Dockerfile.CI". The final stage is called *deploy*, and its function is to log onto the OpenShift server and use helm to create or configure and install a Kubernetes application.

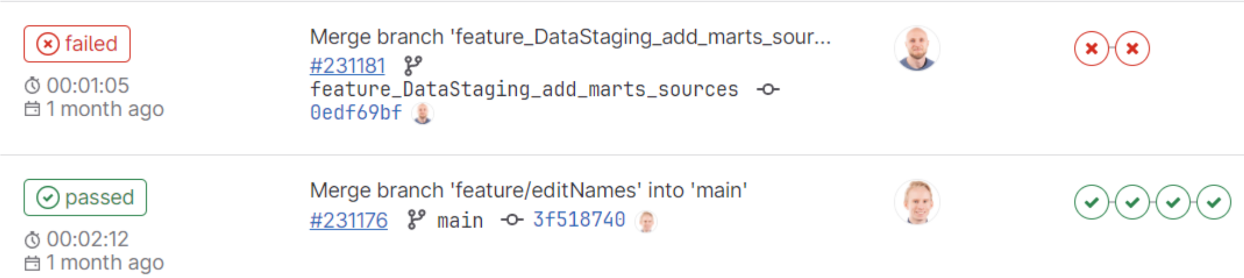


Figure 5.1 Gitlab example, pipeline failure and success

Figure 5.1 shows an example of a pipeline failing when a change is pushed to Gitlab. The pipeline runs each time a change is pushed to Gitlab and gives quick feedback with a status "*failed*" when a stage fails to complete. This means that within a couple of minutes we have feedback whether the branch is ready to merge with the main branch and be deployed.

5.1.2 OpenShift and Microsoft Azure

Running the application in OpenShift as a Kubernetes cluster is advantageous as the Kubernetes load balancer can scale our application by increasing the of pod count (AVINetworks, n.d.). "A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers" (Kubernetes, 2023). Since the load balancer scales

the number of pods as our front-end or back-end requires more compute power, we handle the requirements specified earlier in chapter 0 Scalability.

Secrets

The scripts contain many secret variables, such as the OpenShift token used to login to OpenShift from Gitlab. These secrets need to be added as a variable in the Gitlab settings. Storing variables in Gitlab has the advantage of being flexible, giving us higher cohesion in being able to apply changes in one place instead of in each script. More importantly it is a secure environment with encryption which ensures sensitive data remains confidential. Other secrets such as connection string used to connect the application to the database needs to be added to OpenShift as a secret. This is because it is used at application runtime rather than compilation and needs to be available to the container.

Microsoft Azure

We also had to register our application in Microsoft Azure which works as a proxy and authenticator. A crucial setting to keep non-Intility employees from using the application is shown below in Figure 5.2. This setting sets the application to be a *single organizational application,* and allows us to open the application strictly to authorized Intility users, which again helps us comply with the requirements specified in chapter 2.3.5 Security.

Supported account types

Who can use this application or access this API?

- (•) Accounts in this organizational directory only (Intility AS only - Single tenant)
- ( ) Accounts in any organizational directory (Any Azure AD directory - Multitenant)

Figure 5.2 Microsoft Azure API access

Conclusion

This chapter has not gone into depth onto the different technologies used, as it would be disingenuous to take credit for work heavily helped by templates and colleagues. Therefore, we

have focused on broadly describing the deployment and the configurations we made specific to our application.

We have no previous experience with CI/CD from previous subjects or projects, and it was truly eye-opening how helpful and productive it made us. Previously we have experienced the difficulty of manually configuring two separate front-end and back-end servers. Testing which normally is only run after creation or specific changes is now run automatically and prevents regression.

5.2 REST API

The API is designed as a REST (Representational State Transfer) HTTP (Hypertext Transfer Protocol) service and works as an intermediary step between the underlying resources and the front-end client. REST is a set of standards that specify certain principles the API should follow.

5.2.1 HTTP and REST Principles

An important REST principle is the *uniform interface* principle which states that each resource should be uniquely identified (CloudFlare, n.d.). Below you can see an example of how Data Marts and Data Exposures are identified by different URIs (Uniform Resource Identifier).

Figure 5.3 Resource URIs from the Swagger tool

The HTTP methods on the left of the image describe the CRUD (Create, Read, Update, Delete) operations available for every given resource. Designing the API in this way makes it easier to comply with the requirements covered in the use-case diagram chapter 2.2.1 and security chapter 2.3.5. Since all *regular-user* interactions are read-only operations we can design our application with this in mind and make non-read-only operations available to certain authenticated users. You can read more about this in the authentication chapter 5.3.

DataSources

Method	Path	Description
GET	/api/v1/DataSources	Method `GetDataSources` Fetch all dpm.backend.Models.DataSource from DB
PUT	/api/v1/DataSources	Method `AddBusinessSystemSource` Adds the SourceDto to DB
POST	/api/v1/DataSources	Method `AddBusinessSystemSources` Adds the sourceBusinessLinksDtos to DB
GET	/api/v1/DataSources/{id}	Method `GetDataSource` Retrieve a single data source with its business system information

Figure 5.4 "Data Sources" resource from swagger API

In the image above you can see that all the URIs for a given resource are the same, but the usage is different. The first GET request is an API call to fetch all data sources from the system. The second is a PUT request to update a data source which will update the data source's overlying business system. Next, the third is a POST request which adds multiple data sources to a specific business system. Finally, the last GET request takes in a path parameter at the end of the URI to specify the given data source you wish to fetch.

Another important principle is separation of concerns between the client (API interface) and the server (underlying data storage). The objective of our API is to provide data for our front-end to display, and this may change without the underlying data storage changing. There could be a need to bring in *additional* information from the database or to process data already stored which would change the responses. But by having the API as a mediator between the database and the interface, we can make changes faster and less costly. Below in Code Snippet

5.1 there is an example of a DTO (Data Transfer Object) used to send data through the API which separates our data storage from the API interface.

```
/// <summary>
/// Class <c>MartDto</c> is data tranfer object used in the API
/// </summary>
public class MartDto : IDataModel
{
    // General Data Model attributes
    public string UniqueId { get; set; } = "";
    public string Name { get; set; } = "";
    public string Description { get; set; } = "";
    public string PackageName { get; set; } = "";

    // Specific Mart attributes
    public string Language { get; set; } = "";
    public string CheckSum { get; set; } = "";
    public string Database { get; set; } = "";
    public string Schema { get; set; } = "";
    public string RawCode { get; set; } = "";
    public string CompiledCode { get; set; } = "";
    public string Relation { get; set; } = "";
    // Referenced by
    public List<string> Tests { get; set; } = new();
    public List<string> Stagings { get; set; } = new();
    public List<string> Exposures { get; set; } = new();
    public List<string> DependsOnMarts { get; set; } = new();
    public List<string> ReferencedByMarts { get; set; } = new();

    // Depends on
    public List<string> Macros { get; set; } = new();

}
```

Code Snippet 5.1 "MartDto" from DataMart controller

The code above shows a single data mart resource in the API. The properties do not match the underlying database where data is compiled from the “*Models*”, “*marts”* and “*relation*” tables. As stated earlier the resources do not have to match the underlying data structure of the system or database and in many cases, this is advantageous as it can be a security risk by describing the internal data architecture.

5.2.2 Response body and status codes

We chose to use JSON (JavaScript Object Notation) as our way of communicating requests and responses because it is more accessible in JavaScript and C# as they support reading JSON to objects. XML (The Extensible Markup Language) would have meant introducing an XML-parser to the front-end and add unnecessary complexity. JSON also has the advantage of being more

human-readable and uses less memory, which is beneficial for the requests which exceed thousands of data objects (Osman, 2022).

Status codes gives the front-end information about whether a request was successful or not. A successful request is in the 200 to 299 range and unsuccessful requests are from 400 and up. Requests between 400 and 499 are client errors, meaning something is wrong from the client side. Server-side error status codes are in the 500 to 599 range and indicate an internal server error.

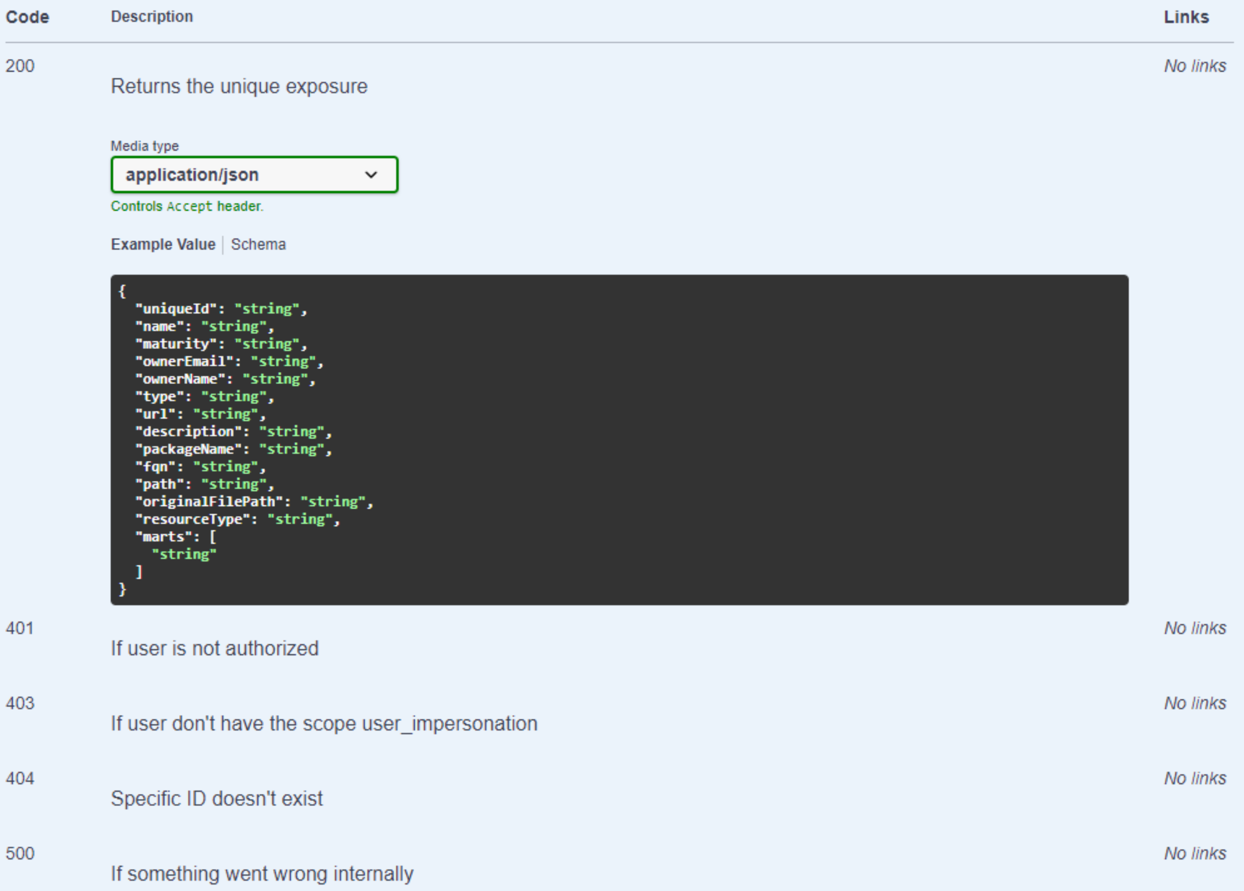


Figure 5.5 Response example and status codes from Swagger API

Figure 5.5 shows the JSON response from fetching an exposure and the possible status codes. As one can see status code 200 "*OK*" means everything is as expected. Whilst 401 "*unauthorized*" means that the user is not authorized through azure authentication and needs to authenticate before proceeding. 403 "*forbidden*" is like 401 but means that the user is not scoped and allowed to access the application. 404 "*not found*" means that there specified

resource was not found. All these responses give the front-end knowledge of what went wrong and in the case of "*not found*" messages, a response of what is wrong in the request is given to the client to display. Though the final error code 500 "*internal server error*" does not give any indication of what went wrong, since a response could give an indication of internal architecture and vulnerabilities. Generally, all our API routes follow this template of response messages and codes which can be viewed more closely in the application code in the folder "Controllers".

5.3 User authentication

As mentioned in the previous chapter, we used a REST API in order to communicate between front-end and back-end. To make sure there was a secure communication between the two layers, we use an authorized fetch function with a bearer token in the header of the request. This token is obtained through an authentication process that requires the user to log in with their Intility credentials. Once authenticated, the user receives a token and stores it in the browser's local storage. This token is then included in the header of all subsequent requests to the back-end where an authorized fetch is used.

5.4 Database

The database is created in Microsoft Azure and populated using Entity Framework through .NET. All the database tables are either added manually by the user of the application or added automatically from the manifest file. Examples of information added manually are the business systems or the business glossaries.

The purpose of our database is to store all information related to the data pipelines and other information related to the application like the business glossaries. Keeping the data for the data pipelines in organized in different tables was essential for this application. This organization of the database makes it easy to access, update and manage.

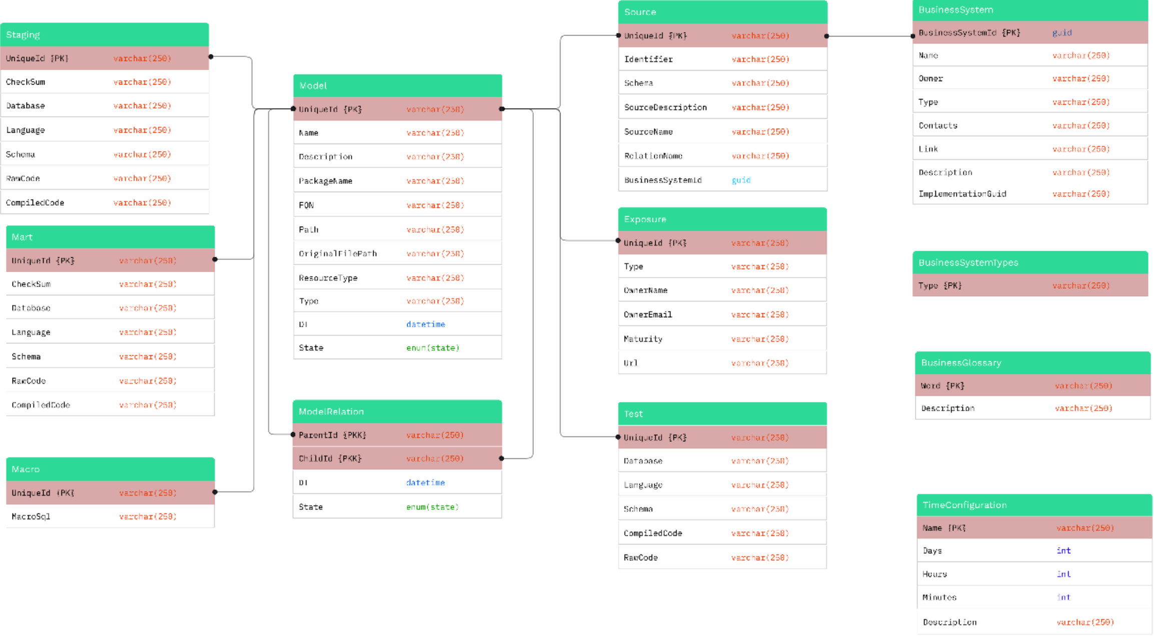


Figure 5.6 Logical Model

As you can see in Figure 5.6 above the database is 3NF normalized and each data model has a foreign key relation to the “Model” table. The “ModelRelation” table is a many-to-many relation which indicate ever parent to child relationship. Both the *Model* and the *ModelRelation* table has DT and State to indicate when last change was and what state it currently is in. As we deliver the thesis this is either added indicating it is new, changed indicating the *model* has changed, and deleted indicating the *model* or *relation* is removed.

It was important to extract and analyze the data in our database in an easy way. Efficient queries enable the user to extract the needed data for their use. The queries are returning the data quickly to ensure that we follow our goals in chapter 1.6.1 and 1.6.2.

When sending requests from the front-end to the back-end, there are functions checking the input fields. If the input is wrong, the request would fail and give an error message to the user. This will make sure that the user will try to fill in the input again, but with only the allowed characters. More about this in chapter 5.3 User authentication.

It is still possible so send requests through tools like Swagger or Postman, if you want to avoid this input validation. To protect for these cases, we had to create input validation functions in the back-end code as well to check for invalid inputs before altering the database. This is done

by precompiling the regex, then checking the input fields with the allowed characters in the regex. The reason behind precompiling the regex is to follow best practice when working with it, and to have the code work as fast as possible. A cached regex can be reused, and this will be faster than compiling the same expression repeatedly.

The database normalization means database size is not an issue in terms of scalability. Many users of the application could call for scaling in form of the underlying architecture and the computing capabilities if we want to have a fast database. The data security also needs an upgrade. As the userbase of the application grows, so does the chance of cyber-attacks. Changes to prevent this could be to have more and better input validation checks, better access controls, more robust encryption and a more detailed monitoring and logging for the database.

5.5 Implementation of GUI design

During the initial stages of the development process, we collaborated with the employees at Intility to produce preliminary sketches of the application using Figma. These sketches were based on the Functional requirements elicited in chapter 2.2 and can be found in Appendix 0 Kristian Senneset (Data Science Expert). Following their approval, we proceeded with the development process. However, during the user testing phase, Intility requested a layout that differed significantly from the original sketches, at a relatively late stage of development. As a result, there was limited time to redesign the application entirely to their specifications, and we had to establish constraints and communicate the extent of what we could accomplish with the available resources. This experience taught us the importance of producing more comprehensive sketches before embarking on the development process.

5.6 Business glossary page

To ensure that everyone using the application is speaking the same language when it comes to key business terms and concepts, we made a page containing a business glossary. Having a centralized location for definitions and explanations, reduces the likelihood of misunderstandings and miscommunications that could negatively impact the business. The business glossary can also improve collaboration and decision-making by promoting a shared understanding of key terms and concepts across different departments and teams. Additionally,

it can help with onboarding new employees, as it provides a quick and easy reference for them to learn the language and key terms related to the application.

The implementation of the glossary page was relatively straight forward, and a visualization of the result can be seen in Figure 5.7. We used a Bifrost table with two columns to display the words and descriptions. A button containing a red cross is displayed on each table row, signifying that the user can delete the word from the glossary. A button with a plus sign can also be found below the table. This is used to add a new word to the glossary, and when pressed, a new table row with input fields is displayed. These two features will later be limited only to users with administrator roles, as part of the user roles discussed in chapter 9.4 Future development. Input validation is in place whenever a user is trying to add a new word, and if illegal or no input is entered, a floating error message will be displayed for in the bottom right corner for three seconds (see Figure 3.12).

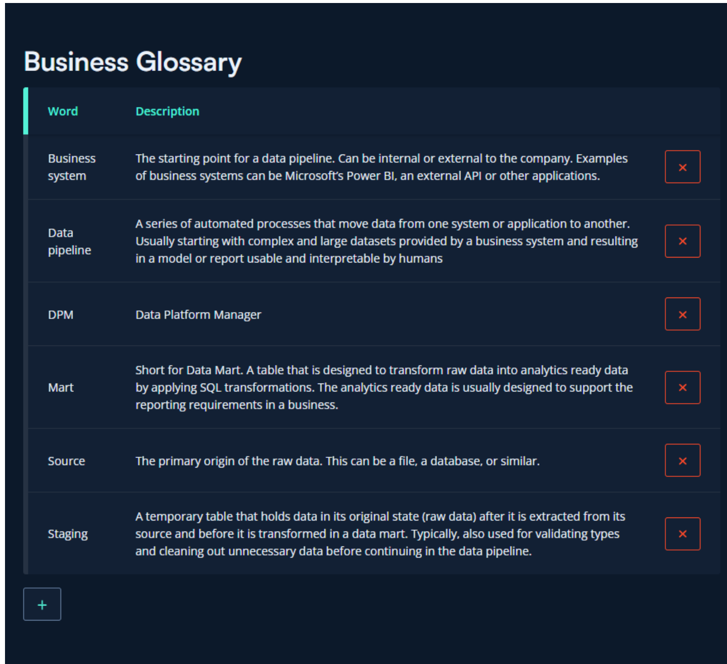


Figure 5.7 Business glossary page.

5.7 Business system information page

A key feature in our data platform manager is to be able to see basic information about the various stages in the data pipeline. The business systems are no exception.

Intility did not initially specify how they wanted the information page for the business systems to look, which meant we had to make a first draft ourselves. We got feedback from the end users to find out if they wanted the information to be displayed differently. As mentioned in chapter 7, the client would like the setup to be more like DBT docs, which the data scientists already use and are familiar with. Based on this feedback, we chose to change all the information pages described in chapter 5.10 Individual Information Pages, as well as the information page showing the business systems. You can see the first draft in Figure 5.8 and the reworked version in Figure 5.9.

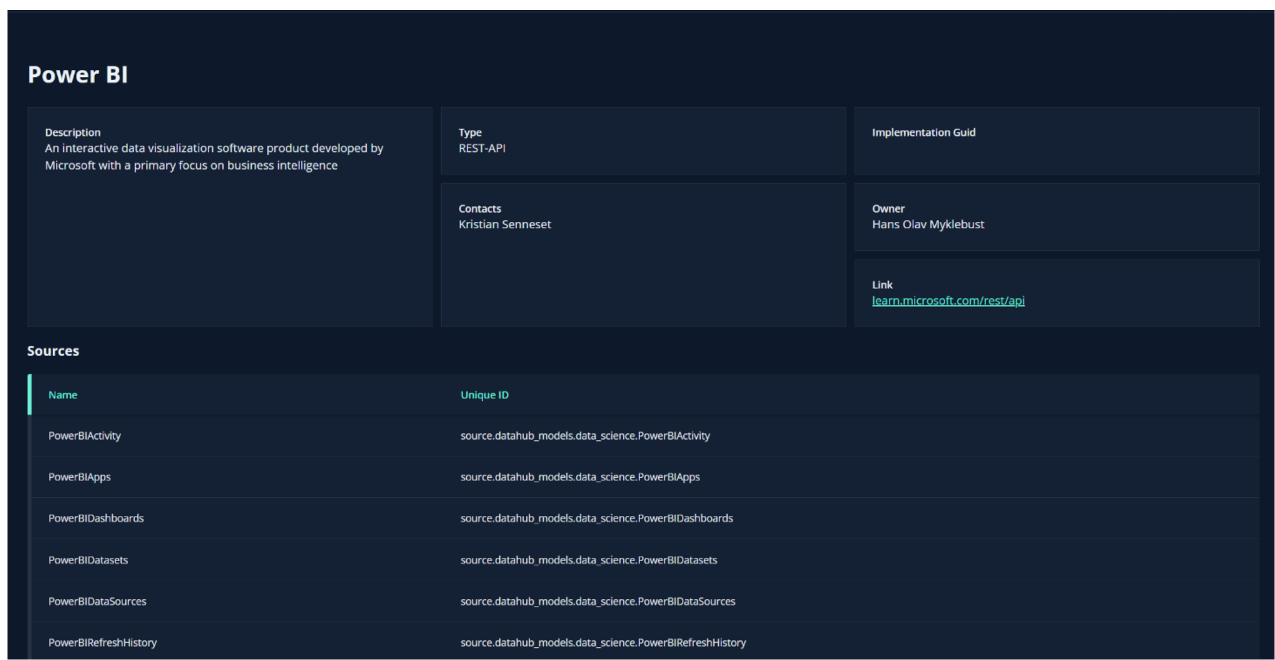


Figure 5.8 First draft of information page

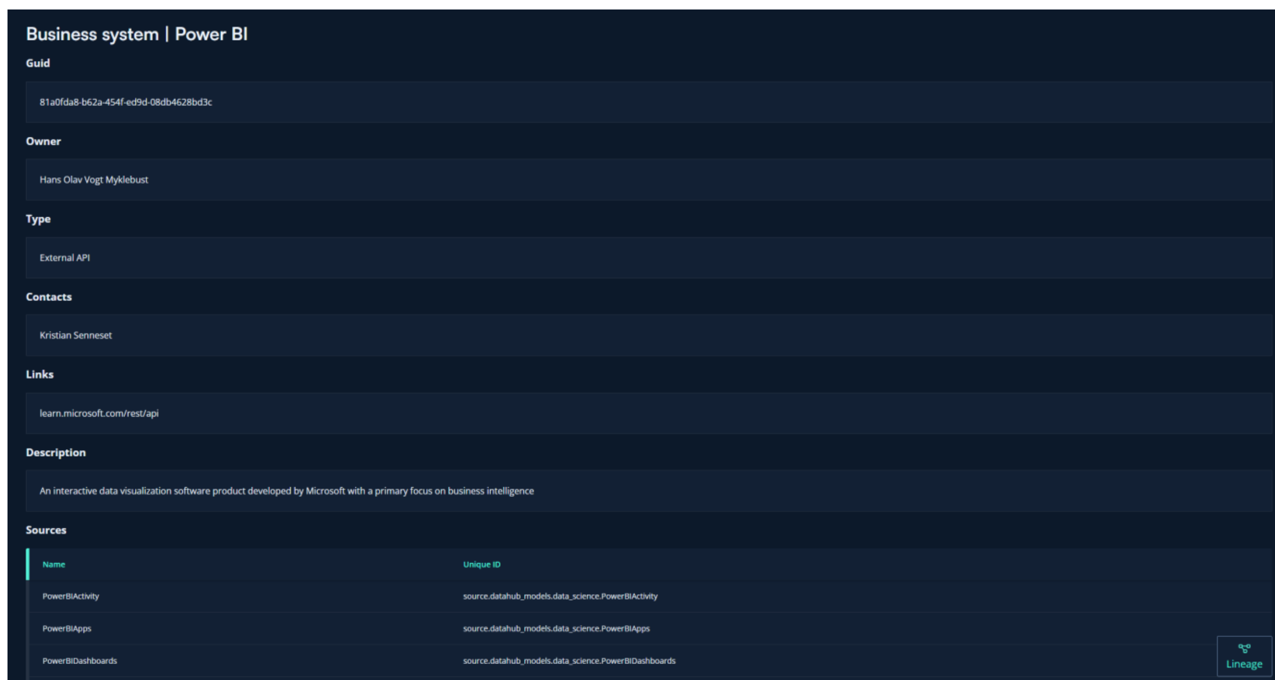


Figure 5.9 Reworked information page to be more like DBT docs

5.7.1 Adding sources through the business system information page

Through conversation with the data science team, we received a request to add new functionality to the business system info page. The issue was that whenever a new business system is created, the user often wants to add a lot of links from the newly created system to existing sources. This would take a long time if the user was to add the links one by one on the data sources page, which meant we had to introduce an option to add several at once.

This option is found at the bottom of the business system info page in our final version. An image of the functionality can be seen in Figure 5.10 Functionality to add several sources at once to a single business system In this component, we have added a table containing all the sources which are not currently linked to a business system. Each row has a checkbox allowing the user to select one or more sources before adding all the checked sources to the business system using the “Add checked sources” button. We have also included a search function to filter the sources based on their name and id. This functionality further increases the efficiency of the application, as the time to integrate business systems with sources drastically decreases.

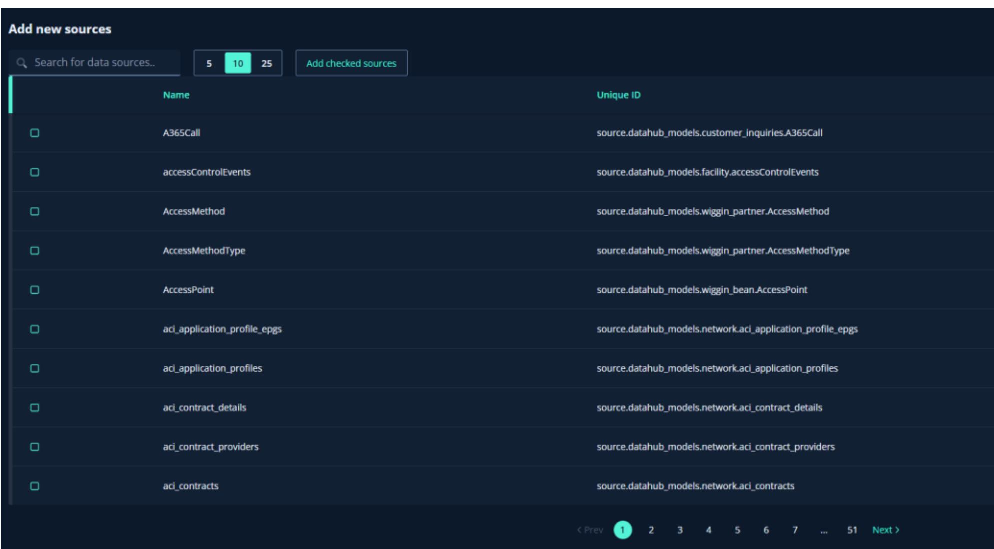


Figure 5.10 Functionality to add several sources at once to a single business system

5.8 Data Assets page

A fully dynamic table of all data assets with multiple functionalities including a search function, filtering, and pagination. An example of the design can be seen in Figure 5.20.

5.8.1 Search

For the search functionality we have introduced a simple input field with a label and a search icon which has a loading animation upon doing a search:

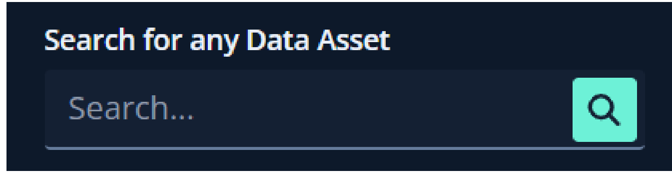


Figure 5.11 Data asset search input

Figure 5.12 Illustration of the loading animation during a search

See chapter 5.9 Search function Front-end for more information about how we use the search input to alter the displayed data assets in the table.

5.8.2 Filtering data assets

Furthermore, we have implemented the ability to filter data assets based on different categories such as folder and data type.

The folder filter is made using a multi select component from the Bifrost library and is filtering all assets based on the folder names you select (Intility, 2023). Once the user has checked off the folders (Figure 5.13), only assets from selected folders are presented in the table (Figure 5.14). It is possible to select several folders at the same time.

