Design Project

Bricksyncer

Authors:

Connor Bleumink (s1948717)
Daniël Huisman (s1697471)
Lian van Kesteren (s2003279)
Remus Niculescu (s2189542)

Supervisor:

Dr. Faiza Bukhsh

UNIVERSITY OF TWENTE.

April 16, 2021

Abstract

Bricksyncer aims to be a user-friendly cloud-based software application that Unbrickable and possibly other sellers can use to synchronize their local stock of LEGO bricks with any selling platform. This report provides an overview of the context where this type of application is desired and states all requirements it has to fulfil to operate properly and satisfy the user's needs. At the same time, it describes logical steps that were considered in the approach of the project and goals for each development phase. Moreover, the report contains an analysis, thorough discussion of design choices and the global design structure of the project. Additionally, it includes testing procedures and techniques used to verify its functionalities. In the end, this document presents an evaluation of the whole process, suggestions for maintenance and future improvements.

Contents

Chapter 1: Introduction

This chapter will reveal the social context in which the project came into fruition as well as general information about the main focus of the project, the already existing solution, and the problems that it aims to solve. Having a clearer understanding of the background that generated the necessity for this project will aid in understanding the requirements that are going to be presented thoroughly throughout the upcoming chapters.

1.1 Unbrickable

Unbrickable is a social enterprise that is focused to help mentally challenged youths from ages 14 to 21 to develop their social and work-related skills. They try to achieve this by grouping them up with non mentally challenged youths and have them work together as a team. Within the team, they have to share responsibilities and try to reach their developmental goal, by working together in a certain branch in the company which would be related to the interests of the youths or is directly linked to what they are studying. Youths that have worked within Unbrickable are provided help with looking for further education or a new employer. [1]

1.2 Brickconnect

LEGO is a toy that millions of people use, ranging from children to adults. The adult fans of Lego (AFOL) not only collect Lego sets, but sometimes also create their own builds. For these builds they need specific bricks which are difficult to find in LEGO stores. Due to this difficulty they buy their bricks online, which sparked the creation of LEGO specific online stores like Brickowl and Bricklink. In these stores a wide range of people sell Lego pieces from individuals who just want to sell a few bricks to companies like Unbrickable. Brickconnect aims to be a software application to help these sellers to manage their shops.

1.3 Kulla

The warehouse is located in Enschede. Here new items come in for sale. These have to be placed in the correct locations within the warehouse. Right now these locations are memorized by staff. This makes it difficult for new employees, as they have to memorize all locations in order to know which item is placed where. Also orders come in that need to be assembled and shipped. Assembling each order also requires employees to know the locations of the items.

[1] https://unbrickable.info/sociaal/

Previous year, another group of students from UT worked with Unbrickable for their design project to automate some processes in order to be more efficient and easier for newly hired employees. Their project was called Kulla. They developed software to be able to digitize bricks, create digital item locations, and assign items to these locations. This meant that not every location had to be memorized by the employees and that items could be found much faster. When an order was placed, it also created an overview of where each item could be found so that it was easier for the employees to assemble the order.

1.4 Current solution

Unbrickable currently uses a software called Bricksync to synchronize the local stock with Brickowl and Bricklink. This software changes the stock on the websites when an order is made, and can be used to keep track of local stock. The software however does not automatically increase the stock when an order has been cancelled, and thus having the manager have to add the stock manually.[2] Bricksync also does not allow for having different prices on either website, nor does it allow the freedom to synchronize local stock with online stock when the seller doesn't want to use both websites to sell a certain brick. Lastly the tool is operated by text line commands making it difficult to use for certain individuals.

[2] http://www.bricksync.net/#source

Chapter 2: Requirements

In order to design a new system that works on full capacity, it is necessary to extract what the users/stakeholders want from the system and what functionalities are expected to be fulfilled by it. After a couple of meetings with the client, we managed to divide all the requirements in 4 categories of priority according to the MoSCoW principles - must, should, could, won't - depending on how crucial they are for completing the system. In the upcoming chapter, the requirements will be presented along with a brief description for each one of them.

2.1 Priority list

2.1.1 Must

These are the requirement specifications that are mandatory for a fully operational system that can fulfil all the necessary functionalities. The primary aim of the project is to fulfill these requirements with no exception.

2.1.2 Should

This set of requirements will increase the efficiency of the overall system and will make it easier to use. Moreover, adding additional options will reduce the necessary time to complete the task for an employee.

2.1.3 Could

These requirements are a nice addition to the original purpose of the system if there is enough time left for implementation, but their absence won't interfere with any critical functionality.

2.1.4 Won't

These requirements are outside the scope of this project, but they might prove useful upgrades in the future.

2.2 Definitions

- A **platform** is an online store where LEGO is sold. This can be an external store or Unbrickable's own website.
- **Local** is used to refer to the Brickconnect system. For example, a local order is an order in the Brickconnect system.
- A **batch** is one or multiple new items that are going to be added to the stock and/or platforms.

2.3 Roles

- A **bricksyncer** is a person allowed to add stock to the platforms and to change the stock on the platforms.

2.4 User requirements

2.4.1 Must have

1. As a bricksyncer, I want an intuitive visual interface.
   An intuitive visual interface is crucial to have since it allows the user to clearly understand every element of the design, navigate and complete tasks with ease. Also, it makes the system less susceptible to human errors since every action only requires a few simple steps to be completed.

2. As a bricksyncer, I want to see an overview of batches that have not yet been added to the platforms.
   It is important to keep track of which batches have already been added to the platforms and having a list in this regard offers a good overview of what items are included. This leads to an easier update of item details and gives the chance for double-checking before putting everything up for sale.

3. As a bricksyncer, I want to upload a BSX file to the batch overview.
   Inside the BrickConnect system, there are two ways in which a batch can be added to the overview. It can either be fetched from the Warehouse, or it has to be uploaded manually via a BSX file. In the latter case, the system needs a functionality to handle it.

4. As a bricksyncer, I want to set the price of each item.
   Items can have different prices on different sales platforms, prices that can deviate from their default value. Moreover, these prices can be changed constantly, and it is important that the update is fast, and it is synchronized along desired locations.

5. As a bricksyncer, I want to approve batches for uploading to the platforms.
   To further eliminate the possibility of human errors, the system offers the possibility to review batch details before uploading them on the platforms.

6. As a bricksyncer, I want to use the system at the same time as other users.
   There are multiple users of the system, so it has to make sure that it supports multiple simultaneous logins and changes to batches, items, etc without affecting the overall correctness of the actions performed.

7. As a bricksyncer, I want to see an overview of the operations performed by the system.
   The system is loaded with a lot of updates and changes daily, automatically or manually. For a better understanding of what happened within the system, what actions were performed and the results, it is important for an employee to be able to see an overview/log/history of what happened with proper details and timestamps.

