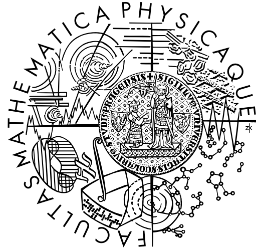

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Baroš

BrickSnoop: Optimizer of LEGO® Brick Orders

Department of Software Engineering

Supervisor of the bachelor thesis: Mgr. Ing. Robert Husák
Study programme: Computer Science
Study branch: Programming and software development (IPP2)

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In . . . . date . . . .

Author’s signature

I dedicate this thesis to my parents, brother, and dear friends. Thank you for your support and encouragement throughout this journey.

For the assistance in creating this bachelor thesis, I would like to thank my supervisor, Mgr. Ing. Robert Husák, for his valuable advice, experience, constructive criticism, willingness, and time.

Title: BrickSnoop: Optimizer of LEGO® Brick Orders

Author: Martin Baroš

Department: Department of Software Engineering

Supervisor: Mgr. Ing. Robert Husák, Department of Software Engineering

Abstract: LEGO enthusiasts often buy sets, but there are also dedicated fans who build their own creations and need to purchase individual bricks. These fans are trying to buy their desired bricks for the best price from multiple places as the price differs from store to store. However, they do this optimisation manually by comparing offers from multiple platforms as the existing tools are limited to their respective platforms and do not compare with LEGO's official listings. This bachelor thesis aims to create an optimisation tool that accepts offerings from any source in a predefined format.

The resulting web application addresses this issue by accepting offers from various platforms, allowing users to customize the optimization process by selecting algorithms and platforms, and enabling exclusions of specific countries or stores. Users can view the optimization results and download the optimized offers. Due to the NP-hard nature of the problem, approximation algorithms like greedy algorithms and simulated annealing were used. The frontend was developed in Next.js with TypeScript, while the backend for data manipulation and offer optimization was implemented in Python. The remainder of the backend was written in Node.js with TypeScript. The application is deployed on the Google Cloud Platform.

Keywords: LEGO optimisation bricks Google Cloud Platform

Contents

1 Introduction **5**
- 1.1 Current workflow . . . 5
  - 1.1.1 Obtaining a model . . . 5
  - 1.1.2 Finding the needed parts . . . 6
  - 1.1.3 Finding the best offers - Competing solutions . . . 7
- 1.2 Issues with the current optimisation solutions . . . 9
- 1.3 Objectives of the Project . . . 10

2 Requirement analysis **11**
- 2.1 End user . . . 11
- 2.2 User scenarios . . . 11
  - 2.2.1 User Access and Information Requirements . . . 11
  - 2.2.2 Brick Upload . . . 12
  - 2.2.3 Brick Optimisation . . . 12
  - 2.2.4 Results . . . 12
- 2.3 Author's requirements . . . 12
- 2.4 Functional Requirements . . . 12

3 Optimisation Analysis **15**
- 3.1 Optimisation goal . . . 15
- 3.2 Input Data . . . 15
- 3.3 Output data . . . 16
- 3.4 Optimisation Problem Definition . . . 16
- 3.5 Possible Optimisation Algorithms . . . 17
  - 3.5.1 Greedy Algorithms . . . 17
  - 3.5.2 Simulated Annealing . . . 18
  - 3.5.3 Genetic Algorithms . . . 18
  - 3.5.4 Ant Colony Optimisation . . . 19
  - 3.5.5 Algorithm choice . . . 19
- 3.6 Summary . . . 19

4 Technical Solution Analysis **21**
4.1 Processes overview 21
4.1.1 Signing in 22
4.1.2 Data upload 23
4.1.3 Optimisation customisation, running, results viewing . . 24
4.2 Data model 25
4.3 Technology stack 26
4.3.1 Frontend 27
4.3.2 Backend 28
4.4 Summary 31

5 User Interface Design **33**
5.1 Design System 33
5.1.1 Microsoft Fluent Design System 34
5.1.2 Apple Human Interface Guidelines 34
5.1.3 Google Material Design 35
5.1.4 Summarization 35
5.2 Colour scheme and Typography 36
5.3 Views 37
5.3.1 About Page 37
5.3.2 Sign-in Page 37
5.3.3 The Account Actions Dialog 39
5.3.4 The Upload Page 39
5.3.5 The Algorithm Customisation Page 40
5.3.6 The Results views 41
5.3.7 Summarization 43

6 Development documentation **45**
6.1 Application architecture 45
6.2 Authentication 46
6.3 Database architecture 46
6.3.1 Users 46
6.3.2 Requests 46
6.3.3 Security rules 47
6.3.4 TypeScript types 48
6.4 Storage architecture 48
6.5 Backend functions 49
6.5.1 Database communication and data management 49
6.5.2 Data manipulation and preprocessing 50
6.5.3 Optimisation 53
6.6 Frontend 57

6.7 Deployment . . . 58
6.8 Summary . . . 58

7 User documentation **59**
7.1 Access to the Page . . . 59
7.2 Learning about the application . . . 59
7.3 Authentication . . . 60
7.3.1 Returning user . . . 60
7.3.2 New user . . . 61
7.3.3 Upload . . . 61
7.4 Algorithm customisation . . . 62
7.5 Algorithm run overview . . . 63
7.6 Algorithm run statistics . . . 64
7.7 User account customisation . . . 65

8 Conclusion **67**
8.1 Evaluation of Functional Requirements . . . 67
8.2 Meeting the objective of the thesis . . . 69
8.3 Possible improvements . . . 69

Bibliography **71**

A Directory structure **75**

B Code examples **77**
B.1 Firebase Functions Code . . . 77
B.2 TypeScript types . . . 79
B.3 Frontend Component Example . . . 80
B.4 Input JSON schema . . . 84

Chapter 1

Introduction

In today's world, the love for LEGO building sets is growing. Many people spend countless hours assembling intricate models and dreaming up new designs or creations. However, this passion comes with a significant challenge: efficiently and affordably gathering all the LEGO bricks needed from different sources.

This bachelor's thesis aims to solve this problem by developing a tool that helps users purchase LEGO bricks cost-effectively from multiple platforms by incorporating optimisation algorithms. Its scope includes three main tasks: defining the input format for such a tool, implementing a cross-platform optimisation algorithm, and implementing a graphical user interface (GUI) that allows users to customise the optimisation process and view results.

1.1 Current workflow

This section will present the process of creating and building LEGO creations. It will also discuss currently available options for optimisation and the problems of using them. The usual workflow consists of creating or finding a model, finding the needed parts and buying them.

1.1.1 Obtaining a model

For model creation, these are two currently used tools [1]:

- BrickLink Studio - downloadable program
- MecaBrick - an online tool

After finishing modelling, the user can export parts into one of the offered data formats, and the chosen tool also generates a building manual for the creation.

However, LEGO's big community of enthusiasts has already created thousands of models. Their creations are mainly published on these websites:

- Rebrickable - the most popular one, also providing the opportunity for the creators to sell their model to others
- BrickLink Studio Gallery - some models do not provide a parts list

Chosen model usually includes a parts list exportable into a usual data format and building instructions.

1.1.2 Finding the needed parts

Having found the desired model, users currently have four main sources from which they can buy them.

BrickLink

BrickLink is currently the leading marketplace for LEGO bricks [2]. Besides its most common usage of buying parts, it also provides the functionality of Wanted Lists, which provide a way to save a set of bricks with their desired quantities for later use. Furthermore, BrickLink owns BrickLink Studio and BrickLink Studio Gallery, which are mentioned above. Its coverage of these areas makes it an easy recommendation and a go-to place for most enthusiasts.

BrickOwl

BrickOwl is a marketplace that supports Wanted Lists. Its lack of other features of BrickLink makes it less favoured, yet many stores only offer their bricks on BrickOwl [3], so checking the price on both of these websites has been a part of the usual workflow for the past years.

Pick a Brick

Pick a Brick is an official LEGO service providing an opportunity to get the needed part. However, the bricks offered highly depend on LEGO's current sets. Also, it was unavailable for customers from smaller countries such as Slovakia until recently.

Independent stores

Independent stores provide another option for finding the needed parts, but they have limited offerings, and it is difficult to find them as they are not on any platform.

1.1.3 Finding the best offers - Competing solutions

After finding all the necessary parts, users need to buy them; however, the same brick can be purchased from hundreds of stores on BrickLink and BrickOwl and can also be found on Pick a Brick. Here comes the problem of choosing the right vendor. As the number of different bricks in our creation can vary from tens to hundreds, finding the optimal solution becomes harder and harder. Fortunately, some tools can do such optimisation.

BrickLink Auto-select

BrickLink offers an Auto-select feature that utilises a proprietary algorithm to search the offers on the BrickLink platform for bricks listed in the user's wanted list and select stores from which to buy these bricks. The Auto-select feature allows users to specify store location, select preferred currency, and exclude stores that do not ship to the user's country or those that the user has previously disliked.

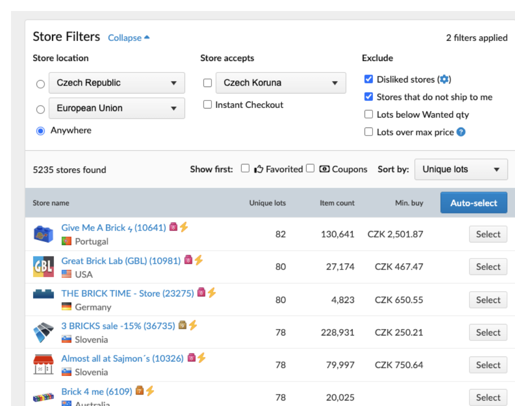

Figure 1.1 BrickLink Auto-select algorithm customisation UI

After running the algorithm, the Auto-select feature presents optimisation results, detailing the number of unique bricks available from each selected store. Additionally, it facilitates the creation of shopping carts for each store containing the desired bricks.

Figure 1.2 BrickLink Auto-select results UI

BrickOwl Buy Wishlist feature

The BrickOwl Buy Wishlist feature provides users with a list of stores on the BrickOwl platform ordered by the number of unique bricks that could be bought. This feature offers a filter for store location and exclusion of store options. The list is updated after manually selecting a store, excluding the allocated bricks. This process repeats until all the bricks in the wishlist are assigned to stores. The final price includes estimated shipping by the BrickOwl platform.

Figure 1.3 BrickOwl Buy Wishlist feature UI

Rebrickable Multi-Buy

The Rebrickable Multi-Buy feature offers the most comprehensive optimisation configuration among the currently available options. Users can customise their search by including offers from BrickLink and/or BrickOwl, selecting stores from all regions, and choosing whether to include second-hand or used parts. Additionally, the Multi-Buy feature allows users to select an optimisation goal from the following options:

- choosing the cheapest stores
- choosing the largest stores

- minimising the delivery costs (requires manual input of average delivery cost)

This advanced configuration gives advanced users more control over the results and better insight into the algorithm's optimisation strategy.

After the optimisation is completed, the user is presented with the table of bricks and stores with the chosen offers highlighted and with the optimisation results, including total cost, number of stores used, and total number of parts.

Figure 1.4 Rebrickable Multi-Buy UI

1.2 Issues with the current optimisation solutions

After analysing these optimisation options, several issues were identified:

- BrickLink optimisation includes only offers from the platform itself
- BrickLink's algorithm is proprietary, and the optimisation goal is non-configurable
- BrickOwl optimisation includes only offers from the platform itself
- BrickOwl only offers a list and no automatised optimisation algorithm
- Rebrickable's algorithm does not consider minimum buy, so manual checking is required, which when there are thousands of offers can take long

- only Rebrickable allows independent store offers. However, they have to be registered on their platform
- there is no Pick a Brick integration

These problems tend to make users painstakingly compare all the stores manually or overpay by buying all the bricks from one big vendor while saving time.

1.3 Objectives of the Project

The objective of this bachelor's thesis is to **implement cross-platform LEGO brick shopping optimisation tool**. This project's scope encompasses the following process: the user uploads offerings from various platforms in a predefined format, the tool then optimises the purchasing options, and finally, it shows result statistics, and the user can export the list of bricks for each store.

Chapter 2

Requirement analysis

In this chapter, we establish who the end user is and delve into the requirement analysis phase of the cross-platform LEGO brick shopping optimisation tool development. This stage is crucial as it identifies the necessary features of the application to ensure it meets the user's expectations and needs. Finally. we summarise these in the form of functional requirements that will guide the development of this application.

2.1 End user

The usual end user of the LEGO Brick Optimiser is a LEGO enthusiast with a basic knowledge of computers. The user often manually compares offers from different sources to achieve the best possible result, spending considerable time doing so. The user is comfortable working with computer programs such as Excel to streamline this process but lacks the programming skills.

2.2 User scenarios

Through multiple conversations with LEGO enthusiasts at different conventions and a forum, we had the opportunity to listen to their needs and understand what would make an ideal LEGO Brick Optimiser application. Based on these insights, we have outlined the following user scenarios:

2.2.1 User Access and Information Requirements

The user wants to be able to access the application on the internet, get information on its prerequisites, and get a clear understanding of its functionality.

2.2.2 Brick Upload

The user wants to be able to upload a list of offers from various sources in the form of files. The user also wants to learn about the format that these offers should be in. Furthermore, the user wants to see a list of files to be uploaded.

2.2.3 Brick Optimisation

The user wants to have the capability to choose the optimisation algorithm that will be used to determine the best combination of stores from which to buy the bricks and to customise said algorithm run by selecting desired countries and stores. Finally, they want to find the optimised combination of bricks and stores and have the option to rerun the algorithm on the same dataset with different parameters.

2.2.4 Results

The user wants to be notified when the optimisation is complete and to view an overview of all the optimisation runs for a dataset. Moreover, the user wants the ability to view the specific run results statistics. This should entail details such as total cost, total cost with estimated shipping, and number of selected stores and platforms. Finally, the user should be able to download the results for each store in a common format.

2.3 Author's requirements

The application will require users to sign in to use the optimisation algorithm as part of its access control mechanism. This is necessary because the application should be accessible on the internet; thus, controlling access can prevent the misuse of application resources. To balance this need for sign-in, the application will offer personalisation and an option to view all users' optimisation results from their accounts. Also, the application will offer a typical username and password option to sign in and provide one of the social media sign-in options: Google Login, one of the most popular social media sign-in options [4].

2.4 Functional Requirements

This section summarises the system's functional requirements based on the typical scenarios and the author's requirements. (section 2.2)

1. The user must be able to access the application on the web.

2. The user must be able to navigate inside the application.
3. The user must be able to learn about the application.
4. The user must be able to learn about the algorithm customisation.
5. The user must be able to learn about the prerequisites of using the application.
6. The user must be able to find the predefined format for the offer upload.
7. The user must be able to create an account.
   (a) The user must be able to create an account using a username and password.
   (b) The user must be able to create an account using their Google account.
8. The user must be able to sign in to an account.
   (a) The user must be able to sign in to an account using a username and password.
   (b) The user must be able to sign in to an account using their Google account.
9. The user must be able to reset their password.
10. The signed-in user must be able to change their username.
11. The signed-in user must be able to log out.
    (a) The user must be prompted to confirm their logout action before logging out.
12. The signed-in user must be able to see the upload page.
13. The signed-in user must be able to upload a list of offers in the predefined format.
    (a) The signed-in user must be alerted when their data are not in the predefined format.
14. The signed-in user must be able to upload a list of offers from multiple sources.
15. The signed-in user must see the list of files to be uploaded.