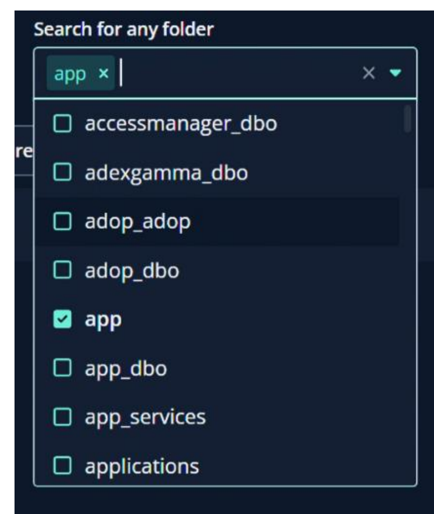


Figure 5.13 Folder selection field, with possibility to search for folders and select multiple at once

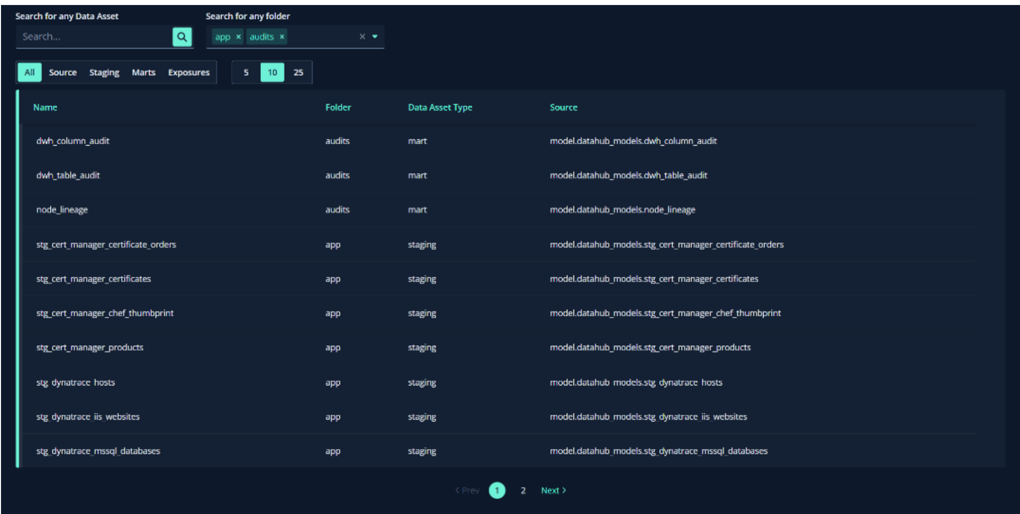


Figure 5.14 Filtered table based on the folders selected

```
{searchResult.length > 0 || searchValue.length === 0 ? (
  <Pagination
    className='bfl-padding'
    totalPages={totalPages(showTable(), amountPerPage)}
    currentPage={page}
    onChange={setPage}
    displayPages={11}
  />
) : null}
```

Code Snippet 5.2 Pagination with properties

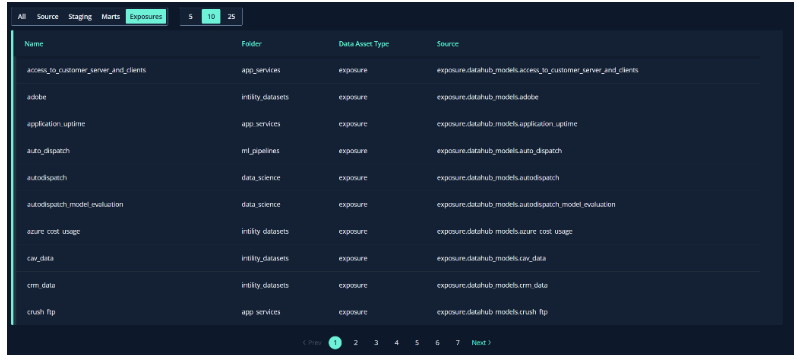


Figure 5.15 Example of filtering based on type

```
/**
  *
  * Filters dataAssets data based on type and any applied search or folder filters
  *
  * @returns data to be displayed in table
  */

 function showTable(): any {
   let data = searchResult.length > 0 ? searchResult : dataAssets
   if (selectedFolders.length > 0) {
     data = data.filter((asset: { folder: string }) =>
       selectedFolders.includes(asset.folder)
     )
   }

   switch (activeBtn) {
     case 'all':
       return data
     case 'source':
       return data.filter(asset => asset.type === 'source')
     case 'staging':
       return data.filter(asset => asset.type === 'staging')
     case 'mart':
       return data.filter(asset => asset.type === 'mart')
     case 'exposure':
       return data.filter(asset => asset.type === 'exposure')
     default:
       break
   }
 }
```

Code Snippet 5.3 Function filtering data assets based on selected asset type

For filtering based on the data types, we used a button group component so that the user can click on the preferred type they want displayed (Figure 4.1) (Intility, n.d.). Both filtering methods including a search can be used together if the user wants to easily narrow down the assets depending on the users' needs.

5.8.3 Pagination

To navigate between the table pages we are using the Bifrost pagination component (Intility, n.d.). To make it dynamic based on all the filtering and sorting we needed to make some additional functions. These include "totalPages" and "showTable" as shown in Code Snippet 5.2.

Figure 5.16 Pagination options

The "showTable" function is crucial for the data to be displayed correctly. Without the "showTable" function, the data would not be filtered as intended. The function applies filters to the data and returns the correct array of data assets so that the "totalPages" function can calculate the correct number of pages needed. The "totalPages" also takes the number of assets per page to be displayed as argument. By default this is 10 per page, but there is an option to change it to 5 or 25, by clicking on buttons in the button group as shown in Figure 5.16.

5.8.4 Caching data assets table

To enhance performance and minimize API calls in our application, we implemented caching for the data assets on the front-end side. This feature comes in handy if the user experiences slow internet. For example, on a public Wi-Fi where multiple people are connected. This allows the application to serve the data from the cache, boosting the user experience by not relying on the connection and reducing the overall time needed to load the assets. Reduced network traffic will not only boost the user experience but also reduce load on the servers and improve the applications scalability and overall cost. Considering the amount of data assets, we saw the need to implement a caching mechanism.

```
/**
 * Gets total pages
 * @param array data
 * @param amount number of pages
 * @returns
 */
const totalPages = (array: any[], amount: number) => {
  return Math.ceil(array.length / amount)
}
```

Code Snippet 5.4 Function calculating the total amount of pages

The data is cached using "sessionStorage" to store the data on the client-side. This means that if the data assets have been previously loaded, they will be fetched directly from the cache instead of making additional API calls. The session storage will store the data as long as the website is not closed, and the user can navigate freely through the website without the data assets being removed from cache (Mozilla, 2023).

In summary, implementing caching for the data assets has proven to be highly effective for enhancing the performance, reducing network traffic, and optimizing the scalability and cost-efficiency for our application. It significantly improves the user experience, particularly in situations where the internet might be slow such as on a public Wi-Fi.

```
// loads on mount
useEffect(() => {
  // If data is cached gets it from cache, otherwise make a request
  const cachedData = sessionStorage.getItem('dataAssets')
  if (cachedData !== null) {
    setDataAssets(JSON.parse(cachedData))
    setFolderOptions(getFolderOptions(JSON.parse(cachedData)))
    setspinnerLoading(false)
  } else {
    const requestOptions = {
      method: 'GET',
      headers: { 'Content-Type': 'application/json' }
    }
    // Fetch data from API
    const getDataAssets = async () => {
      const res = await authorizedFetch(
        configuration.api + configuration.dataAssets,
        requestOptions
      )
      const data = await res.json()

      // Sorts data alphabetically
      data.sort((a: any, b: any) =>
        removeSpecialCharacters(a.name.toLowerCase()).localeCompare(
          removeSpecialCharacters(b.name.toLowerCase())
        )
      )
      setDataAssets(data)
      setFolderOptions(getFolderOptions(data))
      setspinnerLoading(false)

      // Cache the data
      sessionStorage.setItem('dataAssets', JSON.stringify(data))
    }
    getDataAssets()
  }
}, [])
```

Code Snippet 5.5 Useeffect function running on mount of data asset page. Used to fetch data, process it, and set the assets to display

5.9 Search function Front-end

5.9.1 Overview and Purpose

The “simulateSearch” function is an implementation of a search algorithm which filters and sorts through an array of objects to find results that match a given search input value. It takes in several input parameters and uses built in array functions. “simulateSearch” also uses another function we have made, called “removeSpecialCharacters”. This will remove special characters from the input to ensure that the search results are accurate and consistent.

5.9.2 Input Parameters

The parameters of 'simulateSearch' can be seen in .

```
/**
 *
 *  Simulates a search
 *
 * @param value search value/input
 * @param data data state
 * @param setLoading loading state
 * @param setSearchResult search result state
 * @param setPage page state
 */
export const simulateSearch = (
  value: string,
  data: any[],
  setLoading: (loading: boolean) => void,
  setSearchResult: (result: any[]) => void,
  setPage: (page: number) => void
) => {
```

Code snippet 5.6 SimulateSearch parameters

value – The search input the user is typing into a search box, this parameter is of type string.

data – An array of all the data that is going to be searched through. Can be any type of array.

setLoading – Function used to set a loading state. This sets the "isLoading" variable to the given boolean.

Figure 5.17 Spinning search button while loading

setSearchResult – Function to set the search results to be displayed in the table. Can be any type of array.

setPage – Function used to update the table's pagination. Sets page state to first page when doing a search. Is of type number.

5.9.3 Function Logic

First, the function receives the search input, the data array, and several state-setting functions as parameters. It sets the loading state to true to indicate that a search has started.

Next, it checks the search value to determine what search strategy to use. If the value is not empty and does not contain an underscore, it performs a filtered search on the data array. The data is filtered based on the condition that the lowercase name of each asset starts with the lowercase version of the search value when special characters are removed. Filtered results are then sorted alphabetically and added to the "filteredResults" array. Then, it performs another

filter to include items with lowercase name containing the search but does not start with it and adds them to the “filteredResults” as well. Which again gets sorted alphabetically.

```
if (value !== '' && !value.includes('_')) {
    const filteredResults = [
      ...data
        .filter((result: any) =>
          removeSpecialCharacters(result.name.toLowerCase()).startsWith(
            removeSpecialCharacters(value.toLowerCase())
          )
        )
        .sort((a, b) =>
          removeSpecialCharacters(a.name.toLowerCase()).localeCompare(
            removeSpecialCharacters(b.name.toLowerCase())
          )
        ),
      ...data
        .filter(
          (result: any) =>
            removeSpecialCharacters(result.name.toLowerCase()).includes(
              removeSpecialCharacters(value.toLowerCase())
            ) &&
            !removeSpecialCharacters(result.name.toLowerCase()).startsWith(
              removeSpecialCharacters(value.toLowerCase())
            )
        )
        .sort((a, b) =>
          removeSpecialCharacters(a.name.toLowerCase()).localeCompare(
            removeSpecialCharacters(b.name.toLowerCase())
          )
        )
    ]
    setSearchResult(filteredResults)
```

Code Snippet 5.7 Part of the simulate search function which creates filtered and sorted results, without special characters

If the search value is not empty but includes an underscore, it performs a similar search strategy, but this time it checks if the lowercase name of each asset starts with the lowercase search value without removing special characters. Again, it then adds the matching assets to the “filteredResults” array and sorts them alphabetically. Then it performs a second filter to include items where the lowercase name contains the search value but does not start with it. While maintaining the alphabetical order it then gets added to the “filteredResults” array as well.

```
} else if (value !== '' && value.includes('_')) {
   const filteredResults = [
     ...data
       .filter((result: any) =>
         result.name.toLowerCase().startsWith(value.toLowerCase())
       )
       .sort((a, b) =>
         a.name.toLowerCase().localeCompare(b.name.toLowerCase())
       ),
     ...data
       .filter(
         (result: any) =>
           result.name.toLowerCase().includes(value.toLowerCase()) &&
           !result.name.toLowerCase().startsWith(value.toLowerCase())
       )
       .sort((a, b) =>
         a.name.toLowerCase().localeCompare(b.name.toLowerCase())
       )
   ]
   setSearchResult(filteredResults)
```

Code Snippet 5.8 Part of the simulateSearch function which filters the data based on the input provided

If the search value is empty. It initializes an empty array as “filteredResults”.

```
} else {
    const filteredResults: typeof data = []
    setSearchResult(filteredResults)
  }
```

Code Snippet 5.9 If there are no filtered results an empty data array is set

Once the “filteredResults” array is constructed, it will update the search result state using the “setSearchResult” function, passing “filteredResults” to it. The page state is then set to 1 using the “setPage” function to reset the pagination to the first page.

```
setPage(1)

 setTimeout(() => {
   setLoading(false)
 }, 1000)
```

Figure 5.18 Setting the page to one, in order to display the first results

Finally, it will set a timeout of one second to simulate a loading delay, before setting the loading state to false using the “setLoading” function, to make the search complete.

5.9.4 Additional functions and features

```
/**
 * Removes special characters
 *
 * @param str a string
 * @returns string without speical characters
 */
export function removeSpecialCharacters(str: string) {
  return str.replace(/([^\w ]|_)/g, '').toLowerCase()
}
```

Figure 5.19 "removeSpecialCharacters" function which removes all special characters using regex

The "removeSpecialCharacters" function is crucial for the search function logic and uses regex to replace all special characters with an empty string.

```
const trimmedValue = searchValue.replace(/(\s)/g, '')
simulateSearch(
  trimmedValue,
  dataAssets,
  setLoading,
  setSearchResult,
  setPage
)
```

Code Snippet 5.10 The input from the user is trimmed to remove whitespaces before being sent to the "simulateSearch" function

The search value typed into the search bar is first trimmed before it is passed to the simulate search function so that the user can type in a value with spaces if they want. E.g., "Power BI". This was one of the issues we faced making the search algorithm. First when the function was implemented if the user typed a search value with white spaces no result would be displayed.

```
/**
 * Handles input change for search
 *
 * @param event Change event
 * @see simulateSearch
 */
const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
  const value = event.target.value
  setSearchValue(value)
  // Remove spaces from the value parameter
  const trimmedValue = value.replace(/(\s)/g, '')
  simulateSearch(
    trimmedValue,
    dataAssets,
    setLoading,
    setSearchResult,
    setPage
  )
}
```

Code snippet 5.12 "handleInputChange" function handling the change of input from the user and calling the "simulateSearch"

```
onChange={handleInputChange}
onKeyPress={e => {
  if (e.key === 'Enter') {
    const trimmedValue = searchValue.replace(/(\s)/g, '')
    simulateSearch(
      trimmedValue,
      dataAssets,
      setLoading,
      setSearchResult,
      setPage
    )
  }
}}
onIconClick={(event: React.MouseEvent<HTMLButtonElement>) =>
  handleIconClick(event)
}
```

Code snippet 5.11 The input field has props which allow for continuous updates of the searched list whenever the input is changed

We implemented the search so that when a user is typing, it runs a search every time the input is changed by calling the "handleInputChange" function which again calls the search function. This improves user experience and allows for faster navigation. On top of this, it is possible to run a search again by either clicking enter or clicking on the search icon. This is to ensure that the user will be able to reload if needed.

5.9.5 Search Strategy

In the "simulateSearch" function there are two separate search strategies depending on if the search value contains an underscore or not. Since many of the data assets names includes an underscore, we thought it would be good for the search to have this option for the user to get the names with underscores displayed first. The first strategy aims to provide more relevant results by placing a greater importance on the initial match, while the second strategy is useful for when the user wants to specifically find an asset which has a more specific sequence containing underscores. Additionally, it would be possible to add more strategies to make a more custom search experience or improve our implementation.

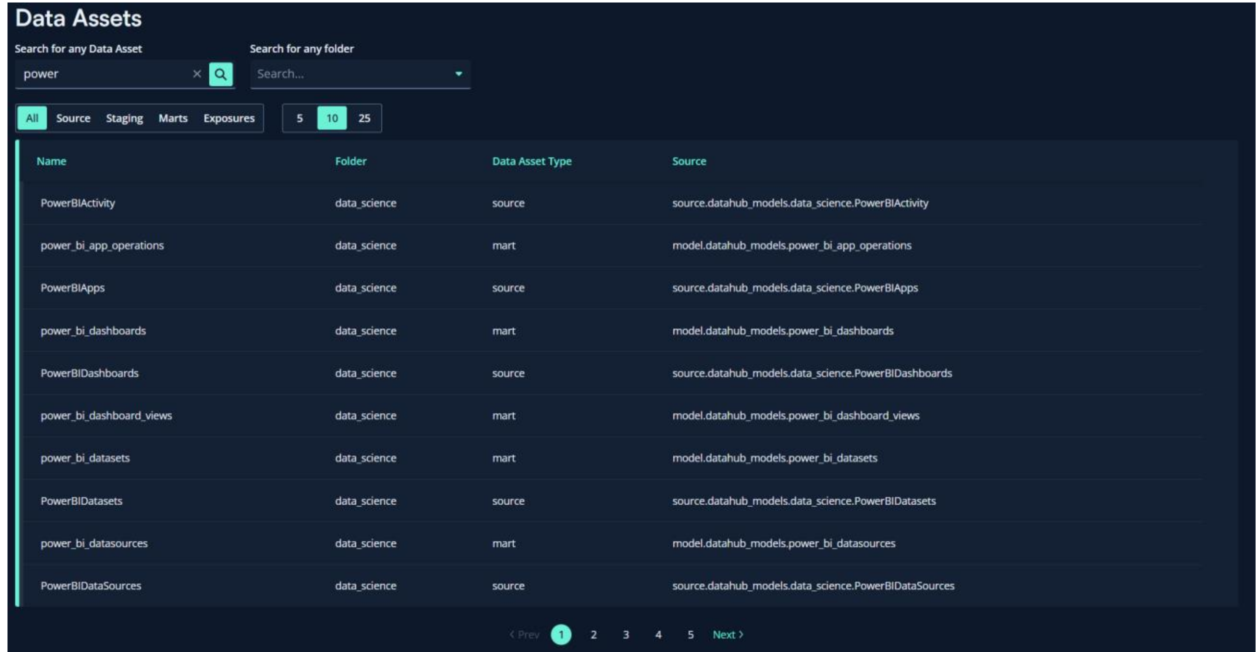


Figure 5.20 Example of the data asset page with search input

In Figure 5.20, we are searching for the word "power". This provides the assets starting with the word "power" at the beginning of the result list. Later we are appending all assets where "power" is part of the name in alphabetical order. By implementing the search function like this there is a higher chance of finding the asset you are looking for effectively. The users are more likely to look for an asset starting with their input, rather than one which has the search word in the middle of its name. As mentioned in chapter 1.1.2, "The current solution – DBT and DBT docs", this is an issue with DBT docs, as all results containing the search word are provided in alphabetical order, rather than the most relevant results appearing first. An example of this can be seen in Figure 5.21 on the right.

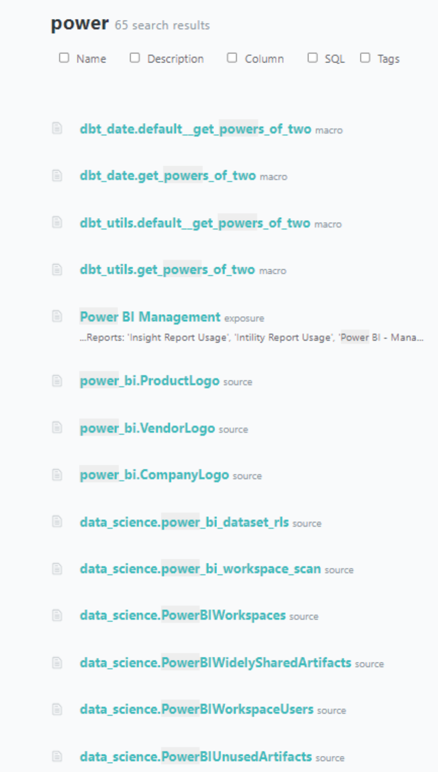


Figure 5.21 Example of search functionality on DBT docs

5.9.6 Efficiency

The efficiency of the function depends on the size of the data array. We decided to use the built in JavaScript array functions for our search algorithm. This is because they are highly optimized functions and therefore efficient enough for our needs. We are using functions such as “filter”, “startsWith”, “includes”, “localeCompare” and “sort” for the search function. According to D. Austin’s article from 2021, we can see that the performance of the “includes” function is faster than a for loop if the array items are not in the hundred thousand. The arrays we are searching through, only contain a couple of thousand items. This means if we had a lot more data it would be ideal to change our search function to use for loops instead.

Another potential point of improvement is the JavaScript sort method. This could be replaced with a better sorting algorithm like quicksort which would be able to handle larger arrays more effectively. The downsides of using for loops in this instance would be that the readability and simplicity of the code would be sacrificed, and the code would be harder to manage. Since the circumstances do not require a complicated search algorithm to be implemented, we chose a simpler implementation.

5.9.7 Conclusion

In conclusion, our search algorithm is based on simplicity using JavaScript functions to do searches. Even though we could have made it quicker and more complex, we did not see the need to add further functionality or features and wanted to keep the search efficiency good while still having readable code.

5.10 Individual Information Pages

The data assets page has a table with all the data assets displayed. By clicking on the data assets, you can navigate to the clicked asset, where each will have additional information related to the asset. All the individual pages are all remarkably similar but have been divided into the four types of data assets which are sources, stagings, mart, and exposures. All of them have some common details, but there are also some distinct attributes for each asset type. On the bottom of the information page, you can see what other assets the selected one is referenced by and/or depends on, with links navigating to each one. Below are images which display the differences and how all the pages are set up with relevant information for each:

Source

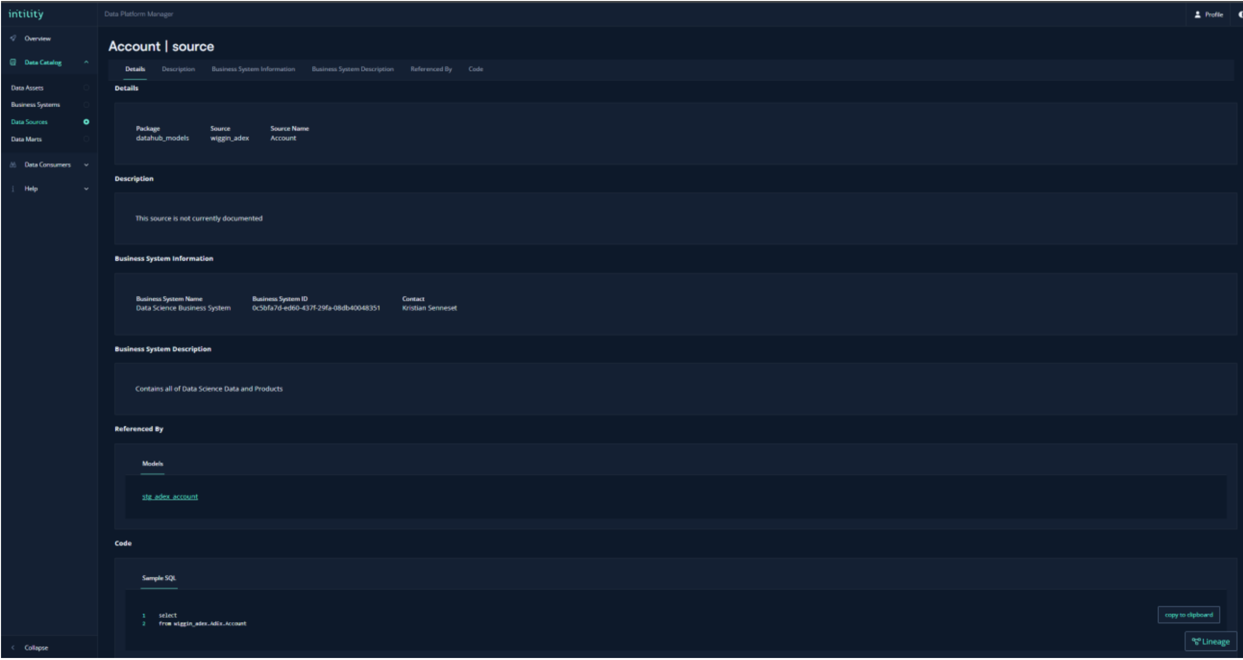


Figure 5.22 Individual Source page

Staging

Figure 5.23 Individual Staging page

Mart

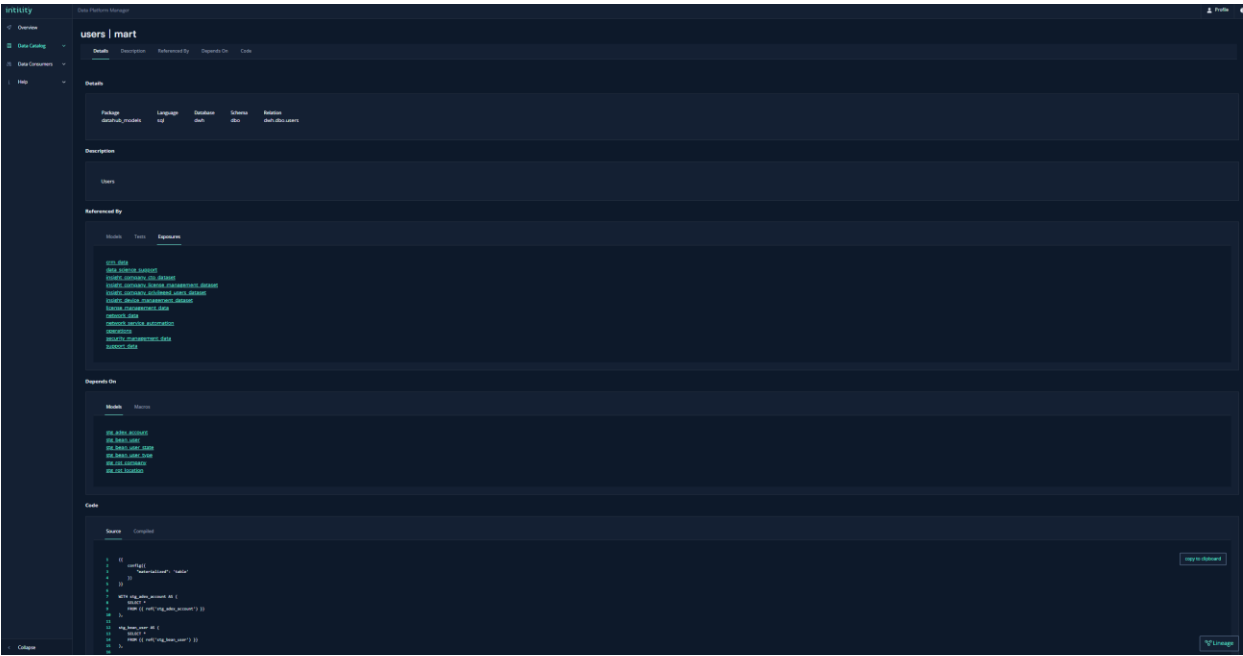


Figure 5.24 Individual Mart page

Exposure

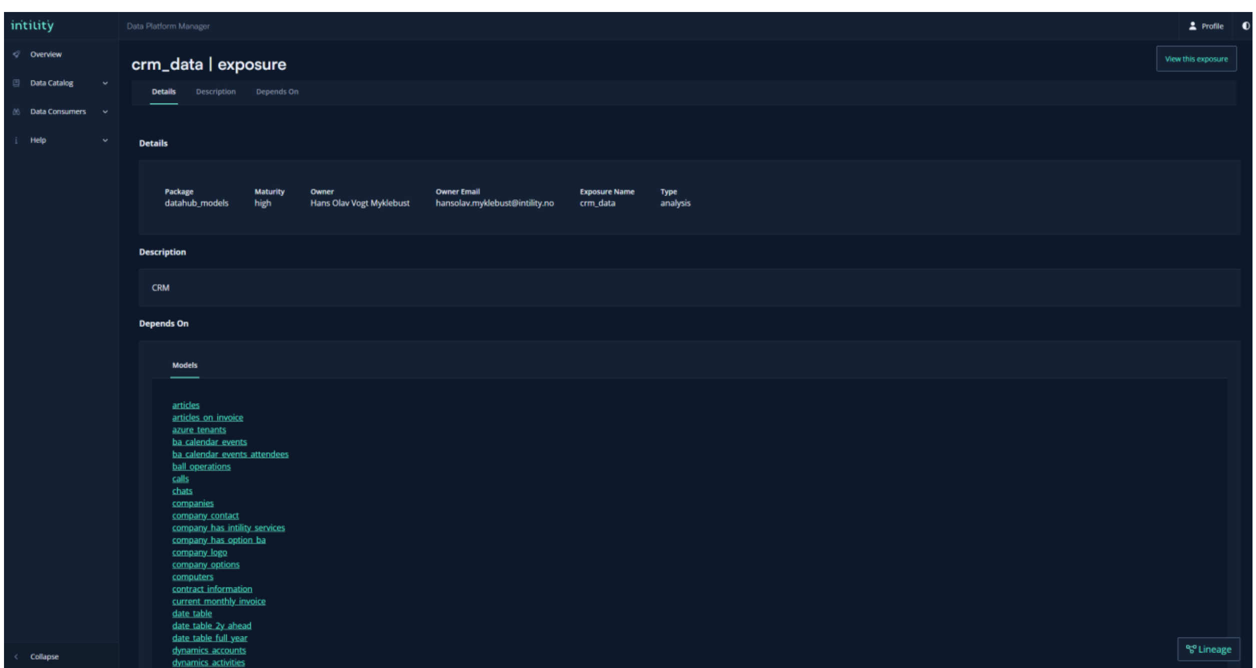


Figure 5.25 Individual Exposure page

5.10.1 Features

Anchor link scrolling: Anchor link scrolling is a feature we implemented for the individual pages. This is something we spent some time trying to figure out a good solution for. For the solution we used the Bifrost component Tabs. Inside the tab we added anchor tags to the different sections on the pages that will go down to the sections specified with an id. For our needs we needed to adjust it a little bit and had to use a state to set the active tab and handle the click for scrolling. By clicking on one of the buttons in the tab the page will scroll down to the correct section.

The handleTabClick function makes sure the scroll is working as intended adding a 150px offset to the scrolling from the top of the page and adds a smooth scroll animation when clicking on the tab button.

```
<Tabs>
          <a
            className={activeTab === 'details' ? 'active' : ''}
            href='#details'
            onClick={e => {
              setActiveTab('details')
              handleTabClick(e, 'details')
            }}
          >
            Details
          </a>
          <a
            className={activeTab === 'description' ? 'active' : ''}
            href='#description'
            onClick={e => {
              setActiveTab('description')
              handleTabClick(e, 'description')
            }}
          >
            Description
          </a>
          <a
            className={activeTab === 'referenced' ? 'active' : ''}
            href='#referenced'
            onClick={e => {
              setActiveTab('referenced')
              handleTabClick(e, 'referenced')
            }}
          >
            Referenced By
          </a>
          <a
            className={activeTab === 'depends' ? 'active' : ''}
            href='#depends'
            onClick={e => {
              setActiveTab('depends')
              handleTabClick(e, 'depends')
            }}
          >
            Depends On
          </a>
          <a
            className={activeTab === 'code' ? 'active' : ''}
            href='#code'
            onClick={e => {
              setActiveTab('code')
              handleTabClick(e, 'code')
            }}
          >
            Code
          </a>
</Tabs>
```

Code snippet 5.13 Code for the anchor scrolling tab

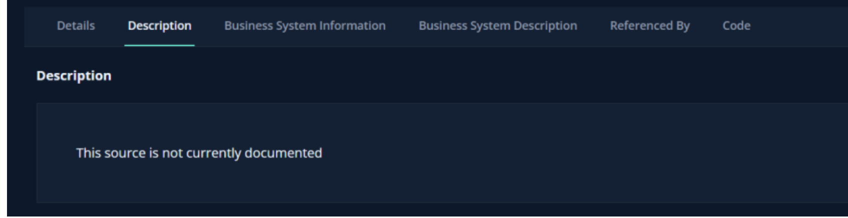


Figure 5.26 Visual representation of the anchor scrolling tab

Copy to clipboard:

Another feature we made was a copy to clipboard component to use for the code section. It is set up basic with a Bifrost button component[1] and a state to display if the content is copied. It takes in an object with the code as a string and copies it as the button is clicked, changing the button text to “copied”.

```
/**
 *  Handles anchor links to be displayed 150px under top of page when clicking on tab
 *
 * @param e - click event
 * @param id - id of the clicked tab
 */
export function handleTabClick(
  e: React.MouseEvent<HTMLAnchorElement>,
  id: string
) {
  e.preventDefault()
  const content = document.getElementById(id)
  if (content) {
    window.scrollTo({
      top: content.offsetTop - 150,
      behavior: 'smooth'
    })
  }
}
```

Figure 5.27 Code for the handling of tab click.