8. As a bricksyncer, I want to be able to revert any operations performed by the system.
   Communication errors can occur, within the system or with external systems like the warehouse and that might lead to the necessity to revert certain actions performed. It can happen that some updates were not processed correctly or typing mistakes occurred or there are some last-minute changes that were not taken into consideration, thus it is important that every action is reversible.

2.4.2 Should have

1. As a bricksyncer, I want to schedule when items are added to the platforms.
   Whenever new batches and items are added, they have to be manually put on sale with corresponding prices and quantities for different platforms. But, any kind of change like this requires API calls and there is a limited number that can be executed daily. A scheduler would be useful because it can act as a priority setter depending on the user preferences and intentions.

2. As a bricksyncer, I want to set the price of each item that is added per platform.
   All the items have a standard price when they are fetched into the system for the first time. But environmental changes can lead to price changes, this includes higher traffic on a specific platform or higher demand for a certain item. The system should be able to handle this type of situation where the price of an item differs from platform to platform, and it should give the option to set the individual price per store before putting the item for sale.

3. As a bricksyncer, I want to change the quantities of items that are added to the platforms.
   A batch contains a certain amount of items and their corresponding quantities, but it is not mandatory that everything goes on sale. Thus, the possibility to change the quantity that will be sold is welcome.

4. As a bricksyncer, I want to select a priority option when adding items to the platforms. This will add the items as soon as possible.
   Due to the limited amount of daily API calls, it is not always possible to instantly add items to platforms. In situations where an item is on high demand, selecting higher priority for the resupply will allow the system to try to update quantities of that specific item first.

5. As a bricksyncer, I want to change the public comment of items that are on the platforms.

Sometimes people leave ill-intended comments on the platforms that neither describe the product nor the experience. Thus, the system could act as a sanitizer for this kind of situation.

6. As a bricksyncer, I want to change the prices of items that are on the platforms.
   New changes in prices might come at any time and can affect only certain platforms, thus price updates for an item listed for sale should be possible.

2.4.3 Could have

1. As a bricksyncer, I want to see an overview of how fast items are selling.
   This option can help create statistics to generate better marketing strategies.

2. As a bricksyncer, I want to change the quantity per platform of items that are added to the platforms.
   In general, there is a common stock for all the platforms which reflects the total amount of items in inventory. Whenever a new order is received, no matter on which platform, the amount is deducted from the total available stock, and it is updated accordingly to all other platforms. This option will give the possibility to split the total amount and associate each store with its stock.

2.4.4 Won't have

1. As a bricksyncer, I want to receive tips for improvements and optimizations on the sale of items.
   Collecting data of which items are preferred by customers, or how many sales there are a day on each platform or the distribution of sales across stores and generating tips for improvement and making more efficient resupply or update choices can be a nice future feature.

2. As a bricksyncer, I want a guided tutorial on how to use the system.
   To get a good grip on how to use the system, a video tutorial would be helpful but not necessary. A manual with instructions on how to use the system, including a step-by-step configuration and explanation of actions will be provided.

2.5 System requirements

2.5.1 Must have

1. The system must support at least two platforms, namely BrickLink and BrickOwl.
   Currently, there are only two platforms, namely BrickLink and BrickOwl, so the main purpose of the system is to handle the requests from and to these websites accordingly. Also, the system has to cope with the different API structures for each one of them in order to function properly.

2. The system must create a local order when an order is placed on a platform.
   Whenever an order is placed, the data of the order has to be stored in a new entry in the database with the right format to have a complete view of what happened in case of need.

3. The system must update the stock of the other platforms when an order is placed on a platform.
   All platforms share a common stock and that is the total quantity of an item that has been included in the inventory. If an order is placed and items are sold on any of them, the available quantity in stock inventory has to be updated and synchronized with all the other sales platforms.

4. The system must add the items of a cancelled order back to the stock of the other platforms.
   If an order is cancelled, either by a customer or an operator for particular reasons, the original amount of items before the order was placed has to be restored in the inventory and all platforms.

5. The system must add back the items that are refunded to the stock of other platforms if that option is selected on a platform.
   Customers have the option to ask for a refund for the items of an order if something does not correspond with the specifications or the bricks are damaged. In the first case, the system will have an option to make the items available for sale on other platforms.

6. The system must change the stock of the platforms when record about change in inventory is added.
   When there is a change in the inventory, this change has to be reflected in the available stock of all platforms immediately.

7. The system must create local changes in orders, for cancellations and (partial) refunds.
   In case of cancellations and (partial) refunds, the local entry of the database has to be changed to reflect the latest updates.

8. The system must use Brickconnect accounts for authentication.
   There is going to be a single central point of authentication for the whole BrickConnect system, so the synchronization software has to allow users to access it with the same credentials.

9. The system must log all operations performed on the platforms.
   To keep track of all the activity that happened on the platforms, every time there is something new, the system will create a log entry with details of the type of action executed and the results.

10. The system must be able to reverse any performed operations.
    If any mistakes are made regarding updating item details or listing batches for sale, the system has to be able to keep track of all the actions performed and revert them if needed. For every method there has to be an inverse equivalent that can be called to cancel the effect of previous actions.

11. The system must not exceed the daily API calls per platform and delay operations if needed.
    At any point of time, the system needs to keep track of how many API calls were used since there is a daily limit and delay actions that would exceed this amount until it becomes possible again.

12. The system must keep a configurable amount of daily API calls left per platform to handle orders.
    When the maximum amount of daily API calls, no action can be further performed on the platforms. Thus, the system has to reserve an amount of calls to make sure new orders can be handled.

13. The system must not upload items without a price.
    In case an item has no price, or it is 0, the system has to return an error and abort uploading items for sale.

2.5.2 Should have

1. The system should include the average price from the last 6 months as a suggested price.
   Whenever a new batch arrives, the suggested price or the default price for an item should be computed as the average price for the last 6 months.

2. The system should support exporting items that are being added to the platforms as a BSX file. The BSX file can be opened in Brickstock which will add average prices to the items. The system should then support uploading the file containing the items with prices. These prices should then appear as the suggested price in case there was no suggested price in the database.
   New batches and items can be added manually via a BSX file. The data of the BSX file should be converted into a format that can be accepted by the platforms, containing a suggested price either from the file or from the database.

3. The system should use left over API calls to add/update average prices to the database.
   Since the system has a reserved amount of calls to handle orders daily, it might happen that at the end of the day there are left over API calls. In this case, the system should use them to update average prices in the database instead of wasting them.

4. The system should give error and warning notifications to the user.
   Sometimes action can fail for different reasons. In case an error occurs, a small description of what happened will be made available for the user to read and also users will receive a notification that something did not finish properly.