16. The signed-in user must be able to see the algorithm customisation page.
17. The signed-in user must be able to choose the optimisation algorithm that will be used.
18. The signed-in user must be able to choose which platforms they want the stores to be from.
19. The signed-in user must be able to choose which countries they want the stores to be from.
20. The signed-in user must be able to choose which stores will be excluded.
21. The signed-in user must be able to start the optimisation run.
22. The signed-in user must be able to see the overview of all optimisation runs with their statuses.
23. The signed-in user must be able to see the statistics of the specific run.
    (a) The statistics must include the name of the chosen algorithm.
    (b) The statistics must include the price of the bricks to be bought.
    (c) The statistics must include the price of the bricks to be bought with the estimated shipping.
    (d) The statistics must include the number of bricks to be bought.
    (e) The statistics must include the number of stores chosen by the optimisation algorithm.
    (f) The statistics must include the number of platforms chosen by the optimisation algorithm.
24. The signed-in user must be able to create a new run of the optimisation on the already used dataset with different parameters.
25. The signed-in user must be able to download the results for each store in a common format.

Chapter 3

Optimisation Analysis

The last chapter specified functionalities our application must include. In this chapter, we will analyse our application's main functionality, the optimisation of LEGO brick offers. We will go through the input and output of the optimisation, define the optimisation problem, and go over possible algorithms to use.

3.1 Optimisation goal

The optimisation goal is to find the best combination of stores from which to buy the bricks, taking into account user customisation factors from the functional requirements (section 2.4):

- algorithm selection
- platform choice
- store location specification
- store exclusion option

Additionally, coming from problems with the competing solutions (section 1.2), the optimisation should consider the Minimum Buy if it has one specified. Furthermore, if the store provides a shipping price, the algorithm should consider it.

3.2 Input Data

Our algorithm will get the list of lots (unique parts), including their wanted quantities, with all possible offers. To be able to achieve the optimisation goal, these offers for unique parts must include the following:

- price per piece - a price of one piece of that specific lot
- platform name - the platform from where the offer is
- store name - the name of the store which offers the brick for that price
- store country - the country where the store is located
- quantity offered - the number of pieces in stock
- Minimum Buy - the minimum order value from the store, if specified by the store
- shipping - the price of shipping if the store provides this information

3.3 Output data

After the optimisation algorithm run has ended, the user must be able to see optimisation statistics (requirement 23), including:

- algorithm used
- total price of the bricks
- total price of the bricks with estimated shipping
- number of bricks to be bought
- number of stores chosen by the algorithm
- number of platforms chosen by the algorithm

Also, the user must be able to download the optimised combination of bricks and stores where to buy them (requirement 25), so the output data must include such mapping.

3.4 Optimisation Problem Definition

The optimisation problem in our application is an instance of the Internet Shopping Optimisation Problem (ISOP). The ISOP is defined as follows:

Definition 1 (Internet Shopping Optimisation Problem [5]). *A single buyer looks to purchase a multiset of products $N = \{1, \ldots, n\}$ from m shops. Each shop l (where $l = 1, \ldots, m$) offers a multiset of available products N_l, with a cost c_{jl} for each product $j \in N_l$, and a delivery cost d_l for any subset of the products from the shop to the buyer. If a product j is not available in shop l, the cost c_{jl} is assumed to be ∞.*

The problem is to find a sequence of disjoint selections (or carts) of products $X = (X_1, \ldots, X_m)$, which we call a cart sequence, such that $X_l \subseteq N_l$ for each shop l, $\bigcup_{l=1}^{m} X_l = N$, and the total product and delivery cost, denoted by

$$F(X) := \sum_{l=1}^{m} \left(\delta(|X_l|)d_l + \sum_{j \in X_l} c_{jl} \right),$$

is minimised. Here, $|X_l|$ denotes the cardinality of the multiset X_l, and $\delta(x)$ is a function that equals 0 if $x = 0$ and 1 if $x > 0$.

We denote this problem as the ISOP, its optimal solution as X^*, and its optimal solution value as F^*.

This definition captures the core challenge of our optimisation problem: minimising the total cost of purchasing all required LEGO parts while considering the price per piece and the shipping costs from different stores. Given that our problem is essentially the same as the Internet Shopping Optimisation Problem (ISOP), and given that the ISOP is NP-hard [5], we can conclude that **our optimisation problem is NP-hard** as well.

3.5 Possible Optimisation Algorithms

Given that our optimisation is NP-hard, as stated above, finding an exact solution in a reasonable time is infeasible for larger sets of bricks. Therefore, we turn to heuristic and approximation algorithms that can provide good, if not optimal, solutions within practical time constraints. This section will discuss potential algorithms to tackle our problem, including greedy algorithms, simulated annealing, genetic algorithms and ant colony optimisation.

3.5.1 Greedy Algorithms

Greedy algorithms are a straightforward and intuitive approach to solving optimisation problems. They work by making a sequence of choices, each of which looks best at the moment, aiming to find a global optimum [6].

In the context of our problem, a greedy algorithm might iteratively select the store offering the lowest price for a needed part while also considering shipping

costs and minimum order values. The other option is always to choose the store that offers the most bricks, which optimises for the largest stores. Although greedy algorithms are fast and straightforward, they do not always find the best solution. However, they can quickly give us a good starting point.

3.5.2 Simulated Annealing

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. Inspired by the annealing process in metallurgy, it allows for occasional worsening moves to escape local optima, gradually reducing this likelihood over time [7]. The algorithm was introduced by Kirkpatrick et al. in 1983 and has since been widely applied to various optimisation problems.

Simulated annealing works by starting with an initial solution and then exploring neighbouring solutions by making small random changes. If the new solution is better than the previous, it is accepted; otherwise, it may still be accepted with a certain probability, which decreases over time. This probabilistic acceptance helps the algorithm avoid getting stuck in local optima, allowing it to explore a broader search space [8].

For our problem, simulated annealing can be applied by initially selecting a random combination of stores and then iteratively making minor changes to the selection. The algorithm evaluates the total cost at each step, including the parts' price and shipping costs. Changes that reduce the total cost are always accepted, while changes that increase the cost are accepted with a probability that decreases as the algorithm progresses. Simulated annealing is flexible and can be adapted to various optimisation problems.

3.5.3 Genetic Algorithms

The process of natural selection and genetics inspires genetic algorithms. They work by evolving a population of potential solutions over several generations, using operations such as crossover and mutation to introduce variety and promote the survival of the fittest individuals [9].

In our case, a genetic algorithm would generate various combinations of parts and stores to buy them from, combine them in new ways, and introduce small changes. Over successive generations, this process can yield increasingly better solutions. Genetic algorithms are particularly effective for problems with large and complex search spaces.

3.5.4 Ant Colony Optimisation

Ant Colony Optimisation (ACO) is a bio-inspired algorithm based on the foraging behaviour of ants. It uses a colony of artificial ants to explore the solution space and find optimal paths by mimicking the pheromone trail laying and following the behaviour of real ants [10].

For our problem, ACO can construct solutions by having each ant select stores and parts based on cost and pheromone levels. Over time, the pheromone trails help guide the search towards more promising regions of the search space. ACO has been successfully applied to various combinatorial optimisation problems.

3.5.5 Algorithm choice

These algorithms provide a range of approaches for tackling our NP-hard optimisation problem. While no single method guarantees an optimal solution, each offers a way to find good solutions within practical time constraints. **In this bachelor's thesis, we have chosen two algorithms to implement: Greedy Algorithm and Simulated annealing, with the possibility of implementing more in the future**.

3.6 Summary

In this chapter, we reviewed the optimisation's input and output, defined the optimisation problem, learned that the problem is NP-complete, found possible algorithms to use and decided which ones we would implement.

Chapter 4

Technical Solution Analysis

This chapter will determine the technology stack and examine the application's architecture. This involves analyzing the processes within our application and reviewing and selecting appropriate frontend and backend frameworks, cloud platforms, programming languages, and databases. We aim to ensure that our technology choices align with the project's requirements and provide a robust, scalable, and maintainable solution.

4.1 Processes overview

This section will review and analyse our application's user path. Based on the requirements analysis (chapter 2), we have outlined the path that captures the steps necessary to achieve the user's goal of getting the optimal LEGO brick offers. These steps are:

1. signing in - we require the user to sign in to use the optimisation feature (section 2.3)
2. uploading the data
3. customising and running the optimisation
4. viewing results
5. optionally, if the user wants to rerun the optimisation with different customisations (requirement 24), he must be able to come back to step 3.

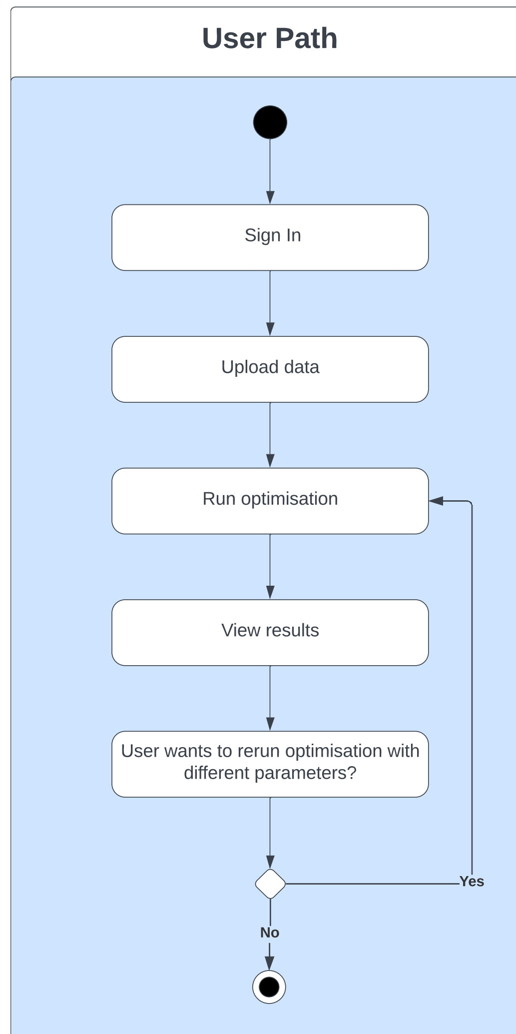

Figure 4.1 User Path

4.1.1 Signing in

To fulfil functional requirements 7 to 9, which encompass the ability to create an account, sign into it, reset the password if forgotten and username choice, we have created entity **User** that will represent the user in our application and hold the necessary data. For a visual representation of this process of signing in, see Figure 4.2.

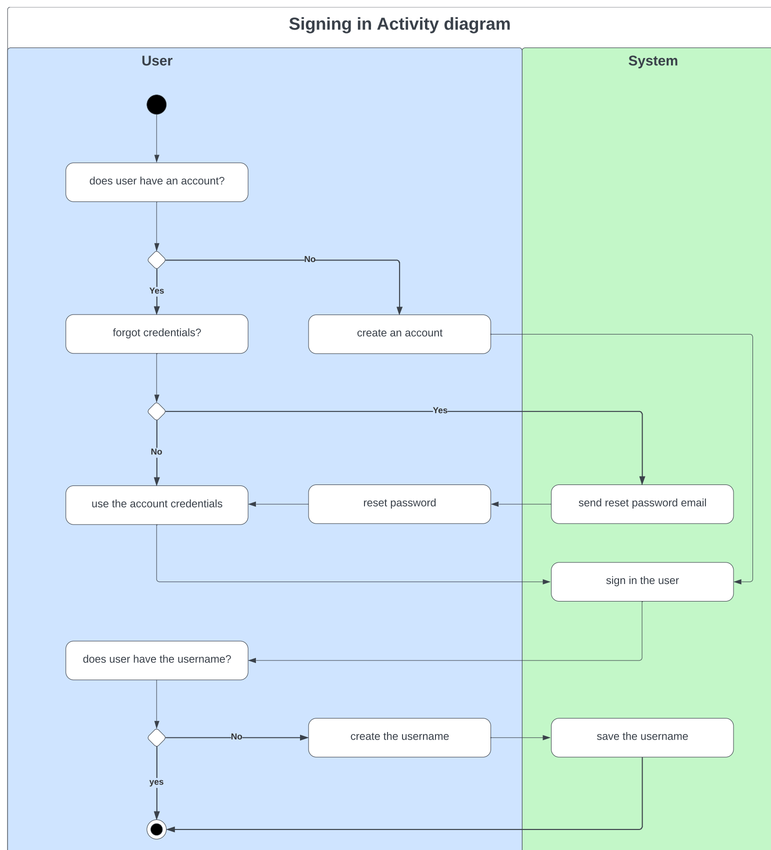

Figure 4.2 Signing in activity diagram

4.1.2 Data upload

As specified in functional requirements (section 2.4, requirements 13 to 15) user must be able to submit data (offers) from multiple sources in a predefined format in form of files, be alerted if these files do not adhere to the predefined format, see the list of files to be uploaded and finally upload them.

The fact that these files are from multiple sources and there might be an inconsistency between them means that our application will need to preprocess the data before optimising it. This preprocessing should include merging the offers based on the brick and filtering out the duplicate offers.

To be able to accommodate the users' ability to rerun the optimisation on the already used dataset (optional step 5), we will create an entity named **Request**. This entity will serve as a data structure to which the preprocessed data will be bound, enabling easy reuse of the dataset for future optimisation runs.

To illustrate and capture this process, we created the Upload data activity diagram(Figure 4.3).

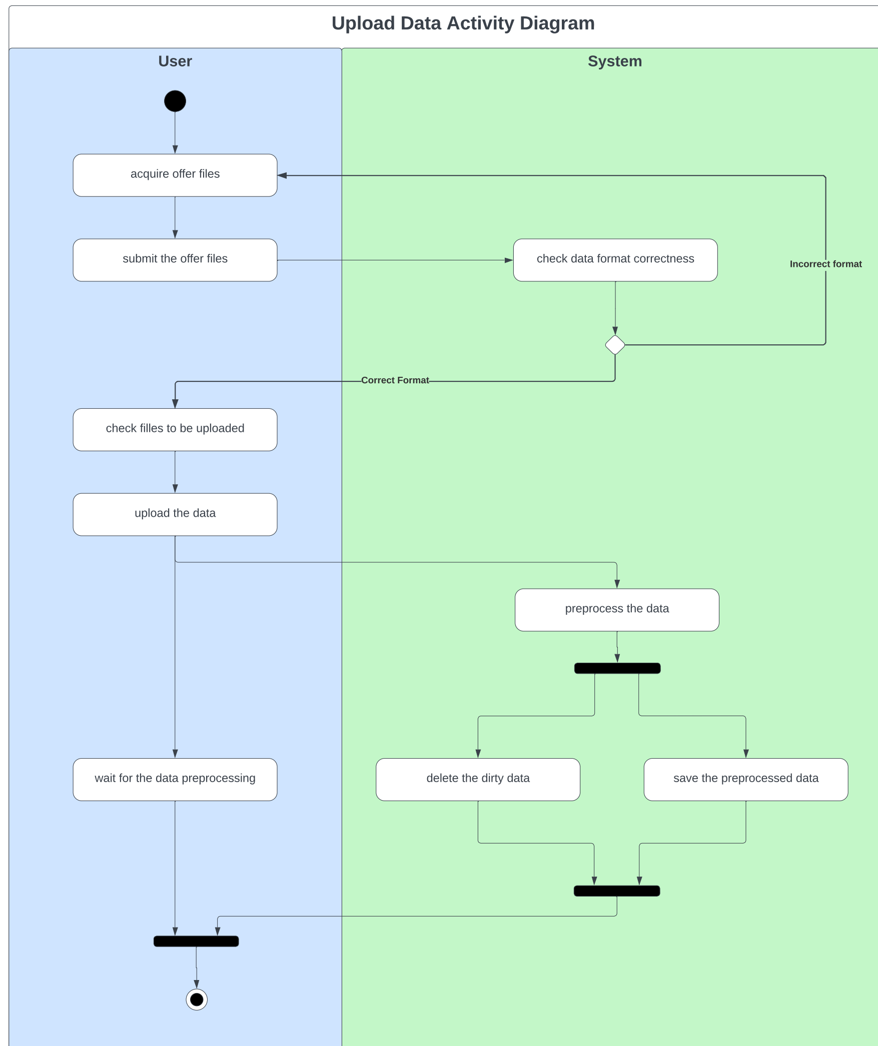

Figure 4.3 Upload data activity diagram

4.1.3 Optimisation customisation, running, results viewing

From the functional requirements analysis, we know that the user must be able to customise the optimisation run, start it, view optimisation result statistics and download the optimised offers (requirements 16 to 25).