Code

Sample SQL

```
1    select
2    from customer_inquiries.dbo.A365Call
```

copy to clipboard

Lineage

Figure 5.28 Code section where copy to clipboard is used

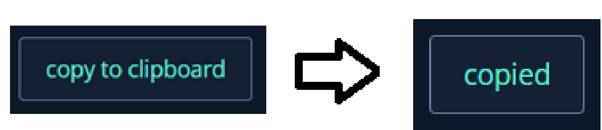
```
/**
 *
 * Copy to clipboard component
 *
 * @param data - an object containing a
 * string to be copied to the clipboard
 * @returns - a button component that copies the provided
 * string to the clipboard when clicked
 *
 */
export const CopyToClipboard = (data: { data: string }) => {
  const [copied, setCopied] = useState('copy to clipboard')

  return (
    <Button
      className='bfc-base-3-bg'
      onClick={() => {
        navigator.clipboard.writeText(data.data)
        setCopied('copied')
      }}
    >
      {copied}
    </Button>
  )
}
```

Code snippet 5.14 Code for the copy to clipboard component

copy to clipboard

copied

Figure 5.29 The button changes when clicked from "copy to clipboard" to "copied".



```
<div className='copy-btn'>
  {<CopyToClipboard data={dataMart.rawCode} />}
</div>
```

Code snippet 5.15 How the copy to clipboard component is used

[1] Bifrost Button Component

Our own custom code formatting:

We implemented our own custom formatting for the code. It takes in the code as a string and will add an index for every line in the code. The <pre> tag preserves both spaces and line breaks for the code. If there is no code data "n/a" will be displayed. The line number will have a certain width and margin on the right for it to line up properly and is using a color theme from Bifrost.

```
1   {{
2       config({
3           "materialized": 'view'
4       })
5   }}
6
7   WITH source_data AS (
8       SELECT
9           [correlationid] AS [call_id],
10          CASE
11              WHEN [caller] LIKE 'sip:%' THEN REPLACE(caller, 'sip:', '')
12              WHEN [caller] LIKE 'tel:%' OR [caller] IS NULL THEN REPLACE(caller, 'tel:', '')
13              WHEN [caller] LIKE '@' THEN NULL
14              ELSE [caller] END AS [customer_phone_number],
15          CASE
16              WHEN [caller] LIKE '@' THEN [caller]
17              ELSE NULL END AS [customer_email],
18          REPLACE([initialagent], 'sip:', '') AS [intility_worker_id],
19          CAST([starttime] AS DATETIME) AS [started_at],
20          [talktime] AS [duration_sec],
21          [queuetime] AS [response_time_sec],
22          CASE
23              WHEN ([ucc_name] = 'Outbound calls' OR modality = 'Dialer' OR modality = 'OutboundDialer') THEN 'Outbound'
24              ELSE 'Inbound' END AS [call_direction],
25          NULL AS [cs_ticket_id],
26          '' AS [problem_description],
27          'A365' AS [system_producer],
28          [actiontype] AS [action_type],
29          [skillchosen] AS [skill_chosen],
30          [ucc_name] AS [workflow_name]
31      FROM {{ source('customer_inquiries', 'A365Call') }}
32  )
33
34  SELECT *
35  FROM source_data
```

Figure 5.30 Image of the formatted code with line numbers.

```
/**
 *
 * Formats a string of code and adds numbers to each line
 *
 * @param data - the code data to be displayed
 * @returns - a pre-formatted block of code
 *
 */

export const CodeSnippet = (data: string) => {
  return (
    <pre>
      {data
        ? data.split('\n').map((line, index) => (
            <span key={index}>
              <span className='line-number'>{index + 1}</span>
              {line}
              {'\n'}
            </span>
          ))
        : 'n/a'}
    </pre>
  )
}
```

Code snippet 5.16 CodeSnippet component for formatting the code

```
/* CSS for number for each line in code sections */
.line-number {
  display: inline-block;
  width: 2em;
  margin-right: 1em;
  color: var(--bfc-base-c-theme)
}
```

Code snippet 5.17 CSS for the line numbers

Link rendering:

For rendering the links in the “depends on” and “referenced by” sections we made a useful component that displays the name of the asset to be linked to and creates a link to navigate to the asset. Under is a list of depends on links for models. Models consists of both marts and stagings. With the function we can easily specify what type of data to be displayed.

The function takes in an array of the link ids, the specified type to be in the base URL and the number to split the id to display only the name in the id. It maps over the array and uses a hyperlink tag to set the path and the id and displays the name of the asset that it links to.

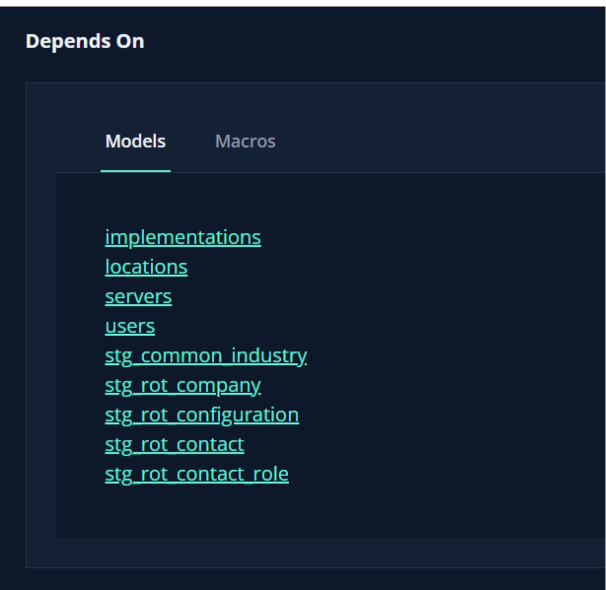


Figure 5.31 Depends on section with links to the marts and stagings it depends on

```
/**
 * A component for printing links
 *
 * @param data - array of strings to be used to generate links
 * @param path - base path in url
 * @param num - number used to split the data to only print out the name
 * @returns - list of links with their own path
 *
 */
export const RenderLinks = (
  data: string[] | null | undefined,
  path: string,
  num: number
) => {
  return (data || []).map(id => (
    <a href={`/${path}/${id}`} className='bf-link' key={id}>
      <div key={id}>{id.split('.')[num]}</div>
    </a>
  ))
}
```

Code snippet 5.18 Code for the RenderLinks component

```
{RenderLinks(dataMart?.dependsOnMarts, 'datamarts', 2)}
{RenderLinks(dataMart?.stagings, 'datastagings', 2)}
```

Code snippet 5.19 How the RenderLinks function is called

5.11 Lineage graph

Displaying the lineage graph front-end was a major part of the assignment, and finishing it was a primary objective from the very beginning. It was important for Intility that the lineage graph was easy to use and understand. A request from them, was that our lineage graph looked similar to the already existing lineage graph on the DBT docs site. They also wanted some extra features like Intility standardization (WCAG) and earlier stages of the data pipelines, namely the business systems.

First, we did some sketching in Figma to have some visual on how it would be possible to create the function. We made a simple sketch, and it gave us a great understanding of how the result could look. From previous experience, we have learned that it is important to have a good sketch before the programming starts. This ensures that developers know what to create, and can reduce the amount of iterations needed before reaching the goal.

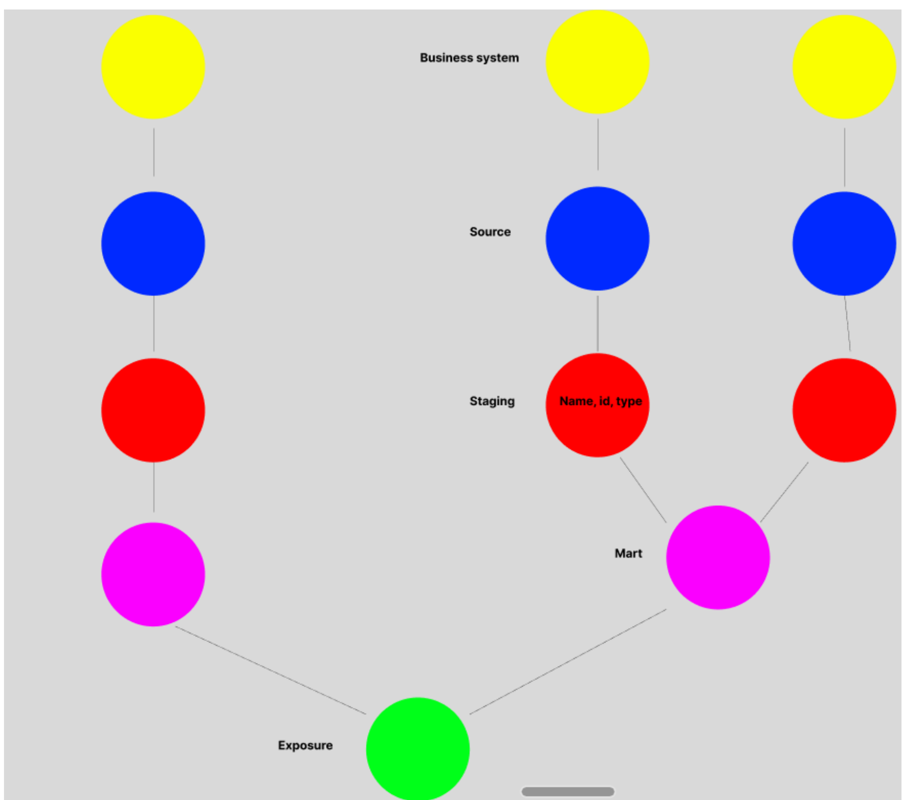


Figure 5.32 Early first draft of how the lineage graph should be presented.

After the sketching, we had to find a way to implement it to our application. The data science department let us know that another department had created a similar graph to our sketch.

The picture below is from their graph, which we used as inspiration when we started creating our own.

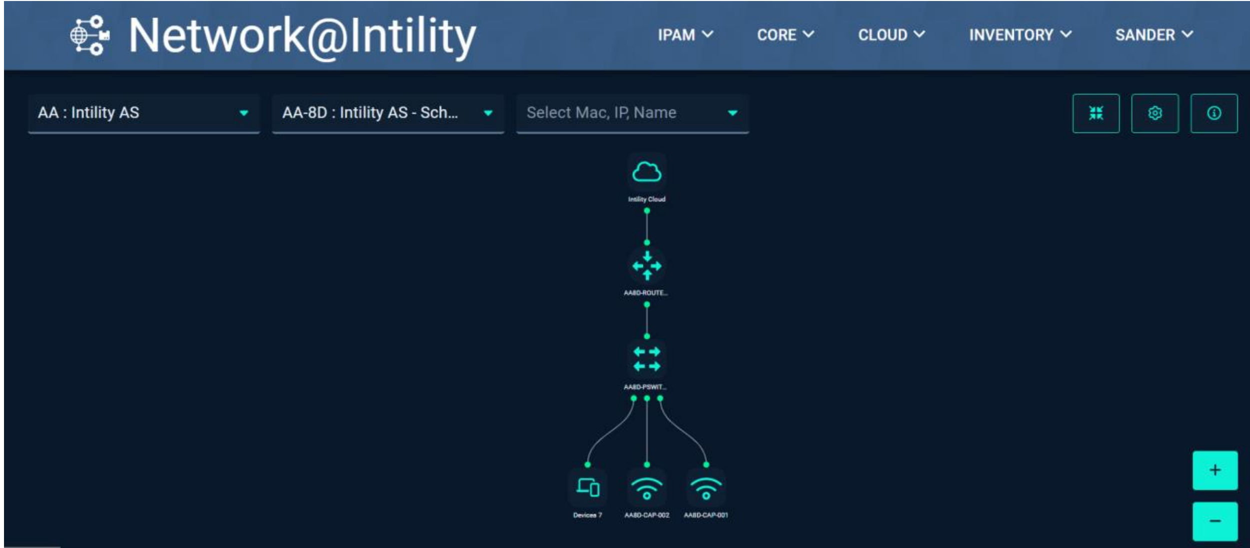


Figure 5.33 Intility Topology graph used for inspiration.

Although their solution was different from what we had envisioned, we were able to overcome the challenge of creating the lineage graph as we intended. Despite it being a complex coding task, we were able to create the lineage graph somewhat similar to our original sketches.

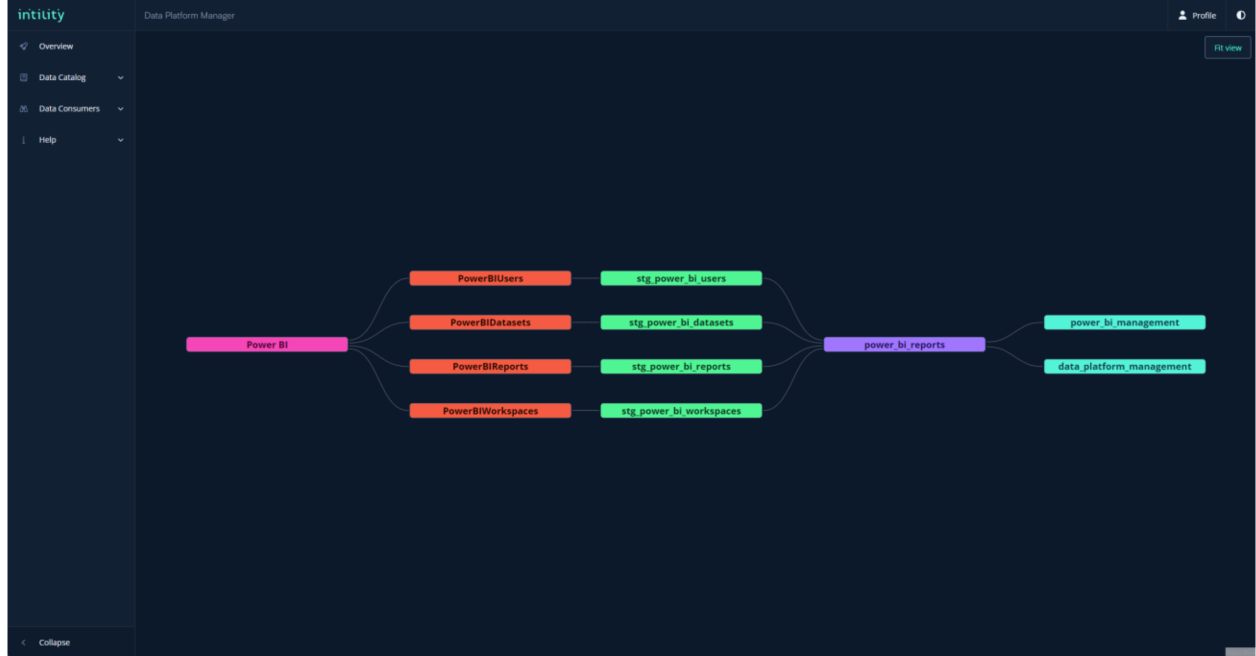


Figure 5.34 Lineage graph example showing the mart "power_bi_reports" as the targeted node.

Figure 5.34 above is an example of the results of our lineage graph. The graph starts at a business system and finds all sources, stagings, marts and exposures. There are some similarities between our inspiration and result, but when the two figures above are compared, it is easy to see that our graph is used for another purpose with less details, options and noise, as well as another way to display the nodes. If we compare our lineage graph to the one in DBT docs (Figure 5.35), you can see our version fits the Intility design standards better. The DBT docs alternative also has a lot more options and filters at the bottom of the graph, which according to the data scientists at Intility rarely gets used.

Figure 5.35 Example of lineage graph from DBT docs.

We wanted to create a graph which was easy to use and had clickable interactive functionality to display information. There are several interactive functionalities on the lineage graph. All the nodes are clickable and will display a drawer with basic information about the clicked node, and inside this drawer is a button to display the full information. The "Fit view" button located in the top right corner of the screen will fit the screen back to default if the user has been navigating the graph by dragging or zooming. As mentioned, zooming the graph is also an option in order to make larger graphs more readable. It is also possible to move all the nodes around without losing the relations. The line between the nodes will not disappear if a node is moved.

Figure 5.36 Lineage graph with information drawer extended

The drawer is shown in Figure 5.35. When the 'See full info' button is pressed, the website will be redirected to a page with more detailed information about the type. You can see the page you will be redirected to in Figure 5.37.

There are different colors corresponding to the WCAG guidelines for the different node-types which are shown in Table 5.1 below. By implementing different colors, the user does not need to click every node to find their type. At first, all the nodes were the same color. This made it almost impossible to differentiate the types in the lineage graph, and made the relations harder to understand.

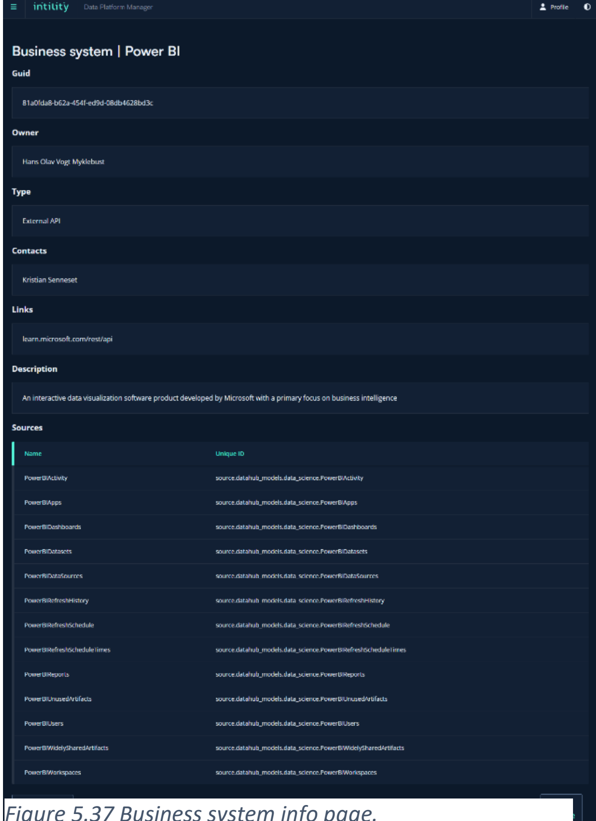


Figure 5.37 Business system info page.

Pink	Business system

Red	Source
Green	Staging
Purple	Mart
Turquoise	Exposure

Table 5.1 The different colored nodes and their corresponding types.

We are extremely proud of how the lineage graph ended up looking. As mentioned, we have created a good looking, easily readable and interactive graph within the Intility standard.

The graph is faster and more effective compared to the existing graph as DBT docs, which was one of our goals in chapter 1.6.2. Upon testing, we found that loading a medium-sized lineage graph in DBT docs takes 5-6 seconds. Loading a graph of the same size in our application takes less than 1 second, improving the overall load time to less than 20% of the original. This is a major improvement over the previously used solution, and is highly valued by the people who are using the feature every day.

In our opinion, the lineage graph is a very innovative addition to this application. Making useful features is very rewarding in our eyes, and it motivated us to make our application as good as possible. In the paragraphs below, we will elaborate on why we think the lineage graph is a very innovative aspect to this application.

Our lineage graph is solving a problem that the users of DBT docs have been struggling with. In DBT docs, it is not possible to see the relations between the business systems and the sources. This problem is solved in our solution, by linking the business systems with the corresponding data sources. For further work, it would also be possible to connect more nodes to the graph in the other direction, including reports which use the exposures. The users of the application previously used two different interfaces, one for the business systems and one for the DBT part of the lineage graph. With both previously used interfaces integrated into one application, you can see the relations more clearly, and have a better overview of what is affected by each business system. Integrating these two interfaces into a single application will also be time saving for the users.

The lineage graph generated by DBT docs also had some problems with the load time for larger data pipelines. As mentioned in 4.1 Back-end, we chose an efficient programming language and cached the data pipelines in our back-end in order to ensure faster reload time than DBT docs alternative.

In the next two chapters, we will break down how the front-end and back-end are programmed in order to display the graph shown on the previous figures in this chapter.

5.11.1 Lineage graph front-end

Little to no experience with visualizing graphs quickly proved to be a challenge after starting the process of implementing our own lineage graph front-end. Luckily, there is a React library called React Flow (React Flow, n.d.), which was made for this purpose. After thorough research, the front-end team displayed a simple graph to the screen. Navigation throughout the graph and moving the nodes are part of the react flow component out-of-the-box.

The next major challenge was finding a way to arrange the nodes correctly in a tree shape for easy visualization and understanding. To accomplish this, we utilized a package called "The Eclipse Layout Kernel" (ELK). This package introduces a collection of layout algorithms to simplify the process of getting the initial diagram layout right.

After the diagram showed the correct nodes and edges in the right positions, the next issue was making the nodes interactive, to display a drawer of basic information whenever a node is clicked. This proved to be a major challenge. We started by placing a button inside of each node on the lineage graph, which allowed the nodes to still be draggable. By clicking on the button it would then activate.

The next challenge was to open the drawer when a node is clicked. This was achieved through extensive research about react contexts, which was a new concept to us. React contexts provide a way to pass data through a component tree without having to pass props down manually at every level. They are used to share data which can

```
// Properties to manage the state of the lineage graph drawer
// containing information about its state and content
type DrawerProps = {
  isOpen: boolean
  node: Node | null
  toggleDrawer: Function
  setDrawerNode: Function
  nodes: Node[]
}

// Create the drawer context
export const DrawerContext = createContext({} as DrawerProps)
```

Code snippet 5.20 Type containing the drawer properties.

be considered "global" to components within a particular context. By passing functions inside the context, the buttons in the nodes were able to call functions to toggle the drawer and populate it with the right information located further up the react component tree (Code snippet 5.20)

5.11.2 Lineage graph back-end

We had to do a lot of research in order to be able to find the entire lineage graph with different nodes and information for each node. The solution became recursive functions. We learned about recursive functions during the Algorithmic Methods course previously in our degree.

```
/// <summary>
/// Method <c>FetchRelations</c> retrieves information about a single source and its relations.
/// </summary>
/// <param name="id">The id of the source to get the relations from</param>
/// <returns></returns>
/// <exception cref="ArgumentException">Throws ArgumentException if Relation id not found </exception>

1 reference
public EdgesAndNodes FetchRelations(string id)
{
    dataCache = _cacheDB.GetCache();
    // If id doesnt exist, an error has occured
    if (!dataCache.TryGetValue(id, out _)) throw new ArgumentException($"Error: No Data model with given id{id}")
    SearchParents(id); // Search through parents & children relations (runs recursively)
    SearchChilds(id);

    EdgesAndNodes result = new EdgesAndNodes(Nodes.Values.ToList(), Edges.Values.ToList());

    return result;
}
```

Code snippet 5.21 Function used to find all parent and child relations based on given node.

```
/// <summary>
/// Method <c>SearchParents</c> recursively searches all parents and adds them to a list of edges
/// </summary>
/// <param name="id"></param>
2 references
public void SearchParents(string id)
{
    if(dataCache.TryGetValue(id, out DataInfo? data))
    {
        Info info = new();

        info.Label = data.Name;
        Nodes.TryAdd(id,new Node(info, id, data.Type, data.Description, data.Folder));

        if (data.Parents == null || data.Parents.Count == 0) return; // Will stop when there are no more parents
        foreach (var parent in data.Parents)
        {
            string ID = "e-" + parent + "-" + id;
            Edges.TryAdd(ID, new Edge(ID, parent, id));
            SearchParents(parent);
        }
    }
}
```

Code snippet 5.22 Recursive function searching through the parents of a given node and adds all relations to a list.

The “SearchParents” and “SearchChilds” functions, are looping through to find all relations. If there are any relations, an edge is created between the current node and its relative. The edge gets an id consisting of an “e-“, and the combination of the parent-id and the child-id. The “e” stands for edge and lets us separate the edge-ids from the node-ids when humans read the information. When the edges are added, the function runs recursively to find all the relations. This means that the function calls itself, but the parameter is changed to be the relative’s id. The “SearchParents” function is shown in Code snippet 5.22, and “SearchChilds” is made in a similar way, only looking through children instead of parents.
The two functions mentioned above are used together in the “FetchRelations” function which is used by the controller whenever a call to the endpoint is made.

These functions took a long time to create, and we are very proud of how they turned out. They are complex functions which needs to be carefully coded to find all child and parent nodes. A node can have many nested children, so the function needed to continue to check the children’s nodes for more children, which was why we chose to use recursive functions.

Recursive functions are effective when traversing trees, which was our objective with these functions, but can however introduce issues if not implemented correctly. If you do not check the right conditions, there is a chance of a recursive algorithm never ending, which would lead

to memory issues such as a stack overflow. This is because each recursive call adds a new frame to the call stack, which can eventually cause the stack to exceed its memory limits. (GeeksForGeeks, 2023)

Below is a screenshot of what the JSON response looks like when a call for a specific lineage graph is received. The response contains all the edges and nodes for the specified graph with all necessary data about the lineage graph, as well as the basic information about each node which is displayed in the front-end lineage graph drawer.


```
▼{nodes: [,…],…}
  ▼edges: [{id: "e-model.datahub_models.power_bi_reports-model.datahub_models.stg_power_bi_users",…},…]
    ▼0: {id: "e-model.datahub_models.power_bi_reports-model.datahub_models.stg_power_bi_users",…}
        id: "e-model.datahub_models.power_bi_reports-model.datahub_models.stg_power_bi_users"
        source: "model.datahub_models.power_bi_reports"
        target: "model.datahub_models.stg_power_bi_users"
    ▶1: {id: "e-exposure.datahub_models.data_platform_management-model.datahub_models.power_bi_reports",…}
    ▶2: {id: "e-exposure.datahub_models.power_bi_management-model.datahub_models.power_bi_reports",…}
    ▶3: {id: "e-model.datahub_models.power_bi_users-model.datahub_models.stg_power_bi_users",…}
    ▶4: {id: "e-exposure.datahub_models.power_bi_management-model.datahub_models.power_bi_users",…}
    ▶5: {id: "e-model.datahub_models.power_bi_workspaces-model.datahub_models.stg_power_bi_users",…}
    ▶6: {id: "e-exposure.datahub_models.power_bi_management-model.datahub_models.power_bi_workspaces",…}
    ▶7: {id: "e-model.datahub_models.stg_power_bi_users-source.datahub_models.data_science.PowerBIUsers",…}
    ▶8: {id: "e-source.datahub_models.data_science.PowerBIUsers-81A0FDA8-B62A-454F-ED9D-08DB4628BD3C",…}
  ▼nodes: [,…]
    ▼0: {data: {label: "stg_power_bi_users"}, id: "model.datahub_models.stg_power_bi_users", type: "staging",…}
      ▶data: {label: "stg_power_bi_users"}
        description: "Power BI Users"
        folder: "data_science"
        id: "model.datahub_models.stg_power_bi_users"
        type: "staging"
    ▶1: {data: {label: "power_bi_reports"}, id: "model.datahub_models.power_bi_reports", type: "mart",…}
    ▶2: {data: {label: "data_platform_management"}, id: "exposure.datahub_models.data_platform_management",…}
    ▶3: {data: {label: "power_bi_management"}, id: "exposure.datahub_models.power_bi_management",…}
    ▶4: {data: {label: "power_bi_users"}, id: "model.datahub_models.power_bi_users", type: "mart",…}
    ▶5: {data: {label: "power_bi_workspaces"}, id: "model.datahub_models.power_bi_workspaces", type: "mart",…}
    ▶6: {data: {label: "PowerBIUsers"}, id: "source.datahub_models.data_science.PowerBIUsers", type: "source",…}
    ▶7: {data: {label: "Power BI"}, id: "81A0FDA8-B62A-454F-ED9D-08DB4628BD3C", type: "businessSystem",…}
```


Code snippet 5.23 JSON response from lineage endpoint.

5.12 Changelog

In order to keep track of changes made to the DBT project and ensure transparency and ease of communication among the data scientists, we decided to implement a changelog feature. This feature displays a table of all the models that were added, deleted, and edited recently. By having a clear overview of all the changes made to the project, the users can easily identify and resolve any potential issues or conflicts that may arise. Additionally, it will ensure that all team members are aware of the latest updates without needing a detailed sync meeting every day. In

this chapter, we will describe the process of implementing our changelog feature both back-end and front-end.

The idea of a changelog was a completely new feature, not found in similar applications. We asked the employees at Intility how they handled changes to different models with the application they used today. The answer was that they did not have any solutions for this problem with the current application. This could prove to become a problem if this application and the number of data scientists in the team would be expanded in the future.

There are several reasons why this is an innovative idea to our application. Firstly, it makes it possible for multiple users to collaborate on the different models and see the recent changes without comprehensive communication with their colleagues. This prevents the users from modifying and working on the same models.

Secondly, the changelog allows for easy overview if errors or inconsistencies have been introduced in the process of editing the models. Tracking the changes over time makes it easy to see where bugs could have been implemented, and lets the users know what changes have been made since the last stable version. When the users utilize this feature together with version control from Gitlab, error management becomes significantly easier.

The employees at Intility were very surprised that we came up with an idea like the changelog and wanted to provide a confirmation on how this feature could help them every day. See appendix Data Science Expert on the *Change Log* feature for the confirmation.

5.12.1 Back-end Changelog

In the database, we represent changes made to models using an Enum named “State” which exists in the Constants folder as described in 3.3.1 Back-end system design. The three types of changes are “added”, “deleted” and “changed”. The three Enums represent whether a data model or relation has been modified or deleted, and is updated each time a new manifest is updated. To track when a change has been made, data models and data relations also has a timestamp called “DT”. This timestamp represents when the last change was made and helps us filter newer and more relevant changes.

To track changes we use the checksum attribute of data models which is compiled by DBT docs and can be found in the manifest file. “To produce a checksum, you run a program that puts that file through an algorithm. Small changes in the file produce very different looking checksums.” (Hoffman, 2019). This means that we can compare the newly produced checksum to the old one and discover changes. By using this method, our program only compares one attribute instead of every attribute. With the system having thousands of data models and some very large string properties this is helpful to the performance of processing the manifest.

The API endpoint sorts recent changes based on configurations set in the database, which at the time of writing is set to 7 days. The most recent changes are sorted by date and sent to the front-end to display on the overview page.

5.12.2 Front-end Changelog

Implementation of the changelog front-end was a relatively simple matter. After developing the other pages and functionality, we started getting very comfortable using the Bifrost React library. To match the rest of the application, we created a component with the usual Bifrost colors and layout. When the entries to the changelog exceeded the space available, we chose to use a scrollable list instead of pagination which we used for other pages. We split the component into two lists, one containing change made to models, the other one containing changes to relationships between models. You can see an example of how the changelog looks in the image below.

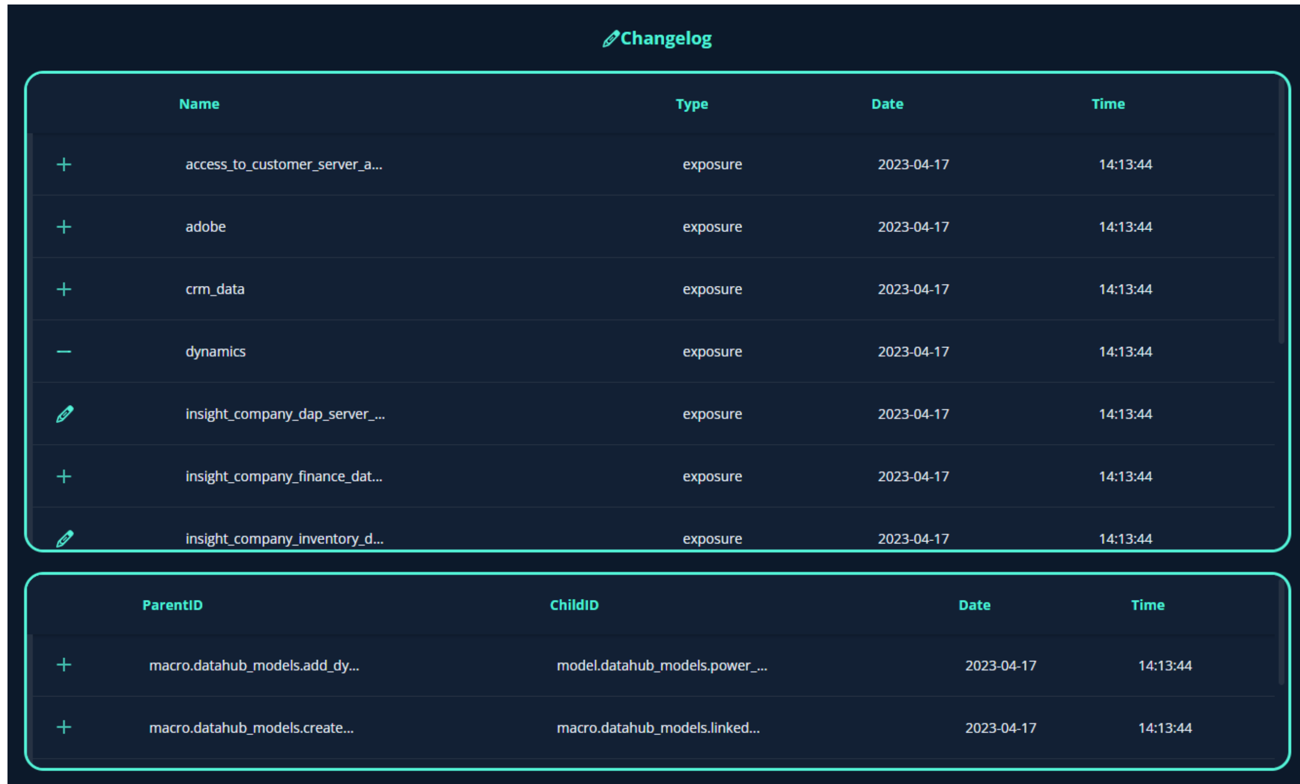


Figure 5.38 Changelog example. Changed nodes above, changed relations below.