5. The system should automatically add complete (containing all necessary data) batches and changes in inventory to the platforms if the option no approval is selected.
   There are going to be two options that can be selected in the settings menu, approval or no approval. If the second one is selected, then whenever all the necessary information for items in a batch is filled in or there are changes in the inventory, the system will make the API calls to put everything on the platforms. If the first option is selected, then someone else has to check the data before committing it to the platforms.

6. The system should have an optimized algorithm to reduce API calls.
   Limited API calls are the biggest constraint of our project, thus making an efficient algorithm to reduce them will greatly increase the effectiveness of the general system and will allow more changes to be made daily.

2.5.3 Could have

1. The system could support more platforms, for example Bol.com and the Unbrickable website.
   In the future, the system might be extended to include other platforms than BrickLink and BrickOwl, so making it easily extendable will be helpful in the future.

2.5.4 Won't have

1. The system will not handle order picking.
   Order picking is handled by other parts of the BrickConnect system and it is not necessary to change it.

2. The system will not handle financial information.
   The system aims to synchronize stock across all platforms, it does not look to compute anything related to the total price of sales or items.

3. The system will not include mobile support
   For the time being, the system will only work on computers.

Chapter 3: Organisation

In this chapter, details on how we organize ourselves in order to work efficiently and deliver the best possible final product are discussed. The goals of the project are divided into smaller objectives as a result of working in an Agile environment. This strategy facilitates concurrent development and testing of the product and involves constant communication with stakeholders and supervisors to ensure the quality of the product.

3.1 Scrum

Development work in Scrum is done in short iterations, called sprints. There are 4 sprints of 2 weeks each. For each sprint, a list of features is selected for implementation and the ultimate goal is to deliver a shippable product at the end of each phase.

3.1.1 Sprint Planning

At the beginning of each sprint, we discuss which functionalities are the most critical to implement, divide them into smaller tasks and assign group members for each one of them.

3.1.2 Sprint Review

At the end of each sprint, we make a small presentation to discuss our progress with the company, leaving room for feedback along the way.

3.1.3 Sprint Retrospective

After the presentation, we discuss as a team what went well and what did not work out for the previous sprint and adjust accordingly. Also, we try to point out strengths and weaknesses and give suggestions for improvement for each team member.

3.1.4 Daily Stand-up

Each morning at ten o'clock, the team meets for a small recap of what has been accomplished so far by every member. The general purpose of this meeting is to make sure everyone is on the right track and to help anyone who got stuck.

3.2 Meetings

Along the module, we meet with Unbrickable, our UT supervisor and other students working on the project.

3.2.1 Unbrickable

We schedule a meeting every week with the supervisors from Unbrickable to make sure that all the requirements have been appropriately approached and to discuss any question that may arise during the development.

3.2.2 Supervisor

Also, we keep close touch with our UT supervisor Faiza since she can give us guidance to keep the progress going, help extinguish any conflicts within the team and give us valuable opinion into the design choices we have to make.

3.2.3 Other Students

Collaboration with the other student groups working on the project is crucial to ensure the quality of the final product since it has to be compatible with all the other systems. We have set up a Discord channel for this purpose, and we meet whenever necessary.

Chapter 4: Planning

The key behind every successful project is proper planning and setting achievable intermediate goals. The following section will provide an overview of the main stages of development, the time allocated for each phase as well as which requirements were targeted for completion along the way.

4.1 Deliverables

Table 4.1 shows an overview of important dates and deliverables during the project.

Week	**Date**	**Description**
Week 1	Feb 4	First client contact
Week 2	Feb 12	Discuss project proposal with client
	Feb 14	Finish project proposal
Week 3	Feb 15	Start sprint 1
	Feb 16	Peer review 1: project proposal and planning
Holiday	Feb 22	-
Week 4	Mar 1	Start sprint 2
Week 5	Mar 8	Peer review 2: requirements and test plan
Week 6	Mar 15	Start sprint 3
Week 7	Mar 24	Peer review 3: design document and first prototype
Week 8	Mar 29	Start sprint 4
Week 9	Apr 5	
Week 10	Apr 13	Final poster presentation

Table 4.1: Overview of deliverables

4.1.1 Sprint 1

Sprint 1 is two weeks long and will focus on the project design, setup and the first set of must-have requirements. Table 4.2 shows the tasks for this sprint.

Number	Reference	Description
1	-	Finish system design
2	-	Finish database model
3	User - Must 1	Set up basic visual interface
4	User - Must 2	Overview of batches
5	User - Must 4	Add batches to platforms
6	User - Must 5	Remove batches from platforms
7	User - Must 6	Multiple simultaneous users
8	System - Must 1	Support BrickLink and BrickOwl platforms
9	System - Must 3	Updating stock across platforms
10	System - Must 6	Updating stock for inventory changes
11	System - Must 8	Brickconnect account authentication
12	System - Must 9	Log platform operations
13	System - Must 11	Not exceed daily API calls
14	System - Must 12	Configurable amount of leftover API calls

Table 4.2: Overview of tasks for sprint 1

4.1.2 Sprint 2

Sprint 2 is two weeks long and will focus on finishing must have requirements and adding should have requirements. Table 4.3 shows the tasks for this sprint.

Number	Reference	Description
1	-	Finish previous tasks if necessary
2	System - Must 2	Local orders
3	System - Must 4	Cancellations
4	System - Must 5	Refunds
5	System - Must 7	Synchronize order changes
6	System - Must 10	Reverse platform operations
7	User - Should 1	Per platform scheduling
8	User - Should 2	Per platform pricing
9	User - Should 3	Updating pricing
10	System - Should 1	Suggest price based on platform prices

Table 4.3: Overview of tasks for sprint 2

4.1.3 Sprint 3

Sprint 3 is two weeks long and will focus on finishing and improving requirements from previous sprints. Optionally, *could* requirements are also implemented in this sprint. Table 5.4 shows the tasks for this sprint.

Number	Reference	Description
1	-	Finish previous tasks if necessary
2	-	Make application production ready
3	System - Should 2	Error and warning notifications
4	System - Should 3	Reduce API call usage
5	System - Could 1	Additional platforms, e.g. Bol.com
6	User - Could 1	Overview of item sale speeds
7	User - Could 2	Guided tutorial

Table 4.4: Overview of tasks for sprint 3

4.1.4 Sprint 4

The last phase of the project is about finishing what was started and fixing bugs. Since there are three teams working on the BrickConnect system, all the parts, including the BrickSyncer, had to be integrated into the final system. Merging different projects into a single operational and final product is not an easy task since conflicts might arise and new problems can occur at any time. Thus, the entire last sprint is focused on testing, debugging and also presenting the final product to the client.