After the user finishes the customisation phase, we will start the optimisation run, represented by an entity called **Mission**. The Mission will belong to the request, and the result of the optimisation run will be bound to it. As our problem is NP-hard, the optimisation process can take longer. We included notifying a user of the optimisation termination for the user's convenience.

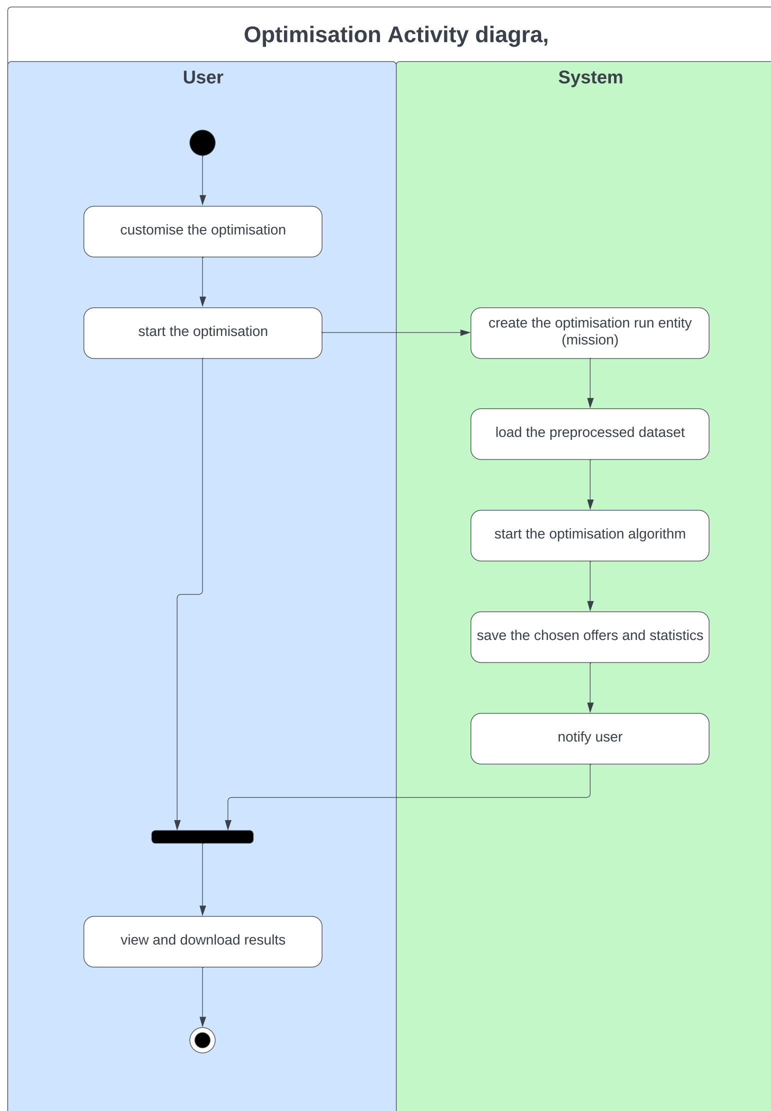

Figure 4.4 Optimisation customisation, running and results viewing activity diagram

4.2 Data model

The data model for our application is designed to support the core functionalities outlined in the requirements analysis and processes overview. It consists of several key entities that capture the essential data structures and relationships necessary for the application's operations.

User: Represents the user of the application and contains all relevant user information, including account details, username and email address of a user.

Request: Each request corresponds to a dataset of offers the user uploads for optimization. This entity ensures that data can be reused for multiple optimization runs. It contains basic information about the dataset, such as countries, stores, and platforms present.

Preprocessed data: After uploading files to our application and cleaning them, we need to store the offers somewhere. As the data can get large, for example, some sets can have three thousand bricks, each with a thousand offers or more, and we will be storing thirty million offers. Because of that, we will need to utilise file storage on the backend.

Mission: Represents a single optimisation algorithm run initiated by the user. It is associated with a specific request and stores the optimisation results, including the optimised offers, directly or by link.

These entities and their relationships form the backbone of our data model, enabling efficient data management and processing. The following diagram (Figure 4.5) illustrates the relationships between these entities.

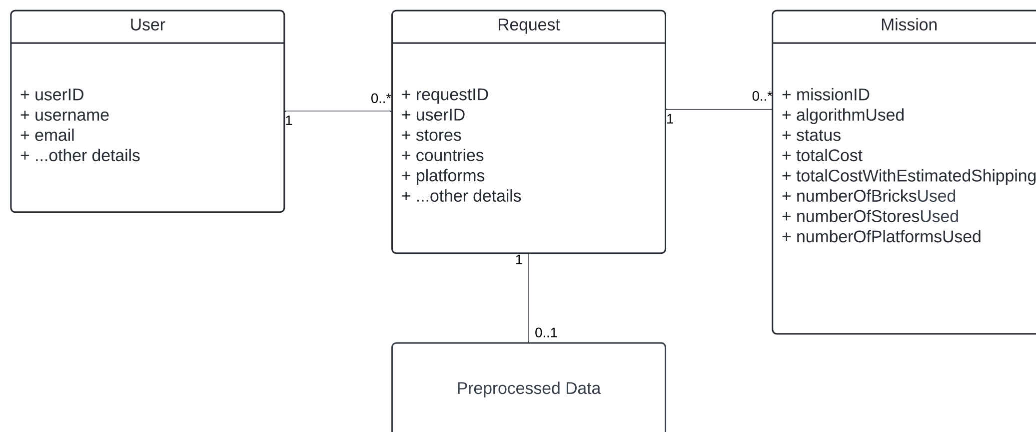

Figure 4.5 Data model

4.3 Technology stack

A technology stack combines programming languages, frameworks, libraries, and tools to develop and run a software application. It encompasses both the frontend and backend technologies, as well as any middleware, databases, and cloud services that support the application's infrastructure [11]. In this section, we will review our application's technology choices.

4.3.1 Frontend

The choice of the right frontend framework for web development can be complex due to the many available options, each with its advantages and disadvantages. Key deciding factors include language, performance, community support, security and flexibility.

Frontend framework

JavaScript remains the most common language for building web applications. However, many developers turn to **TypeScript** as applications become complex. By allowing developers to write strongly typed code, TypeScript can improve maintainability and reduce errors. The most popular JavaScript frameworks, which include Angular, React, Next.js, and Vue.js [12], also support TypeScript, providing additional benefits for large-scale projects. Each framework offers unique benefits and drawbacks, and the choice largely depends on the project's needs.

Angular is a comprehensive framework developed by Google for building dynamic single-page applications (SPAs). Built on TypeScript, Angular offers a robust toolkit for creating scalable web applications, including a component-based architecture, extensive libraries, and developer tools for building, testing, and updating code. Angular applications follow the Model-View-Controller (MVC) architecture, where the model represents the data, the view represents the UI, and the controller manages the interaction between the model and the view. This separation of concerns helps maintain organized and manageable codebases. Angular is well-suited for large-scale applications and provides features like two-way data binding and dependency injection, making development more efficient and manageable [13].

React is a JavaScript library developed by Facebook for building user interfaces, particularly single-page applications. React focuses on creating reusable UI components, which manage their own state, allowing for more efficient and manageable code. React uses a virtual DOM to optimize rendering performance and can be used with various other libraries and frameworks for building complex applications. Its component-based architecture and emphasis on declarative programming make it popular among developers [12, 14].

Next.js is a React-based framework that enables server-side rendering and static site generation for React applications. Developed by Vercel, Next.js enhances the performance and SEO of React applications by rendering pages on the server before sending them to the client. It also offers automatic code splitting, API routes, and a robust routing system. Next.js is ideal for building modern web applications that require high performance and SEO optimization [15].

Vue.js is a progressive JavaScript framework for building user interfaces created by Evan You. Vue is designed to be incrementally adoptable, focusing on the view layer with an easy integration capability with other libraries or existing projects. Vue uses a component-based architecture, offering features like reactive data binding and a simple yet powerful templating syntax [16].

After reviewing the abovementioned options, the author thinks our application could be implemented in every single one. Due to Next.js's sudden rise in popularity and to it being built on React, which is currently the most popular framework [12] with great community support, **we have chosen Next.js to be the framework of our application**. The application will be **written using Typescript**.

4.3.2 Backend

Choosing the right backend architecture and programming language is crucial as it impacts the application's performance, maintainability, scalability and ease of development. In this section, we will explore the key considerations for making an informed decision, focusing on the aspects of functional requirements and optimisation algorithms chosen.

Server vs Serverless Architecture

This section explores the key differences, benefits, and drawbacks of server-based and serverless architectures to help make an informed decision.

Server-based Architecture involves managing dedicated servers to run the backend application. This approach provides complete control over the server environment, enabling custom configurations and optimizations. It is well-suited for applications requiring consistent performance and high customization. However, it also requires significant maintenance effort, including managing server uptime, scaling, and security updates [17].

Serverless Architecture allows developers to build and deploy applications without managing the underlying infrastructure. In a serverless setup, the cloud provider dynamically allocates resources as needed, and the application is charged based on actual usage rather than pre-allocated capacity. This model offers automatic scaling, reduced operational overhead, and cost-efficiency for variable workloads. However, it can introduce latency due to cold starts and limits control and visibility as the developers rely on third-party cloud providers [18].

Based on the fact that we want to focus more on developing the actual application rather than managing the infrastructure, the on-demand pricing advantage as well as the author's previous experience with serverless technologies **we have chosen the serverless architecture**.

Cloud Platform

A cloud platform provides a range of cloud computing services that enable developers to build, deploy, and manage applications without the need to manage underlying infrastructure. These platforms support serverless architectures. The most popular cloud platforms include the Google Cloud Platform (GCP), Amazon Web Services (AWS) and Microsoft Azure [19]. Each platform offers a comprehensive suite of services and tools but has unique strengths and weaknesses, making the choice largely dependent on the project's specific needs.

Google Cloud Platform (GCP) provides a range of cloud computing services, including computing, storage, data analytics and machine learning. GCP is known for its strong emphasis on serverless technologies, such as Cloud Functions, Cloud Storage, and Workflows, which enable developers to build scalable applications with minimal overhead. Cloud Functions allow for event-driven serverless computing, Cloud Storage provides secure and scalable object storage, and Workflows easily orchestrate complex workflows. Additionally, GCP's free tier is particularly generous, allowing new users to access various services without incurring significant costs, which is advantageous for startups and small projects [20].

Amazon Web Services (AWS) is the most mature and widely adopted cloud platform, offering an extensive array of services, including computing, storage, databases, machine learning, and IoT. AWS is known for its flexibility, scalability, and reliability, with a global infrastructure supporting many use cases. Services like EC2, S3, and RDS are well-regarded for their robustness and performance. However, AWS can be complex to navigate, with a steep learning curve for new users. Additionally, while AWS offers a free tier, it is often considered less generous compared to GCP's free offerings [21].

Microsoft Azure is a strong contender in the cloud space, particularly for enterprises already using Microsoft products. Azure offers comprehensive services, including computing, storage, AI, and DevOps. It integrates seamlessly with Microsoft's software ecosystem, such as Windows Server, Active Directory, and Office 365. Azure's hybrid cloud capabilities and strong support for Windows and Linux environments make it versatile for diverse workloads. However, Azure's pricing and billing structure can be complex, posing challenges for cost management [22].

After evaluating the options, the author believes that our application would benefit most from using Google Cloud Platform. GCP's generous free tier, strong serverless capabilities, and the author's familiarity with the platform make it the ideal choice for our project. Because of that **we have chosen GCP as the cloud platform for our application**.

Backend languages

This section will provide an overview of popular languages for backend development, namely Javascript, Python, PHP and Java. Based on this overview, we will decide which languages will be used for our use case.

Python is a versatile and widely-used programming language known for its simplicity and readability. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. It has a rich ecosystem of frameworks and libraries. Libraries such as NumPy, Pandas, Matplotlib, and SciPy are widely used for data manipulation, analysis, visualization, and optimization. Its robust community support and extensive documentation make it a popular choice for beginners and experienced developers [23].

JavaScript (Node.js) allows developers to use JavaScript for server-side scripting. Node.js is known for its asynchronous, event-driven model, making it lightweight and efficient for building scalable applications. With npm (Node Package Manager), developers can access a vast library of open-source packages. Node.js also supports TypeScript. Using TypeScript on both the backend and frontend can enable type sharing, enhance code consistency and reduce development time. Also, Node.js offers direct integration with most GCP products and these SDKs are well-documented. [24].

PHP is a server-side scripting language designed primarily for web development, well-suited for web content management systems like WordPress, Joomla, and Drupal. PHP follows a procedural programming paradigm but also supports object-oriented programming. It integrates well with databases such as MySQL, PostgreSQL, and SQLite[25]. Despite its strengths, PHP has been criticized for its inconsistent syntax, which can lead to maintenance challenges. Additionally, PHP applications can suffer from performance issues if not optimized properly.

Java is a robust, object-oriented programming language widely used for building enterprise-scale applications. Java applications are typically known for their portability across platforms due to the Java Virtual Machine (JVM). It offers a comprehensive set of libraries and frameworks. However, Java can be verbose and requires a significant amount of boilerplate code, which can slow down development. Additionally, Java applications can consume more memory and resources than other languages, potentially leading to higher operational costs. Despite these drawbacks, Java's strong performance, security features, and extensive ecosystem make it a reliable choice for large-scale applications [26].

Our application's back-end functionality includes data preprocessing, offer optimisation, user management and communication with the database and front-end. **For the data manipulation and optimisation** parts of our application, **we have chosen Python** for it is a widely used language for these tasks with many libraries available to streamline the process. **For user management**

and communication with the database and front-end, we have chosen Typescript for the ability to use the same types on the front and for the availability of many useful SDKs. This decision will enable us to develop our application faster by using the right tool for the problem, which is, in our case, the correct programming language for the use case.

Database and data storage

Selecting the right database technology is crucial for efficient data storage and management. The two primary types of databases are SQL (Structured Query Language) and NoSQL (Not Only SQL).

SQL databases, such as MySQL, PostgreSQL, and Microsoft SQL Server, are relational databases known for their strong consistency, ACID compliance, and support for complex queries and transactions. They use a predefined schema, ensuring data integrity and relationships between tables. However, they can struggle with scalability and flexibility, particularly with large volumes of unstructured data or rapid schema changes.

NoSQL databases, including MongoDB, Cassandra, and Cloud Firestore, are designed to handle large volumes of unstructured or semi-structured data. They offer horizontal scalability, flexible schemas, and high performance for read and write operations. NoSQL databases support various data models like document, key-value, column-family, and graph, making them ideal for rapidly changing data requirements and real-time analytics applications.

Based on the data model, our application needs to manage not strictly structured data (for example, variable length of number of stores stored in requests) with a possibly large volume; **we have decided to use NoSQL type of a database** and as our application will be served on Google Cloud Platform, we will use **Cloud Firestore**. We will use **Cloud Storage to store the offers**, which, as mentioned in the data model part, can be a large amount of data.