One interesting feature regarding the changelog is the ability to directly navigate to an added or edited model. The user can see if a model is added, deleted, or edited by the icon on the corresponding row. Rows containing deleted models are not clickable, as you cannot see info about a deleted model, but the other ones are. This is done by having a function return a clickable or unclickable row based on the type of change (See code snippet in Figure 5.38).

The navigation feature makes the changelog a lot more powerful, in the sense that it provides an easy access and overview of the model, as well as its relationships with other models. This means that the users of the application can get a full overview of what has been changed, and what other models are affected with only a couple of clicks.

```
function GetModelTableRow(entry: ModelChange) {
  // If the state is not delete, make the row clickable
  if (entry.state !== 1) {
    return (
      <Table.Row
        key={entry.id}
        id='table-row'
        onClick={() => navigate(getLink(entry))}      You, last week • Make deleted changes unclickable
      >
        <Table.Cell id='change-icon'>{getIcon(entry.state)}</Table.Cell>
        <Table.Cell>{maxlengthName(entry.name)}</Table.Cell>
        <Table.Cell>{entry.type}</Table.Cell>
        <Table.Cell>{getDate(entry.dt)}</Table.Cell>
        <Table.Cell>{getTime(entry.dt)}</Table.Cell>
      </Table.Row>
    )
  } else {
    // Else make the row unclickable
    return (
      <Table.Row key={entry.id} id='table-row'>
        <Table.Cell id='change-icon'>{getIcon(entry.state)}</Table.Cell>
        <Table.Cell>{maxlengthName(entry.name)}</Table.Cell>
        <Table.Cell>{entry.type}</Table.Cell>
        <Table.Cell>{getDate(entry.dt)}</Table.Cell>
        <Table.Cell>{getTime(entry.dt)}</Table.Cell>
      </Table.Row>
    )
  }
}
```

Code snippet 5.24 Front-end function returning a clickable- or non-clickable table row based on if the model still exists.

5.13 Stats

In addition to the changelog, we have implemented statistics tracking for the application. These statistics include the numbers of each data type in the system. The data include the number of total models, business systems, sources, staging tables, marts, and exposures. This tracking will help keep track of data and see system growth. In the future we could also expand the statistics to include more things, such as graphs to see the increase in data over time, etc.

5.13.1 Back-end Stats

In order to display the stats front-end, we have created back-end functions which count the different entities in the database tables. Macros and tests are filtered out of the models-list and removed from the model counter. This is because macros and tests do not have any relevance for the stats at the front-end page. Business systems, sources, stagings, marts and exposures are counted with LINQ commands. See the figure below for the function which retrieves all the information from the database.

```csharp
/// <summary>
/// Method <c>GetStats</c> fetches different statistics of data in the application
/// </summary>
/// <returns></returns>
1 reference
public StatsDto GetStats()
{
    // Find total number of models
    var models = DataQueries.FilterTestsAndMacros(_context.Models);
    var modelsFilterDeleted = DataQueries.FilterDeletedModels(models);
    int modelCount = modelsFilterDeleted.Count();

    // Find number of business systems, sources, stagings, marts, exposures
    int bsysCount = _context.BusinessSystems.Count();
    int sourceCount = modelsFilterDeleted.Where(model => model.Type == Constants.Source).Count();
    int stagingCount = modelsFilterDeleted.Where(model => model.Type == Constants.Staging).Count();
    int martCount = modelsFilterDeleted.Where(model => model.Type == Constants.Mart).Count();
    int exposureCount = modelsFilterDeleted.Where(model => model.Type == Constants.Exposure).Count();

    // Number of Data Model Changes previous 7 days
    DateTime sevenDaysAgo = DateTime.Now.AddDays(-7);
    int modelChangeCount = models.Where(model => model.DT > sevenDaysAgo).Count();
    int relationChanges = _context.ModelRelations.Where(model => model.DT > sevenDaysAgo).Count();

    StatsDto statsDto = new (bsysCount, modelCount, sourceCount, stagingCount, martCount, exposureCount, modelChangeCount, relationChanges);

    return statsDto;
}
```

Code snippet 5.25 Function using LINQ to get the count of the different models

5.13.2 Front-end Stats

The statistics component was also relatively simple to implement. We utilized multiple of the Bifrost components such as icon[2], grid[3], card[4], and spinner[5]. Below you can see an example of how the components are used:

```jsx
<Grid cols={2} id='overview-stats'>
  <Card>
    {loading ? (
      <Spinner />
    ) : (
      <>
        <Icon icon={faChartColumn} id='change-icon' />
        <div id='change-icon'>{stats?.dataModelCount.toString()}</div>
        <h3>Total models</h3>
      </>
    )}
  </Card>
```

Figure 5.40 Code for one element of the statistics UI

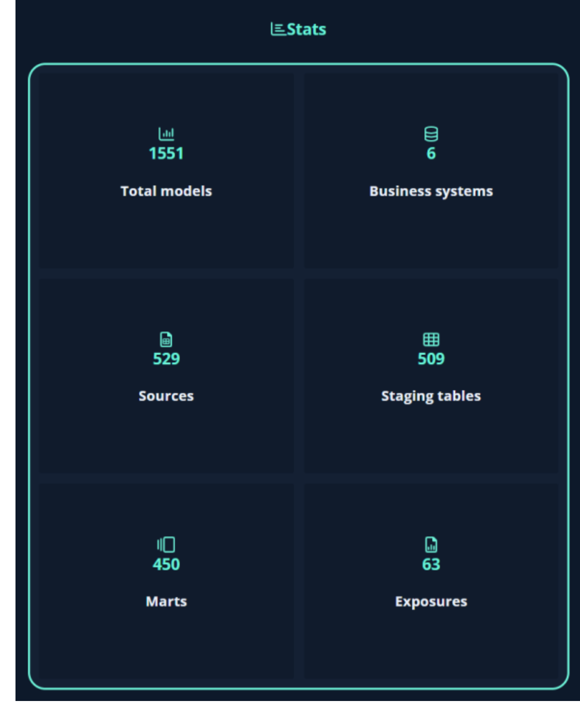


Figure 5.39 Statistics UI

[2] Icon Component
[3] Grid Component
[4] Card Component
[5] Spinner Component

5.14 Sustainability

Sustainability is an important topic in today's society, and this is especially relevant in the software development industry. Minimizing the carbon footprint of the software development processes will reduce the environmental impact of our application. Sustainability is important for Intility as a company (Intility, n.d.), and they take several measures to reduce their carbon footprint. Consequently, we ensured that our application adheres to standard sustainability practices.

This application uses Azure as a cloud-computing service. Azure is used by many services and provides a shared service scalable and optimized for efficiency. This helps companies reduce their carbon footprint and energy consumption by utilizing a cloud pool without the need for internal servers. This means that we avoid having potential unused resources.

We have aimed to develop an efficient application to reduce the computing power. The application has also been designed with sustainability in terms of development. By being easy to update, upgrade and maintain, which will reduce extra work. The code is also available to other employees at Intility which can potentially make development on similar applications more efficient and sustainable.

6 Testing

Testing the application is essential to find problems and bugs in the application. For this application, we have chosen to only test the most essential functionality. This is due to the time limitations discussed in chapter 1.4.1. We will discuss both front-end and back-end testing, where we utilized unit testing for individual components, and integration testing for the interaction between them.

6.1 Back-end Testing

Back-end testing involves ensuring quality and functionality of the API which the front-end communicates with. This involves unit testing, to verify that individual functions work as intended and provides the correct results. As well as integration testing to ensure that components, dependencies, and the database work together. These tests are run automatically when pushed to Gitlab and need to pass in order to merge with the main branch as described in chapter 5.1.1 CI/CD. The tests primarily focus on the high-risk functionality used to update the system with a new manifest file.

6.1.1 Unit testing

Unit testing individual components without dependencies has the advantage that it tests smaller units of code at once. This makes debugging errors easier as the code tested is more isolated. Meaning that when an error occurs, fewer tests are going to fail compared to integration tests, giving better indication of exactly where an error has occurred.

Another advantage of unit testing is that it forces you to write testable code without unnecessary complexity which in turn is more maintainable (Kolodiy, 2015). An example where a simple refactor made the code more testable and less dependent as shown in Code snippet 6.1.

```
/// <summary>
/// Method <c>FindNewModelIds</c> finds all the ids which are NOT present in the DB
/// </summary>
/// <param name="newIds"></param>
/// <returns></returns>
7 references
public HashSet<string> FindNewModelIds(HashSet<string> newIds)
{
    var dataModels = _context.Models;
    // We find the models which does NOT exist in DB but exist in list of ids
    IQueryable<string> dbIds = dataModels.Select(model => model.UniqueId);
    var result = newIds.Where(id => !dbIds.Contains(id)).ToHashSet();
    return result;
}
```

Code snippet 6.1 Early iteration of FindNewModelIds

The image above shows the original function before the refactor. It belonged in the "*ManifestUpdateDB*" class which was already growing quite large. The function also depended on the "BackendContext" shown as "_context" which is used to query the database. This meant that for unit testing you would need to setup a mock "*BackendContext"* class which is an added dependency and provides a much larger context than what is needed.

```
/// <summary>
/// Method <c>FindNewModelIds</c> finds all the ids which are NOT present in the DB
/// </summary>
/// <param name="newIds"></param>
/// <param name="dataModels"></param>
/// <returns></returns>
7 references
public static HashSet<string> FindNewModelIds(HashSet<string> newIds, IQueryable<DataModel> dataModels)
{
    // We find the models which does NOT exist in DB but exist in list of ids
    IQueryable<string> dbIds = dataModels.Select(model => model.UniqueId);
    var result = newIds.Where(id => !dbIds.Contains(id)).ToHashSet();
    return result;
}
```

Code snippet 6.2 Later Iteration of FindNewModelIds

The rework consisted of creating a static class "MUpdateHelper" which handles this and many other frequently used functions within the original class. Creating the class as static means that any function within it can be called without the object being instantiated. The functions are therefore usable in any context with the correct parameters. Instead of using the dependency the function now accepts a parameter that can be queried. Passing parameters in this way is not any less performant, as C# uses lazy loading, and the query is not evaluated until the second last line in both functions.

```
[Theory]
[InlineData(new string[] { "test1", "test2" }, new string[] { "test2" })]
[InlineData(new string[] { "test1" }, new string[] { })]
[InlineData(new string[] { "test1", "test2", "test3" }, new string[] { "test1", "test3" })]
0 references
public void FindNewModelIds_FindSingle(string[] newIds, string[] dbIds)
{
    // Arrange
    List<DataModel> oldModels = new();
    foreach (var id in dbIds)
    {
        var model = new DataModel();
        model.UniqueId = id;
        oldModels.Add(model);
    }

    // Act
    var result = MUpdateHelper.FindNewModelIds(newIds.ToHashSet(), oldModels.AsQueryable());

    // Assert
    Assert.Single(result);
}
```

Code snippet 6.3 Testing FindNewModelIds

Above you can see that the new function is now easy to test without the dependency.

When dependencies are integral to the class or function that we want to test, we will need to mock these dependencies. “Mocking is the process of emulating the behavior of a real object” (Kralj, n.d.). The advantage of this is that we are making sure the dependency behaves as expected and we test smaller pieces of the system. The disadvantage is that it can be quite costly in terms of developer workload to mock dependencies. We used the open-source library Moq to mock our objects because it allows us to mock the “DbContext” class we use for our data layer.

```
[Fact]
0 references
public void DataMartDBTest_GetById_FromListOfFakeDataMarts()
{
    // Arrange
    var mockContext = new Mock<BackendContext>();

    // Arrange DataModels and DataMarts
    var marts = GetFakeDataMarts();
    mockContext.Setup(c => c.Models).ReturnsDbSet(marts.Item1);
    mockContext.Setup(c => c.Marts).ReturnsDbSet(marts.Item2);

    // Arrange ModelRelations
    mockContext.Setup(c => c.ModelRelations).ReturnsDbSet(new List<ModelRelation>());

    var dataMartsDB = new DataMartsDB(mockContext.Object);

    // Act
    var mart1 = marts.Item1.First();
    var resultMart1 = dataMartsDB.GetById(mart1.UniqueId);
    var mart2 = marts.Item1.Last();
    var resultMart2 = dataMartsDB.GetById(mart2.UniqueId);

    // Assert
    Assert.Equal(mart1.UniqueId, resultMart1.UniqueId);
    Assert.Equal(mart2.UniqueId, resultMart2.UniqueId);
}
```

Code snippet 6.4 Testing Data Mart fetching

```
3 references
private static Tuple<List<DataModel>, List<DataMart>> GetFakeDataMarts()
{
    var martId1 = "id1";
    var martId2 = "id2";
    var models = new List<DataModel>()
    {
        new DataModel { UniqueId = martId1 },
        new DataModel { UniqueId = martId2 }
    };

    var marts = new List<DataMart>()
    {
        new DataMart { UniqueId = martId1 },
        new DataMart { UniqueId = martId2 }
    };
    return Tuple.Create(models, marts);
}
```

Code snippet 6.5 Function to create fake data used in data marts unit test

The image shows that the first half of the code is setting up the mock object and making sure that the object returns the fake data marts. This means if the function fails, we have a smaller scope to debug, although less of the system is tested.

6.1.2 Integration testing

Integration testing is a type of software testing that focuses on evaluating the interaction between different dependencies, modules or the database of a system. The advantage of integration testing is that we test a larger part of the system at once. As time is a real constraint this was important to get a good test coverage of critical functionality.

Integration testing also requires some set-up. We decided to use an in-memory database for our integration tests as these are cheap and allows easy build-up and tear-down.

```
18 references
public static class SetUpTestEnvironment
{
    9 references
    public static BackendContext CreateMockBackendContext(string dbName)
    {
        var options = new DbContextOptionsBuilder<BackendContext>().UseInMemoryDatabase(databaseName: dbName).Options;
        var backendContext = new BackendContext(options);
        return backendContext;
    }

    9 references
    public static ManifestUpdateDB CreateMockManifestUpdateDB(BackendContext backendContext)
    {
        var manifestUpdateDB = new ManifestUpdateDB(backendContext);
        return manifestUpdateDB;
    }
}
```

Code snippet 6.6 Setting up test environment

The first function in Code snippet 6.6 shows how a “BackendContext” is built using an in-memory database. Then it is injected into the “ManifestUpdateDB” class allowing us to use it like the code in production. In the future a test database could be generated and used instead, to fully simulate the production system.

The data layer of our API uses LINQ (Language Integrated Query) when fetching and transforming data from the database. This means that our code is translated to SQL by Entity Framework. While our code passes unit tests, they might not in integration tests or production. This can happen because Entity Framework fails to translate the query to valid SQL. When this happened, it meant that we either had to translate our code to do more in memory by evaluating parts of the query early, or more-often that we had to refactor the code.

```
[Fact]
0 references
public void UpdateManifestDB_TestManifestThenEmpty()
{
    var backendContext = SetUpTestEnvironment.CreateMockBackendContext("TestManifestThenEmpty");
    ManifestUpdateDB manifestUpdateDB = SetUpTestEnvironment.CreateMockManifestUpdateDB(backendContext);
    JsonManifest jsonManifest = ReadJSON.Read(@"ExampleData/testManifest.json");
    // Act
    manifestUpdateDB.UpdateManifestDB(jsonManifest);

    JsonManifest emptyManifest = new();
    manifestUpdateDB.UpdateManifestDB(emptyManifest);


    // Assert
    var relations = backendContext.ModelRelations.Where(rel => rel.State != State.Deleted);
    var models = backendContext.Models.Where(model => model.State != State.Deleted);

    Assert.Empty(relations);
    Assert.Empty(models);
}
```

Code snippet 6.7 Manifest test function

In Code snippet 6.7 you can see one of the integration tests updating with an example manifest. Then an additional update is run with an empty manifest which means all models should be set to the state *deleted*, but not fully deleted as they are used in the change log. This 15-line function only took 20 minutes to set up after research was made and tested hundreds of lines of code.

Integration testing allowed us to test the entire “ManifestUpdateDB” class which is responsible for updating the database with changes from a manifest. This provided large value in terms of quality control and as a debugging tool for errors. Also since most of our codebase is heavily dependent writing and reading the database it was efficient to focus on this type of testing.

Stress Test

To ensure that our application meets the scalability demands discussed in chapter 2.3.4 Scalability regarding a growing manifest we needed to stress test our system. To do this we used an example manifest file which is almost two hundred thousand lines. Reading the actual file is not a memory issue as we use “FileStream” to read into objects (Microsoft Learn, n.d.). This means we do not have to hold the entire file in memory, while slightly slower this is advantageous as we avoid running out of memory.

```
/// <summary>
/// This method aims to stress test that an update runs successfully on future larger manifest
/// It reads a manifest of almost 200.000 lines and populates the database with this information.
/// Then it copies and creates dummy values for all sources, nodes, exposures and macros to double the amount of data models in the manifest.
/// Lastly it asserts that the system is able to update without failures such as DB connection timeout.
/// </summary>
[Fact]
0 references
public void UpdateManifestDB_StressTest()
{
    // Arrange
    var backendContext = SetUpTestEnvironment.CreateMockBackendContext("StressTestManifestUpdate");
    ManifestUpdateDB manifestUpdateDB = SetUpTestEnvironment.CreateMockManifestUpdateDB(backendContext);

    JsonManifest manifestWithDummyData = ReadJSON.Read(@"./ExampleData/manifest.json");
    JsonManifest manifestInitial = ReadJSON.Read(@"./ExampleData/manifest.json");

    // Duplicate all data models to fill the manifest with twice the number of data models
    string dummyId = "test";
    foreach (var newSource in manifestInitial.sources)
    {
        newSource.Value.unique_id += dummyId;
        manifestWithDummyData.sources.Add(newSource.Key + dummyId, newSource.Value);
    }

    foreach (var newNode in manifestInitial.nodes)
    {
        newNode.Value.unique_id += dummyId;
        manifestWithDummyData.nodes.Add(newNode.Key + dummyId, newNode.Value);
    }

    foreach (var newExposure in manifestInitial.exposures)
    {
        newExposure.Value.unique_id += dummyId;
        manifestWithDummyData.exposures.Add(newExposure.Key + dummyId, newExposure.Value);
    }

    foreach (var newMacro in manifestInitial.macros)
    {
        newMacro.Value.unique_id += dummyId;
        manifestWithDummyData.macros.Add(newMacro.Key + dummyId, newMacro.Value);
    }

    // Act
    try
    {
        // Update manifest with initial manifest
        manifestUpdateDB.UpdateManifestDB(manifestInitial);

        // Update manifest with the large manifest with dummy data
        manifestUpdateDB.UpdateManifestDB(manifestWithDummyData);
    }
    catch (Exception ex)
    {
        // Assert
        Assert.Fail(ex.Message);
    }
}
```

Code snippet 6.8 Stress test

Code snippet 6.8 shows how we utilize a “dummyId” which we append to the primary identifier “uniqueId”. Even though this only changes one property in the data models, it serves it purpose as we are only testing if the system can handle the number of updates.

This simple test gave us confidence in the scalability and future development of our application regarding updates from new manifests.

6.2 Front-end Testing

For front-end testing we performed unit testing, using the framework Vitest. Which is fast and excellent for unit testing. Additionally, Vitest also includes many features including features from the jest framework (Vitest, 2022). For front-end we have not prioritized to test all the functionality, because of the time constraint discussed in chapter 1.4.1. We have however demonstrated how to make front-end tests, which tests some of the crucial functions. The reason why the tested functions are so crucial, is to get a proper display of the correct data. Tests have been made for the linear search function from chapter 5.9, as well as one function used inside the search function which removes special characters. We tested the search function as it is most likely to have flaws and issues. During the initial testing phase, we figured out that some of the functionality did not work as intended and had to change the function accordingly.

```
// Testing for removeSpecialCharacters
describe('removeSpecialCharacters function', () => {
  it('should remove all special characters from a string', () => {
    const str = 'hello-world!'
    const expectedResult = 'helloworld'
    expect(removeSpecialCharacters(str)).toEqual(expectedResult)
  })
})
```

Code snippet 6.9 Test for the "removeSpecialCharcaters" function

This unit test checks the functionality of a function that removes special characters from a string. The input string is "hello-world!", and the expected result should be "helloworld" after removing the dash (-) and the exclamation mark (!). The test succeeds if the "removeSpecialCharacters" function produces the expected result shown in Code snippet 6.9.

```
// Testing for simulateSearch function
describe('simulateSearch', () => {
  const dataAssets: DataAssetsType[] = […
  ]

  it('filters and sorts search results correctly when value is not empty and does not contain an underscore', () => {…
  })

  it('filters and sorts search results correctly when value is not empty and contains an underscore', () => {…
  })

  it('sets search results to an empty array when value is empty', () => {…
  })

  it('sets loading state to true when simulateSearch is called', () => {…
  })

  it('sets loading state to false after a 1000ms delay', () => {…
  })
})
```

Code snippet 6.10 The 5 tests for the simulateSearch function

We performed five tests for the simulateSearch function, which involves filtering, sorting and loading. This function simulates a search and returns all matching data assets. More about the search functionality is explained further in chapter 5.9.

```
const dataAssets: DataAssetsType[] = [      You, last month • caching dataassets
  {
    name: 'Test 1',
    folder: 'folder1',
    type: 'Type 2',
    uniqueId: 'uniqueId2'
  },
  {
    name: 'test_1',
    folder: 'folder2',
    type: 'Type 5',
    uniqueId: 'uniqueId5'
  },
  { name: 'test', folder: 'folder3', type: 'Type 1', uniqueId: 'uniqueId1' },
  {
    name: 'Test 2',
    folder: 'folder4',
    type: 'Type 3',
    uniqueId: 'uniqueId3'
  },
  {
    name: 'test_2',
    folder: 'folder5',
    type: 'Type 6',
    uniqueId: 'uniqueId6'
  },
  {
    name: 'Test 3',
    folder: 'folder6',
    type: 'Type 4',
    uniqueId: 'uniqueId4'
  },
  { name: 'test_3', folder: 'folder7', type: 'Type 7', uniqueId: 'uniqueId7' }
]
```

Code snippet 6.11 Mock data used in the "simulateSearch" test

Test 1 (filters and sorts the search results correctly when value is not empty and does not contain an underscore)

```
it('filters and sorts search results correctly when value is not empty and does not contain an underscore', () => {
  const value = 'test'
  const setSearchResult = vi.fn()
  const setLoading = vi.fn()
  const setPage = vi.fn()
  vi.useFakeTimers()
  simulateSearch(value, dataAssets, setLoading, setSearchResult, setPage)
  expect(setLoading).toHaveBeenCalledWith(true)
  vi.runAllTimers()
  expect(setLoading).toHaveBeenCalledWith(false)
  expect(setSearchResult).toHaveBeenCalledWith([
    {
      name: 'test',
      folder: 'folder3',
      type: 'Type 1',
      uniqueId: 'uniqueId1'
    },
    {
      name: 'Test 1',
      folder: 'folder1',
      type: 'Type 2',
      uniqueId: 'uniqueId2'
    },
    {
      name: 'Test 2',
      folder: 'folder4',
      type: 'Type 3',
      uniqueId: 'uniqueId3'
    },
    {
      name: 'Test 3',
      folder: 'folder6',
      type: 'Type 4',
      uniqueId: 'uniqueId4'
    },
    {
      name: 'test_1',
      folder: 'folder2',
      type: 'Type 5',
      uniqueId: 'uniqueId5'
    },
    {
      name: 'test_2',
      folder: 'folder5',
      type: 'Type 6',
      uniqueId: 'uniqueId6'
    },
    {
      name: 'test_3',
      folder: 'folder7',
      type: 'Type 7',
      uniqueId: 'uniqueId7'
    }
  ])
  expect(setPage).toHaveBeenCalledWith(1)
})
```

Code snippet 6.12 Test 1 - filters and sorts based on string "test"

The first unit test filters and sorts the search results based on the string “test”. The test simulates the search, starts a timer, sets loading to false and checks whether the search function returns the expected results, which should be displayed alphabetically.

Test 2 (filters and sorts the search results correctly when value is not empty and contains an underscore)

```
1 it('filters and sorts search results correctly when value is not empty and contains an underscore', () => {
2     const value = 'test_'
3     const setSearchResult = vi.fn()
4     const setLoading = vi.fn()
5     const setPage = vi.fn()
6     vi.useFakeTimers()
7
8     simulateSearch(value, dataAssets, setLoading, setSearchResult, setPage)
9
10    expect(setLoading).toHaveBeenCalledWith(true)
11
12    vi.runAllTimers()
13
14    expect(setLoading).toHaveBeenCalledWith(false)
15    expect(setSearchResult).toHaveBeenCalledWith([
16      {
17        name: 'test_1',
18        folder: 'folder2',
19        type: 'Type 5',
20        uniqueId: 'uniqueId5'
21      },
22      {
23        name: 'test_2',
24        folder: 'folder5',
25        type: 'Type 6',
26        uniqueId: 'uniqueId6'
27      },
28      {
29        name: 'test_3',
30        folder: 'folder7',
31        type: 'Type 7',
32        uniqueId: 'uniqueId7'
33      }
34 ])
```

Code snippet 6.13 Test 2 - filters and sorts all that matches string "test_"

The second unit test filters all the search results matching the string “test_” (including underscore). Like the first test, it simulates the search, manages loading, and checks that the expected results are correct.

Test 3 (sets search results to an empty array when value is empty)

```
1  it('sets search results to an empty array when value is empty', () => {
2      const value = ''
3      const setSearchResult = vi.fn()
4      const setLoading = vi.fn()
5      const setPage = vi.fn()
6      vi.useFakeTimers()
7
8      simulateSearch(value, dataAssets, setLoading, setSearchResult, setPage)
9
10     expect(setLoading).toHaveBeenCalledWith(true)
11
12     vi.runAllTimers()
13
14     expect(setLoading).toHaveBeenCalledWith(false)
15     expect(setSearchResult).toHaveBeenCalledWith([])
16     expect(setPage).toHaveBeenCalledWith(1)
17 })
```

Code snippet 6.14 Test 3 - sets the search result array to empty if no searches

The third unit test clears the search results array if there is no search. It simulates the search, sets loading to true, runs timers and sets loading to false. The expected result is an empty array.

Test 4 (sets loading state to true when “simulateSearch” is called)

```
1  it('sets loading state to true when simulateSearch is called', () => {
2      const value = 'test'
3      const setSearchResult = vi.fn()
4      const setLoading = vi.fn()
5      const setPage = vi.fn()
6      vi.useFakeTimers()
7
8      simulateSearch(value, dataAssets, setLoading, setSearchResult, setPage)
9
10     expect(setLoading).toHaveBeenCalledWith(true)
11 })
```

Code snippet 6.15 Test 4 - sets loading state to true if function is called

The fourth unit test verifies the loading state. When the function is called, it expects the loading state to be true. This test simulates the search and checks whether "setLoading" is called with a true value.

Test 5 (sets loading state to false after a 1000ms delay)

```
1 it('sets loading state to false after a 1000ms delay', () => {
2     const value = 'test'
3     const setSearchResult = vi.fn()
4     const setLoading = vi.fn()
5     const setPage = vi.fn()
6     vi.useFakeTimers()
7
8     simulateSearch(value, dataAssets, setLoading, setSearchResult, setPage)
9
10    expect(setLoading).toHaveBeenCalledWith(true)
11
12    vi.runAllTimers()
13
14    expect(setLoading).toHaveBeenCalledWith(false)
15 })
```

Code snippet 6.16 Test 5 - after 1 sec set the loading state to false

The fifth test checks whether the loading state changes to false after one second. The test first simulates the search and expects the loading state to be true, then runs the timer and checks whether the results is false or not.

7 Methodology – Usability Testing

In order to evaluate the usability of our application, we conducted a user test involving a group of data scientists from Intility, who will be the primary users of our application. This is crucial to deliver the best possible product to our end users, and performing the test in the middle of the development process lets us alter the direction we are going with the application before it is finished. The purpose of this test was to identify any usability issues and gather feedback on how to improve the application.

7.1 User test preparation

Before conducting the user test, we had to prepare questions and tasks for the participants of the test. We conducted a thorough study on user testing by reading relevant articles and literature such as Handbook of Usability Testing (Rubin & Chisnell, 2008) and Usability testing 101 (Moran, 2019). This allowed us to create a targeted plan to find the answers we were searching for.

We invited a group of four data scientists who have experience working with data pipelines and DBT docs in their daily work. We provided them with a brief overview of our tool and gave them specific tasks to complete. During the test, we observed their interactions and documented their actions and behaviors with the application and noted any issues or difficulties they experienced.

7.2 User testing theory

According to Moran (Moran, 2019), the goal of user testing is to "measure how well users can complete tasks and find areas where they get stuck."

One key aspect of user testing is creating scenarios or tasks that the user will be asked to complete while using the product. These scenarios should be designed to mimic real-life situations and reflect the intended use of the product. For example, if the product is a financial app, a scenario could be to have the user add a new bank account or track an expense. In our case, we created a list of specified objectives for the test-participants to complete which were all relevant to the desired final functionality of the application.

It is important to recruit participants who reflect the target audience for the product. Moran recommends using a mix of both current users and potential users to gain a well-rounded perspective. In our usability test, all the participants were users of the current and future users of the new documentation tool and could therefore provide valuable insight into how the tool would be used daily. Their experience also allowed them to give helpful advice on the desired new features of the application.

During the testing process, it is crucial to observe the user's behavior and gather both qualitative and quantitative data. The user’s behavior could provide us with useful feedback and Moran suggests recording the user's actions and taking notes on their comments, facial expressions, and body language. Additionally, the surveys and questions from the user test are placed in chapter 7.3, and were used to gather feedback on the user's overall experience.

After the testing was complete, the data was analyzed and used to identify areas where improvements can be made. Moran advises categorizing and finding similarities based on themes, such as usability, design, or functionality. This allows developers to prioritize areas that need the most attention. This was done by going through all the feedback and find the most common issues related to the applications. Then we created issues in Jira in order to resolve the most important feedback from the user test.

In conclusion, by designing realistic scenarios, recruiting appropriate participants, gathering both qualitative and quantitative data, and analyzing the results, we can gain valuable insights into how users interact with the product and make improvements to better the overall experience.

7.3 Test questions

The following instructions (bold characters) were given to the test participants, and the observants were noting down the answers to the points below each instruction. Many of the questions originate from the use-case diagram and use-cases mentioned in chapter 2.2 Functional requirements.

Find out what business system the data source “intility_webpages” originates from.

- How easy is it for the user to know what is clickable?
- How long does it take to navigate to the desired pages?
- Do they use the search functionality as intended?

Return to the home page.

- What is the user’s first reaction when asked to return to the home page / previous page?

Find the description of the mart “intility_webpages”.

- Does the user find the correct mart?
- Does the user access the mart from the data asset tab, or the data mart tab?
- Is the description easy to see?

Create a new business system.

- Does the user find the option to add a business system?
- Are the labels descriptive enough to understand what to input?

Add 5 sources to your newly created business system.

- Does the user navigate to the business system info page to add several at once?
- If not, do they make a comment that this is a difficult way to add several sources?

Add a new word to the business glossary and delete it again.

- Does the user navigate straight to the glossary page?
- Do they have any comments on the design of the glossary list?

7.4 Test Scenario

During our user test, we aimed to assess user experience and overall functionality presented to the users with a set of scenarios to complete using the application. These scenarios covered different aspects of the application such as navigation, search functionality, adding a new business system and adding glossary words. We observed the users' behavior and asked them to think aloud while they completed the tasks. This helped us better understand their decision-making process and identified any difficulties they encountered.