Chapter 5: Global Design

In this chapter the global design of the project will be discussed. Considering functionalities, the project can be divided into three main parts: database, frontend and backend. All of these components have to be interconnected and properly synchronized to achieve a working product.

5.1 Database

The Kulla database was used as a starting point for this project. Certain structures of the database were kept the same, but other structures were completely changed. Working together with the other student groups working on the BrickConnect system, a new design was made. This report will go into more detail for tables related to the Bricksyncer module of the BrickConnect system. The complete database diagram together with a description for every table used by the Bricksyncer can be found in the Appendix

5.1.1 Database queries

The Kulla database was an SQL database so SQL queries were used to communicate with the database. For this project we decided to switch to using an ORM, namely SQLAlchemy. ORMs have the benefit of automatically generating SQL queries for the database requests made in the code. An additional benefit is the ability to automatically generate migrations for database schema changes. These migrations can then be applied to a previous version of the database without having to manually alter the database schema.

As it was new to most of us, some SQL queries were used as well. The routes for API calls using SQL queries start with ‘v1’ and the routes using the SQLAlchemy models start with ‘v2’. The goal was to completely switch to using SQLAlchemy models, but in the end both were used. To still keep everything organized, everything was clearly separated.

5.2 Frontend

5.2.1 Frameworks

The frontend was based on the previous Kulla project. Kulla was made using Vue.js, a front end JavaScript framework, and Materialize, a CSS framework. The decision was made to build upon the existing Kulla frontend and to continue using Vue.js and Materialize to do so.

5.2.2 API

Just like the previous Kulla project, Axios was used to send API requests in order to communicate with the database through the backend. HTTP GET, PUT, POST or DELETE requests can be used for the frontend pages.

5.2.3 Pages

The pages allow the user to upload items to the stores and to manage the items. This process should be user-friendly and easy. More pages were also added to keep track of the actions of the system and to see any problems that can possibly arise. Furthermore, a page was added to manage the stores on the platforms. Below will be a description of each page in more detail.

5.2.3.1 Navigation bar

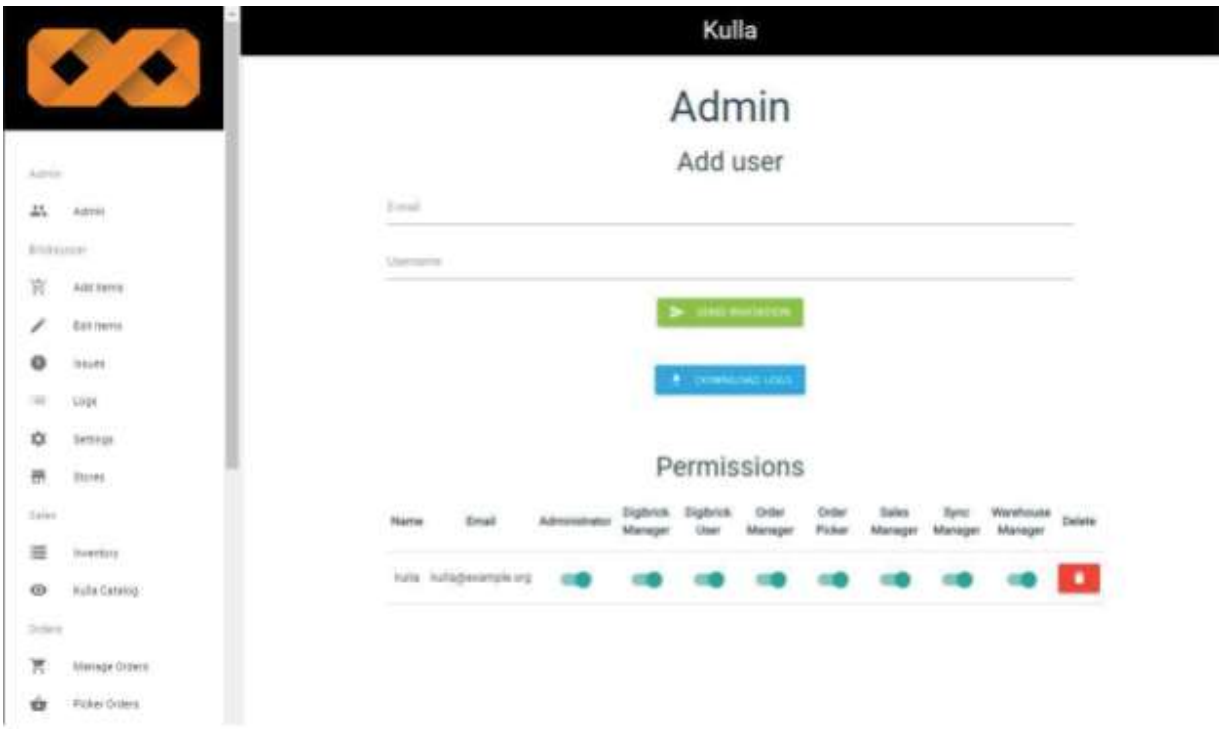

Figure 1. Home page with navigation sidebar

All the pages are easily accessible through the navigation bar on the left side. When a user has the bricksyncer permission and is allowed to manage online stores, the navigation bar will include a section with all the bricksyncer options like seen in figure 1.

5.2.3.2 Batch overview

Figure 2. Batch overview page

It is important for the user to have an overview of every item that still needs to be added to the stores. This page will show the batches, groups of items, that are ready to be added to the stores. It can be found in the navigation bar at add items. It is paginated to keep the overview simple and clear. Each batch will have the option to view the batch in more detail. This will forward the user to the batch page.

Normally the batches will be provided by the other modules within the BrickConnect system. Batches can also be created using a BSX file, explained further in the section on the BSX batch page.