4.4 Summary

In this chapter, we have analysed the processes in our application and reviewed possible frameworks, architectures, languages, and databases to be used. We have decided to utilise Google Cloud Platform, Firestore as our database, Python and Node.js for the backend parts and Next.js for the front end of our application.

Chapter 5

User Interface Design

Before programming the application, we need to establish how it will look. After reviewing the requirements from section 2.4 and the iterative design process, we have developed a user interface design, which will be presented in this chapter alongside the principles for developing a user-friendly user interface.

5.1 Design System

For our bachelor's thesis project, we needed to select a design system to meet the project's requirements and help create a successful user interface for our product.

A design system is a set of interconnected patterns and shared practices coherently organized to achieve the purpose of digital products. Patterns include elements such as user flows, interactions, buttons, text fields, icons, colours, typography, and microcopy. Practices refer to how these patterns are created, captured, shared, and used, particularly in team settings [27].

Selecting the right design system is a crucial part of modern product development. Its benefits lie in providing a consistent and efficient design across all platforms and devices. A well-chosen design system significantly impacts the popularity and overall success of the product. Offering a set of predefined design patterns and components accelerates the design and development process, improves speed, quality, and consistency, and helps teams focus on solving more significant problems rather than minor design details [28].

Given the importance of this choice, we had to carefully evaluate various options, each with its strengths and weaknesses, to find the design system that best suits our project and aligns with its goals. This section focuses on comparing popular design systems and selecting the appropriate one to ensure the success of our project.

5.1.1 Microsoft Fluent Design System

The Microsoft Fluent Design System is a design language developed by Microsoft that focuses on creating visually appealing and intuitive interfaces across devices and platforms. It emphasizes light, depth, motion, material, and scale to deliver a cohesive and dynamic user experience.

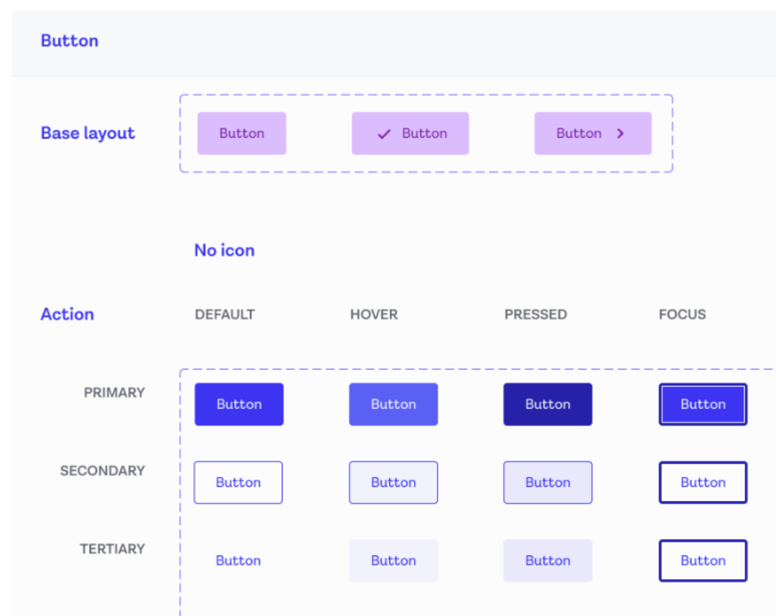

Figure 5.1 Screenshot of button design in Fluent Design System

5.1.2 Apple Human Interface Guidelines

Apple's Human Interface Guidelines are a set of principles and best practices that Apple provides to help developers create intuitive and visually pleasing apps for iOS, macOS, watchOS, and tvOS. These guidelines prioritize clarity, deference, and depth, ensuring that interfaces are aesthetically consistent with Apple's ecosystem and easy to use.

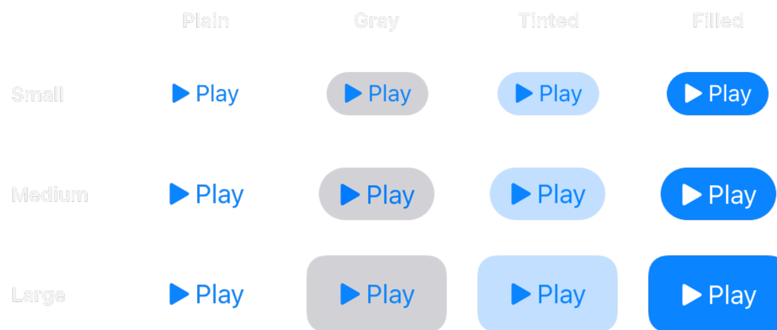

Figure 5.2 Screenshot of button design in Human Interface

5.1.3 Google Material Design

Google Material Design is a comprehensive design system developed by Google that aims to create a unified user experience across all platforms and devices. It emphasizes the use of grid-based layouts, responsive animations, and transitions, padding, and depth effects such as lighting and shadows to produce a consistent, bold, and adaptable design language.

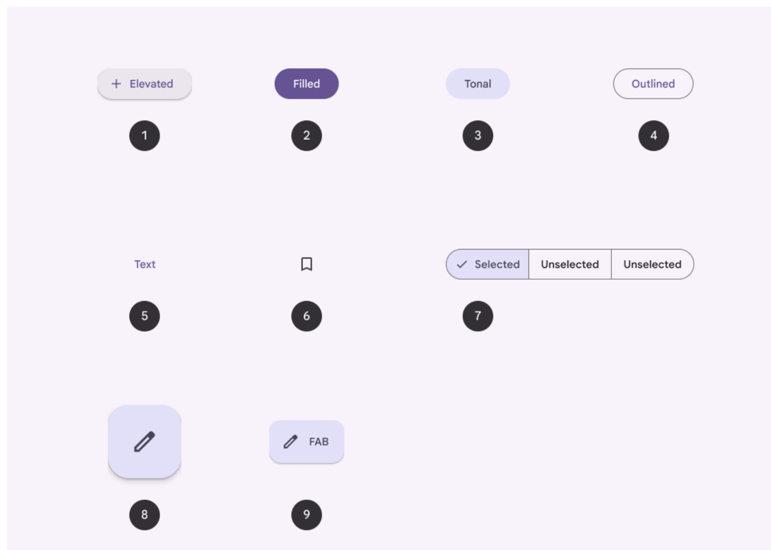

Figure 5.3 Screenshot of button design in Material Design

5.1.4 Summarization

Currently, Material Design is the most widely used design system [29], featuring well-developed UI frameworks, and the author has experience with it; therefore, we have decided to use it for the purposes of this project.

5.2 Colour scheme and Typography

Typography and colour schemes are crucial in web design as they significantly impact the readability, accessibility, and overall aesthetic appeal of an application, enhancing user experience and engagement. We went for a more professional look with dark blue as the main colour and user-available online tools for the secondary and complementary colours (Figure 5.4). Furthermore, we have themed the Material design components according to our colour pallet (Figure 5.5).

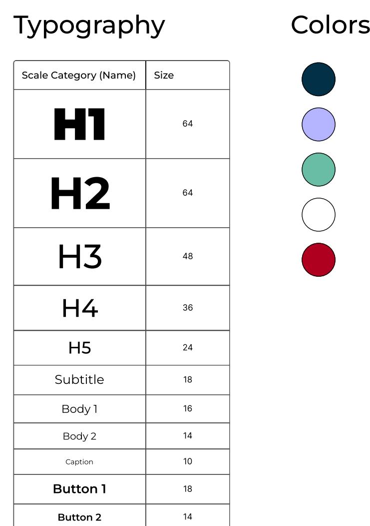

Figure 5.4 Colour scheme and typography

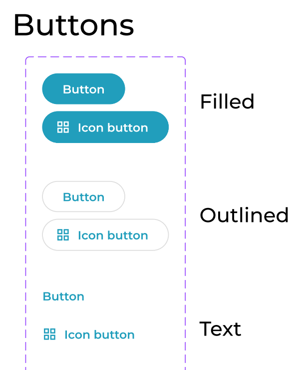

Figure 5.5 Stylized Components

5.3 Views

In this section, we will go through the main views/pages of the application and will demonstrate how they incorporate the functional requirements from the section 2.4. The chosen Google Material Design (subsection 5.1.3) design system's user interface and user experience principles were used to design each page.

5.3.1 About Page

The About Page provides a user with an overview of the application's features, its purpose, and the needed prerequisites for using it. Key elements include:

- An Overview Section, briefly explaining the application's goals (requirement 3)
- Feature Highlights, listing algorithm customisation options (requirement 4)
- Prerequisites, showing that the user needs to have the offers list and including the predefined format that data needs to be in (requirement 5, requirement 6)

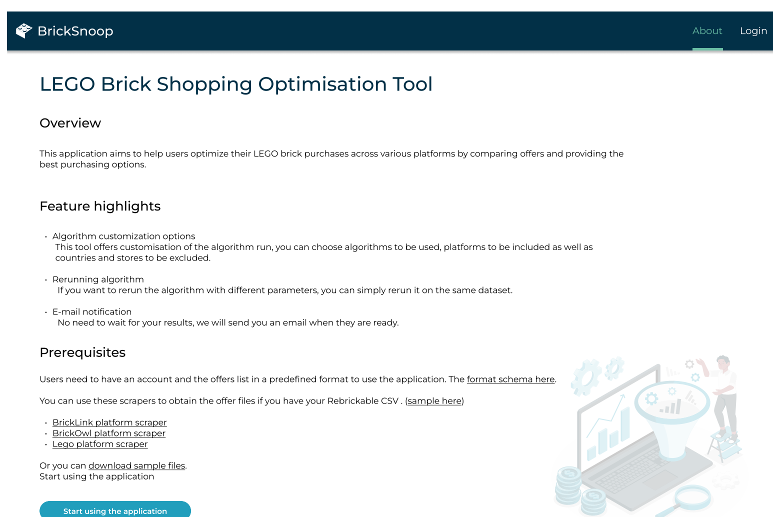

Figure 5.6 The About Page UI

5.3.2 Sign-in Page

From requirement 7 (the user must be able to create an account using username and password or by using the Google Account), requirement 8 (the user must

be able to sign to an account using the same methods) and requirement 9 (the user must be able to reset their password) we know our design must include two authentication methods. These methods are:

- Username and Password Authentication
- Google Account Authentication

To accommodate the abovementioned requirements, the Sign In page will feature the following:

- Input Fields for Username and Password: Users can use a combination of username and password to log in.
- Google Sign-In Button: Users can create an account or sign in to our application via their Google Account.
- Create Account Form: Users can fill in their name, email and desired password to create an account.
- Forgot Password Link: Users can recover their account if they forget their password.

These features ensure that the application is accessible and convenient for many users, meeting the authentication requirements specified.

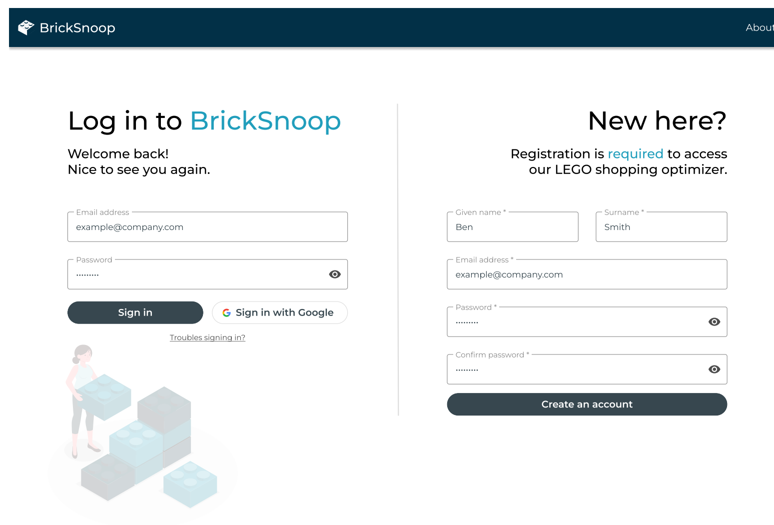

Figure 5.7 The Sign-in Page UI

5.3.3 The Account Actions Dialog

As requirements 10 (the user must be able to change their username) and 11 (the user must be able to log out and be prompted with log out confirmation) state, we need to allow the user to access these features. By designing the Account Actions dialog with buttons Change username and Log out, we satisfy these requirements (Figure 5.8).

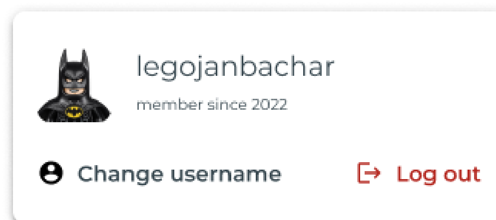

Figure 5.8 The Account Actions Dialog

This dialog will be accessible after logging into the application and clicking on the user icon in the AppBar (Figure 5.9).

Figure 5.9 Logged in user's AppBar

5.3.4 The Upload Page

To fulfil requirements 12 to 15, which encompass the ability to see the Upload Page, upload data and see the uploaded files, we designed the user interface of this page with the main components being:

- A Stepper: This component will show users where they are currently in the process of using our application.

- An upload section: This section will allow users to upload files using drag and drop or the browse feature (Figure 5.10).

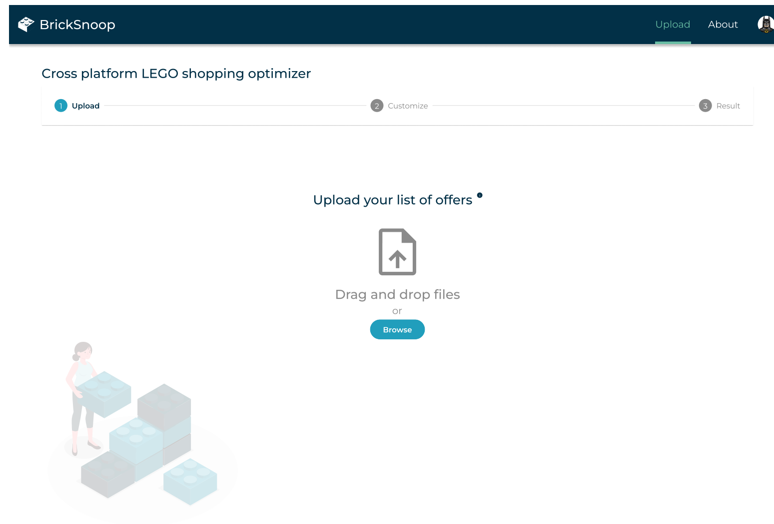

Figure 5.10 Upload Page UI - before uploading files

- Files overview component: This will allow users to clearly see their uploaded files (Figure 5.11).

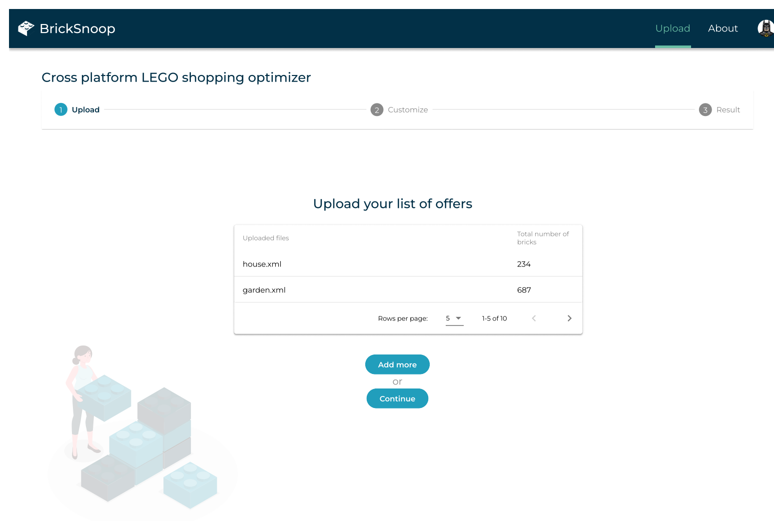

Figure 5.11 Upload Page UI - after uploading files

5.3.5 The Algorithm Customisation Page

The functional requirements 16 to 21 state that there must be a page that allows users to customise the algorithm run. Such customisation must include the ability

to choose platforms and countries from which the stores will be and the capability to exclude specific stores. Furthermore, users must be able to choose which algorithms will run with these parameters. Finally, they need to be able to start this optimisation run.