7.5 Test results

Overall, the feedback from our users was positive. The participants found the application itself intuitive and easy to navigate, and they were all able to complete the tasks within reasonable time. A reasonable time to complete the different tasks for a new user should be no more than 15 seconds. However, we did identify a few areas for improvement.

Even though the application was easy to navigate, they missed a navigation bar on the left of the screen whenever they were looking at information about a specific data asset. This navigation bar would make it easier to see what folder the data asset belongs to, and to navigate to similar assets. As mentioned in chapter 3.2 Prototyping with Figma, such a feature was part of our initial sketch suggestions. After discussing this with the test participants, we agreed that implementing such a major new feature would be time consuming, and they had to let us know what they wanted us to prioritize. We landed on a compromise where we added an option to filter the data assets based on folders within the Data Assets page.

Also, we encountered some minor bugs when trying to add a new business system. These included duplicate entries in the select field lists, it was possible to add several business systems with the same name and the participants requested form validation feedback when typing in the “add business system” form.

Unintuitive icons were another issue identified, where a plus which was meant to indicate “Add new source” was interpreted as “More information”. We took all these responses into consideration and made necessary changes accordingly.

In a discussion session after the user test, the data science department wanted an overview at the landing page. We started to come up with solutions to implement an innovative but useful feature that was not a standard in any similar tools. We quickly agreed that statistics of how many models there are of each type would be useful to have on the overview landing page. Following up on this idea, a member of our group asked if the data scientists have seen a changelog in a similar application before. They answered that they were not familiar with such a feature, and we all agreed that it would be an interesting and innovative element to add to the overview page. We have discussed the implementation of this in chapter 5.12 Changelog earlier in the report.

Lastly, we got a request we chose not to implement, mostly because of the time constraint mentioned in chapter 1.4.1. This feedback was made available to the team at Intility who are going to continue the development. The test participants appreciated the useful functionality of changing relationships to business systems, adding new business systems and changing the glossary directly in the application. However, they requested that only members of the data science team were able to make changes which impacts the application, while regular users should only be able to view the documentation. As stated in the Access to sensitive information chapter, we did not have the permission to implement such a feature.

7.6 Ethics

When conducting user tests, it is important to be aware of the ethical manners which respects the users' rights and privacy. We spoke to the attending participants, and they all signed a research consent form which can be found in Appendix A . We had to ensure that all the rights and privacy is taken care of in order to follow GDPR law. There are several critical principles related to the ethical part of a user test. In the paragraphs below, we discuss how we have taken care of these ethical principles.

For this user test, it was important that the employees attending to the test was aware of the purpose of the test, what data we collected, and how we were using the data. The user test was voluntary, and the participants was free to withdraw from the test at any time. Before the tests

started, the participants had been given clear and concise information about the test and what we were going to do with the given feedback.

At the user test, we collected data in the form of feedback from the participants. We did not collect any personal data, but all the feedback from the user test was anonymized before it was documented in this report. This is to ensure the privacy of the participants.

We wanted to avoid bias and respect diversity and therefore had to be mindful while conducting the user test. We gathered as a diverse group as possible, after taking experience and relevance into consideration. The diverse group was gathered to meet the needs of as many end users as possible.

It was ensured that the questions from the user test did not discriminate any of the participants. We did not want to waste our participants time, due to their busy workdays. In order to have the most efficient user test, we had prepared the necessary questions, and we had a plan for how the documentation process should be done. In total, we had planned a meeting for one full hour, and stayed within this timeline. Read more about how this was done in chapters 7.1 and 7.2.

8 Development process and communication

The development process and communication chapter discuss the project management techniques and tools that were utilized to develop the application. The development process followed Agile practices, specifically the Scrum framework. The team used Jira, a project management tool, to plan and track the progress of the project. Additionally, regular meetings were held within our group, Intility, and with our bachelor counselor. The chapter will also discuss the communication and collaboration processes used within our group to ensure the successful completion of the project. Overall, this chapter provides an overview of the approach taken to manage the development of the application.

8.1 Scrum

The Scrum process consisted of sprint- review, retrospective and planning. The three parts were discussed and planned in the weekly and biweekly scrum meetings. Often, we combined the meetings to save time, and to provide feedback while memory was still fresh. Initially, we employed a one-week sprint duration, but later extended it to two weeks due to not having enough time to solve the more complex issues. The shorter sprint periods made it difficult to predict how much work we could do, and we often underestimated our work-speed. With longer sprints of two weeks, it was easier to fit in more complex issues with longer timeframes. We also found that we saved time by having less frequent meetings. The meetings with Intility and the bachelor guidance counselor were scheduled every week. We also tried to have daily stand-up meetings internally in the group to talk about potential blockers, such as API features needed for the front-end development to progress.

8.2 Jira

As mentioned earlier in this chapter, we used Jira to keep track of our issues. The group as a whole had responsibility to allocate enough issues for each member. All issues had a specific type and was moved between sections in the board as it progressed to *complete* as shown in Figure 8.1 below. Issues were assigned at the start of the sprint to avoid having more group members working on the same issue. Some issues were left open and allocated later in the sprint as we finished our set tasks. We did not manage to finish every sprint according to plan.

The reasons why, as well as the measures we took was discussed in the retrospective meetings, and a summary of those meetings can be found in Appendix 0 Meeting minutes.

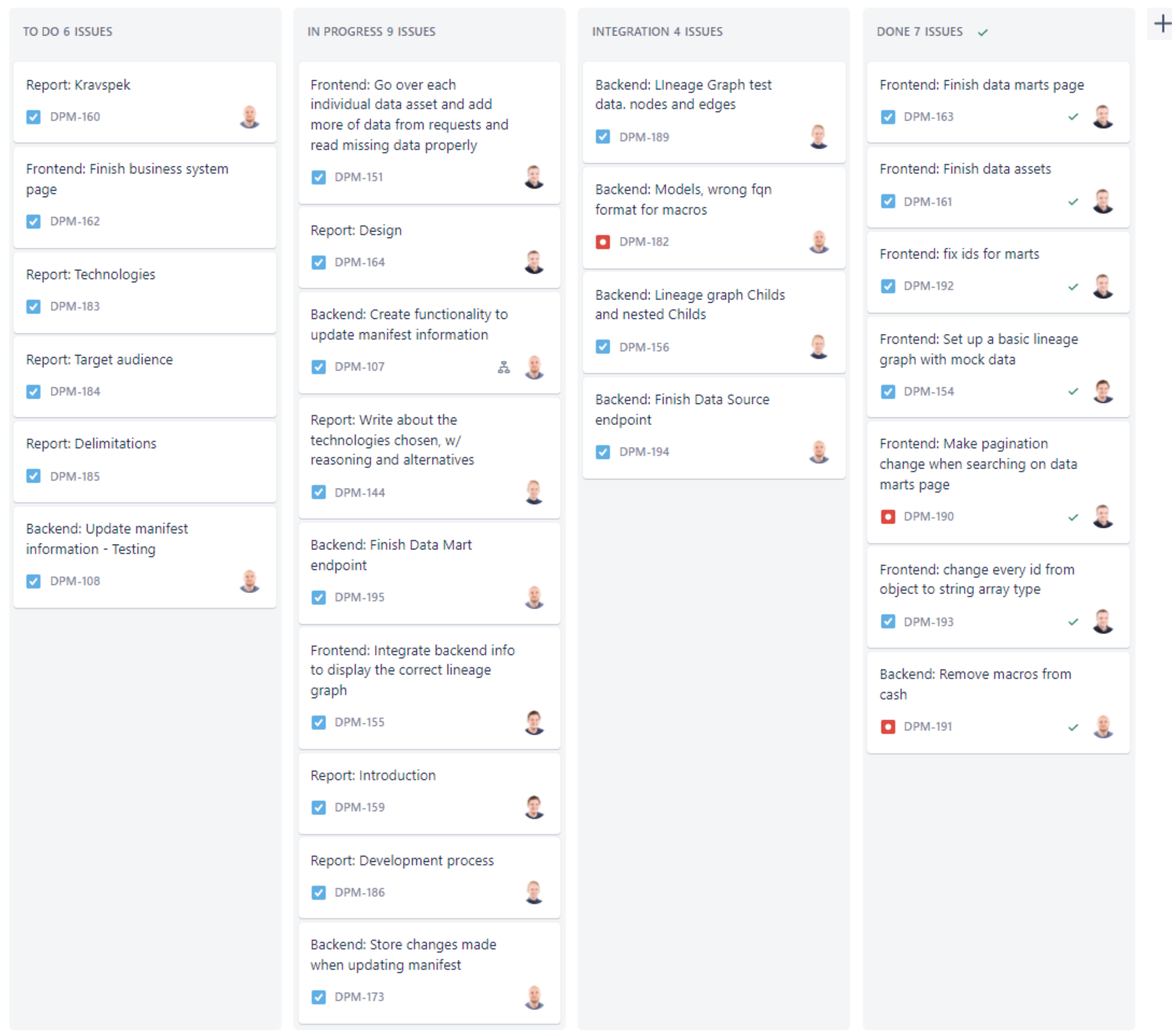


Figure 8.1 Jira board from sprint 7

Prioritizing issues according to sprint goals, keeping track of- and adding issues to the backlog was important. We utilized the backlog to track the progress of our project, prioritize tasks, and ensure that the project's objectives were met. The backlog was used to capture all requirements and tasks, and it was continuously updated throughout the project's lifecycle, most often during sprint planning meetings. Organizing the backlog in Jira allowed us to visualize the progress of each task and identify tasks not finished within the set timeframe. The backlog was also used to assign tasks to team members, set due dates, and monitor the status

of each task. We found that utilizing the Jira backlog helped us to maintain focus on our project objectives and ensured that we completed all tasks within the project's timeline.

8.2.1 Issue types

As mentioned above, there were different types of issues. “Task” was the most used type, which was used for all the regular features on the application. Whenever issues related to bugs were created, they were created with the type “Bug”. The “Report and models” type was related to the project plan, the bachelor report and other documentation. “Testing” was the last of the frequently used types, and is related to front-end, back-end, and user testing. There is also a predefined “Story” issue type in Jira which is used to capture a user or business need in a simple and non-technical format. This issue type was not utilized in our project, as we all had the technical understanding, as well as the sprint goals to lean on in order to understand the objective of our issues.

Task

Story

Bug

Reports and models

Testing

Figure 8.2 Jira issue types

Having different issue types made it straightforward to see what issues were connected and what their objective was.

8.3 Training period

A key obstacle at the beginning of our work with Intility was that we were not familiar with all the tools needed for creating an application according to their standards. It was important that we understood how the developers at Intility managed their workflow and used the internally developed tools. To gain the necessary knowledge, Intility arranged various workshops and talent onboarding for all the bachelor groups which will be discussed in this chapter.

8.3.1 CI/CD

A workshop about git and CI/CD was arranged at the very start of our project. During this two-hour session we learned how to use git branches and best practices. The host of the workshop also taught us to setup of a Gitlab project with pipeline and CI/CD settings, as explained in chapter 5.1 Deployment.

feature/DPM-104_individualMarts merged
75a7bd7a · individual exposure styling · 2 months ago

Figure 8.3 Git branch naming example

Git branches are used to separate different lines of development and isolate changes made to the codebase, allowing developers to work on new features or fix bugs without affecting the main codebase until ready for integration. This allowed us to keep the development progress iterative and collaborate seamlessly within the same codebase. Git branches included the related issue number from the Jira board and looked like the example from Figure 8.3 above.

Whenever we wanted to push a change to main, another team member had to review the changes and approve something called a “merge request”. This way we had an extra layer of protection against merge conflicts (Gitlab, n.d.), bugs and were made aware of progress.

Good commit messages were important in terms of communication and collaboration purposes. The messages let us know what changes the given commit introduced and how the application was affected. An example of a commit message is shown in Figure 8.4 below.

Create DB functions to cache DataModels

Figure 8.4 example commit message

8.3.2 Bifrost Figma

We also attended a workshop about how we could use different Bifrost components within Figma. Figma is a powerful collaborative interface design tool which we used to create sketches. With the Bifrost design system integrated into Figma, it is possible to create sketches using the same components as the Bifrost React library. This made the sketches match what the website would look like. An employee at Intility helped us with creating basic wireframes and explained how we should design an Intility application using WCAG standard with similar design as other Intility applications. This was very useful to learn before we started sketching the first draft in Figma. Figure 8.5 below shows a design in Figma which we created using Bifrost

components. The menu on the left side of the figure is a Bifrost component which we have filled with suitable text and information based on the existing data at DBT docs.

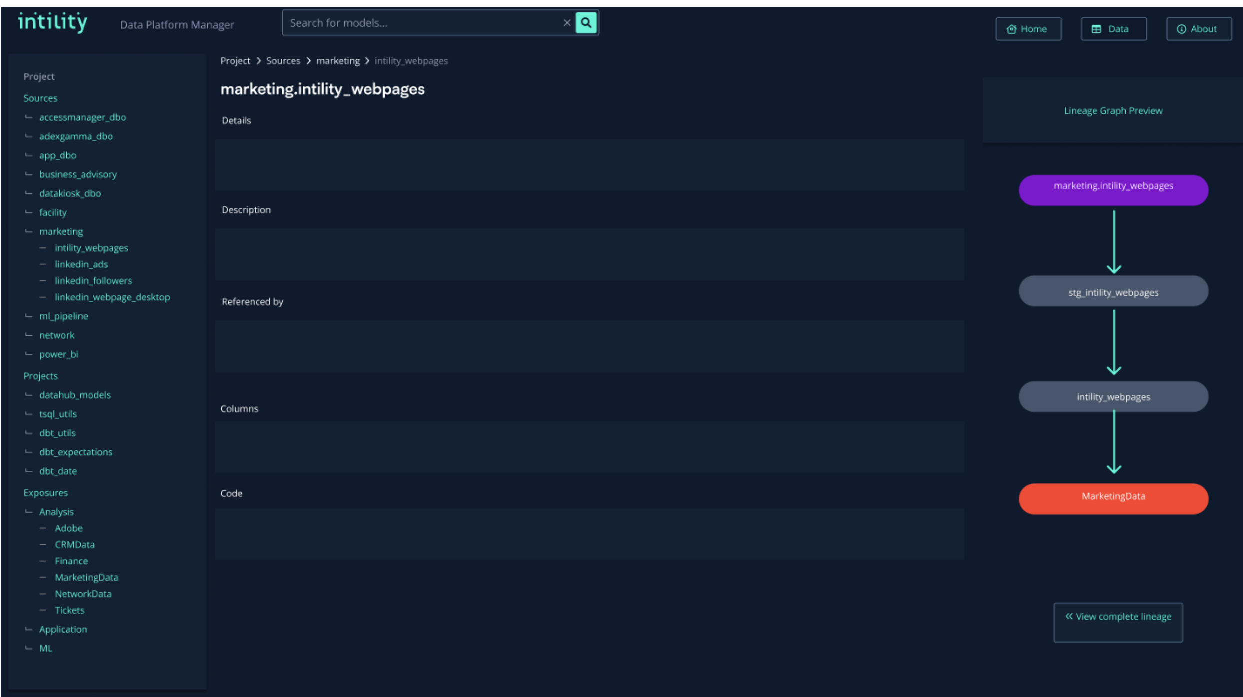


Figure 8.5 A sketch we designed in Figma after the Bifrost Figma workshop

8.3.3 Bifrost React

The last workshop worth mentioning in this report was the Bifrost React workshop, which explained the most useful and common Bifrost components and how we should implement the actual website in React based on Figma sketches. The host of the workshop demonstrated various examples of using components. We were also taught how to use Bifrost variables and properties in our components to style our applications with margins, padding, colors, and other design choices. The knowledge acquired at the workshop, gave us a general understanding of how we should utilize Bifrost React in our application.

Code snippet 8.1 shows an example of a Bifrost search box component from the documentation. The code is a starting point, and some of the code in the components needed a lot of changes to

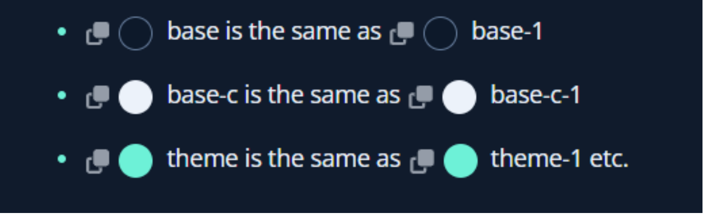


Figure 8.6 Naming conventions for CSS styling of Bifrost components

fit the needs of our application. Figure 8.6 shows an example of naming conventions used in Bifrost components in order to get the correct color theme.

8.4 Meetings with Intility

We had weekly synchronization meetings with two Intility employees, one developer and one data scientist. The developer handled technical and development-related questions, while the data scientist handled general domain and task related questions. These meetings were scheduled for one hour every Friday, but varied in length. Another important part of these meetings was to give understanding of how our application was built, as Intility will continue to develop it further.

By spending as much time as possible at the office it meant that we could have casual conversations and talk about ideas during lunch. This helped collaboration and gave us feedback aside from the user testing. See appendix Meeting minutes (Meeting minutes 13.01.2023, technical sync) for an example of meeting minutes from the meeting with Intility.

```
Search for...

Type in the search box, then click the button or hit Enter to initiate a search

Edit code

import { useState } from 'react'
import { faSearch } from '@fortawesome/free-solid-svg-icons'
import { Input, Grid } from '@intility/bifrost-react'

const [searchValue, setSearchValue] = useState('')
const [loading, setLoading] = useState(false)
const [submittedSearch, setSubmittedSearch] = useState('')

const simulateHttpRequest = () => {
  // do nothing if input field is empty
  if (!searchValue) return
  setLoading(true)
  // pretend to send a http request that takes 1s
  setTimeout(() => {
    setSubmittedSearch(searchValue)
    setLoading(false)
  }, 1000)
}

;<Grid>
  <Input
    value={searchValue}
    onChange={e => setSearchValue(e.target.value)}
    // trigger search on Enter keypress
    onKeyPress={e => {
      if (e.key === 'Enter') simulateHttpRequest()
    }}
    // trigger search on button click
    onIconClick={simulateHttpRequest}
    icon={faSearch}
    iconButton
    rightIcon
    label='Search with button'
    hideLabel
    placeholder='Search for...'
    clearable
    loading={loading}
    variant='outline'
  />
  <p>
    {submittedSearch
      ? `[search results for ${submittedSearch}]`
      : 'Type in the search box, then click the button or hit Enter to initiate a search'}
  </p>
</Grid>
```

Code snippet 8.1 Screenshot of the Bifrost component used to display the search box

8.5 Meetings with bachelor guidance counselor

A weekly status meeting with our guidance counselor Johanna Johansen was planned every Wednesday at 13.30. We sent a status report before each meeting, showing what we were working on and our progress. The focus of the meetings was feedback on the report, work done in the given sprint period and reporting on the progression of our work. Getting feedback on the report early, meant we could make corrections earlier. Two examples of helpful feedback

are: that we had to structure our text according to academic writing and explain the domain better. As we had worked with the project for months, a lot of concepts foreign to us in the beginning became self-explanatory. Based on this feedback, we decided to implement a glossary in the beginning of the thesis and write more descriptive introductions to our chapters.

An example of guidance meeting minutes can be found in Appendix C Meeting minutes (Bachelor meeting with guidance counselor, 25.01.23).

8.6 Internal meetings

We had several meetings when it was required, such as after meetings with Intility or the guidance counselor. This was to discuss the notes from the meetings and talk about any implications and further improvements we could make.

We had weekly or bi-weekly meetings scheduled for scrum meetings where we discussed the previous and upcoming sprint. In these meetings we set goals for the sprint, estimated how much time we had available, as well as potential blockers as described earlier in chapter 8.1 Scrum.

An example of internal meeting minutes can be found in Appendix C Meeting minutes

We also held daily standup meetings, talking about what we planned to work on that day. By doing so, all the group members were up to date on each other's progress, and everyone knew what they should work on. Occasionally, there was not enough work for everyone. In these cases, we extended the standup meetings to allocate more issues or re-allocate current ones. Appendix Meeting minutes (Meeting minutes 12.01.2023) shows examples of meeting notes from internal meetings.

8.7 Work environment

At the start of the project we were given a tour around the office and was introduced to the different departments within Intility. All group members got their own designated workspace with necessary equipment like monitors, headset, mouse, and keyboard. We also received laptops to be used for everything related to project work and the company.

Intility's offices are in Oslo, and therefore we were unable to travel daily from our homes in Gjøvik and Lillehammer. The commute takes about two hours each way by public transport. To help with the commute, the company offered for us to stay in their commute apartments every now and then to attend the office physically several days in a row. The expenses related to the trips were also covered by Intility which allowed us to attend the office three to four days a week on average.

An effect of being present physically in the office was a better relationship and collaboration with the client, and especially the data science department. The workspaces were located right next to the data science department, which led to seamless communication and assistance with the task. By having lunch with the department, we also had daily interaction with the employees about project related and non-project related topic

9 Summary

This chapter provides a summary of the results achieved during the development of the data platform manager. We present our results based on the goals we set ourselves in the Objective chapter, reflect on the process and discuss the potential for further work on the application. Furthermore, we reflect on what we have learned during this project and how it will benefit us in our future careers. Finally, we conclude with a discussion on the significance of this project and its potential impact on the organization.

9.1 Results

The purpose of this bachelor thesis was to create an application that would allow for a better understanding and collaboration within teams when working with data pipelines. We set out to achieve this goal by creating an application that could tie together automatically generated documentation from DBT with business system information and reports, while also including a feature displaying a visual representation of the relationships between various parts of a data pipeline. Additionally, we aimed to reduce the loading time of large lineage graphs and ensure that the data in the system was accurate and reliable.

Based on the final application, we have successfully achieved all our result goals. The application we developed can display all relevant information about a data pipeline by tying together automatically generated documentation from DBT with business system information which can be manually entered into the application. Furthermore, we were able to include a feature beyond the initial expectations, displaying a visual representation of the relationships within a data pipeline, while ensuring that the loading time of the visual representation takes less than 20% of DBT docs' alternative, as mentioned in chapter 5.11. We have confidence that the application we have developed will successfully achieve our effect goals in the long term, as it is yet to be launched for the entire company to use. We expect that the application will increase the adoption and usage of data documentation tools outside of the Data Science department, resulting in increased overall understanding and collaboration across teams and departments. The application's design complies with WCAG and Intility's brand, making it more accessible and easier to use.

We are pleased to report that we have also achieved our learning goals through this project. We gained valuable experience and knowledge in agile development and Scrum, Git best practices, version control, CI/CD and project management with strict deadlines. Additionally, we learned how to collaborate with a company to develop an application that meets their requirements and create a useful product in an unfamiliar domain such as Data Science.

In conclusion, the application we developed has successfully achieved all our goals, including our result, effect, and learning goals. Intility has already made plans on how to take the application in use, and we are confident that it will prove to be a useful tool for teams working with data pipelines and documentation. We are very grateful for the opportunity to work with Intility and the skills we learned along the way.

9.2 Process

The development process was very educational, and we learned to work on a project in the development industry (see chapter 9.3 for more details). However, if we were to carry out the project again, we would make some changes.

Primarily we would choose to spend more time in the planning phase, before starting to implement the functionality of the application. To utilize scrum more efficiently, we would start by composing a more thorough backlog of all known issues for the project's duration. This would help contain a better overall flow throughout the development. One reason we struggled to create a complete backlog was that the domain and current solution was so unknown to us. Another aspect of the task that could have been improved was communication with the client and gaining a better understanding between us and the client about specifications at an earlier stage. The requirements given at the start of the process did not fully align with what the client wanted, which led to some changes being introduced midway through the development. As a result, we had to adjust and iterate, to successfully incorporate changes based on the newly reformulated specifications. This taught us how requirements might change along the way, and about the advantages of using an agile approach as opposed to a waterfall methodology. Moving forward, we ensured that the requirements were made clear, and that there were no misunderstandings in the communication with the client.

9.3 Learning outcomes

During our work with Intility, we have had the opportunity to gain valuable insights and experiences that will undoubtedly benefit us in our future careers. Working in an established and relatively well-known technology company has provided us with a unique and valuable learning environment.

9.3.1 Work-experience in the industry

As previously mentioned in chapter 8.7 Work environment, we were fully integrated into the company, having our own designated desks, access to meeting rooms, and enjoying lunches with the Data Science department. Through these interactions, we gained a soft introduction to possible roles in our future work-life and learned how to effectively collaborate with stakeholders, clients and users of the final application. While we recognize that our future workplaces may present different circumstances, the knowledge and skills we acquired through our work with Intility have provided us with a solid foundation for our future careers.

Creating a product on demand was also new for us. Throughout previous courses, the larger assignments have not been strictly defined but encouraged a free approach on how to solve the task. For this project, we been having weekly meetings with Intility in order to meet their needs with the product that we developed for them. It is worth mentioning that the assignment has been very open, and we have decided a lot in discussion with the data scientists. This allowed us to have a say in what we were able to create with our designated time and the competence we had.

9.3.2 Collaboration

We had previous experience with working in teams before the bachelor's project. Working with a larger team and having the access to additional resources and assistance from different departments was a new aspect to us. When we encountered issues related to our development process, we were given the opportunity to contact the Intility employees who possessed the most knowledge about our problem. This made it possible for us to have significant progress within the given timeframe, as we were given the necessary support to solve issues that we

could not overcome on our own. Efficient collaboration was a major reason we were able to create such an extensive application in a relatively short timeframe.

9.3.3 Time management

During the development of our project, we faced several challenges that forced us to manage our time effectively. As we have never completed such a comprehensive and long-lasting project, we had to learn how to prioritize tasks, set realistic deadlines, and manage our workload to ensure that we stayed on track. We also had to learn how to balance our project work with our other academic and personal commitments, which was a valuable lesson in time management.

To help us manage our time effectively, we utilized a number of tools and techniques, including stand-up meetings most days, sprint planning sessions, and regular check-ins with our supervisor, as discussed throughout chapter 8 Development process and communication

We also tracked how much time we spent within each category of the project (Appendix F Timesheets). This helped us monitor our progress and identify what we spent most of our time on.

Overall, the project taught us how to manage our time effectively and work efficiently under pressure. These skills will undoubtedly be valuable later, as we face similar challenges in managing complex projects and meeting tight deadlines.

9.3.4 Project management

Throughout the project, we gained valuable experience in project management. We learned how to use agile methodologies such as Scrum to organize our work and prioritize tasks. We also developed our skills in communication and collaboration by having regular meetings with both our group members and the company stakeholders. Additionally, we gained knowledge in using project management tools like Jira to track our progress and manage tasks. These skills and experiences will be useful in our future careers as software developers, as project management is an important part of any software development project.

9.3.5 Technical skills

We were able to enhance and develop our technical skills in the development field, especially within .NET and TypeScript. We gained practical experience in building a front-end application using React TypeScript, which included creating components, working with state management, and implementing routing. We also learned how to work with various modern libraries and tools to make our development process more efficient, such as Bifrost and React Flow for React and Entity Framework and Swagger for .NET. Additionally, we learned how to use Visual Studio and Visual Studio Code more efficiently as our integrated development environments (IDEs). Overall, we gained valuable experience in developing a real-world application using these technologies and are now better prepared for future projects that require similar technical skills.

9.4 Future development

As mentioned in the introduction, part of why Intility wanted their own data platform manager application was to have the opportunity to customize and improve it with further features. During several meetings with the data science department, plenty of unique and useful features have been discussed. Not all these features were implemented during this bachelor project, mainly because of limited time, but also due to the other constraints described in chapter 1.4 Constraints.

To allow for further development in our code, we have made some conscious choices when programming. Front-end we have made sure to divide as much code as possible into components, which makes the code reusable and easier to maintain. We have also made smaller changes throughout the process, with further development and ease of change in mind. One specific example is whenever a new business system is added, the user must select what type of business system they want to add. When first implementing the form, the user had three hard-coded options to choose from, but we were told that there might be more types added later. We therefore made a change from radio buttons to a select list, where the options inside the select are stored in the database and can be fetched through the API. This way the

types can easily be changed inside the database, which will alter the options given in the select front-end.

We also made sure the back-end code was prepared for further development. We structured the project in folders separating API controllers, data layer and model layer as described in chapter 3.3.1 Back-end system design. This structure gives understanding both of how the system is built, and the separate domains within each layer. We also translated code which was frequently used into static classes where applicable. This means that changes to a function can be made in one place, making the code more modular and maintainable. As described in the chapter Back-end system design, we also used constants instead of hard-coding values. During the final weeks of the project, we also did a lot of small refactors to create documentation code and remove dead unreferenced code.

The main feature discussed for future development, is to make the data platform manager more interactive by adding an option to refresh a report. This would be very useful for anyone using the data within Intility, and potentially also for Intility's customers, by providing updated data in seconds, without the hassle of manually finding every model needing an update. Upon clicking a button, the data platform manager back-end would trigger a refresh on every model a given report is dependent on. Starting from the business system, and going through every source, staging, mart and exposure the selected report depends on.

Another useful feature would be to implement user roles, in order to add custom functionalities for users with administrator roles. Features such as creating and editing business systems, changing the glossary, or linking a source to a business system should be made available only to employees in the data science department. These features are currently available for to all users of the data platform manager, as we do not have the privileges to integrate roles, as discussed in chapter 1.4.2 Access to sensitive information.
Adding these features would make the application more complete, secure and user-friendly.

9.5 Significance

The data documentation tool we have developed during the bachelor thesis will make the workday for the employees easier. Firstly, with our tool, they will have less fragmented documentation. The tool will eventually make it possible to review and control the entire data pipeline. Secondly, the tool will make their work of finding various connections related to troubleshooting much easier.

The data science department gets many requests to find data and compile reports. This tool will make it possible for other Intility employees to find the data on their own, hence, reducing request going through the data science department. Instead, they could use more time on further development, and more meaningful tasks.

In order to have a competitive advantage in a marked with much competition, it is important to have a tool like the data platform manager tool. The tool makes it possible to have a good overview over the data pipeline and it will make it easy for users to find the correct data. If Intility starts using the data in this tool more frequently, it will be important to have an intuitive and user-friendly tool like data platform manager.

10 References

Ascendle Team. (2020, November 3). *Ascendle*. Retrieved from User Testing: How It's Done and Why: https://ascendle.com/ideas/user-testing-how-its-done-and-why/

Austin, D. D. (2021, March 11). *Towards Data Science*. Retrieved from Should You Use .includes or .filter to Check if An Array Contains an Item?: https://towardsdatascience.com/should-you-use-includes-or-filter-to-check-if-an-array-contains-an-item-1a8365dfc363

AVINetworks. (n.d.). *Kubernetes Load Balancer*. Retrieved from AVINetworks: https://avinetworks.com/glossary/kubernetes-load-balancer/

Barcheski, J., & Collis, S. (2023, March 30). *Analytics8*. Retrieved from dbt (Data Build Tool) Overview: What is dbt and What Can It Do for My Data Pipeline?: https://www.analytics8.com/blog/dbt-overview-what-is-dbt-and-what-can-it-do-for-my-data-pipeline/

Bellomy. (2020, June 9). *Bellomy*. Retrieved from Ethical research in usability testing: https://www.bellomy.com/blog/ethical-research-usability-testing

Berkely edu. (n.d.). *Most in Demand Programming Languages*. Retrieved from Berkely Education: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

Bhadwal, A. (2023, April 26). *hackr.io*. Retrieved from C# vs . Java: Which Language is Better to Learn?: https://hackr.io/blog/c-sharp-vs-java

Borozenets, M. (2022, April 8). *Vue vs. React — Comparison. What is the best choice for 2022?* Retrieved from Fulcrum: https://fulcrum.rocks/blog/vue-vs-react-comparison

Burnam, L. (2022, August 25). *user interviews*. Retrieved from Consent Forms for UX Research: A Starter Template: https://www.userinterviews.com/blog/how-to-write-research-participant-consent-forms

Campbell, A. (2021). *Data Science For Beginners.* Alex Campbell. Retrieved from Data Science For Beginners.