5.2.3.3 Batch

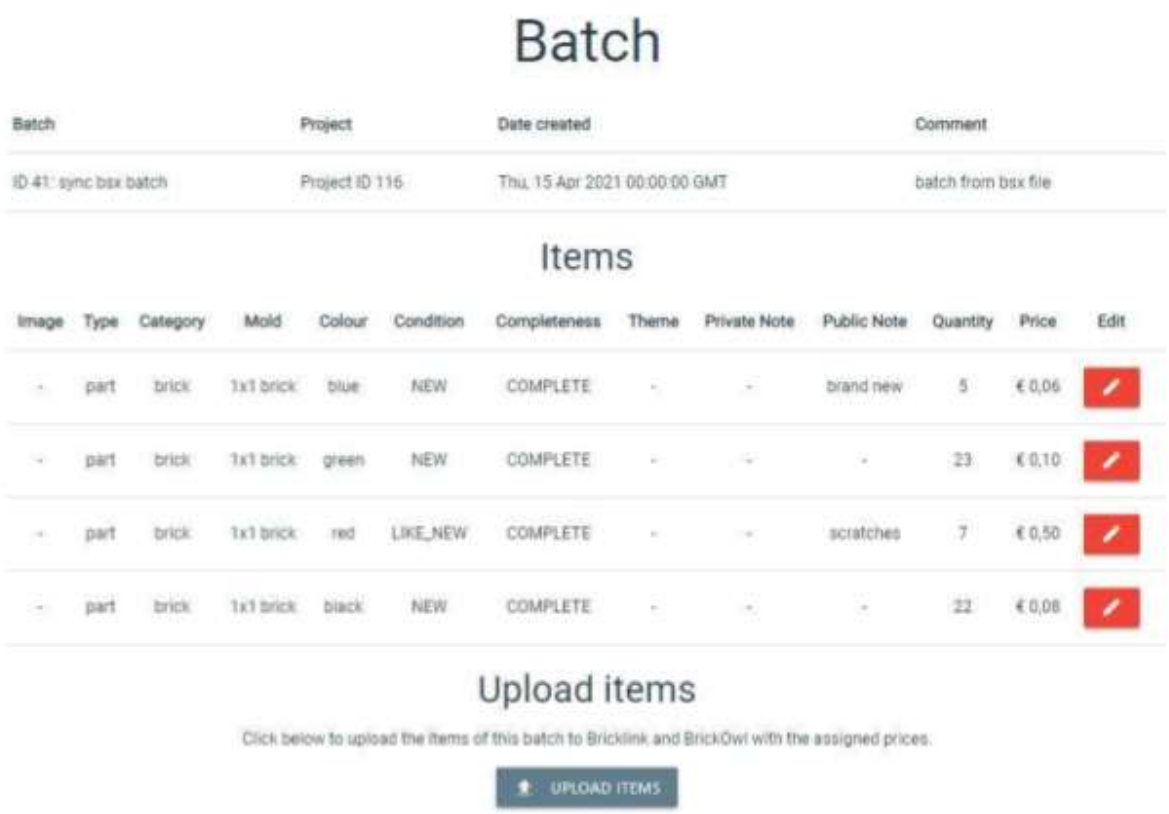

Figure 3. Batch page

To add items to the stores quickly and easily, the user can upload the items of a batch at once. The page shows some information about the batch and the items that belong to the batch. If an item has a suggested price saved in the database, the price of the item will be the suggested price. Otherwise, the price will be set to zero. The user can change the prices of the individual items and upload the batch to the stores. If any of the items have a price of zero, the user will get an error warning as selling items for nothing is not possible. The changes are not saved unless the batch is uploaded. When the user navigates to another page without uploading the batch, a warning pops up asking for confirmation to leave the page.

5.2.3.4 BSX Batch

Figure 4. Import BSX file

Figure 5. Create a BSX batch

To not rely on the other modules of the BrickConnect system for creating batches and entering them into the database, it was important to have a BSX import functionality. It also gives the user the option to create batches of items that are only seen and processed by the Bricksyncer module. The BSX file can be selected by the user, seen in figure 4, after which convert can be pressed and the file is parsed. The items from the file will appear. After the user is done editing the fields of the items, the user can create the batch and will be routed back to the overview. If certain fields of an item are missing, incomplete or incorrect, the user will get an error warning. The fields have to be fixed before the batch can be created.

5.2.3.5 Edit items

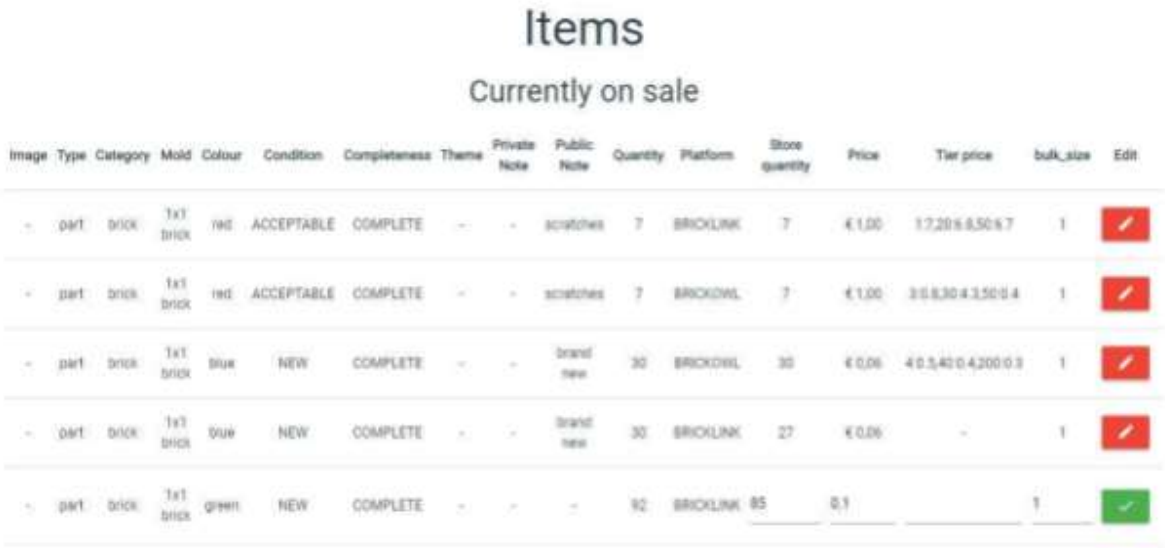

Figure 6. Item overview

Items added to the platforms can be changed on this page found in the edit items option in the navigation bar. It is an overview of all items that are on sale. The items are sorted by the kind of item and the page is paginated to keep it clear and organized. For each item the quantity on the platform and the price can be changed. It also allows for the addition of tier prices and bulk size to the item.