To accommodate these requirements, we designed the Algorithm Customisation Page, which, besides using Stepper to show users where they are currently in the process of using the application, also contains four main sections:

- Algorithm choice section: In this section, users will select all the algorithms that will be used to optimise the offers.
- Platform selection section: Users will choose all platforms to be included in the optimisation results.
- Country choice section: After receiving a list of all countries the data contains, users can select those they want to be used for the optimisation run.
- Store exclusion section: This section will give users a list of all stores from the data provided, and they will be able to pick those they want to leave out of optimisation.

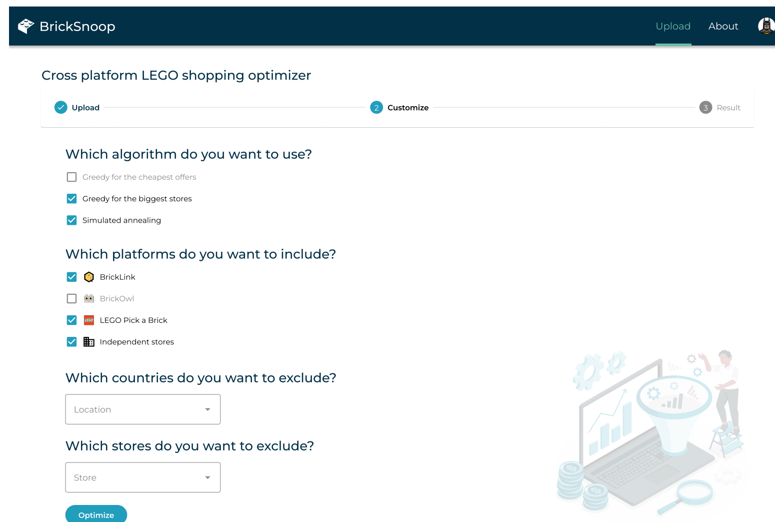

Figure 5.12 The Algorithm Customisation Page UI

5.3.6 The Results views

To fulfil the requirements item 22 to item 25 we have created two views.

Run Overview

As stated in requirement item 22 and requirement item 24, the user must be able to see the overview of all optimisation runs on the dataset and be able to rerun the optimisation with different parameters. We have created the run overview view, where all the runs are listed, and there is a button that will navigate the user back to the algorithm customisation page.

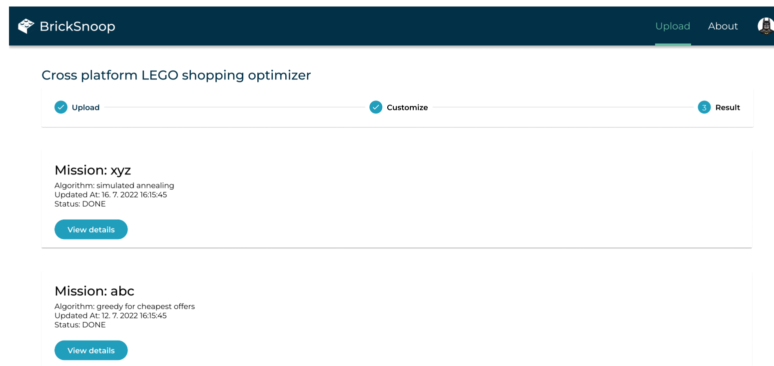

Figure 5.13 The Results Page - run overview UI

Specific Run details

As for requirement item 22 and item 25: the user must be able to see optimisation statistics and download the results. We have created the specific run details view, where the statistics like: the number of bricks to be bought, number of chosen platforms, number of chosen stores, as well as the price with and without estimated shipping, are shown. Also, there is a button for downloading the results file.

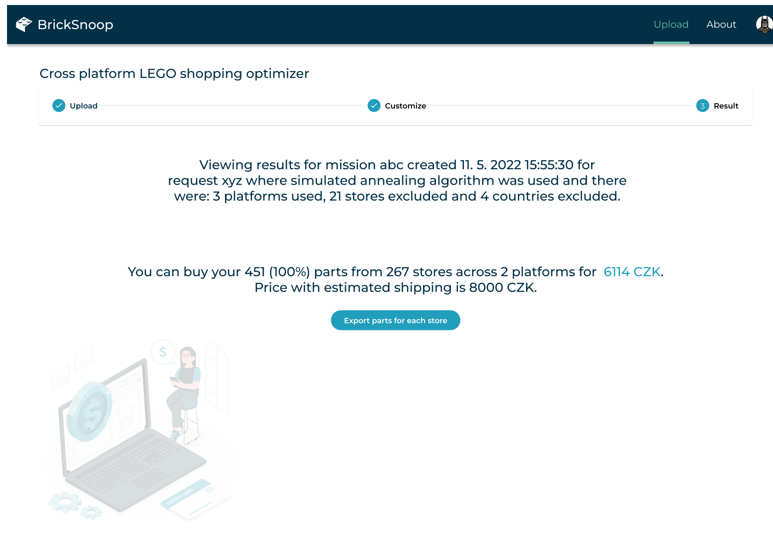

Figure 5.14 The Results Page - specific run details view UI

5.3.7 Summarization

By implementing a web application (requirement 1) where the user can navigate between these pages and views (requirement 2) based on the created design that satisfies the requirements from 3 to 25, we will be able to access all functionality required, thus enabling fulfilment of functional requirements stated in section 2.4.

Chapter 6

Development documentation

In this chapter, we will describe the development of the application in more detail, focusing on the specific solutions for the database model, authentication, backend optimisation and how they are integrated with the web interface. We will also thoroughly analyse each part of the development process.

6.1 Application architecture

As mentioned in the technical solution analysis in chapter 4, we utilise the Next.js framework and Typescript for the frontend part, Python for the data manipulation and analysis part, Node.js with Typescript for other parts of the backend. We have chosen the Firestore NoSQL database for the database, and for the storage of offers, we utilise Cloud Storage. For the authentication, we have selected Firebase Authentication. These parts are communicated through SDKs (Software development kit) and APIs (Application programming interface). See the overview of the architecture:

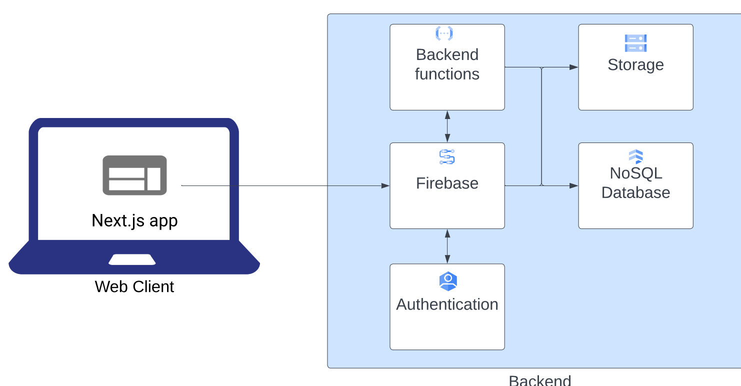

Figure 6.1 Application architecture overview

6.2 Authentication

We use Firebase Authentication, a service provided by Google that enables developers to quickly implement secure authentication in their applications because it integrates seamlessly with our Firebase and GCP infrastructure, offering a unified and convenient solution. It supports various authentication methods, including email/password, Google, Facebook, and more, ensuring flexibility for our users. By leveraging Firebase Authentication, we ensure that only authenticated users can perform write operations to the database, enhancing the security of our application.

6.3 Database architecture

Based on the functional requirements, we have defined a basic data model in the analysis part of this bachelor's thesis (section 4.2). Later in the section 4.3.2, we have decided that for our application, the NoSQL type of database, namely Firebase Firestore, will be used. Since we use a NoSQL database that uses collections and documents and does not enforce a strict schema, we ensure that only correctly typed and structured data is present in the database through security rules and TypeScript, which we will discuss later.

6.3.1 Users

Collection users contain documents for each user of our application. Each document contains these:

```
uid: string - user id
email: string
username: string - user-visible name
avatar: string - profile picture
createdAt: Date
updatedAt: Date
```

6.3.2 Requests

Collection requests contain documents for each request made by the user:

```
requestId: string
createdBy: string - user id of creator
dataStatus: enum - status of data preprocessing
missions: { missionId: Mission }
```

```
    - representing all missions connected to the dataset
validUntil: Date - when will the preprocessed data be deleted
createdAt: Date
updatedAt: Date
stores?: string[] - stores in the dataset
countries?: string[] - countries in the dataset
platforms?: string[] - platforms in the dataset
numberOfBricks?: number - bricks present in the dataset
errorMessage?: string - if there is an error
```

Missions

The dictionary missions in the request contain all missions connected to the specific dataset. Each mission consists of:

```
missionId: string
executionId: string - internal id of the algorithm run
status: enum - status of the optimisation run
algorithm: string - chosen algorithm
platforms: string[] - chosen platforms to be used
countriesToExclude: string[] - chosen countries to be excluded
storesToExclude: string[] - chosen stores to be excluded
createdAt: Date
updatedAt: Date
resultsPath?: string - path to the results file in GCP Storage
result?: {
    bestCost: number
    noPenaltiesCost: number
        - we use penalties to guide the optimisation
    noShippingNoPenaltiesCost: number
        - price of the bricks only
    numberOfPlatforms: number - platforms to be bought from
    numberOfStores: number - stores to be bought from
    numberOfBricks: number - bricks to be bought
}
errorMessage?: string - if there is an error
```

6.3.3 Security rules

In Firestore, security rules control access to the database, ensuring that data is read and written securely. These rules determine who can access specific parts of

the database and what operations they can perform. Below are the security rules we have implemented and their descriptions:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
    match /users/{id}{
     allow read: if request.auth != null
        && request.auth.uid == id;
    }

    match /requests/{id}{
     allow read: if request.auth != null;
    }
  }
}
```

These Firestore security rules enforce strict access control for the database. All documents' read and write operations are denied by default, ensuring a secure baseline. The rules allow authenticated users to read their documents in the "users" collection by matching their user ID with the authenticated ID. Additionally, authenticated users can read documents from the "requests" collection, allowing access to request details while maintaining security.

6.3.4 TypeScript types

We have created the corresponding TypeScript interfaces for each of the entities presented (see section B.4).

6.4 Storage architecture

We use GCP Storage to store large files in our application as the Firestore has a limitation of a maximum of 1 MB per document [30], and the Cloud SQL pricing model is unsatisfactory. Such files are the offer files, which can have hundreds of bricks, each with thousands of offers.

The GCP Storage uses buckets to store the data for our application. We have created three buckets:

- dirty data bucket - used to store the user-uploaded data before preprocessing
- clean data bucket - used to store preprocessed data
- results bucket - used to store the results of the optimisation run

6.5 Backend functions

Our backend functions are responsible for these three functionalities:

- communication with the database (write)
- data manipulation and preprocessing
- optimisation

In this section, we will go over these parts in detail.

6.5.1 Database communication and data management

As mentioned in the previous chapter, our backend is responsible for all write operations to the database as well as for creating entities (users and requests). In our implementation, we use Firebase as our backend service and Firestore as our database. Below is a detailed explanation of the key components we use to handle write operations to the database.

Firebase Functions: We utilise Firebase Functions to handle HTTP requests and background functions. These functions allow us to write server-side logic that responds to events triggered by Firebase features and HTTPS requests. For instance, when creating a new request, a Firebase Function is triggered to process the request data, validate it, and write it to the Firestore database.

Firestore: Firestore is a flexible and scalable database from Firebase and Google Cloud Platform. In our implementation, we use Firestore to store requests and user data. Each request is stored as a document in a collection, with fields representing the request's details.

Data Validation: We employ the Superstruct library for data validation. Superstruct allows us to define the structure of our data and validate it before processing. This ensures the data adheres to the expected format and contains all necessary fields. If the data is invalid, the function throws an error, preventing invalid data from being written to the database.

Authentication: Authentication is handled through Firebase Authentication. We ensure that only authenticated users can perform write operations to the database. The user's authentication context is checked before any data is processed. If the user is not authenticated, the function throws an error.

Error Handling: Error handling is a crucial part of our implementation. We use structured error handling to provide informative error messages for various failure scenarios, such as invalid data or authentication failures. These error messages help in debugging and provide meaningful feedback to the client.

TypeScript: We chose TypeScript for our backend development due to its static typing, scalability, and robust tooling. TypeScript helps catch errors at compile time, enhances code readability, and ensures better codebase maintainability. Using TypeScript with Node.js provides a powerful combination for building a reliable and scalable backend and enabling type sharing with the front end.

In summary, our backend part focuses on database communication and entity creation and leverages Firebase Functions, Firestore, data validation with Superstruct, authentication with Firebase Authentication, structured error handling, and logging to ensure secure, reliable, and efficient write operations. This approach helps maintain data integrity and security and provides a robust system for managing database communication. Using TypeScript further strengthens our backend by providing type safety and improving developer productivity. Refer to the appendix (see section B.1) for an example of the code used for creating a request.

6.5.2 Data manipulation and preprocessing

As mentioned in section 3.2, the offer input data for the algorithm should contain these properties:

- price per piece
- platform name
- store name
- store country
- quantity offered
- minimum buy - if defined
- shipping - if defined

This data represents a singular offer on any platform bound to a brick. Each brick is defined by its brick id and colour. However, each platform's IDs and colours are different; for example, the simple brick that can be found in more than 7000 LEGO sets [31] shown on Figure 6.2 has three different IDs on three popular platforms for buying LEGO bricks (subsection 1.1.2):

- BrickLink id is 3023
- BrickOwl id is 44980
- Lego id is 6225

Figure 6.2 Simple LEGO part [32]

If we want to be able to merge the offers from multiple platforms, we will need to introduce a common identifier for a brick. Fortunately, Rebrickable offers a parts catalogue [1] where each brick has a list of all different IDs on the popular platforms. Furthermore, Rebrickable has its own identifier for every brick, which will be used to merge the bricks. However, we will still need the platform-specific ID for user convenience when using the results. Knowing this, we can define the input format of our optimisation tool :

```
Brick:
    rebrickableID: string - identifier of the brick
    rebrickableColor: string - identifier of the brick colour
    wantedQty: number - how many pieces does the user want
    offers: Offer[] - list of all offers

Offer:
    platform - platform name
    platformBrickID - platform-specific brick identifier
    platformBrickColor - platform-specific brick colour ID
```

[1] https://rebrickable.com/parts/