CloudFlare. (n.d.). *Hypertext-Transfer-Protocol*. Retrieved from CloudFlare: https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/

Comparison with Other Frameworks. (n.d). Retrieved from Vue.js: https://v2.vuejs.org/v2/guide/comparison.html?redirect=true

cshark. (2022, April 4). *cshark*. Retrieved from C# VS. GO - Which is better for your next application?: https://cshark.com/blog/c-vs-go-which-is-better-for-your-next-application/

Daniel Roth, R. A. (2022, 11 15). *Overview of ASP.NET Core*. Retrieved from Learn Microsoft: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0

Datatilsynet. (2021, October 12). *Datatilsynet.* Retrieved from Om personopplysningsloven med forordning og når den gjelder: https://www.datatilsynet.no/regelverk-og-verktoy/lover-og-regler/om-personopplysningsloven-og-nar-den-gjelder/

DBT docs. (2023, 05 03). *Exposures*. Retrieved from DBT docs: https://docs.getdbt.com/docs/build/exposures

DBT docs. (2023, 01 26). *Marts: Business-defined entities*. Retrieved from DBT docs: https://docs.getdbt.com/guides/best-practices/how-we-structure/4-marts

Exasol. (2020, September). Exasol Research Finds 58% of Organizations Make Decisions Based on Outdated Data. London.

Firesmith, D. (2020, January 13). *Carnegie Mellon University*. Retrieved from System Resilience Part 3: Engineering System Resilience Requirements: https://insights.sei.cmu.edu/blog/system-resilience-part-3-engineering-system-resilience-requirements/

GeeksForGeeks. (2023, March 31). Introduction to Recursion – Data Structure and Algorithm Tutorials.

Gitlab. (n.d.). *Merge Conflicts*. Retrieved from Gitlab: https://docs.gitlab.com/ee/user/project/merge_requests/conflicts.html

Gitlab. (n.d.). *Pipeline*. Retrieved from Gitlab: https://docs.gitlab.com/ee/ci/pipelines/index.html

Hoffman, C. (2019, 9 30). *What Is a Checksum (and Why Should You Care)?* Retrieved from howtogeek: https://www.howtogeek.com/363735/what-is-a-checksum-and-why-should-you-care/

IBM Corp. (2023, 02 23). *IBM Corporation*. Retrieved from Lesson 1.3: Identify actors: https://www.ibm.com/docs/en/idsa?topic=model-lesson-13-identify-actors

Intility. (2021). *Intility_Sustainable_platform-2021.pdf.* Retrieved from Sustainable Platform: Report on sustainability for 2021: https://intility.no/wp-content/uploads/2021/12/Intility_Sustainable-platform-2021.pdf

Intility. (2023). *Bifrost*. Retrieved from Bifrost - Setup: https://bifrost.intility.com/#/Welcome%20to%20Bifrost/Examples/Setup

Intility. (n.d.). *Intility*. Retrieved from Teknologi og digitalisering for en mer bærekraftig verden: https://intility.no/baerekraft/

Intility. (n.d.). *Baerekraft*. Retrieved from Intility: https://intility.no/baerekraft/

Intility. (n.d.). *Bifrost Design system*. Retrieved from Intility: https://bifrost.intility.com/

Intility. (n.d.). *Components Button*. Retrieved from Bifrost Intility: https://bifrost.intility.com/#/Components/Inputs/Button

Intility. (n.d.). *Components Pagination*. Retrieved from Bifrost Intility: https://bifrost.intility.com/#/Components/Interactive/Pagination

Intility, B. (2023). *Input Select*. Retrieved from Bifrost Intility: https://bifrost.intility.com/#/Components/Inputs/Select

Invedus. (2022, September 5). *Invedus*. Retrieved from Python vs .Net: Choosing the Right Language For Your Project: https://invedus.com/blog/python-vs-net-choosing-the-right-language-for-your-project/

JavaTPoint. (n.d.). *UML - Association*. Retrieved from JavaTPoint: https://www.javatpoint.com/uml-association

Kelleher, J. D., & Tierney, B. (2018). *Data Science.* The MIT Press.

Kolodiy, S. (2015). Unit Testing and Coding: Why Testable Code Matters. Moscow.

Kralj, K. (n.d.). *stub-vs-mock*. Retrieved from Method Poet Web site: https://methodpoet.com/stub-vs-mock/

Kubernetes. (2023, 03 28). *Pods*. Retrieved from Kubernetes: https://kubernetes.io/docs/concepts/workloads/pods/

Learn Typescript. (n.d.). *Controlling transpilation*. Retrieved from Learn TypeScript: https://learntypescript.dev/11/l3-transpilation

Linnæus University. (n.d.). *Linnæus University*. Retrieved from UML Domain Model: https://coursepress.lnu.se/courses/object-oriented-analysis-and-design/02-theory/domain_class_diagram

Loshin, P. (2021, September). *techtarget*. Retrieved from ASCII (American Standard Code for Information Interchange: https://www.techtarget.com/whatis/definition/ASCII-American-Standard-Code-for-Information-Interchange

Microsoft. (2022, 04 14). *Architectural principles*. Retrieved from Microsoft: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion

Microsoft. (2023, March 31). *Microsoft*. Retrieved from What is SQL Server Management Studio (SSMS)?: https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver15

Microsoft. (2023, March 3). *Microsoft*. Retrieved from Quickstart: Connect and query a SQL Server instance using SQL Server Management Studio (SSMS): https://learn.microsoft.com/en-us/sql/ssms/quickstarts/ssms-connect-query-sql-server?view=sql-server-ver15

Microsoft Learn. (n.d.). *FileStream Class*. Retrieved from Learn Microsoft: https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream?view=net-7.0

Microsoft. (n.d.). *What is .NET*. Retrieved from dotnet Microsoft: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet

Moran, K. (2019, Desember 1). *Usability testing 101*. Retrieved from Nielsen Norman Group: https://www.nngroup.com/articles/usability-testing-101/

Mortensen, D. H. (2021). *Interaction Design Foundation*. Retrieved from Conducting Ethical User Research: https://www.interaction-design.org/literature/article/conducting-ethical-user-research

Mozilla. (2023). Window: sessionStorage property.

Mozilla. (n.d.). *Mmdn web docs*. Retrieved from <button>: The Button element: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button#accessibility_concerns

Nishadha. (2022, 11 25). *UML Class Diagram Relationships Explained*. Retrieved from Creately: https://creately.com/guides/class-diagram-relationships/#inheritancegeneralization

Osman, J. (2022, November 11). JSON vs XML.

Paradkar, S. (2017). Understanding NFRs: scalability. In S. Paradkar, *Mastering Non-Functional Requirements* (p. 207). Packt Publishing.

Razvalinov, A. (n.d.). *UKAD*. Retrieved from ASP .NET Core 8 Pros and Cons: https://ukad-group.com/blog/aspnet-core-8-pros-and-cons/

React Flow. (n.d.). *React Flow*. Retrieved from React Flow: https://reactflow.dev/

Rosala, M. (2019, December 29). *NN / G Nielsen Norman Group*. Retrieved from Ethical Maturity in User Research: https://www.nngroup.com/articles/user-research-ethics/

Rubin, J., & Chisnell, D. (2008). *Handbook of Usability Testing.* Indiana: Wiley Publishing.

Rubin, J., & Chisnell, D. (2008). *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests, 2nd Edition.*

Rumbaugh, J. (2004). Domain Class Model. In J. R. Michael Blaha, *Object-Oriented Modeling and Design with UML* (p. 463). Pearson.

Sealights. (n.d.). *Software Maintainability*. Retrieved from Sealights: https://www.sealights.io/software-quality/software-maintainability-what-it-means-to-build-maintainable-software/

Sheth, H. (2021, 03 22). *NUnit vs. XUnit vs. MSTest: Comparing Unit Testing Frameworks In C#*. Retrieved from Lambdatest: https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/

Telerik. (n.d.). *Unit Testing for Software Craftsmanship*. Retrieved from Telerik: https://www.telerik.com/products/mocking/unit-testing.aspx

TutorialsTeacher. (n.d.). *TutorialsTeacher*. Retrieved from What is LINQ?: https://www.tutorialsteacher.com/linq/what-is-linq?utm_content=cmp-true

UMLBoard. (2023, 03 08). *Multiplicity*. Retrieved from UMLBoard: https://www.umlboard.com/docs/relations/multiplicity/

UXtweak. (2022, January 20). *UXtweak*. Retrieved from User research ethics: 7 principles of an ethical study: https://blog.uxtweak.com/user-research-ethics/

Vats, R. (2022, November 22). *upGrad*. Retrieved from Java Vs .Net: Difference Between Java and .Net: https://www.upgrad.com/blog/java-vs-net-difference-between-java-and-net/

Vitest. (2022). *Vitest Docs*. Retrieved from Why Vitest: https://vitest.dev/guide/why.html

11 Appendices

Appendix A Consent forms usability test 1
Appendix B Project Plan 13
Appendix C Meeting minutes 28
Appendix D Task description 35
Appendix E Pictures of Sprint boards 39
Appendix F Timesheets 42
Appendix G Data Science Expert on the *Change Log* feature 44
Appendix H Examples of Figma Sketches 45

Appendix A Consent forms usability test

Research Consent Form

STUDY DETAILS

The purpose of this study is for us to better understand the user experience of our products and services. Your participation in this study will help us modify, develop, or otherwise improve our products and services. This study will consist of a group interview with all four members of our research team.

DATA WE WILL COLLECT

We will ask you questions about your use of our products and services. We will not record the session, but we will take notes to record your comments. We will not request, and you should not provide, any sensitive personal information in this study.

HOW WE WILL USE YOUR DATA

Any data, recording or other personal information collected about you will be treated confidentially. We may use notes for internal purposes as we continue to improve our products and services. We may also anonymize your responses and aggregate them with the responses of other participants in order to share study results externally.

YOUR RIGHTS

Your participation in this study is voluntary. You can take a break or discontinue participation at any time without giving a reason. If you have any questions or concerns about this study or if you wish to withdraw your consent in the future, please email sanderthorsen76@gmail.com.

YOUR CONSENT I give my consent:

For the session to be documented

For individuals at our bachelor group to use the notes for internal purposes

For the bachelor group to aggregate and anonymize my data to share study results externally

By signing below, you acknowledge that you are 18 years of age or older and have read and understood the information in this Research Consent Form.

Transaction 09222115557492042720Signed JT

Verification

Transaction 09222115557492042720

Document

Research Consent Form - Data platform manager
Main document
1 page
Initiated on 2023-05-03 11:23:17 CEST (+0200) by Intility eSign (Ie)
Finalised on 2023-05-03 11:24:51 CEST (+0200)

Initiator

Intility eSign (Ie)
esign@intility.no

Signing parties

Joel Torby (JT)
Joel.Torby@intility.no
90909296

Signed 2023-05-03 11:24:51 CEST (+0200)

This verification was issued by Scrive. Information in italics has been safely verified by Scrive. For more information/evidence about this document see the concealed attachments. Use a PDF-reader such as Adobe Reader that can show concealed attachments to view the attachments. Please observe that if the document is printed, the integrity of such printed copy cannot be verified as per the below and that a basic print-out lacks the contents of the concealed attachments. The digital signature (electronic seal) ensures that the integrity of this document, including the concealed attachments, can be proven mathematically and independently of Scrive. For your convenience Scrive also provides a service that enables you to automatically verify the document's integrity at: https://scrive.com/verify

Research Consent Form

STUDY DETAILS

The purpose of this study is for us to better understand the user experience of our products and services. Your participation in this study will help us modify, develop, or otherwise improve our products and services. This study will consist of a group interview with all four members of our research team.

DATA WE WILL COLLECT

We will ask you questions about your use of our products and services. We will not record the session, but we will take notes to record your comments. We will not request, and you should not provide, any sensitive personal information in this study.

HOW WE WILL USE YOUR DATA

Any data, recording or other personal information collected about you will be treated confidentially. We may use notes for internal purposes as we continue to improve our products and services. We may also anonymize your responses and aggregate them with the responses of other participants in order to share study results externally.

YOUR RIGHTS

Your participation in this study is voluntary. You can take a break or discontinue participation at any time without giving a reason. If you have any questions or concerns about this study or if you wish to withdraw your consent in the future, please email sanderthorsen76@gmail.com.

YOUR CONSENT I give my consent:

For the session to be documented

For individuals at our bachelor group to use the notes for internal purposes

For the bachelor group to aggregate and anonymize my data to share study results externally

By signing below, you acknowledge that you are 18 years of age or older and have read and understood the information in this Research Consent Form.

Transaction 09222115557492041622Signed KS

Verification

Transaction 09222115557492041622

Document

Research Consent Form - Data Platform Manager
Main document
1 page
Initiated on 2023-05-03 11:15:23 CEST (+0200) by Intility eSign (Ie)
Finalised on 2023-05-03 11:15:39 CEST (+0200)

Initiator

Intility eSign (Ie)
esign@intility.no

Signing parties

Kristian Senneset (KS)
kristian.senneset@intility.no
91796583

Signed 2023-05-03 11:15:39 CEST (+0200)

This verification was issued by Scrive. Information in italics has been safely verified by Scrive. For more information/evidence about this document see the concealed attachments. Use a PDF-reader such as Adobe Reader that can show concealed attachments to view the attachments. Please observe that if the document is printed, the integrity of such printed copy cannot be verified as per the below and that a basic print-out lacks the contents of the concealed attachments. The digital signature (electronic seal) ensures that the integrity of this document, including the concealed attachments, can be proven mathematically and independently of Scrive. For your convenience Scrive also provides a service that enables you to automatically verify the document's integrity at: https://scrive.com/verify

Research Consent Form

STUDY DETAILS

The purpose of this study is for us to better understand the user experience of our products and services. Your participation in this study will help us modify, develop, or otherwise improve our products and services. This study will consist of a group interview with all four members of our research team.

DATA WE WILL COLLECT

We will ask you questions about your use of our products and services. We will not record the session, but we will take notes to record your comments. We will not request, and you should not provide, any sensitive personal information in this study.

HOW WE WILL USE YOUR DATA

Any data, recording or other personal information collected about you will be treated confidentially. We may use notes for internal purposes as we continue to improve our products and services. We may also anonymize your responses and aggregate them with the responses of other participants in order to share study results externally.

YOUR RIGHTS

Your participation in this study is voluntary. You can take a break or discontinue participation at any time without giving a reason. If you have any questions or concerns about this study or if you wish to withdraw your consent in the future, please email sanderthorsen76@gmail.com.

YOUR CONSENT I give my consent:

For the session to be documented

For individuals at our bachelor group to use the notes for internal purposes

For the bachelor group to aggregate and anonymize my data to share study results externally

By signing below, you acknowledge that you are 18 years of age or older and have read and understood the information in this Research Consent Form.

Transaction 09222115557492040958Signed HOVM

Verification

Transaction 09222115557492040958

Document

Research Consent Form - Data platform manager
Main document
1 page
Initiated on 2023-05-03 11:10:28 CEST (+0200) by Intility eSign (Ie)
Finalised on 2023-05-03 11:10:46 CEST (+0200)

Initiator

Intility eSign (Ie)
esign@intility.no

Signing parties

Hans Olav Vogt Myklebust (HOVM)
HansOlav.Myklebust@intility.no
45018465

Signed 2023-05-03 11:10:46 CEST (+0200)

This verification was issued by Scrive. Information in italics has been safely verified by Scrive. For more information/evidence about this document see the concealed attachments. Use a PDF-reader such as Adobe Reader that can show concealed attachments to view the attachments. Please observe that if the document is printed, the integrity of such printed copy cannot be verified as per the below and that a basic print-out lacks the contents of the concealed attachments. The digital signature (electronic seal) ensures that the integrity of this document, including the concealed attachments, can be proven mathematically and independently of Scrive. For your convenience Scrive also provides a service that enables you to automatically verify the document's integrity at: https://scrive.com/verify

Research Consent Form

STUDY DETAILS

The purpose of this study is for us to better understand the user experience of our products and services. Your participation in this study will help us modify, develop, or otherwise improve our products and services. This study will consist of a group interview with all four members of our research team.

DATA WE WILL COLLECT

We will ask you questions about your use of our products and services. We will not record the session, but we will take notes to record your comments. We will not request, and you should not provide, any sensitive personal information in this study.

HOW WE WILL USE YOUR DATA

Any data, recording or other personal information collected about you will be treated confidentially. We may use notes for internal purposes as we continue to improve our products and services. We may also anonymize your responses and aggregate them with the responses of other participants in order to share study results externally.

YOUR RIGHTS

Your participation in this study is voluntary. You can take a break or discontinue participation at any time without giving a reason. If you have any questions or concerns about this study or if you wish to withdraw your consent in the future, please email sanderthorsen76@gmail.com.

YOUR CONSENT I give my consent:

For the session to be documented

For individuals at our bachelor group to use the notes for internal purposes

For the bachelor group to aggregate and anonymize my data to share study results externally

By signing below, you acknowledge that you are 18 years of age or older and have read and understood the information in this Research Consent Form.

Transaction 09222115557492041645Signed CRL

Verification

Transaction 09222115557492041645

Document

Research Consent Form Data Platform Manager
Main document
1 page
Initiated on 2023-05-03 11:15:30 CEST (+0200) by Intility eSign (Ie)
Finalised on 2023-05-03 11:16:10 CEST (+0200)

Initiator

Intility eSign (Ie)
esign@intility.no

Signing parties

Christian Richard Lie (CRL)
Christian.Richard.Lie@intility.no
46415092

Signed 2023-05-03 11:16:10 CEST (+0200)

This verification was issued by Scrive. Information in italics has been safely verified by Scrive. For more information/evidence about this document see the concealed attachments. Use a PDF-reader such as Adobe Reader that can show concealed attachments to view the attachments. Please observe that if the document is printed, the integrity of such printed copy cannot be verified as per the below and that a basic print-out lacks the contents of the concealed attachments. The digital signature (electronic seal) ensures that the integrity of this document, including the concealed attachments, can be proven mathematically and independently of Scrive. For your convenience Scrive also provides a service that enables you to automatically verify the document's integrity at: https://scrive.com/verify

PROG2900 – Bachelor Thesis

Appendix B Project Plan

Intility – Data Platform Manager

Benjamin Loxley Wood

Martin Rui Andorsen

Sander Thorsen

Stian Tusvik

31.01.2023

1.0 Goals and restrictions

1.1 Background

Intility is a data driven company with a need to monitor their data and consume data services. By using data for reports and statistics they can make better business decisions and provide data for their customers.

Currently multiple tools are utilized to prepare data from different productions systems to make analysis and visualizations. DBT (Data Build Tool) is utilized to prepare raw data for analysis. DBT transforms raw data and combines it with other data to produce tables which can be utilized by tools such as Power BI for data visualization. DBT provides automatic context between raw data and analysis ready data in a manifest file. However, which production systems the raw data origins from, or in which analysis products the data sets are utilized is not provided. To follow the data from a production system to the final data visualizations you need to use multiple systems. This makes it difficult to track the origin of the data to the final dashboard. Intility wants to assemble these into a single system.

The tool currently in use also has limitations which could be improved upon. This includes the search function where Intility would want more relevant datasets to appear closer to the top of the search results. The UI can be improved upon to follow Intility design principles. Also, the load time of important features such as loading visual graphs could be faster for larger graphs.

1.2 Project Goals

1.2.1 Learning objectives

- Work as part of a larger development team in a professional business.

- Use tools such as React and Intility's tools to create an application.

- Create a user-friendly and well-designed application with Intility's design system but principles learned in classes such as User-Centered Design.

- Improve our domain modelling skills using code first principles from .NET.

- Utilize TypeScript to create safer and more robust software for front-end development.

- Become better at Azure cloud services and OpenShift

1.2.2 Result goals

The goal for this project will be to create a product which will be used by the company on an everyday basis. They utilize similar applications today, but these applications are slow and not very user-friendly to use. They also do not follow Intility's design standard. Our intention is to take instructions from the company as we develop the product in order to make it as useful as possible for them. The plan for the application is also to expand it after we finish our bachelor period, so we need to keep further development in mind.

1.3 Restrictions

Intility uses Gitlab for version control which comes with built in workflows for Continuous Integration/Continuous Delivery (CI/CD) which we will have to utilize in our project.

1.3.1 Back-end

Since Intility is a Microsoft based company, they want to have their technical stacks and knowledge consisting of the Microsoft stack. This means that for our back-end application we are required to use .NET framework and C# as the development platform. Our application will exist in the Intility cloud, and we will therefore be required to use Microsoft Azure as our cloud platform and Microsoft SQL server which is their standard practice.

1.3.2 Front-end

We are required to use React as our front-end library to create the UI, as well as Intility's own design system called Bifrost, which contains components conforming to Intility's design principles.

2.0 Scope of development

2.1 Academic fields

Data-modelling and databases

Our database will be a central part of the finished application. We need to develop an efficient model to store the necessary connections and information. For this our goal is to make good use of the knowledge acquired from the data-modelling and databases course. Another aspect of the course which will be essential is illustrating models in a good way in the final report.

Cloud Services

The application will be run in the cloud using Microsoft Azure. We did not specifically get an introduction to Azure in the cloud course, however we did use similar services. APIs were also a major part of the course, and we will be able to use many of the practices from the Cloud Services course to develop the API we need in this project.

WWW-Technologies

We are set to create an easy-to-use user interface containing many navigation and menu options, with visual representations of data and statistics. This front-end will be created using the React framework, as we learned in WWW-technologies. Instead of using JavaScript as in the course, we have decided to use TypeScript.

User-centered design

With the knowledge of the User-Centered design course, we plan to create user-tests for the intended users of the application to find points of improvement and get feedback throughout the development process.

Software development

To plan the project, we will use the Scrum method, as well as diagrams and models that we learned how to create in the software development course. The models we are going to create for this project include a Gannt chart, domain model, logical model and use case diagram.

Algorithmic Methods

The platform manager will feature a search function, most likely leveraging algorithms for a more efficient search. In this course we learned a lot of algorithms which might come in handy like a sorting algorithm.

2.2 Task description

Our task is to create a web application which provides an overview and documents all necessary information about data jobs. This means you should be able to see information about the process from the raw data, through a set of operations, until the final reports and models. Parts of the task are already present in a tool called DBT docs which Intility already uses, but our task is to create a superior documentation tool to replace DBT docs.

There are several problems with the current solution which our application (DPM) is going to improve on. Firstly, the current solution is limited only to the middle part of the data jobs. You are able to see steps from raw data to exposures, however you are unable to see the source of the raw data or in what reports and models the exposures are used.

Secondly, the current solution does not comply with Intility's design standards, which if changed would provide a more professional and complete feeling to the application.

Last but not least, DPM will be a base which opens the possibility for further development and improvements. Our application should be designed in such a way that it is possible to expand into a bigger and more useful set of tools, including for instance scheduling of data jobs and monitoring of tasks.

The current task description is flexible and opens for the possibility to expand if the students see fit.

2.3 Delimitations

DPM should contain the features specified in the task description and is initially limited to this. Implementation of scheduling and monitoring of tasks is outside the scope of this project. Our project goal is to make a tool to document data jobs, which means we will not be documenting the data jobs ourselves but will provide a way for Intility employees to do the documentation.

3.0 Project organization

3.1 Responsibilities and roles

Group leader – Stian Tusvik

Scrum master – Sander Thorsen

Meeting responsible – Sander Thorsen

Arbitrator – Benjamin Loxley Wood

Back-end responsible – Martin Rui Andorsen

Front-end responsible – Stian Tusvik

Product owner – Hans Olav Vogt Myklebust (Intility)

3.2 Schedule and group rules

Routines and rules within the group

- Document the hours spent, with a corresponding issue and a category in the timesheets (excel)
- Document deviations from the original deadlines.
- Show up on time to meetings and scheduled work on time.
- Give an early notice if you cannot attend a scheduled meeting
- Do the agreed work and let your group members know if you cannot do it on time or on your own so that prioritization can be made.
- Use Jira issues to document all the work you do in each sprint

- Document the code during development
- Take screenshots and document all work that could be relevant to the final report
- Work on the final report consistently

4.1 Project methodology

We have chosen Scrum as our project methodology since our project goal is not set in stone and specifications can change later. We chose to use Scrum as it is agile and based on an iterative workflow. Our project scope may potentially be revised later, and may implement additional features such as monitoring, and scheduling of data jobs. Because of this, scrum is a sensible choice of project management framework.

4.2 Meeting-plan

Meeting setup:

- **Sprint review meeting:**
    - Scrum meetings will be every Friday.
    - Martin will take notes from the meeting and Sander will lead the meetings.
    - Walkthrough of the goal of the sprint.
    - Walkthrough of what we did in previous sprint, demonstrate changes to show to the rest of the group so they can give feedback.
    - Discuss the tasks that were not completed, and the risks introduced by the delays.
    - Discuss what could be improved upon in the Scrum process.
    - Move the unfinished tasks over to the next sprint.
- **Sprint planning:**
    - How much time do we have in this sprint? (e.g., is someone sick, etc.)
    - Discuss main goal of sprint, (most important goal for progression)
    - Move tasks from backlog to sprint or make new tasks
    - Give an estimate time and assign all the tasks to the group members (all tasks should have at least one member assigned to the task)

To maintain good communication between the group and Intility, we plan to have a weekly technical sync every Friday. This is a good opportunity for us to ask about the application or technical issues, as well as getting feedback. There will also be a scrum meeting every Friday, where we will talk about the previous and upcoming sprint. In addition to this, meetings may be scheduled if deemed necessary.

We aim to be at the office in Oslo as often as possible, which should lead to easier collaboration and better group work. A weekly or biweekly meeting will be held with Johanna, our guidance counsellor, with a status report and explanation of further progress.

5.0 Organizing documentation and quality assurance

5.1 Documentation and version control

GitLab will be used for documentation and version control. All group members have experience with Gitlab, and it is Intility's standard version control system. In GitLab, we have resolved to create new branches for features and issues we are working on. Another member of the group will revise the merge requests and accept the merge if they do not find any errors. This will allow us to avoid losing code and committing unfinished work to the main branch. Everyone should have informative commit messages related to the issues and use the standard practice we learned during our introductory GitLab workshop at Intility.

For the issues in this project, we have decided to use Jira. Initially we used Gitlab Issues, which is Intility standard, but we found that this provided less information and was harder to use for Scrum. We have previously worked with Jira and found that it works well in a Scrum environment provided we utilize planning and retrospective meetings.

5.2 Plans for testing

After we have developed a functional MVP with some of the functionalities of the application, there will be performed user tests. Our plan is to have perform a user test on employees of the data science department to get feedback from the users on improvements to be made. The user tests will be performed on Intility employees since are familiar with the current system

and will be using the application going forward. We also plan to implement unit and integration tests directly in the code, with a focus on back-end testing. One of Intility's employees will give instructions on how they integrate tests in their code.

5.3 Risk analysis and countermeasures (technological, business and project group)

Risk Consequence Table

		Consequence				
		Negligible	Minor	Moderate	Significant	Critical
Likelihood	Almost certain					
	likely					
	Possible					
	Unlikely					
	Rare					

Risks

Risk	Description	Likelihood/Consequence
1	Losing a group member to sickness or other circumstances	Rare / Significant
2	Low productivity because of bad estimation / scrum usage	Likely / Moderate
3	Misunderstanding project requirements and business goals	Unlikely / Significant
4	No working product by deadline	Rare / Critical
5	Loss of source code	Rare / Significant
6	Unfamiliar technical stack can slow down development	Likely / Moderate
7	Changes to dependent software (DBT docs)	Unlikely / Significant

Risk Mitigation

Priority	Risk	Mitigation
Medium	1	Team members will be assigned specific responsibilities for parts of the system, while all the members will be working on the entire stack. This allows us to allocate resources in the project as it is needed, such as a critical feature or team member sickness.

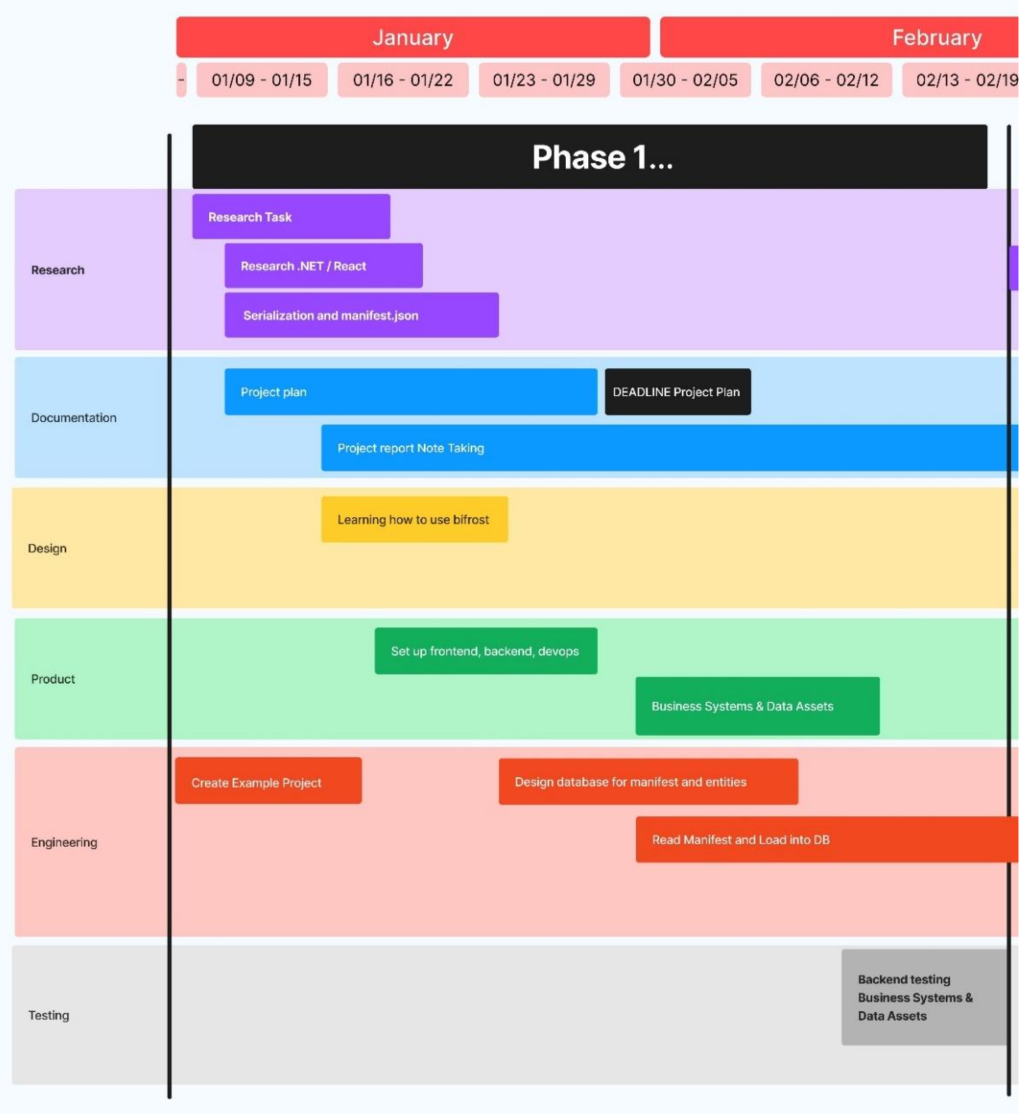
High	2	Utilize Jira for task allocation and provide feedback in the weekly scrum retrospective meeting to quickly adapt practices.
Medium	3	Biweekly meetings with Product Owner to sync our work progress and provide insight to our development.
Medium	4	Utilize CI/CD development in Gitlab to always have an application working as features are developed.
Medium	5	Use Git through Gitlab as source control working on features inside separate branches. Only delete and merge branches which successfully goes through the pipeline and have been reviewed.
High	6	Discuss issues in the weekly technical sync and prioritize problems by importance. Utilize workshops provided by Intility for learning. Technical conversations and discussions with senior engineers.
Medium	7	Breaking changes in DBT docs would make it necessary to make software changes. DBT docs is new software, but they have not had a history of making breaking changes, rather they make additional features. As of today (January 2023) there has been no changes to the manifest since its inception (August 2020). If a breaking change is introduced the project will remain on an older DBT version until it is fixed.