5.2.3.6 Issues

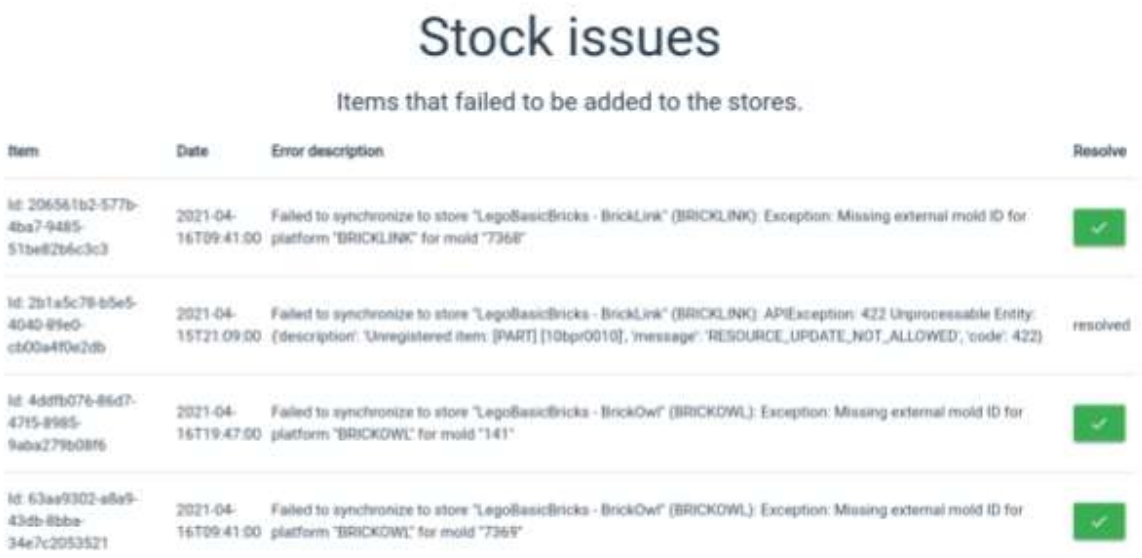

Figure 7. Stock issue overview

This page helps the user to get an overview of items that failed to be added and which problems should still be solved. It can happen that adding an item to the stores was not possible due to missing conversion information or other reasons. When this happens the item will be added to this page with information on the problem and is set to unresolved. The problem has to be solved manually after which the issue can be set to resolved.

5.2.3.7 Logs

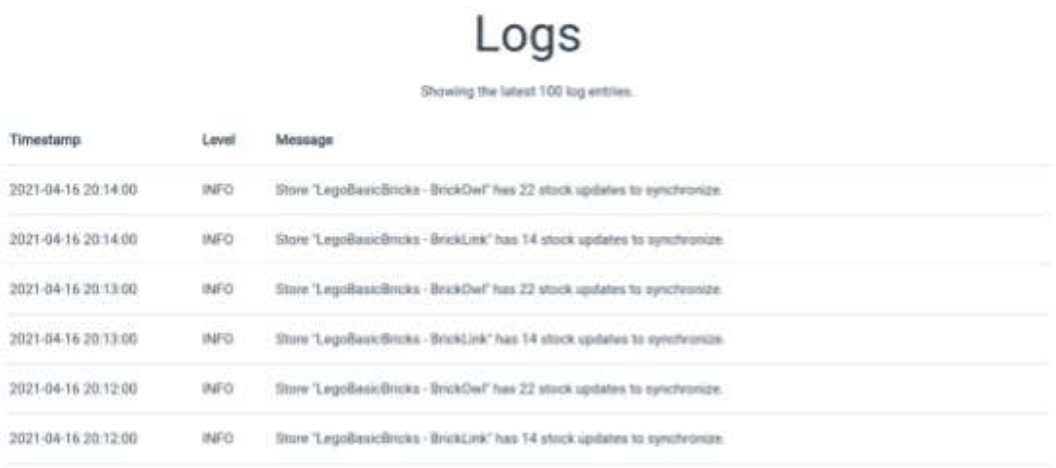

Figure 8. Log of performed actions

To keep records of all actions performed by the Bricksyncer, each action is saved in the database. The last hundred log entries are displayed on the page so that the user can track the actions of the Bricksyncer.

5.2.3.8 Settings

Figure 9. Settings page

To allow the user to configure the API call aspect of the system, a settings page was needed. Each time an item is added, changed or deleted to/from the platform BrickLink or BrickOwl, an API call has to be made to the platform. As each platform has a limited number of calls per day, the user can decide how often calls should be grouped together and how long the delay should be between each call. When more calls are grouped together, fewer calls are necessary, but the delay between calls will be longer. The user can optimize these settings for their specific store.

5.2.3.9 Stores

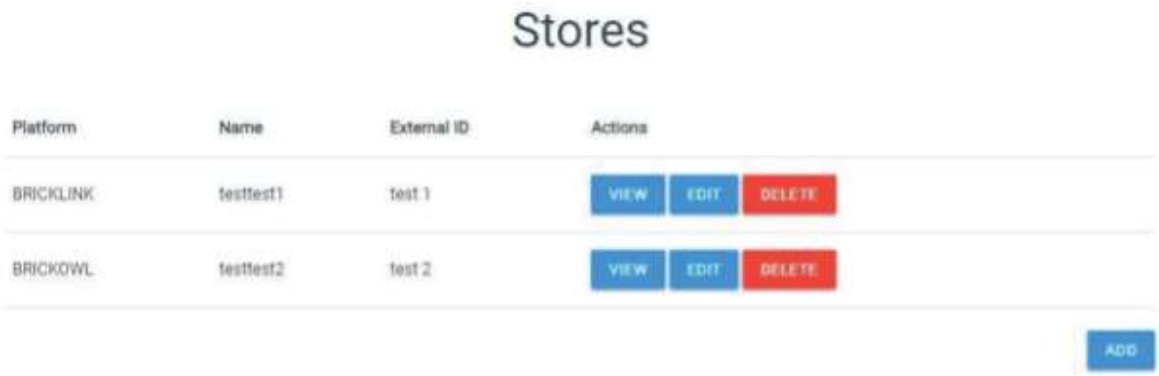

Figure 10. Store overview

Figure 11. Edit the store

To manage the stores on the platforms, an overview of the stores was needed. It allows the user to create new stores on the platforms and to delete existing ones. It also allows the user to see the credential of the stores and to update them.

5.3 Backend

Mainly, the backend can be divided into parts with different functionalities. One of the parts is the backend endpoints used for communication with the web application and the queries to the database. Another part is adding and updating items on the platforms using the APIs of the platforms. Additionally, another part handles notifications of new orders coming in from the platforms.

5.3.1 Languages and environment

To program the backend, the language Python was an obvious choice. Not only was it previously used by the Kulla group, but it also works well for everything that needs to be created. All of us were also familiar with Python.

To allow for an environment that can be set up easily and is maintainable, a Pipenv virtual environment was set up. Configuring PyCharm in combination with the Pipenv environment allows packages to be automatically installed.

A Docker image was created to easily run the backend for production. Docker uses operating system level virtualization to run software in containers. These containers are isolated from each other and the host operating system. A Docker image is a template for a Docker container and contains the definition of all required dependencies. The benefit of Docker images is that they are independent of the host operating system, so it will run anywhere.

5.3.2 Communication platforms

In order to add, remove or update an item from one of the platforms, an API call has to be created. API calls have a specific structure for each platform and need different data on an item than is stored in the Brickconnect database. A few things were necessary to communicate with the platforms.