```
seller - the store name
country - the country that the store is located in
quantity - quantity offered by the store
price - price per piece
```

We have also created a **JSON schema for the predefined input format** (see appendix section B.4). The next chapter will discuss ways to obtain the offer data in such a format.

Having defined the application input format, we will go through the preprocessing of the uploaded data. Firstly, the user uploads their data to the "dirty data" bucket. As multiple files might be uploaded, we zip them before the upload, and then they are uploaded and matched with the Request entity by request, which is the file name. The data cleanup function is triggered on the upload competition using the Google Cloud Storage (GCS) triggers. We implemented this function using Google Cloud Functions - a serverless execution environment for building and connecting cloud services. For the function, as mentioned in chapter 4, we decided to use Python as our programming language. We will provide an overview of the data cleanup function:

1. download the file from the GCS "dirty data" bucket to temporary storage
2. unzip and load the JSON files
3. merge the brick offers between the files by the Rebrickable ID and Rebrickable colour
4. delete duplicate offers
5. delete offers, the currency of which is not CZK
6. save the cleaned offers as JSON to the GCS "clean data" bucket
7. clean the temporary storage
8. delete the original file from the "dirty data" bucket
9. update the request with the data preprocessing being completed and save information about the offers, such as platforms, stores, and countries present in the final file

We have decided to delete all offers whose currency is not in the Czech koruna (CZK) as the shipping destination for our country will be the Czech Republic (see section 6.5.3), and the conversion with the current currency exchange rates is out of the scope of this thesis.

6.5.3 Optimisation

The optimisation functionality of our application requires us to call the optimisation algorithm selected by the user on the provided dataset, save the optimisation results, and email the user about the completion of the optimisation process. We use Google Cloud Workflows to orchestrate these steps seamlessly. Workflows allow us to manage complex sequences of tasks with built-in error handling and retry mechanisms, ensuring a robust and reliable optimisation process. Additionally, using Workflows enables easy integration with other Google Cloud services, enhancing our application's overall efficiency and scalability.

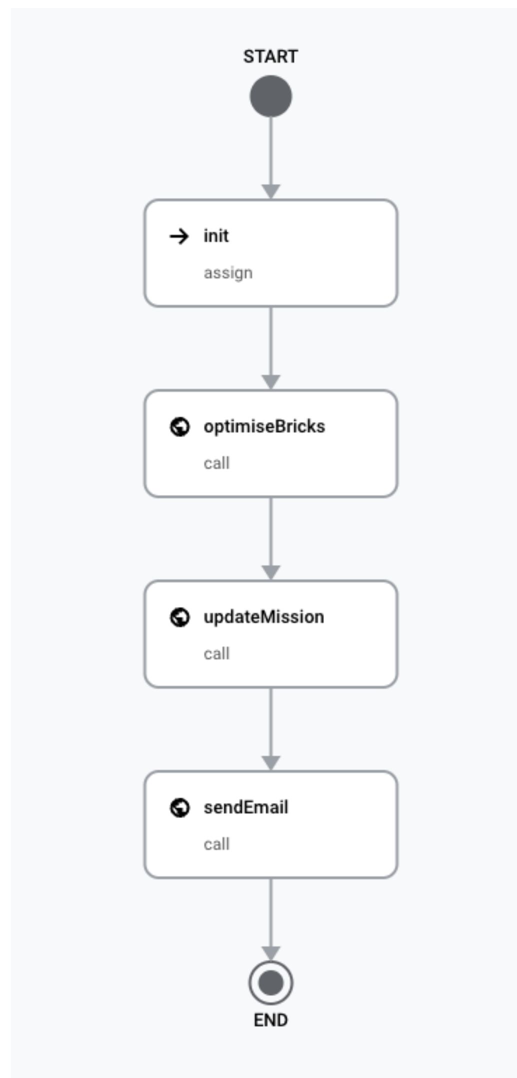

Figure 6.3 Screenshot of the workflow from the GCP console

We will look at each part of the workflow in more detail in the upcoming sections.

Optimise bricks function

Based on the optimisation analysis chapter 3 and technical solution analysis chapter 4, we have decided to implement the brick optimisation function in Python. It will be running using a Cloud Functions environment. Our function will be called from the workflow. An overview of the optimisation function:

1. get the data from the request
2. download the file from the GCS "clean data" bucket to temporary storage
3. unzip and load the JSON files
4. filter the bricks based on optimisation parameters and add estimated shipping to offers
5. optimise the bricks using the algorithm selected
6. if successful, save the resulting offers as JSON to the GCS "results" bucket
7. clean the temporary storage
8. return the optimisation statistics or error message

We will discuss the algorithm implementation process in more detail.

As mentioned in the optimisation analysis part of this bachelor's thesis, we have decided to implement three algorithms, namely: greedy algorithm for the cheapest offers, greedy algorithm for the most prominent stores (the stores that offer the most bricks) and simulated annealing algorithm.

Greedy algorithms implementation

The implementation of the greedy algorithms was very straightforward. For the greedy for the cheapest offers algorithm, we go through all the bricks and choose the cheapest offer. For the greedy for the biggest stores, we first unify the offers by store, then select the store offering the most bricks and finally remove these bricks from the remaining bricks. We repeat this process until there are no more bricks to be bought.

Simulated annealing implementation

Implementing simulated annealing in this bachelor's thesis revealed the complexities involved in parameter selection. The exploratory phase of simulated annealing allows the algorithm to search widely across the solution space. In contrast, the exploitative phase focuses on refining the solutions to find the optimal or near-optimal result. The main challenge lies in balancing the algorithm's exploratory and exploitative phases, which are governed by key parameters such as the initial temperature, the cooling schedule, and the stopping criteria. Each of these parameters is critical to the algorithm's performance. For example, an initial temperature set too high can result in excessive exploration of non-promising solutions. At the same time, a temperature set too low can cause the algorithm to converge prematurely to suboptimal solutions [7].

The cooling schedule, which determines how the temperature decreases over time, is vital. A rapid cooling schedule can lead to faster convergence but risks getting stuck in local minima. Conversely, a slow cooling schedule allows more extensive exploration but increases computational time [33]. Striking the right balance between these extremes requires careful consideration and tuning. The specific dataset used in the process further complicates parameter tuning. The dataset's characteristics, such as its size, variability, and complexity, heavily influence the behaviour of the simulated annealing algorithm.

Finding the optimal parameters involves a mix of empirical testing and heuristic adjustments. Initially, standard values from the literature provide a starting point, but these often need refinement based on the dataset's performance. This iterative tuning process can be both time-consuming and computationally intensive, underscoring the importance of thorough experimentation.

Based on the fact that provided datasets differ a lot and after empirical testing of tuning the parameters, we have decided to utilise the Simanneal library in Python to aid in parameter selection, which supports automatic parameter tuning as the process of fine-tuning the parameters would require us to know the dataset specifics before the upload. This library allowed us to streamline the parameter optimisation process by leveraging built-in functions designed to find suitable initial temperatures and cooling schedules based on the specific characteristics of our dataset; thus, using Simanneal enhances the generality of our solution.

Shipping estimation

Estimating shipping costs can be particularly challenging due to the wide variation in shipping practices and regulations across different countries. For instance, shipping methods can vary significantly depending on factors such as customs procedures and local delivery practices. Given these complexities, a compre-

hensive international shipping analysis is beyond this thesis's scope. Therefore, for our application, we have decided to standardise our shipping estimations by using the Czech Republic as the destination country and by using shipping cost estimations from BrickLink and the author's personal experience.

Optimisation analysis

After analyzing our optimization results, we observed distinct characteristics of the algorithms employed. Greedy algorithms, known for their speed, produced results rapidly. However, these results, while competitive (within a margin of 200 CZK compared to the results from Rebrickable), they often lacked realism and precision. The primary advantage of greedy algorithms is their efficiency, which comes at the cost of accuracy in our problem space.

On the other hand, simulated annealing, which we limited to a runtime of 15 minutes due to cost constraints, demonstrated a more thorough and considerate approach. This algorithm explored the solution space more extensively, yielding more refined results. Despite this, the sheer size and complexity of the problem space meant that the results were still not optimal. Achieving truly optimal results would require significant fine-tuning of the algorithms, a process that is both time-consuming and beyond the scope of this thesis.

Results

As there might be a lot of bricks in the dataset to optimise for our resulting offers, files can be large in file size; as mentioned previously, we have decided to use Google Cloud Storage for its on-demand pricing instead of Cloud SQL's pricing model, which is based on the minutes and the memory per hour used. We save the resulting offers in the same schema as the predefined format has, but with one brick having only one chosen offer. Users can download this file and view it locally on their computers or by using any JSON viewer. Unfortunately, they have to do the final step of buying the bricks from the stores manually as, currently, the platforms do not offer a way of uploading a list of files with the store for them to buy from.

Update mission function

The updateRequestMission function is designed to update the status of a specific mission within a user's request in Firestore. An HTTP request triggers the function. After obtaining the request ID, mission ID and the optimisation results from the request body, the function runs a Firestore transaction to ensure atomic updates. It retrieves the request document, updates the status, error message, results path, and results data for the specified mission, and then commits these

changes. This function ensures that mission statuses are accurately updated in the database, providing up-to-date information for users. Based on the fact that this function works with the Firestore database and that we decided to use Node.js in the database manipulation functions, we also implemented this functionality in Node.js.

Send an email function

The "send email" function is designed to notify users via email when their brick optimisation run finishes. When the function receives an HTTP request from our workflow, it parses the request body to extract user ID (uid), request ID, mission ID, and optimisation results. It then retrieves the user's email from Firestore using the getUserData function. Based on the optimisation results, the function dynamically generates an HTML email summarising the algorithm used, the status of the optimisation, and any relevant cost or error information. We chose SendGrid [2] for sending emails due to its reliable and scalable email delivery service, which ensures our emails reach users promptly and efficiently. The function constructs an email message using SendGrid's API and sends it to the user's email address. This automated email notification enhances user experience by providing timely updates on their optimisation requests. Based on the function's need to access the Firestore database and SendGrids SDK support, we have chosen Node.js as the framework of this function.

6.6 Frontend

Our application's front end uses Next.js with TypeScript (chapter 4), providing a robust and scalable foundation for developing modern web applications. We leverage Firebase snapshots to keep our data updated in real time, ensuring users always have the most current information without needing to refresh the page. We use the Firebase Authentication for authentication. Our design system is built with the Material-UI (MUI) react component library [3], which offers a comprehensive set of components that adhere to Google's Material Design guidelines (subsection 5.1.3), ensuring a consistent and intuitive user experience. The application is deployed using Firebase Hosting. Refer to appendix section B.3 for the code example.

[2] https://sendgrid.com/
[3] https://mui.com/

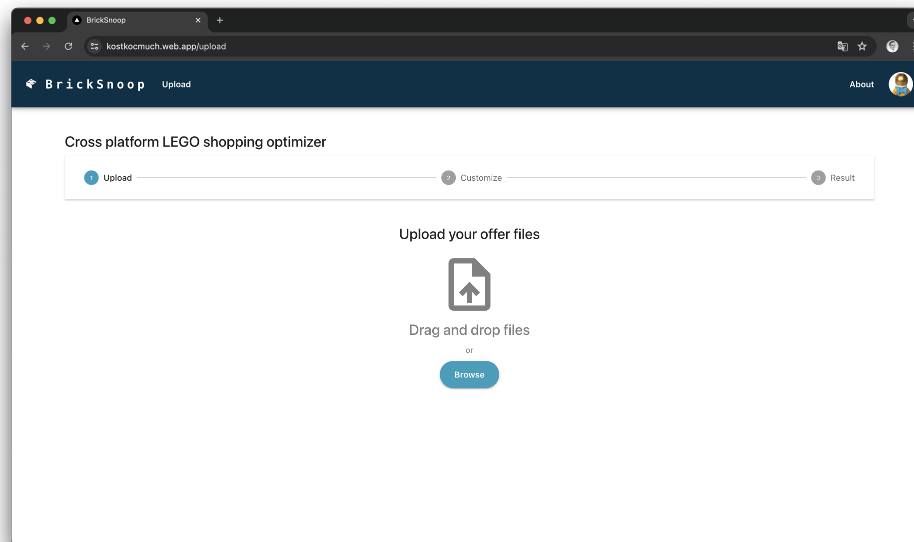

Figure 6.4 Implemented and hosted Upload page

6.7 Deployment

We use the command line tools for Firebase, Next.js and Google Cloud to deploy the frontend and backend parts. The commands to be run can be found in README.md files.

6.8 Summary

In this chapter, we detailed the development of our application, focusing on integrating technologies like Next.js, TypeScript, Python, Node.js, Firebase Firestore, and GCP Cloud Storage. We explained the authentication system using Firebase Authentication and outlined the database architecture and its security measures. The storage architecture was discussed, highlighting the use of GCP Storage buckets for handling large files. We covered backend functions for database communication, data manipulation, and optimisation, including implementing greedy algorithms and simulated annealing using the Simanneal library. We also discussed the details of the front-end implementation using MUI.

Chapter 7

User documentation

In this chapter, we will go over the user path and show how to use our application.

7.1 Access to the Page

During the creation of the bachelor thesis and for a certain period after its submission, the web application will be available at the address `https://kostkocmuch.web.app/`.

7.2 Learning about the application

After accessing the website, the users are presented with the Index page, which, in our case, is the About page. The users can learn about the application (1) and about the features that it offers (2). In the prerequisites section of the Index page, users can download the predefined format schema (3) of the files they need to upload. These files are called offer files in the application. They are a list of bricks with the corresponding offers. The users can obtain such files by using their tools or by using the currently available offer scrapers for the three popular platforms (5) (subsection 1.1.2). These scrapers are published on the Apify store and are not part of this bachelor's thesis. The users need a Rebrickable CSV to use these scrapers. They can either use their own (when they create or find a model, they can export the parts into this format) or download a sample one (4). If the users only want to try out the functionality of our application without the need to scrape the offers, they can download the sample file (6). After having the files ready, users can click on the "Start using the application" button (7), which will navigate them either to the Login or Register page if they are not signed in or will be navigated to the Upload page.

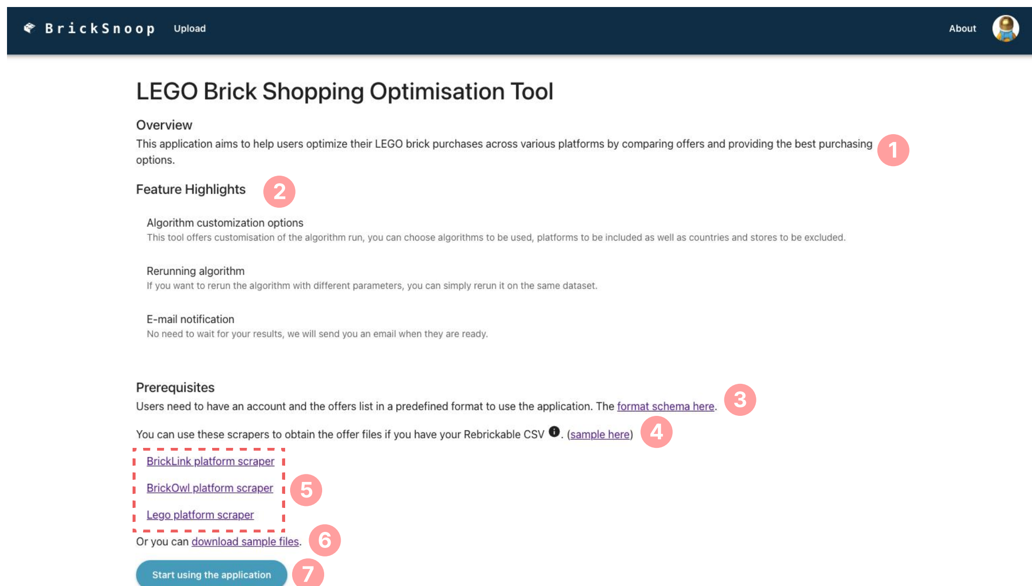

Figure 7.1 Index / About page

7.3 Authentication

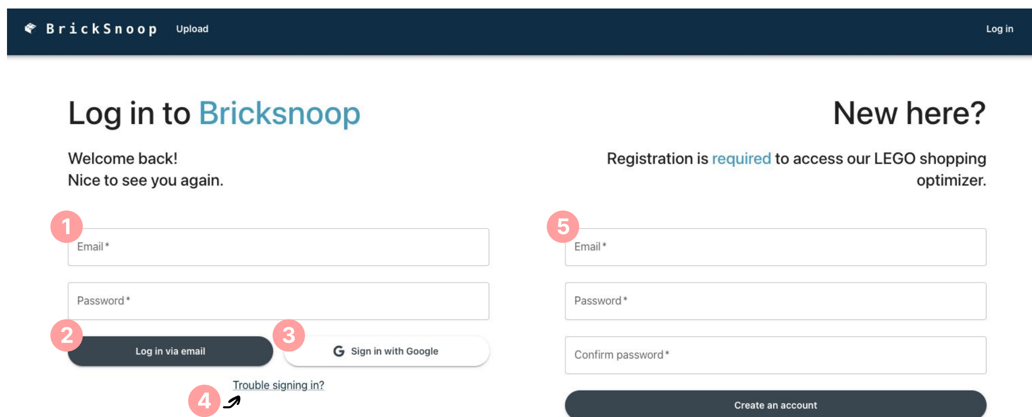

Figure 7.2 Login or Register page

7.3.1 Returning user

After accessing the Login or Register page, the users with an account can sign in using the form ① and pressing the "Login in via email" button ② or by using

the Google Sign In feature (3). If the user has forgotten their email, they will press the "Trouble signing in?" button (4) and be prompted to enter their email address. After that, they will get the reset password email.