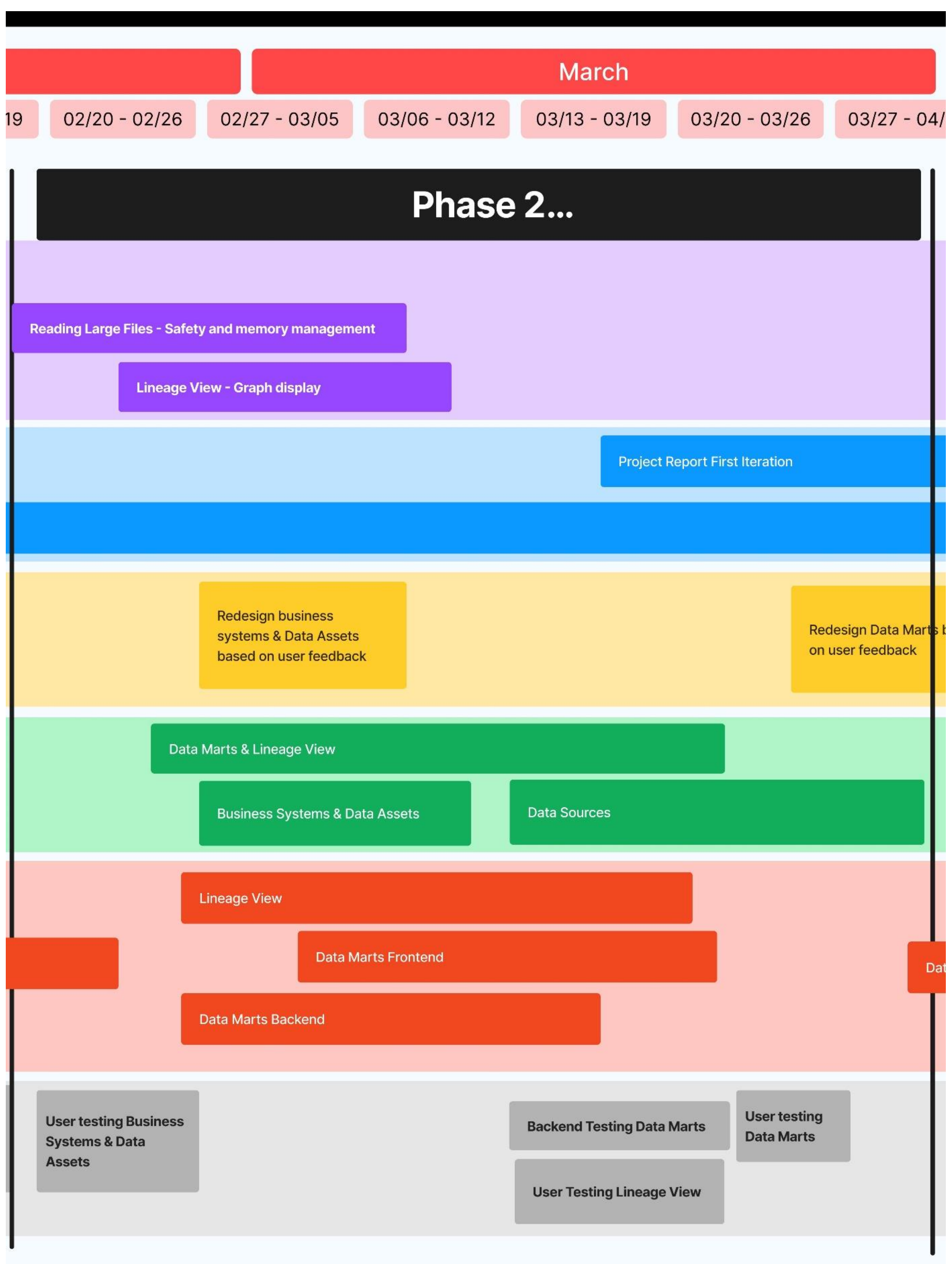
6.0 Plan for project execution

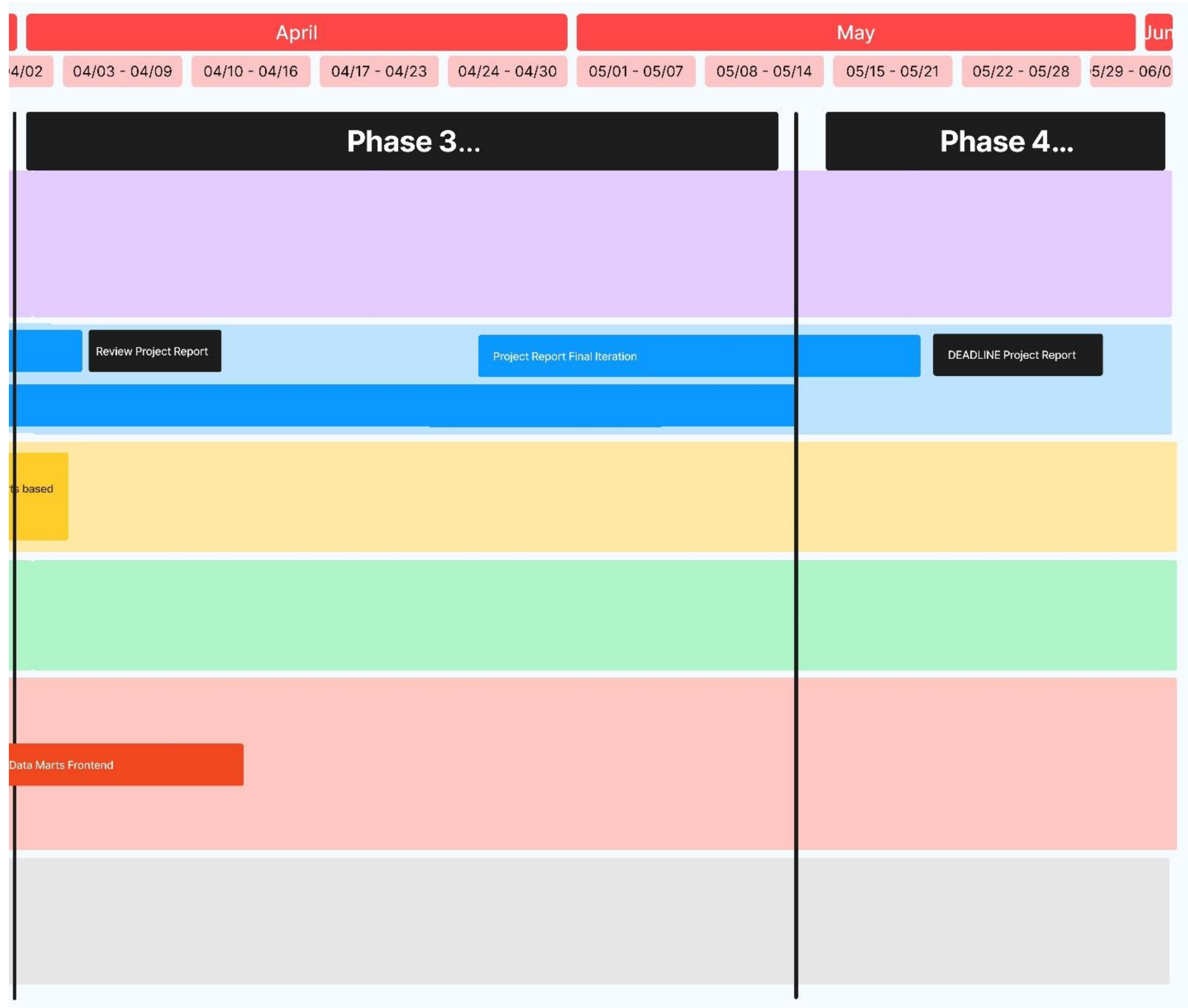
6.1 Gantt chart

Our Gantt chart is quite detailed in the first two phases, but less so in phase three. Because it is difficult to plan in detail for categories such as research, which will depend on specifications not yet set.

As the project develops and takes shape we will alter and add to the Gantt chart. Now in the starting phase the Gantt chart is an excellent tool to let us make a rough plan and make sure we develop features consistently.




March
19
02/20 - 02/26
02/27 - 03/05
03/06 - 03/12
03/13 - 03/19
03/20 - 03/26
03/27 - 04/
Phase 2...
Reading Large Files - Safety and memory management
Lineage View - Graph display
Project Report First Iteration
Redesign business systems & Data Assets based on user feedback
Redesign Data Marts b
on user feedback
Data Marts & Lineage View
Business Systems & Data Assets
Data Sources
Lineage View
Data Marts Frontend
Dat
Data Marts Backend
User testing Business Systems & Data Assets
Backend Testing Data Marts
User testing Data Marts
User Testing Lineage View



6.2 Activities

The different activities in the Gantt chart are all part of the development process. In the first phase, more time will be used for research, this is because there are a lot of new things to learn and choices to make. The documentation will be done incrementally as we develop the application with periods of more detailed report writing. This ensures that we can reflect on the choices we have made when developing the application as we write the report.

The first phase mainly consists of understanding the task, setting up tools, designing the application and domain modelling. This is an important step to make sure we have a unified understanding of our project and work towards the same goals.

In the second phase we shift towards software development and implementing our solution. Having user tests and getting feedback will be important as the third phase will consist of fine tuning our application and start working more on the report.

Some important milestones for this project are the delivery of the project plan and the final project report, MVP and user tests.

7.0 Tools

7.1 Back-end

The tools we are going to use for the back-end are Visual Studio Code and Visual Studio 2022 as the compiler and development environment. The programming language will be C#.

We are also going to use the tool *swagger* which will be used to develop our API. The advantage of Swagger is that you can quickly test the API as it creates documentation, example requests and data from the code. In the development phase this is an important tool to keep cost of change low.

7.2 Front-end

For front-end, Visual Studio Code is our preferred developer environment. For our front-end application we will be using the React framework written in Typescript.

We will design our application with Figma, and we are going to use the library Bifrost which is a design system created by Intility. Bifrost is also a react library meaning we can use the same components from our design system into our actual application. The advantage is that it allows us to create an application which promotes Intility’s design with the freedom to create our own components when suitable for the application.

7.3 Database

Azure SQL will be our chosen database in the internal business cloud. Our plan is to use Microsoft SQL Server Management Studio 18 to administer our database. To develop our database, we are determined to design our database and domain models in .NET, utilizing an Entity Framework to model our database.

7.4 Deployment

For deployment we will create a web application registration in azure to host our application and handle user authorization. Through Gitlab we plan to set up a connection to Azure and OpenShift to continually deploy our application as we develop. This allows us to work better in a Scrum environment as we can quickly deploy changes. We have also decided to set up some tests in Gitlab to ensure our code not only builds but follows best practices.

7.5 Documentation

Jira is our preferred tool for issue-tracking and backlog. Initially, each sprint will last one week with sprint planning meetings every Friday and a mid-week status meeting with our guidance counsellor.

We plan to take screenshots of our Jira board at the start of a sprint, and as we close the sprint to document our process. This will help us see long term trends in issue allocation and which issues we might struggle with.

As we develop and work on our project, we are going to take screenshots of models and code as we make changes. In this way, we can always look back and reflect on previous iterations. This process will be well commented on and discussed in the final report.

In excel, we have created a time log for each group member, where we will write down hours spent related to a category and linked to an issue.

Word will be used to write the project plan, the project report and other documentation.

7.6 Communication

All digital communication with the employees in Intility will be in Teams or Slack. Teams will also be used for workshops and setting up group meetings. We have decided to use Discord as our main communication channel within the group. This is because we have created an organization communication server with different text channels in order to share things like meeting notes and important files between group members.

Appendix C Meeting minutes

Meeting minutes 12.01.2023:

Timesheets, setting up timesheets where everyone should link their hours to their issues, and categories their hours spend with a comment.

Get familiar with the DBT repo and learn how we can get the correct data we need for the data platform manager

Who are going to be group leader, do we actually need a leader, everyone is good at taking responsibility

Sander is meeting and scrum master

Martin responsible for back-end

Stian responsible for front-end

Stian will most likely end up as group leader

Benjamin and Sander are developing hybrid on both front-end and back-end, but mainly Sander on front-end and Benjamin on back-end.

Intern contract, set clear expectations, rules and workflow.

Contact advisor, send mail and schedule weekly meetings.

Keep an eye at the teams calendar, go a lot of meetings the next weeks.

I the start of the project (12.01.2023) get familiar with the assignment and create an informative and good plan before we start to develop.

Put together a document with things we wonder about the assignment, such that we can get answers to this from Intility.

Create issues and start with scrum meetings

Focus a little bit more on the project report than the development process.

Meeting minutes 13.01.2023 (technical sync):

Mocking data from manifest.json file

Want similar solution to the DBT, but need to find out how to get the data from manifest

Want the linear graphs to be similar

Extract the data within each model

Should get familiar with the manifest file

Save the connections in a database

Run the manifest through something, and get the data, the next day, send the file through again and get the updates if any

First step back-end, get info from manifest, put it in the database, run cron jobs on the night to get automated updates.

Bachelor meeting with guidance counselor (25.01.23):

Group rules

what's each groups members responsibility

How often do we meet

Hours work

What's happening if someone do not follow the group rules

Adding the group rules to the project plan

Status report sent to Johanna minimum 1 day before the meeting

Argument around our technologies

Look for research articles to defend our "choice"

look at our assingment in a bigger perspective

Take notes when researching the different technologies

How to take notes: sokogskriv.no

https://i.ntnu.no/bacheloroppgave

Technical sync 27.01.23 (With Christian Richard Lie, Kristian Senneset, Martin Rui Andorsen, Benjamin Loxley Wood, Stian Tusvik and Sander Thorsen):

Prettier config file (look at Chris project)

Set up database and data asset

Data assets could possible only be a searching site, could be hard to have a public search function in the header

Business system:

Data sources: Connection between the information in business system and the content of the staging tables

All the sources in the manifest file

Connect all the sources to the business system (manual work)

No suggestion on how to do this yet

Info on how it's used in DBT and what it gets from marts

Data marts: More like DBT docs

get's details from DBT marts, graph and how everything is connected

Data marts: be a little bit like data assets

Extract and load tasks: Last step after the rest, there are also some points that could be added if we have enough time

Data Consumers: The content of the manifest file

Want the description of which datasets is used in which reports. Could be loaded from a table.

Help site with guides: Explanation of the content on the other sites, links, pages with general information, specific information of power bi

Ideas for something innovative for Intility/the field:

- Front-end testing
- Lineage graph
- Search functions
- Manifest file and updating the database
- Search for new trends and fresh innovation in the code world
- Obtain documentation from Hans Olav on how much the work potentially can help the company in increased efficiency and saved resources
- Notification system for changes made in the last x number of hours/days

Sprint retrospective

Date of sprint-period: 03.02.2023 - 17.02.2023

Group members attending the meeting:

Sander Thorsen

Stian Tusvik

Benjamin Loxley-Wood

Martin Rui Andorsen

Sprint review

Goal of the sprint:

a. Front-end: MVP for business systems, data assets, data source and help Back-end: Integration with business systems, load in the relationship between the models
b. Domain model, Use case diagram(draft) and Logical model

Go through the previous sprint and progress.

Sander has worked on Business System endpoints. We need to create some DTOs for the Database models to avoid data redundancy and messy JSON responses.
Martin gives an overview of how .net DB classes work and the need for DTOs.
Benjamin has worked on Data Assets front-end and sorting functions. Shows us his progress.
Stian shows us his front-end work on Business Systems. Business systems contains sources, and the next functionality is to be able to choose sources from a list of available sources to easily add new ones.
Business Glossary is a simple page with glossary with description of its meaning. Table is currently dynamic in the space usage of the different fields which can look strange. Adding minimum width is an option.

Is there anything we did not manage to do. Could this lead to any risks?

c. Some Endpoints needs optimizing, an example is to be able to add a list of sources to business systems.
d. Setting up a testing environment was more difficult
e. Front-end: Edit owner and contacts to use previously added people.

   Risk of these issues is that things are taking slightly longer than needed, leaving less time for making superb work and creating a good application.

Move the issues in the right section in Jira

Sprint Planning

How much time do we have this sprint? (Are there any that could not attend due to sickness or any other thing)

Nothing to note

1. Main goal of the sprint, what is important for the projects progress?

   Create DTO classes and make.

   Update manifest information functionality.

   Make front-end MVP ready for user testing.

2. Move issues into the sprint from the backlog and create new issues if there are needed new ones for the main goal.

3. Estimate how much time each of the different issues will take and make sure that there is an assignee to the different issues.

Appendix D Task description

Data platform manager

This task belongs to the Data Science department in Intility. As part of the work to become a data-driven company, we need a tool that makes it easy to develop, manage, and consume data services. The Data platform manager should be a tool that includes, among other things:

- Data catalog and documentation
- Data lineage
- Monitoring of data jobs
- Scheduling of data jobs

The project will require both back-end and front-end. The back-end will consist of gathering and compiling data across various tools we use in our data platform. You are not locked into any technology, but it is desirable to use Python or C#. The front-end will display the compiled data in a portal interface. Here, you will use Intility's design system, built on React.

The project is well-suited for those interested in data science.

Detailed description

The project will primarily consist of presenting data that has already been collected and is easily accessible. The project does not involve documenting data and data products but rather compiling what already exists.

Data catalog and documentation

Intility uses the Data Build Tool (DBT) to transform "raw data" into analysis-ready data. Raw data is transformed and combined with other data, producing a table (also called a model in DBT) that goes into, for example, Power BI. All relationships between raw data and analysis-

ready data are automatically documented in DBT in a manifest. This manifest is the basis for our data catalog. When we develop models in DBT, we do the actual documentation in our work in YAML files, and it may look like this:

```yaml
145
146    - name: auto_dispatch_testing_cleaned_messages
147      description: "Unique cleaned messages in AutoDispatch tests."
148      columns:
149        - name: cleaned_message_id
150          description: "The primary key for this table"
151          tests:
152            - not_null        Joel Torby, 4 weeks ago • Auto Dispatch tests data models added to staging …
153            - unique
154        - name: cleaned_message
155          description: "Cleaned message from AutoDispatch model. These need to be unique in order for python script which populates data, to work."
156          tests:
157            - not_null
158            - unique
```

Here, a model has a name and description, and we can provide a description for each column if we wish. We can also run tests on columns to ensure data quality. All this information will also be included in the manifest. An example of what a model looks like in the manifest is in the attached file manifest_example.json.

Data lineage

In DBT, we get a good relationship between raw data and analysis-ready data, but we do not have a good overview of where the raw data comes from (production system) and in which analysis products the data is used. It is therefore difficult to determine the actual use of data and to trace the origin of the data.

In Power BI, the tool Intility uses to visualize data, there is lineage from dataset to report to dashboards (lineage: dataset -> report -> dashboard). A dataset can be used in one or more reports, and a dashboard consists of elements from one or more reports. However, there is no clear link between analysis data (produced by DBT) and datasets in Power BI, and if you want to follow the path from dashboard down to raw data, you have to go through several different systems. We want this to be collected in one place.

The raw data collected for analysis purposes is in various systems owned by a department in Intility. We also want the ability to link the collected raw data to the production system from which the data originally comes.

Production data -> collected raw data -> analysis-ready data -> data product

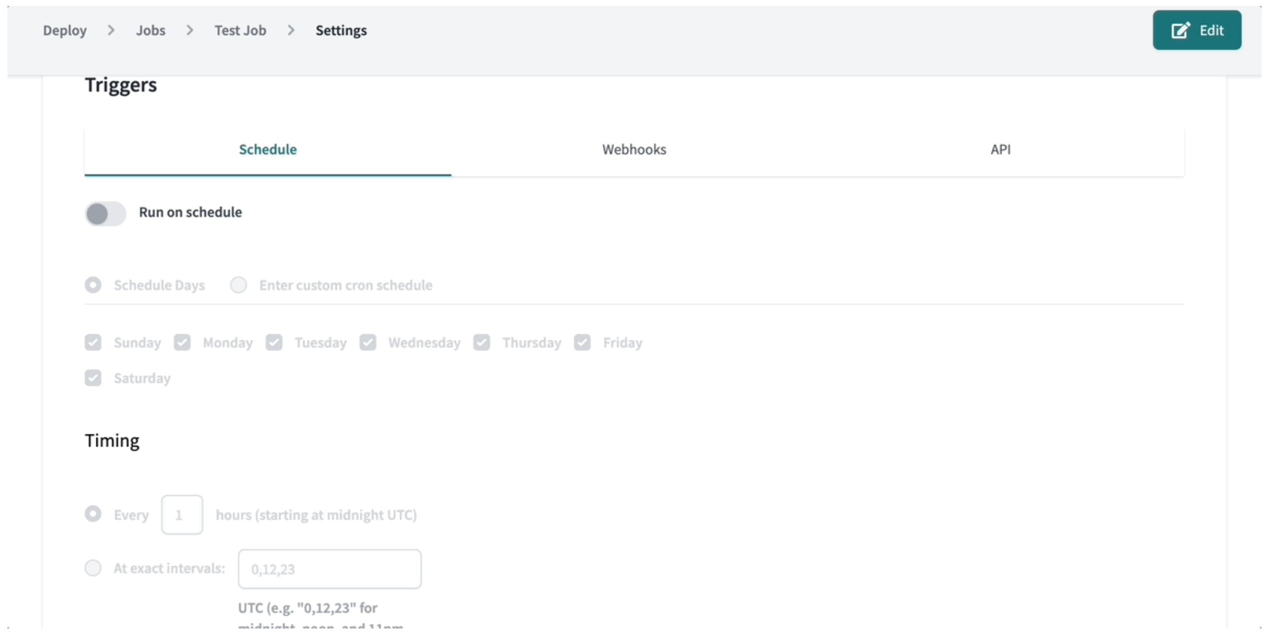
Monitoring of data jobs

During a day, a number of jobs are run that fetch data from various production systems and produce analysis data with DBT. All these jobs can fail at different times, and we want to have as good visibility as possible on this. We have good routines for logging and storing results from different jobs, so the main task here will be to lift this data up and tie it to the data catalog with lineage so that you can see what is affected by this failure.

Scheduling of data jobs

Today we have data jobs running in both openshift and a server environment. These are set up as cron jobs triggered at regular and irregular intervals. It happens that requirements for how often these jobs should run change, and that completely new jobs need to be scheduled.

Initially, the focus is on scheduling DBT jobs. Here we envisage taking some inspiration from DBT Cloud, see the gif below. More information about DBT cloud can be found here: About DBT Cloud | DBT Developer Hub (getdbt.com). You should then be able to change the current schedule of jobs, as well as add new jobs.

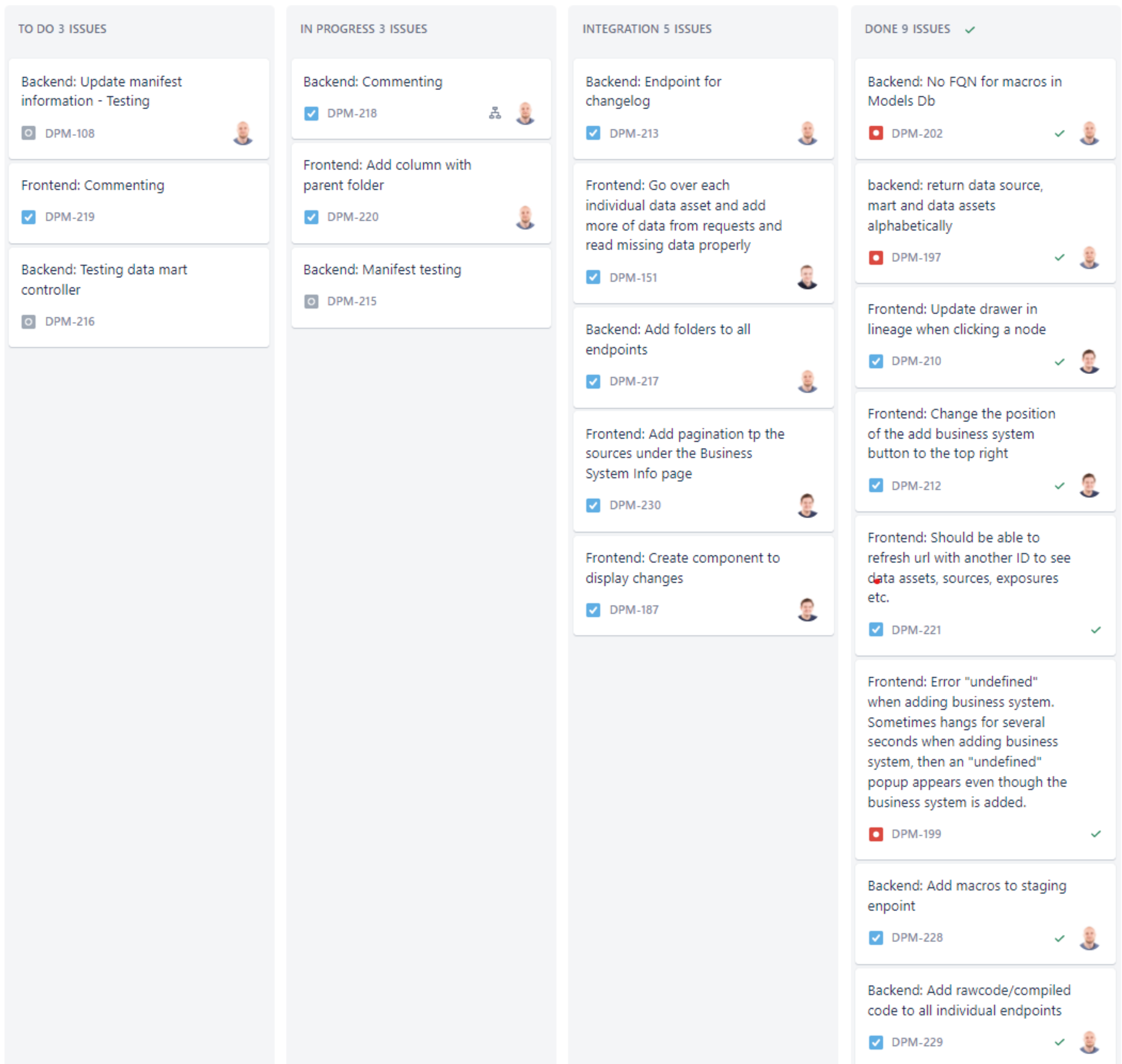


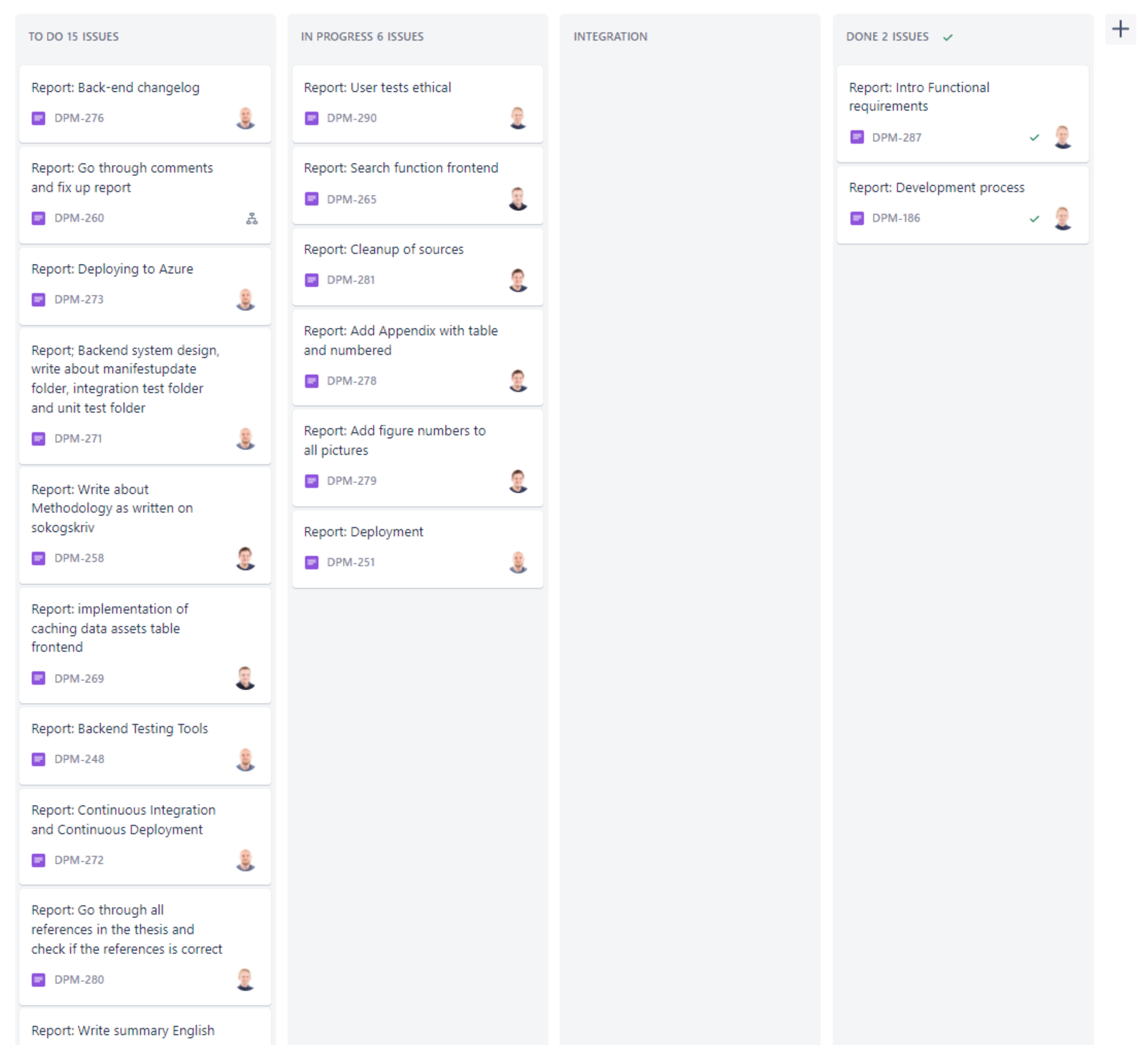
Resources

Introduction to DBT that can provide some better understanding: [Intro to Data Build Tool (DBT) // Create your first project! - YouTube]

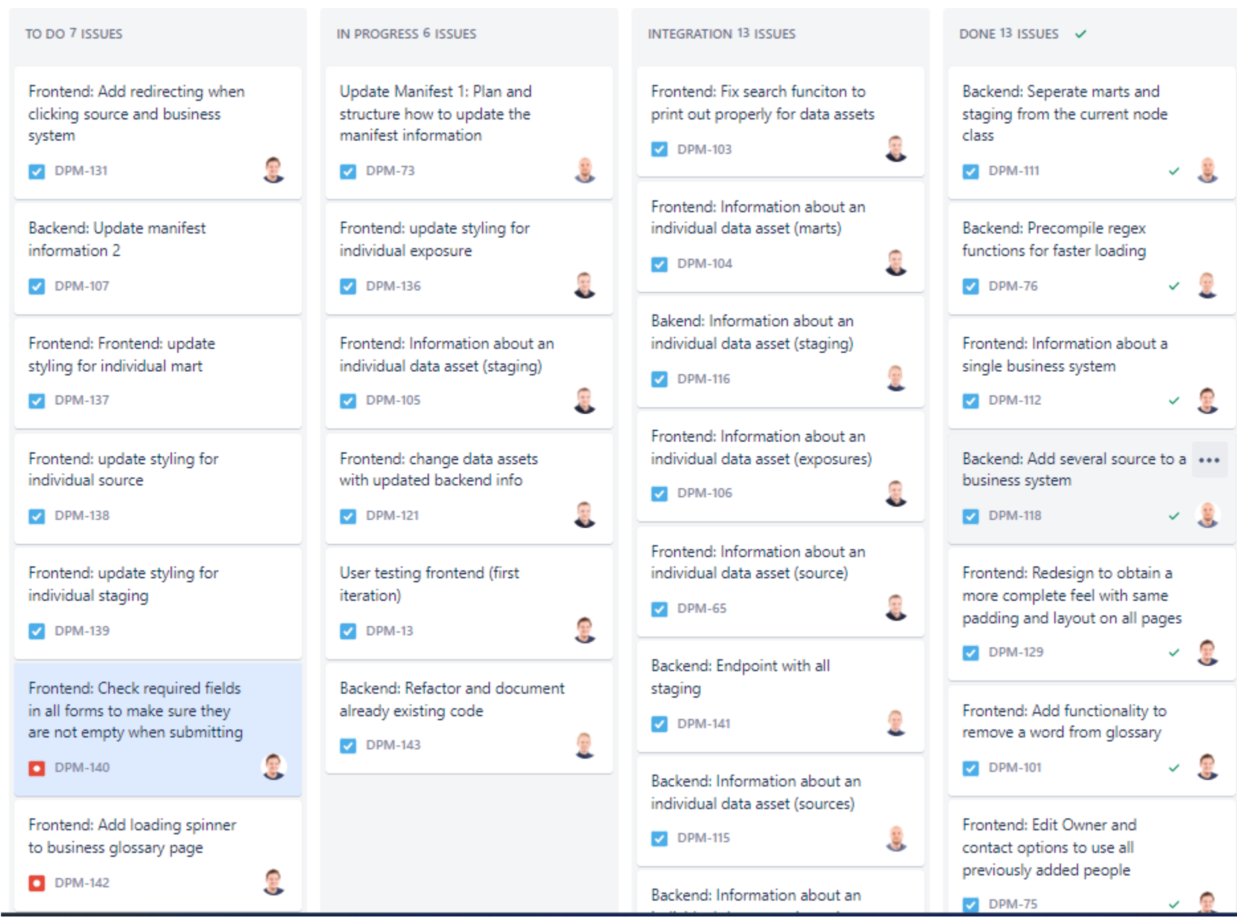
Any questions about the task can be sent to hansolav.myklebust@intility.no.