Firstly, the database contains not only information about the Brickconnect items but also the data of the corresponding item on the platforms. With this data it is possible to convert the item to the version of the platform. For example, in our database the items have a different ID than on the BrickLink platform due to intellectual property laws. To add, change or remove an item on the BrickLink platform, the BrickLink ID is necessary. A negative consequence is that the communication fails if the conversion information is missing in the database.

Additionally, each call needs to be structured differently for each platform. Adding an item to a platform requires generating an API call specifically suited for that platform. The database conversion data is needed to convert the item to the identical version of the platform. The request also uses the keys for the specific platform. The response also needs to be handled differently for each platform.

5.3.3 Notifications orders

Notifications of incoming orders are received through webhooks of the platforms. A webhook can be used to supply information in real-time, meaning that if an order is placed, the platform sends a notification immediately. This means that orders can be handled quickly so that the quantity of the ordered items sold on other platforms as well can be updated in time. It also limits the number of API calls needed, as it is not needed to check if new orders came in. However, a negative consequence is when the backend is not online notifications do not come through.

To process orders that are placed, the order first needs to be retrieved from the platform as the notifications do not include all data that is needed. The items of the order need to be converted back to Brickconnect items in order to assign the items to the order. Then the order and its items are saved in the database. Additionally, the quantity of the items has to be reduced in the database and on the other platforms if they are sold on there as well. The order can then be used by the warehouse to pack the items of the order.

Chapter 7: Testing

To ensure the correctness of the implementation, testing has been done during all phases of the development process whenever possible. The procedures applied, the steps taken and the results will be described in this section.

7.1 Platform API calls Testing

The API calls that are sent out to the platform were tested using unit tests, during these tests an inventory would be created on the platform and item quantities and prices would be changed using the API call methods. In the end, the inventory would be deleted.

7.2 System Testing

To test the collective BrickSyncer module it has been systematically tested, with the help of Unbrickable. These tests were done in the final week as testing the system required multiple tables to be filled correctly, which had to be done either manually or by using the DigiBrick or Warehouse modules. In these tests the following functionalities were tested:

1. Adding Items to stock
2. Changing stock quantities and prices
3. Synchronization with webstores

7.2.1 Adding Items to Stock Testing

There are two ways to add items to stock the first is via batches that have been filled in the warehouse, the second way is by uploading a BSX file, both ways were tested in the following way:
Adding items via a batch that have been filled in the warehouse was tested in the following way:

1. Open the Batch overview page
2. Check whether the Batch that has been filled appears in the list
3. Click on the batch that has been filled
4. Check whether the items and quantities are correct
5. In case no price was set for some items set a price
6. Press the add batch button
7. Go to the stock management page and check whether the items have been added/changed

Adding items via BSX files was tested in the following way:

1. Open the Batch overview page
2. Click on upload a BSX file
3. Check whether the BSX file information is correct
4. Check whether a new Batch appears in the overview
5. Continue with step 3-7 from ‘Adding items via a batch that have been filled in the warehouse’

7.2.2 Changing Stock Quantities and Prices

To be able to test changes to stock, there first have to be items in the stock tables which can either be filled manually or adding items to stock in the way described above. Changing stock information has been tested in the following way:

1. Go to the Stock management page
2. Check whether there are items in stock
3. Make changes to the stock fields
4. Refresh the page
5. Check whether the stock stays updated

7.2.3 Synchronization with webshops

Within the synchronization with the webshops there are two things that needed to be tested: the first is whether inventories were created when new items were added to stock, secondly is whether the inventories got updated when the stock values changed. Both of these were tested by going to the webshops after either adding items to stock or updating stock.

Chapter 8: Evaluation

At the end of the project, a line has been drawn and an evaluation of the current implementation has been made. This chapter will go through the requirements and explain how some requirements have changed, have been partially implemented, or have not been implemented at all. Explanations will be given for every situation. Moreover, it will include a list of personal contributions to the project as well as a brief maintenance and planning overview. Towards the end of the chapter, an overall conclusion of the project results will be given.

8.1 Requirements implementation

8.1.1 Not implemented

All the requirements were prioritized using the MoSCoW method. Must requirements must be completed in order to have a fully functioning product. Almost all of these requirements have been implemented. There was one requirement that was not implemented.

User 8 - As a bricksyncer, I want to be able to revert any operations performed by the system.
During later discussions with the client, we discovered that having a revert button for actions could cause problems. With many stock updates, specifically for one item, it would not be possible to reliably determine whether an action could be reverted without causing the platforms to become out of sync with each other and the database. Therefore, we decided to only log actions and support delta quantities. This way, the user can revert changes manually.

Should requirements are requirements that the product should have. Unfortunately, not all the should requirements were implemented. These are the should requirements that were not implemented with an explanation as to why they were not implemented.

User 1 - As a bricksyncer, I want to schedule when items are added to the platforms.
We simply did not have time to add a scheduling functionality when adding batches to the platforms. However, the stock updates have a “scheduled at” timestamp in the database, so it would be fairly simple for future developers to implement this feature.

User 4 - As a bricksyncer, I want to select a priority option when adding items to the platforms. This will add the items as soon as possible.
We decided against this option as it essentially would not do much. Items that are uploaded will be added to the platform within a configurable time and when the API call limit comes too close, the item is added the next day. A priority setting would not change this process in any way. If the limit has not been reached, the item would be added like normally and if it has, the item would be added the next day.

System 1 - The system should include the average price from the last 6 months as a suggested price.
After the initial proposal, we decided together with the client and the other groups that this should be implemented as part of filling the catalog. Therefore, we did not implement this feature.

System 2 - The system should support exporting items that are being added to the platforms as a BSX file.
We did not have time to implement item exporting. The database should contain all necessary information to allow this feature to be implemented in the future. The main thing required for this feature is the conversion data from the catalog group.

System 3 - The system should use leftover API calls to add/update average prices to the database.
We did not have time to implement this feature. The system does keep track of the amount of API calls that have been used in a day. So implementing this feature in the future could check the amount of leftover API calls at the end of the day and fetch the price updates.

Could have requirements are functionalities that would be nice to have. These functionalities should be added when time is left over. As we did not have enough time to implement all the should requirements, we did not implement many could requirements. There is however one could requirement that we implemented.

User 2 - As a bricksyncer, I want to change the quantity per platform of items that are added to the platforms.
We did implement this requirement as it went together with the requirement of changing the public comment of an item and the price per platform (user should requirement 5 and 6). We made the edit items page where each stock_store_item entry as well as the comments of the item can be edited. This means for each item sold in each store, the quantity on sale, the price, the tier prices, the bulk size, the private comment and the public comment can be changed.

The won't have requirements are not implemented as they were functionalities that were not going to be implemented.