7.3.2 New user

After accessing this page, new users can register using Google Sign In (3) or the form (5). After the creation of the account, they will be prompted to choose their avatar and username (see Figure 7.3).

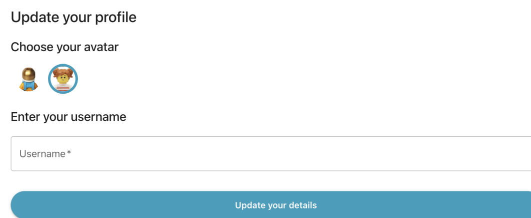

Figure 7.3 Account details prompt

7.3.3 Upload

After the authentication completion, users are navigated to the Upload page, where they can upload their offer files using the drag and drop feature (1) or by searching their computer(2).

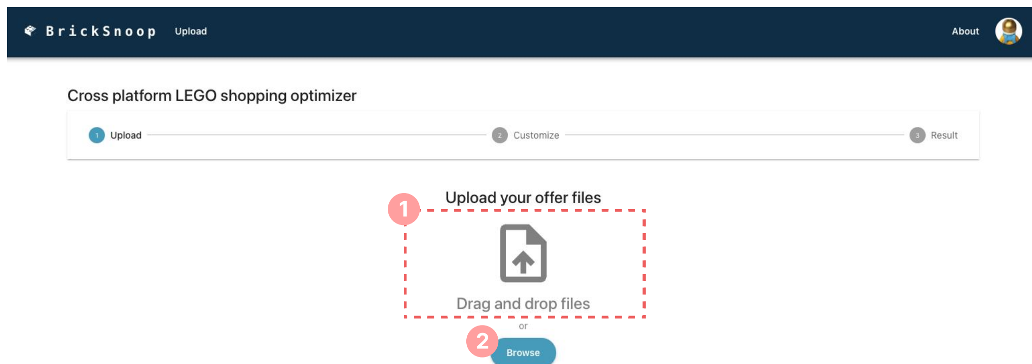

Figure 7.4 Upload page before the upload

When the user chooses the files to be uploaded, they will be checked against the JSON schema. All uploaded files will be displayed in the table (1), where their file name (2) and status (3) can be found. Also, if the JSON schema validation fails, the alert snack bar is shown (4). The user can add more files (5) or use the accepted files for optimisation (6). After choosing to continue with the optimisation process, the files are uploaded and preprocessed. After the preprocessing, the user is redirected to the Customisation page.

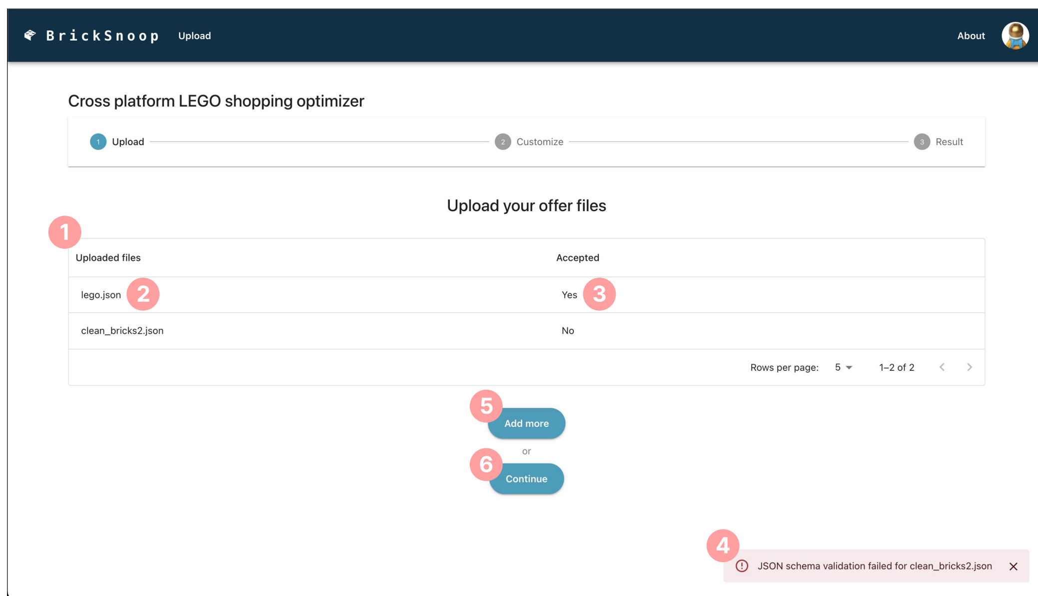

Figure 7.5 Upload page after the upload

7.4 Algorithm customisation

Being redirected to the Customisation page, the users can choose the algorithms to be used for the optimisation (1), the platforms they want to include in the optimisation process (2), the countries they wish to exclude (3) as well as the stores they do not want to include (4). The users can start the optimisation process by pressing the "Optimize" button (5). It will create a mission for each algorithm, which will be displayed on the Request Missions page. The users can also see until when the dataset will be available for the optimisation (6).

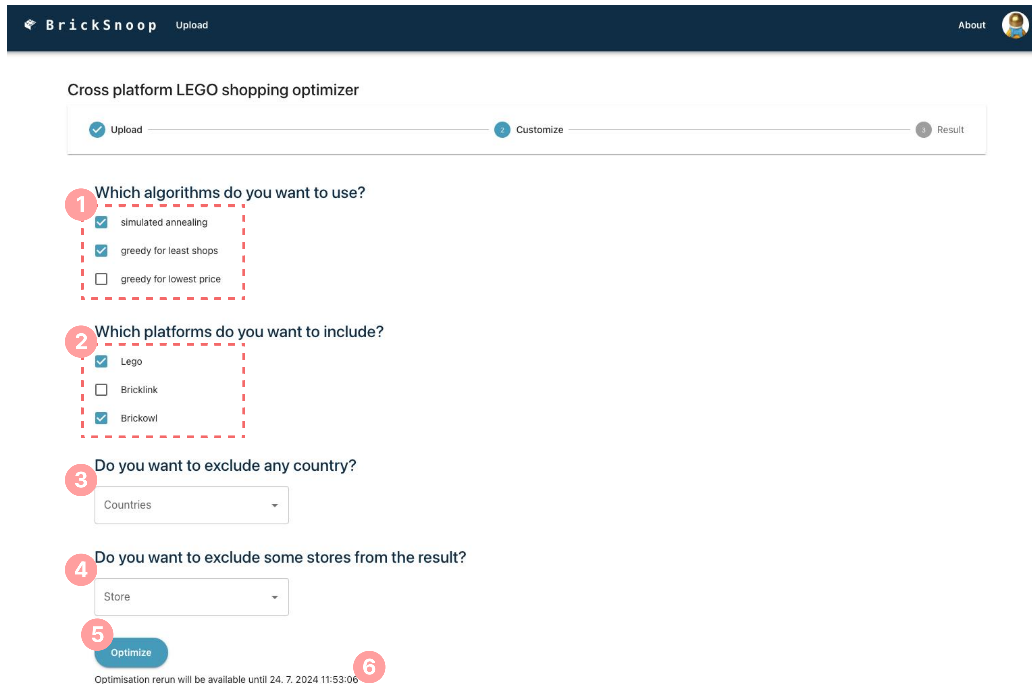

Figure 7.6 Customisation page

7.5 Algorithm run overview

The Request Missions page shows an overview of all algorithm runs (missions) bound to a single dataset (referenced as request). Each algorithm run is displayed as a card (1), where the mission ID, algorithm used, when the mission was created and updated. Also, the status of the optimisation is displayed on the card. The status can be optimisation was successful (3), the optimisation is still running (4), or it can be errored, where an error message will be displayed. If the algorithm run has successfully finished, the users can press the "Details" button (3) and be redirected to the Mission details page. Furthermore, when the algorithm run finishes, the users get an email with the statistics and a link to see the optimisation run details. The users can also click the "Rerun the optimisation with different parameters" button (5) to be redirected back to the customisation page.

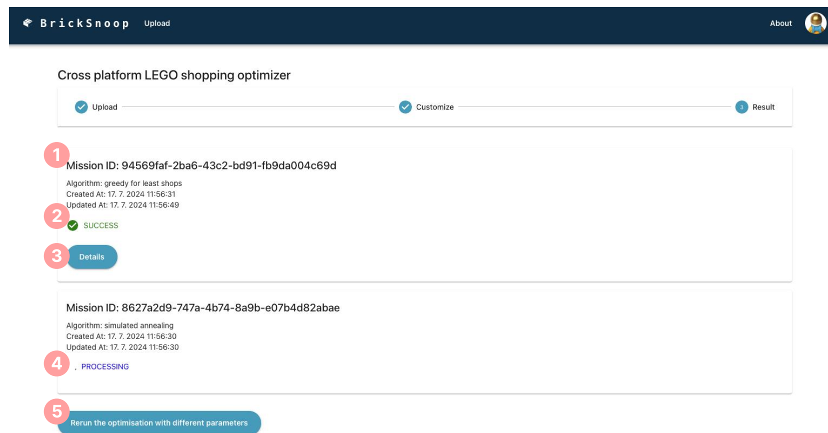

Figure 7.7 Request Missions page

7.6 Algorithm run statistics

The Mission details page shows the mission ID, algorithm used, request ID, and how many platforms the user has chosen to include, how many stores to exclude, and how many countries not to include (1). The final optimised price is displayed, along with the estimated shipping price and the information about how many parts, platforms and stores have been chosen (2). The user can download the results by pressing the "Export parts for each store" button (3). The JSON file will be downloaded, and the user can use the JSON file viewer (4) to view the results. By pressing "See all results"(5), the user can navigate back to the algorithm run overview.

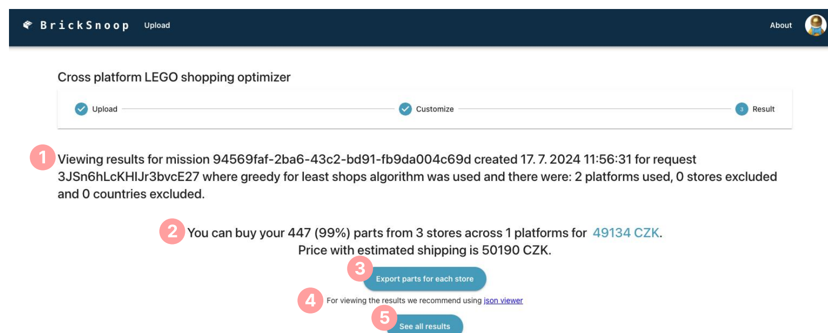

Figure 7.8 Mission details page

7.7 User account customisation

At any point, the signed-in user can click on their avatar in the app bar (1). The modal with information such as username and since when the user is a member will be shown (2). The user can either customise their profile by pressing the "Customise profile" button (3) or log out of the application by clicking the "Log out" button (4). If they choose to customise the profile the dialog with options to select the avatar or change the username will be shown (5).

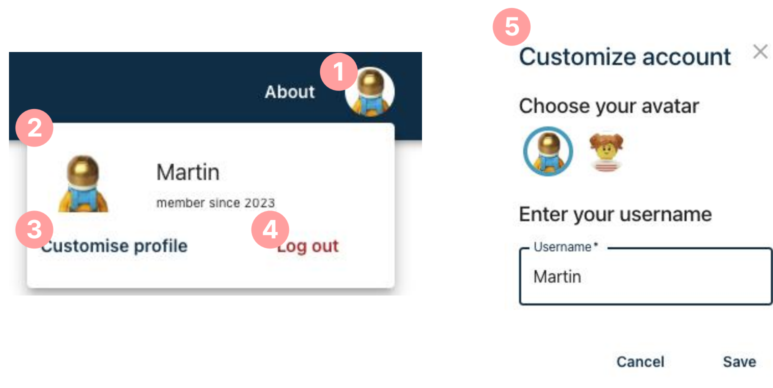

Figure 7.9 User account customisation flow

Chapter 8

Conclusion

In the conclusion part of this thesis, we will evaluate how well we have met the requirements defined in the chapter 2 and whether we have accomplished the goal of this thesis (section 1.3).

8.1 Evaluation of Functional Requirements

In this section, we will assess whether we have successfully met the functional requirements as outlined in the section 2.4.

- **Requirements 1 and 2: the user must be able to access the application on the web and navigate inside it.**

  The application is deployed on the web, and the user is able to access it as well as to navigate inside of it as described in chapter 7.

- **Requirements 3 through 6: the user must be able to learn about the application, algorithm customisation and prerequisites.**

  The system enables a user to access the About page (section 7.2), which contains information about the application, about the possible customisations of the optimisation algorithm as well as shows the predefined format that the offer data needs to be in. Furthermore, it mentions already existing tools for getting the data in such format from the most used platforms (subsection 1.1.2).

- **Requirements 7 through 11: the user must be able to create and access their account as well as to change the password if forgotten. Furthermore, the user must be able to change their username or log out**

The system enables a user to access the Sign in page (section 7.3), which allows a user to create and access the account using both username and password and Google Sign-in. Also, the user is able to reset their password if they forgot it. After signing in, the user can change their username or can log out.

- **Requirements 12 through 15: the user must be able to upload a list of offers from multiple sources as files in a predefined format, see the list of files to be uploaded and be alerted if the files do not adhere to that format.**

  The system allows users to upload the data by creating an Upload page (subsection 7.3.3), with the functionality of testing the format of the data against a schema and seeing the list of files to be uploaded. After confirming, this data is sent to preprocessing and saved to Cloud Storage.

- **Requirements 16 through 21: the user must be able to customise the optimisation algorithm run and start it.**

  Our application allows a user to access the Customisation page (section 7.4), where they can choose:

  - algorithm to be used, available options are greedy for the largest store, greedy for the cheapest offer and simulated annealing
  - platforms to be worked with during optimisation
  - stores to be left out during the optimisation
  - countries from which the user wants the stores and offers to be from

  After the user is satisfied with the customisation, they can start the optimisation process.

- **Requirements 22 through 25: the user must be able to see optimisation run overview, see statistics for a specific run, download the results and rerun the optimisation process with different parameters.**

  The system allows a user to see the Optimisation overview page (section 7.5), where they can find out about all the optimisation runs on the specific dataset as well as about the status they are in. After choosing a specific run, they are shown statistics (section 7.6) such as:

  - chosen algorithm.
  - price of the bricks to be bought

- price of the bricks to be bought with the estimated shipping
- the number of bricks to be bought
- the number of stores chosen by the optimisation algorithm
- the number of platforms chosen by the optimisation algorithm

After reviewing these statistics, users can download the optimisation results or choose the Rerun optimisation option, where they will be navigated to the Algorithm customisation page.

After reviewing the requirements and showing which parts of our application satisfy which requirements, we can say that our application meets the requirements defined in section 2.4.

8.2 Meeting the objective of the thesis

The objective of this bachelor's thesis was to **implement cross-platform LEGO brick shopping optimisation tool**. By creating a functional application and meeting the functional requirements as stated above, we have completed and met the objective of this thesis.

8.3 Possible improvements

Based on the feedback and the author's experience using it, we have created a list of possible enhancements and potential implementation strategies. These enhancements are out of the scope of this bachelor's thesis.

- *Additional algorithms.* As the optimisation problem of finding the optimal offers for bricks is NP-hard (chapter 3), we turned to approximation and heuristic algorithms. These algorithms provide solutions; however, their quality largely depends on the specifics of the dataset and the duration of the optimisation run; thus, choosing the best algorithm for every dataset is impossible. By implementing more algorithms, some with the possibility of customisation of optimisation duration, we would be able to, in some cases, find even better solutions for our users. The application could be extended this way by implementing such an algorithm, adding it to the optimisation workflow and allowing the users to choose it while customising the optimisation run.

- *Direct platform integration.* By adding support of the direct offer integration from the biggest platforms (subsection 1.1.2), the users would only need

to obtain the offers from some niche or new platforms, making it easier for them to get the prerequisites of our application. However, this would require these platforms to enable such integration, for example, by enabling us to get the offers by API and share their data, which they currently do not offer. If they did, it would require only small changes, where we would gather the data before the optimisation.

- *Support for more countries and currencies.* As mentioned in the chapter 6, our application currently supports offers with the price in Czech koruna and takes the Czech Republic as a destination country. On a global scale, currency exchange and shipping estimation present significant challenges due to the wide variation in shipping practices, regulations, and currency fluctuations across different countries. Therefore, expanding support for additional countries and currencies is a complex task beyond this thesis's current scope. It would be possible to incorporate courier and currency exchange APIs to address these challenges in future work, but such integration requires extensive development and testing.

The analysis aimed to show that the application is designed to be easily extendable, an important qualitative requirement of the software system.

Bibliography

[1] Michael Crider. *Tools to Get Started Designing Your Own LEGO Creations*. 2020. URL: https://www.howtogeek.com/37000/tools-to-get-started-designing-your-own-lego-creations/ (visited on 05/15/2023).

[2] BrickLink. *About BrickLink*. 2023. URL: https://www.bricklink.com/v3/about.page (visited on 04/23/2023).

[3] BrickOwl. *BrickOwl Stores*. 2023. URL: https://www.brickowl.com/stores (visited on 04/23/2023).

[4] Auth0. *Social Login Report*. 2023. URL: https://assets.ctfassets.net/2ntc334xpx65/77U9sLF07rD7t9zdI6Q1SV/a8e2054b5affc0280769516eee70b0ea/Social-Login-Report.pdf.

[5] Jacek Blazewicz et al. "Internet shopping optimization problem". In: *International Journal of Applied Mathematics and Computer Science* 20 (June 2023), pp. 385–390. DOI: 10.2478/v10006-010-0028-0.

[6] Thomas H Cormen et al. *Introduction to Algorithms*. MIT press, 2009.

[7] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[8] Emile H. L. Aarts and Jan H. M. Korst. *Simulated Annealing: Theory and Applications*. Springer, 1988.

[9] David E Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Publishing Company, 1989.

[10] Marco Dorigo and Gianni Di Caro. "Ant colony optimization: a new meta-heuristic". In: *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)* (2006), pp. 1470–1477.

[11] Techopedia. *Technology Stack*. 2021. URL: https://www.techopedia.com/definition/28452/technology-stack (visited on 10/12/2023).

[12] StackOverflow. *Popularity of web frameworks.* 2023. URL: https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe (visited on 04/04/2024).

[13] Angular. *What is Angular?* 2023. URL: https://v17.angular.io/guide/what-is-angular (visited on 10/08/2023).

[14] React. *About React.* 2023. URL: https://react.dev/ (visited on 10/08/2023).

[15] Next.js. *What's in Next.js?* 2023. URL: https://nextjs.org/ (visited on 10/08/2023).

[16] Vue.js. *What is Vue?* 2023. URL: https://vuejs.org/guide/introduction.html (visited on 10/08/2023).

[17] Vitaly Makhov. *Server vs. Serverless: Benefits and Downsides.* 2021. URL: https://nordicapis.com/server-vs-serverless-benefits-and-downsides/ (visited on 10/08/2023).

[18] Scout APM. *Serverless Architecture: Pros, Cons, and Examples.* 2022. URL: https://scoutapm.com/blog/serverless-architecture (visited on 10/08/2023).

[19] StackOverflow. *Popularity of cloud frameworks.* 2023. URL: https://survey.stackoverflow.co/2023/#cloud-platforms (visited on 04/04/2024).

[20] Google. *Google Cloud overview.* 2023. URL: https://cloud.google.com/docs/overview (visited on 10/10/2023).

[21] Amazon. *About AWS.* 2023. URL: https://aws.amazon.com/about-aws/ (visited on 10/10/2023).

[22] Azure. *What is azure?* 2023. URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure (visited on 10/10/2023).

[23] Python. *About Python.* 2023. URL: https://www.python.org/about/ (visited on 10/11/2023).

[24] Node.js. *About Node.js.* 2023. URL: https://nodejs.org/en/about (visited on 10/11/2023).

[25] PHP. *What is PHP?* 2023. URL: https://www.php.net/manual/en/intro-whatis.php (visited on 10/11/2023).

[26] Oracle. *What is Java technology and why do I need it?* 2023. URL: https://www.java.com/en/download/help/whatis_java.html (visited on 10/11/2023).

[27] Alla Kholmatova. *Design Systems.* Smashing Magazine, 2017.

[28] Built In. *11 Benefits of Design Systems for Designers, Developers, Product Owners, and Teams.* 2022. URL: https://builtin.com/articles/11-benefits-design-systems (visited on 12/06/2023).

[29] UXPin. *9 Best Design System Examples in 2024.* 2024. URL: https://www.uxpin.com/studio/blog/best-design-system-examples/ (visited on 05/31/2024).

[30] Google. *Usage and limits.* 2024. URL: https://firebase.google.com/docs/firestore/quotas (visited on 05/31/2024).

[31] Rebrickable. *LEGO Part 3023 Plate 1 x 2.* 2024. URL: https://rebrickable.com/parts/3023/plate-1-x-2/ (visited on 05/31/2024).

[32] Jared Hinton. *LEGO Part 3023 Plate 1 x 2 image.* 2018. URL: https://rebrickable.com/media/parts/photos/0/3023-0-0da9dce5-9fc1-4d56-8002-51eca76cc2a9.jpg (visited on 05/31/2024).

[33] Basilis Gidas. “Nonstationary Markov Chains and Convergence of the Annealing Algorithm”. In: *Journal of Statistical Physics* 39 (Apr. 1985), pp. 73–131. DOI: 10.1007/BF01007975.

Appendix A

Directory structure