Appendix E Pictures of Sprint boards





TO DO 15 ISSUES
Report: Back-end changelog
DPM-276
Report: Go through comments and fix up report
DPM-260
Report: Deploying to Azure
DPM-273
Report; Backend system design, write about manifestupdate folder, integration test folder and unit test folder
DPM-271
Report: Write about Methodology as written on sokogskriv
DPM-258
Report: implementation of caching data assets table frontend
DPM-269
Report: Backend Testing Tools
DPM-248
Report: Continuous Integration and Continuous Deployment
DPM-272
Report: Go through all references in the thesis and check if the references is correct
DPM-280
Report: Write summary English
IN PROGRESS 6 ISSUES
Report: User tests ethical
DPM-290
Report: Search function frontend
DPM-265
Report: Cleanup of sources
DPM-281
Report: Add Appendix with table and numbered
DPM-278
Report: Add figure numbers to all pictures
DPM-279
Report: Deployment
DPM-251
INTEGRATION
DONE 2 ISSUES
Report: Intro Functional requirements
DPM-287
Report: Development process
DPM-186

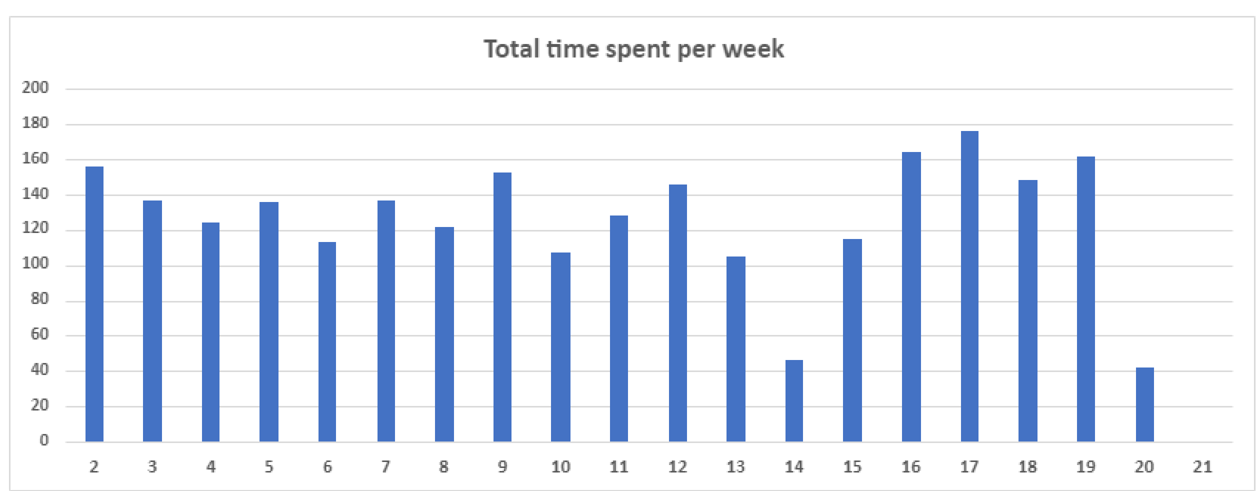


TO DO 7 ISSUES
Frontend: Add redirecting when clicking source and business system
DPM-131
Backend: Update manifest information 2
DPM-107
Frontend: Frontend: update styling for individual mart
DPM-137
Frontend: update styling for individual source
DPM-138
Frontend: update styling for individual staging
DPM-139
Frontend: Check required fields in all forms to make sure they are not empty when submitting
DPM-140
Frontend: Add loading spinner to business glossary page
DPM-142
IN PROGRESS 6 ISSUES
Update Manifest 1: Plan and structure how to update the manifest information
DPM-73
Frontend: update styling for individual exposure
DPM-136
Frontend: Information about an individual data asset (staging)
DPM-105
Frontend: change data assets with updated backend info
DPM-121
User testing frontend (first iteration)
DPM-13
Backend: Refactor and document already existing code
DPM-143
INTEGRATION 13 ISSUES
Frontend: Fix search funciton to print out properly for data assets
DPM-103
Frontend: Information about an individual data asset (marts)
DPM-104
Bakend: Information about an individual data asset (staging)
DPM-116
Frontend: Information about an individual data asset (exposures)
DPM-106
Frontend: Information about an individual data asset (source)
DPM-65
Backend: Endpoint with all staging
DPM-141
Backend: Information about an individual data asset (sources)
DPM-115
Backend: Information about an
DONE 13 ISSUES
Backend: Seperate marts and staging from the current node class
DPM-111
Backend: Precompile regex functions for faster loading
DPM-76
Frontend: Information about a single business system
DPM-112
Backend: Add several source to a business system
DPM-118
Frontend: Redesign to obtain a more complete feel with same padding and layout on all pages
DPM-129
Frontend: Add functionality to remove a word from glossary
DPM-101
Frontend: Edit Owner and contact options to use all previously added people
DPM-75

Appendix F Timesheets

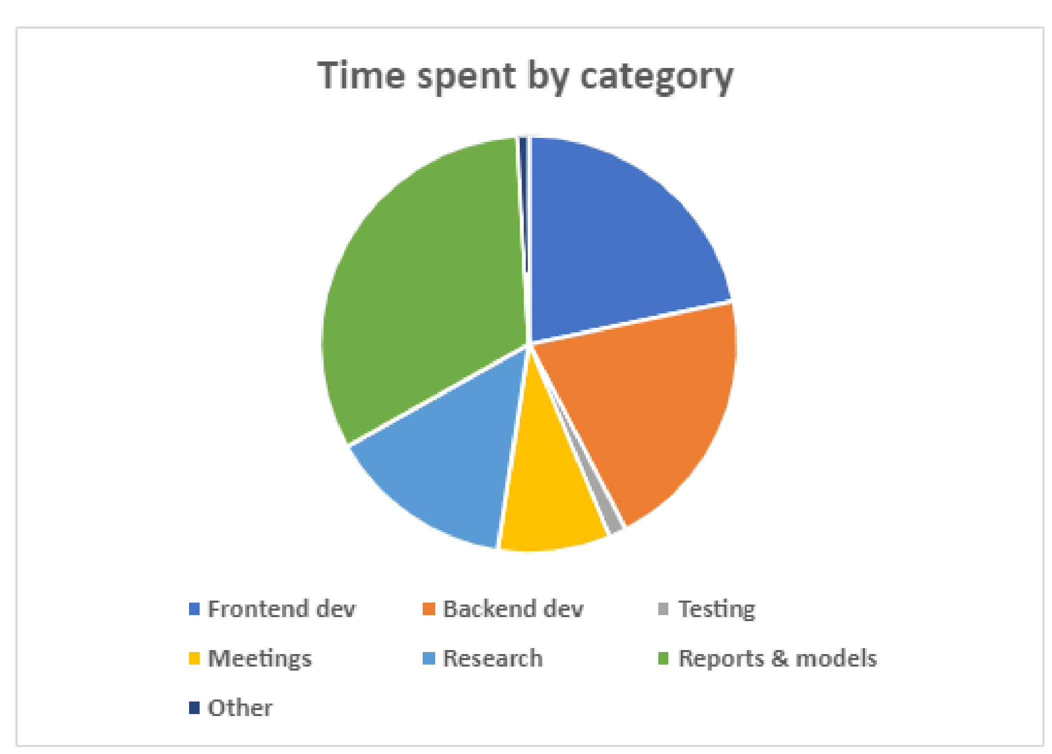
A full overview of detailed time sheets for each team member can be seen by clicking the following link:

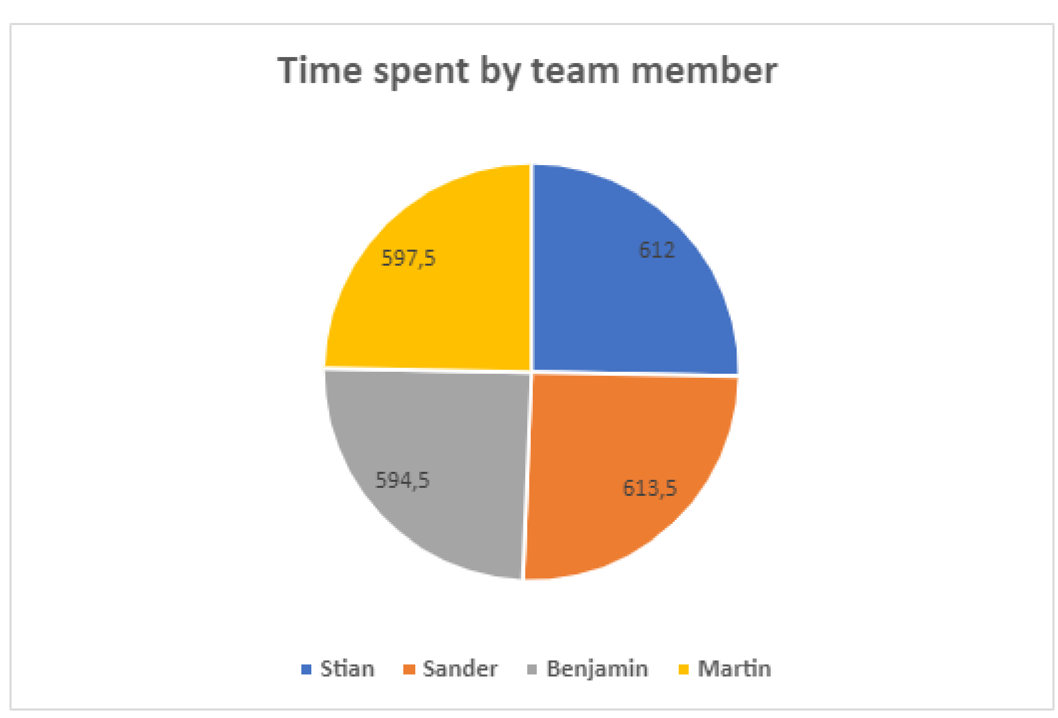
https://studntnu-my.sharepoint.com/:x:/g/personal/sanderth_ntnu_no/EbKCya2cYn1NpGztiwohQiwBWfNu0ijVH4Zc6doQOAeUZw?e=bdLT5A



Total hours

Week \ Category	Frontend dev	Backend dev	Testing	Meetings	Research	Reports & models	Other	Total
2	4	11	0	47	62	15	17	**156**
3	10	38	0	20	64	5	0	**137**
4	21	9,5	0	23	27	42	2	**124,5**
5	40	46	1	9	21	18	1	**136**
6	46	29	0	1	31	6,5	0	**113,5**
7	51	41,5	6	13,5	17	6	2	**137**
8	50	44	3	9	15,5	0	0	**121,5**
9	53	55	6	25	14	0	0	**153**
10	45	29	4	6	9	14	0	**107**
11	39	35	11	7,5	17	19	0	**128,5**
12	45	48,5	0	6	18	28	0	**145,5**
13	20	19	0	6	5	55	0	**105**
14	5	5	0	0	4	32	0	**46**
15	36	42	0	7	3	27	0	**115**
16	59	47	2	11	7	38	0	**164**
17	0	0	0	4	20	152	0	**176**
18	0	0	0	9	12	127	0	**148**
19	0	0	0	7	2	153	0	**162**
20	0	0	0	0	0	42	0	**42**
21	0	0	0	0	0	0	0	**0**
Total	**524**	**499,5**	**33**	**211**	**348,5**	**779,5**	**22**	**2417,5**


Time spent by category
Frontend dev
Backend dev
Testing
Meetings
Research
Reports & models
Other


Time spent by team member
612
613,5
594,5
597,5
Stian
Sander
Benjamin
Martin

Appendix G Data Science Expert on the *Change Log* feature

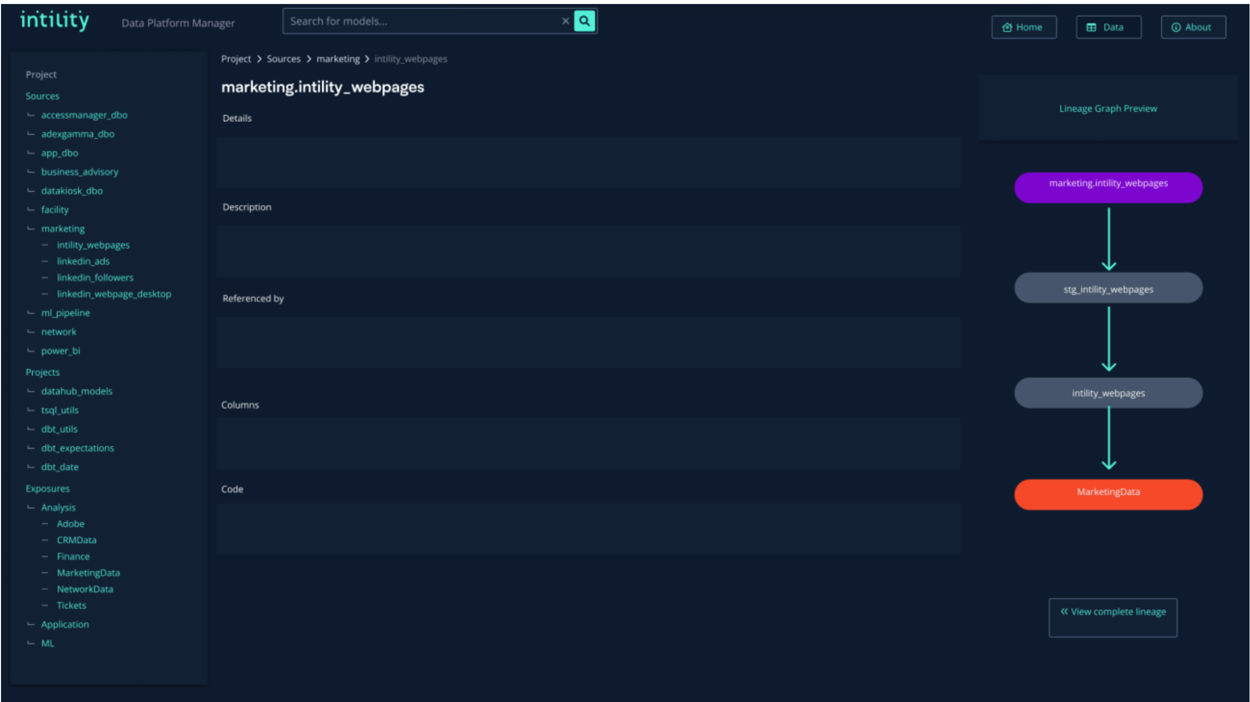
The 'change log' feature in DPM will be a increasingly helpful tool in our data catalog. Currently, we have no clear and easy way to see what new data models, or changes to existing models, that have been added to our data warehouse, which can often lead to problems and errors in downstream use of the data, as well as potential duplicate work. The benefit of the change log will grow as our Data team expands, as more and more people will develop their own models, by adding or changing data in the process, so this will help everyone to be continuously updated on new changes.

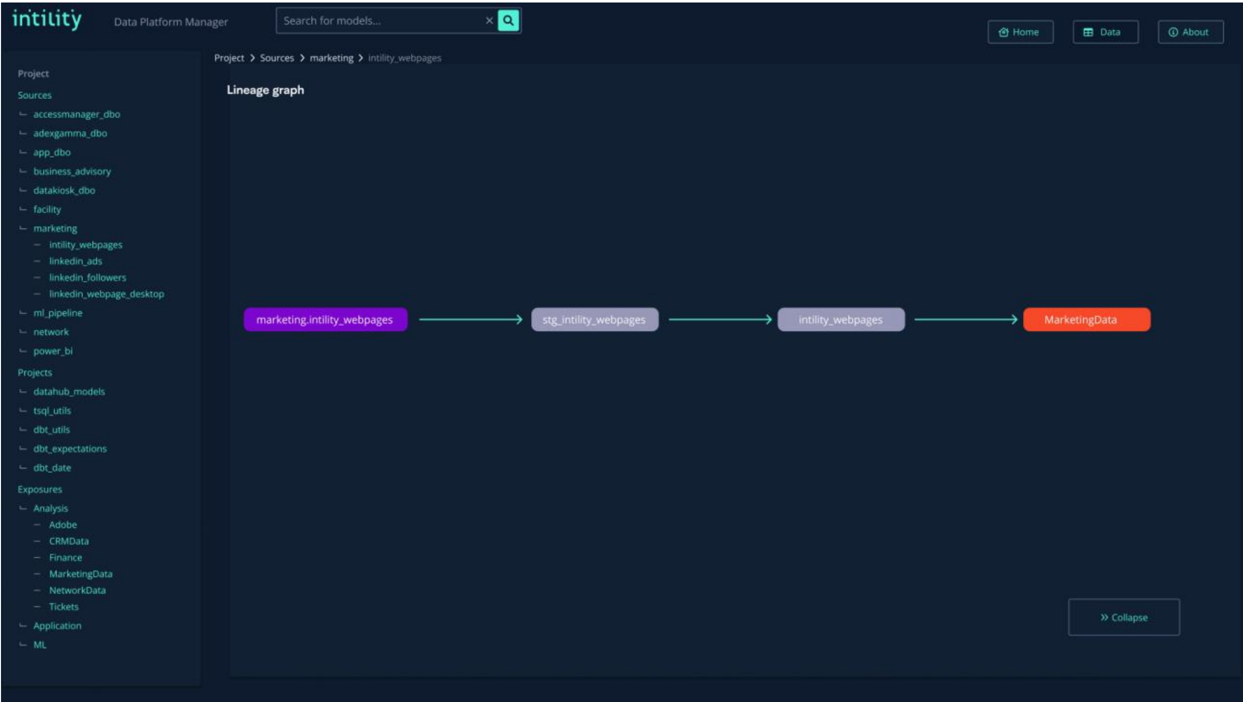
When planning the features of data catalog project, we did not include a change log, since we had not seen this functionality in the data catalogs we used for inspiration. Thus, we were enthusiastic when the group suggested the feature, and accepted it immediately.

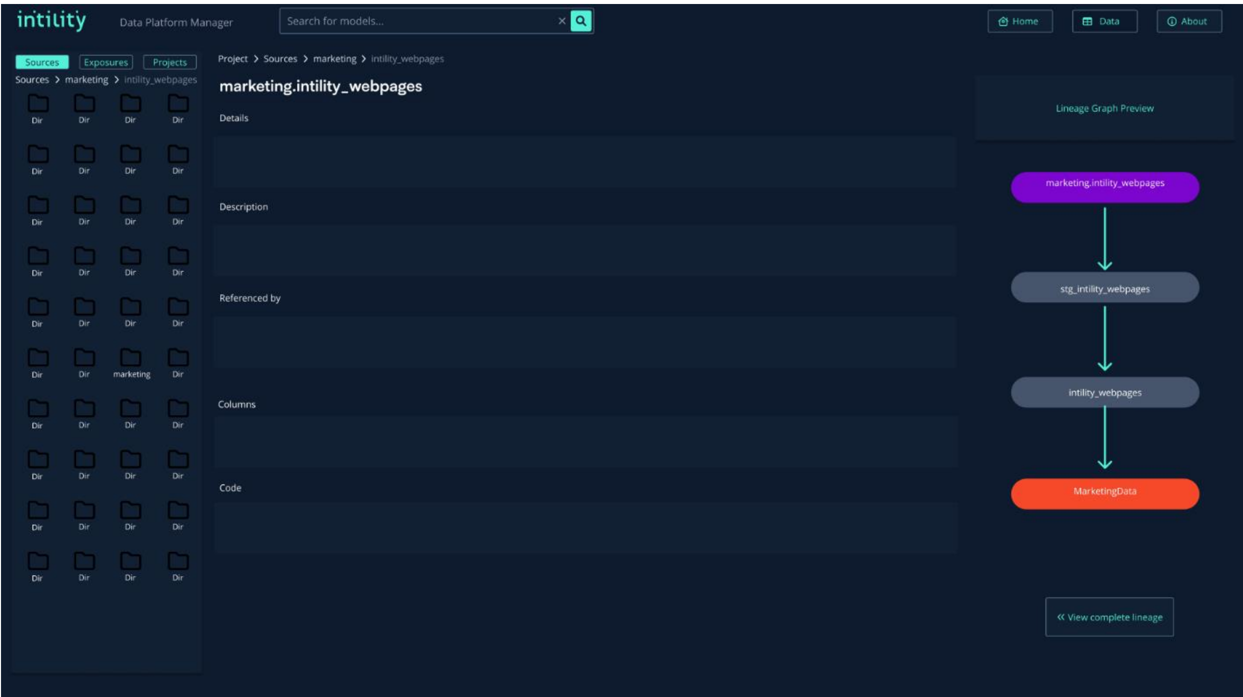
- Kristian Senneset (Data Science Expert)

Appendix H   Examples of Figma Sketches

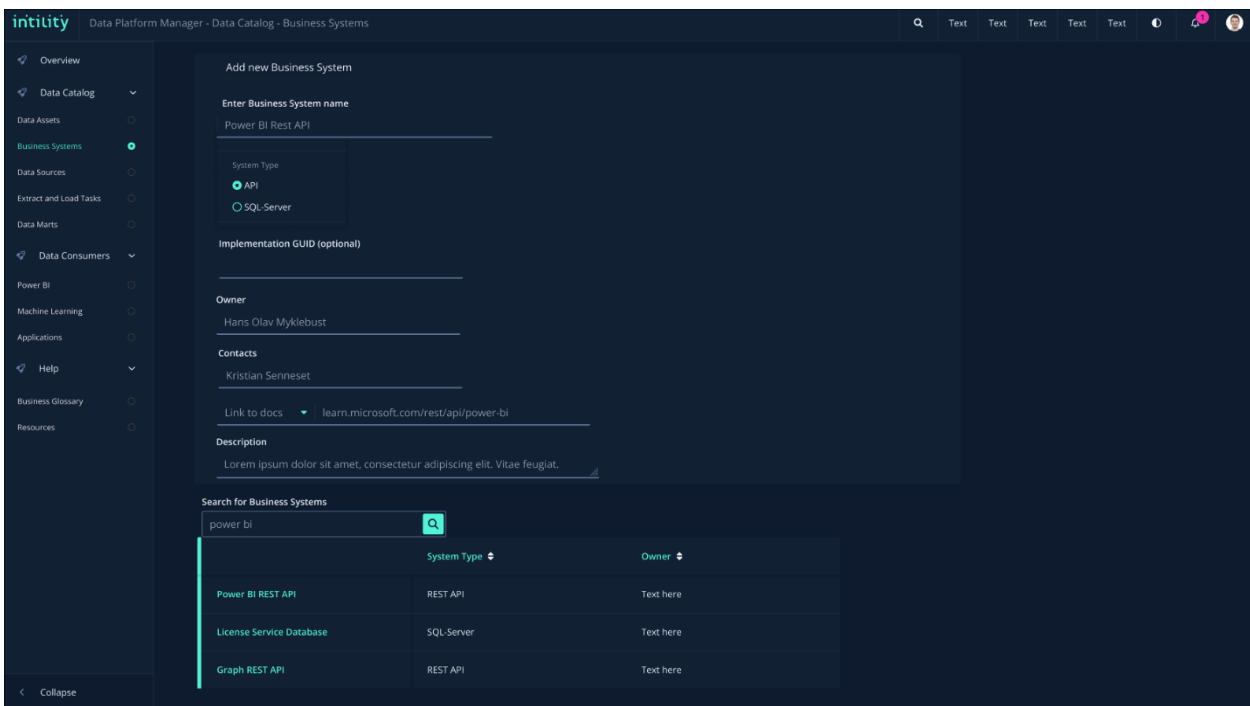
Examples from our sketches:

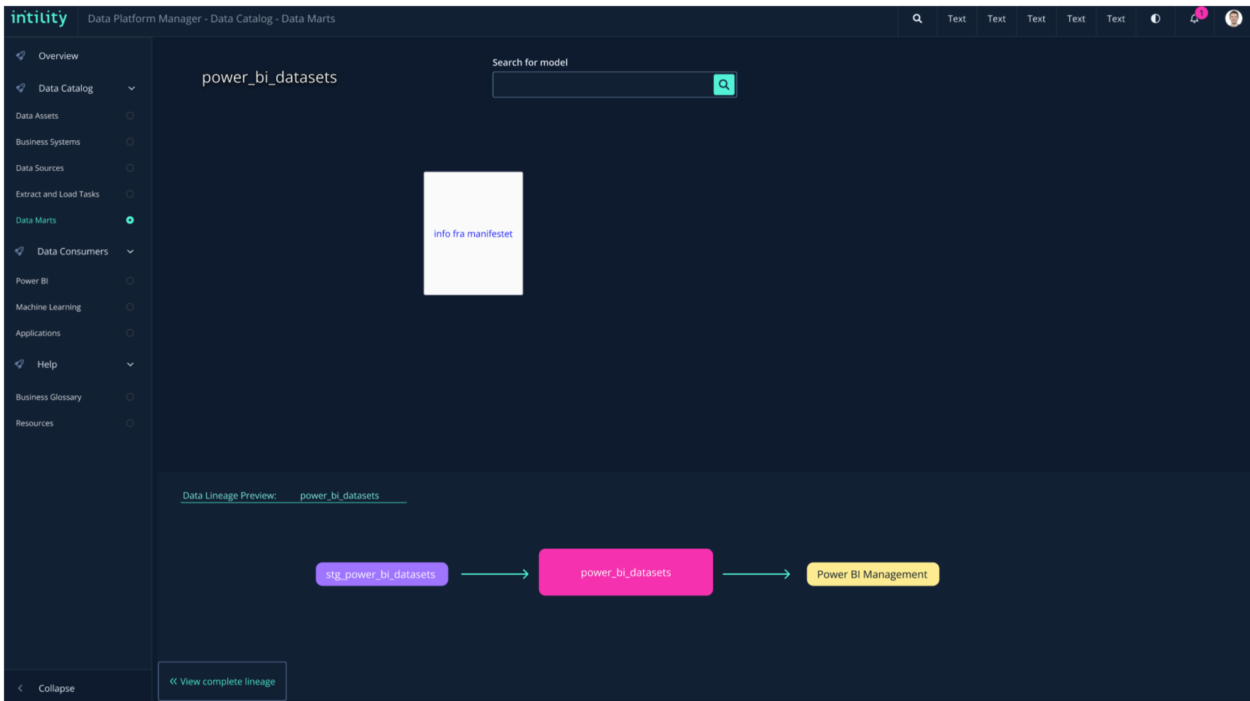


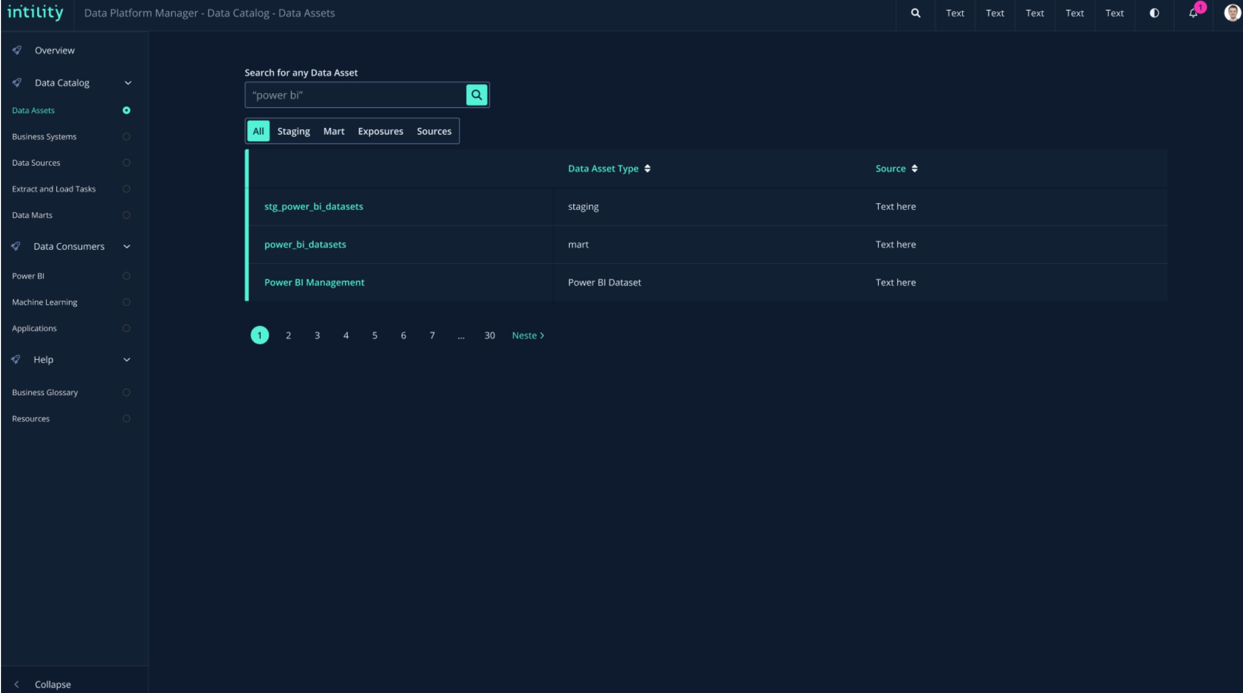




Examples from the data science teams sketches:




intility
Data Platform Manager - Data Catalog - Data Marts
Text
Text
Text
Text
Text
Overview
Data Catalog
Data Assets
Business Systems
Data Sources
Extract and Load Tasks
Data Marts
Data Consumers
Power BI
Machine Learning
Applications
Help
Business Glossary
Resources
Collapse
power_bi_datasets
Search for model
info fra manifestet
Data Lineage Preview: power_bi_datasets
stg_power_bi_datasets
power_bi_datasets
Power BI Management
View complete lineage


intility
Data Platform Manager - Data Catalog - Data Assets
Text
Text
Text
Text
Text
Overview
Data Catalog
Data Assets
Business Systems
Data Sources
Extract and Load Tasks
Data Marts
Data Consumers
Power BI
Machine Learning
Applications
Help
Business Glossary
Resources
Collapse
Search for any Data Asset
"power bi"
All
Staging
Mart
Exposures
Sources
Data Asset Type
Source
stg_power_bi_datasets
staging
Text here
power_bi_datasets
mart
Text here
Power BI Management
Power BI Dataset
Text here
1 2 3 4 5 6 7 ... 30 Neste

NTNU
Norwegian University of
Science and Technology

intility
The Enterprise
Cloud Solution