```
\bricksnoop-backend-nodejs
    \functions
        \src
            \emails
            \requests
            \shared
            \users
            files for functions
\bricksnoop-backend-python
    \clean_bricks
        deploy.sh
        source code files
    \optimise_bricks
        deploy.sh
        source code files
    \workflows
        workflow.yaml
        deploy.sh
\bricksnoop-frontend
    \assets
    \components
    \lib
    \pages
    \public
    \shared
    \styles
    \utility
    source code files
```

Appendix B

Code examples

B.1 Firebase Functions Code

This is an example of Firebase cloud function code written in Node.js, it is used to create the Request entity represented by interface IRequest.

```
import { https } from "firebase-functions";
import {
    RequestDataStatusEnum,
    IRequest,
    firestore,
    createRequestInput
    } from "../shared/shared";
import * as functions from "firebase-functions";
import * as struct from "superstruct";

export async function createRequest(
    data: any,
    context: https.CallableContext
    ) {
        if (data === null)
            throw new https.HttpsError(
                "invalid-argument",
                "no data"
            );
        if (!struct.is(data, createRequestInput))
            throw new https.HttpsError(
                "invalid-argument",
                "invalid data"
```

```
        );
    if (!context.auth?.uid)
        throw new https.HttpsError(
            "invalid-argument",
            "no auth"
        );

    const uid: string = context.auth.uid;
    const id: string = firestore
                    .collection("requests").doc().id;

    const validUntil = new Date();

    validUntil.setDate(validUntil.getDate() + 7);

    const requestData = {
        requestId: id,
        createdBy: uid,
        dataStatus: RequestDataStatusEnum.PROCESSING,
        missions: {},
        validUntil: validUntil,
        createdAt: new Date(),
        updatedAt: new Date(),
    } satisfies IRequest;

    return firestore.collection("requests")
        .doc(id)
        .set(requestData)
        .then((docRef) => {
            functions.logger.info(
                `Created request with id ${id}`,
                { structuredData: true,}
            );
            return id;
        })
        .catch((err) => {
          functions
            .logger.error(err, { structuredData: true });
        });
}
```

B.2 TypeScript types

These are the interfaces representing the entities that our application is working with.

```
export interface IUser {
    uid: string;
    email: string;
    username: string;
    avatar: string;
    isNewUser: boolean; //helper property
    createdAt: Date;
    updatedAt: Date;
}

export interface IRequestMissionResult {
    bestCost: number;
    noPenaltiesCost: number;
    noShippingNoPenaltiesCost: number;
    numberOfPlatforms: number;
    numberOfStores: number;
    numberOfBricks: number;
}

export interface IRequestMission {
    status: MissionStatusEnum;
    missionId: string;
    executionId: string;
    errorMessage?: string;
    algorithm: string;
    platforms: string[],
    countriesToExclude: string[],
    storesToExclude: string[],
    createdAt: Date;
    updatedAt: Date;
    result?: IRequestMissionResult;
    resultsPath?: string;
}


export interface IRequest {
```

```
    requestId: string;
    createdBy: string;
    dataStatus: RequestDataStatusEnum;
    missions: { [id: string]: IRequestMission };
    validUntil: Date;
    createdAt: Date;
    updatedAt: Date;
    stores?: string[];
    countries?: string[];
    platforms?: string[];
    numberOfBricks?: number;
    errorMessage?: string;
    }
```

B.3 Frontend Component Example

This is the Update user details component on the front end. It uses MUI components, Firebase Authentication and Firebase Functions.

```
import { auth, functions } from "../lib/firebase";
import { useContext, useEffect, useState } from "react";
import { UserContext } from "@/lib/context";
import {
  Alert,
  Box,
  Button,
  Grid,
  Snackbar,
  TextField,
  Typography,
} from "@mui/material";
import { useRouter } from "next/router";
import AvatarChoice from "@/components/AvatarChoice";
import { httpsCallable } from "firebase/functions";

export function UpdateDetailsForm() {
  const router = useRouter();
  const { userData } = useContext(UserContext);
```

```
const [username, setUserame] = useState(
  userData ? "" : userData?.username
  );
const [avatarName, setAvatarName] = useState("default");
const [loading, setLoading] = useState(false);
const [error, setError] = useState<string | null>(null);

const handleErrorClose = () => {
  setError(null);
};

useEffect(() => {
  if (userData) {
    setAvatarName(
      userData?.avatar.toString() === ""
        ? "default"
        : userData?.avatar.toString()
    );
    setUserame(userData?.username ?? "");
  }
}, [userData]);

const handleSubmit = async (event: any) => {
  event.preventDefault();
  console.log(event);

  const uid = auth?.currentUser?.uid;
  const email = auth?.currentUser?.email;

  if (uid === null) {
    setError("Please log in to update your details");
    return;
  }

  if (username === null || username === "") {
    setError("Please enter a username");
    return;
  }

  setLoading(true);
```

```
  //Call to the cloud function
  const updateUserDetails = httpsCallable(
                                  functions,
                                  "updateUserDetails"
                                  );
  updateUserDetails({
    email: email,
    username: username,
    avatarName: avatarName,
    isNewUser: userData === null,
  })
    .then((result) => {
      const data: any = result.data;
      router.push(‘/upload‘);
    })
    .catch((error) => {
      const code = error.code;
      const message = error.message;
      const details = error.details;
      console.log(code, message, details);
    })
    .finally(() => {
      setLoading(false);
    });
};

return (
  <>
    <Box sx={{ mt: 4 }}>
      <Grid container spacing={14} alignContent={"center"}>
        <Grid item xs={0} md={2} display={{
              xs: "none", md: "block"
          }}>
          {" "}
        </Grid>
        <Grid item xs={12} md={8}>
          <Typography variant="h5">
              Update your profile
          </Typography>
          <Box sx={{ mt: 2 }}>
```

```
<form noValidate onSubmit={handleSubmit}>
  <Box>
    <Typography variant="h6" sx={{ mb: 1 }}>
      Choose your avatar
    </Typography>
    <AvatarChoice
      avatarName={avatarName}
      setAvatarName={setAvatarName}
    />
  </Box>
  <Box sx={{ mt: 2 }}>
    <Typography variant="h6">
      Enter your username
    </Typography>

    <TextField
      variant="outlined"
      margin="normal"
      required
      fullWidth
      id="username"
      label="Username"
      name="name"
      value={username}
      defaultValue={username}
      onChange={
          (e) => setUserame(e.target.value)
          }
      autoComplete="name"
      focused={
        !(
          username === "" ||
          username === undefined ||
          username === null
        )
      }
    />
  </Box>
  <Button
    sx={{ mt: 3 }}
    type="submit"
```

```
                fullWidth
                variant="contained"
                disabled={loading}
              >
                Update your details
              </Button>
            </form>
          </Box>
        </Grid>
        <Grid item xs={0} md={2} display={{
          xs: "none", md: "block"
          }}>
          {" "}
        </Grid>
      </Grid>
    </Box>
    <Snackbar
      open={!!error}
      autoHideDuration={6000}
      onClose={handleErrorClose}
      anchorOrigin={{
          vertical: "bottom",
          horizontal: "right"
          }}
    >
      <Alert
        onClose={handleErrorClose}
        severity="error"
        sx={{ width: "100%" }}
      >
        {error}
      </Alert>
    </Snackbar>
  </>
  );
}
```

B.4 Input JSON schema

This is the JSON schema of the predefined input of our application.

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "array",
    "items": {
        "type": "object",
        "properties": {
            "rebrickableID": {
                "type": "string"
            },
            "rebrickableColor": {
                "type": "integer"
            },
            "wantedQty": {
                "type": "integer"
            },
            "brickPreviewImage": {
                "type": ["string", "null"]
            },
            "offers": {
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "platform": {
                            "type": "string"
                        },
                        "platformBrickID": {
                            "type": "string"
                        },
                        "platformBrickColor": {
                            "type": "integer"
                        },
                        "seller": {
                            "type": "string"
                        },
                        "country": {
                            "type": "string"
                        },
                        "quantity": {
                            "type": ["integer", "string"]
                        },
```

```
                        "price": {
                            "type": "string"
                        },
                        "originalPrice": {
                            "type": ["string", "null"]
                        },
                        "minBuy": {
                            "type": ["string", "null"]
                        },
                        "minAvgLot": {
                            "type": ["string", "null"]
                        },
                        "shipping": {
                            "type": ["string", "null"]
                        }
                    },
                    "required": [
                        "platform",
                        "platformBrickID",
                        "platformBrickColor",
                        "seller",
                        "country",
                        "quantity",
                        "price"
                    ]
                }
            }
        },
        "required": [
            "rebrickableID",
            "rebrickableColor",
            "wantedQty",
            "offers"
        ]
    }
}
```