8.1.2 Partially implemented

There are also some requirements that were partially implemented. These are mentioned below.

System 4 - The system should give error and warning notifications to the user.
The system does keep track of errors, warning and information messages. These can all be viewed in the frontend. However, the existing Kulla system did not have notifications and there was not enough time left for our group to implement such a notification system.

System 5 - The system should automatically add complete (containing all necessary data) batches and changes in inventory to the platforms if the option no approval is selected.
It was decided to process inventory changes instantly, as it is important to reduce the quantity quickly if there are actually fewer items available than is registered in the system. For batches this was not automated, as prices can differ and not all the items will have suggested prices available. There would be hardly any batches that could be uploaded automatically unless the database contained almost all suggested prices.

System 6 - The system should have an optimized algorithm to reduce API calls.
The system does attempt to reduce API calls by grouping together stock updates of the item within a certain time period. However, more could be done to reduce API calls. For example, using the bulk API request feature of BrickOwl. This could allow multiple stock updates of different items to be grouped together as well.

8.1.3 Extra

There are also some functionalities that were implemented, that were not part of the requirements but should have been. The store managing page is one of these. We added the stores page to manage stores created on the platforms. During testing with the client, we found that there was no way of creating a store with the keys for authentication, only to manually add them to the database. This was missed during the designing and implementing process. We also added the option to change more fields of an item than just the price per store, quantity per store and public comment. Both BrickOwl and BrickLink support bulk and tier pricing. These options were also added to the system.

8.2 Maintenance

Our group also performed maintenance on the existing Kulla backend code. The existing system consisted of only a few files of several thousands of lines of code. This was not easy to work with, because it was difficult to find the code responsible for a certain part of the system. Our group split up these large files into more manageable smaller files separated by purpose. A clear folder structure was used, which helps separate the code by purpose. Additionally, our group took the lead in merging the individual changes and additions of all groups into one integrated system. After the initial merge, each group submitted pull requests to the repositories. These pull requests were reviewed to ensure the changes did not affect other parts of the integrated system. Our group mainly did these reviews as we had the best overview of the merged system.

8.3 Planning

We did not follow the planning exactly. We planned to make a working version with some features each sprint, starting with the must-have requirements as they had the highest priorities, followed by the should-have and the could-have requirements. There were some aspects that took longer than expected. Communication with the APIs of the platforms took some time. BrickOwl had very little documentation on their API, and we could not really test the API calls. We eventually received credentials of a BrickLink store to test the API calls with, but this was quite a bit further into the project. The database was also changed quite a bit after we planned to have a final database design. There were some features that took longer than expected and some situations that caused small delays. We also created the deployment environment and handled the system merge, so that also took some time that was not planned. Fortunately there were no substantial issues that would have prevented us from finishing the project. Overall, it went well, and we implemented most of the features that the client wanted for the product.

8.4 Contributions

Each team member had a main task from the beginning, which made it easy to work on functionalities as they would not overlap much. It also made it clear what each person had to do. Certain things we worked on together, like presentations and reports, as everyone could write about a different aspect of the system.

Connor Bleumink

- Backend
    - API implementation of platforms
    - API testing of platforms
    - Sync design
- Documentation
    - Documenting BrickOwl API Responses
    - Database table specifications

Daniël Huisman

- Backend
    - Database models
    - Sync design and implementation
- Frontend
    - Stores
- Maintenance
    - Reorganisation and cleanup of Kulla code
    - Systems merge
    - Pull request reviews
- Deployment
    - Docker images
    - Server setup and configuration
    - GitHub Actions pipeline
    - Documentation

Lian van Kesteren

- Frontend
    - Add Items
    - Edit Items
    - Issues
    - Logs
    - Settings
- Backend
    - BSX upload
    - Endpoints required for the frontend
    - Endpoint API documentation
- Supervisor communication

Remus Niculescu

- Poster
- Backend
    - Setting up basic webhook server
    - Conversion from platform orders to system orders.

8.5 Conclusion

Ultimately the project went well even though we did not adhere completely to the planning. We managed to get the product finished and had time to fix issues and bugs that were discovered. The product includes most of the requirements that the client wanted. It resulted in a product that is easy to use for users selling Lego items on the platforms BrickLink and BrickOwl. Items can be added to the stores on the platforms, items that are sold on the platforms can be altered, some settings can be configured, and the user can see all actions performed by the system and manually revert them. Orders placed on the platforms are retrieved and put in the database. From the testing process it was clear that the Bricksyncer worked correctly in combination with the other modules from the Brickconnect system.

Chapter 9: Future improvements

As the project was designed with future development in mind, we will discuss some future implementations that can be added for future developers working on the project.

9.1 Add platform support

To add support for another platform, the interface `kulla.platforms.platform.Platform` can be used. This file will contain all methods that need to be implemented in order to communicate with the platform. The existing BrickOwl and BrickLink implementations can be used as reference.

9.2 Multiple companies

We found some parts of the database design that are only indirectly connected to a specific company. Inventory is one of those. Items are linked to a tray and batch which are created by users. The company of the user making the request can be compared to the company of the user that created the tray and/or batch.
Currently, most of the queries are not made to filter on company, so these queries would have to be altered. The stock tables were created with multiple companies in mind, so a filter on company_id can be added to the stock queries to make them usable for multiple companies.

9.3 Scheduling stock updates

Currently, stock updates only have a configurable delay before being executed, but scheduling stock updates for a specific moment was one of the “could have” requirements of the client. The sync system already partially supports this, so the main addition would be an interface to actually schedule an update. Additionally, the system can currently only schedule quantity updates in advance, but scheduling other fields such as bulk/tier pricing might also be desirable. This can be done by extending the stock_update table with more fields from stock_store_item (e.g. bulk_size) and adding a “last” version of these fields to stock_store_item (e.g. last_bulk_size). This could then behave in the same way as the quantity and last_quantity fields do in the current system.

9.4 Order polling

The current system only uses the webhooks from platforms to receive orders. However, in the event that either the platform or the system malfunctions or is unable to reach each other, orders would not be received. This can be improved by also polling the latest orders using API calls. This should probably only occur once a day to save API calls. In general, the webhook system should work well enough, but in case it fails, there would only be a maximum of a day delay for orders to appear in the system. The basics of this are already present in the system, but there was no time to finish it. For example, the store table already has an orders_synced_at field.

Bibliography

[1] BrickSync. BrickSync - Inventory Synchronization Software. http://www.bricksync.net/. Accessed on 11 February 2021.
[2] Unbrickable. Sociaal - Unbrickable. https://unbrickable.info/sociaal/. Accessed on 11 February 2021.

Appendix

A. Database design diagram

The complete database diagram of the database can be seen in figure 1.

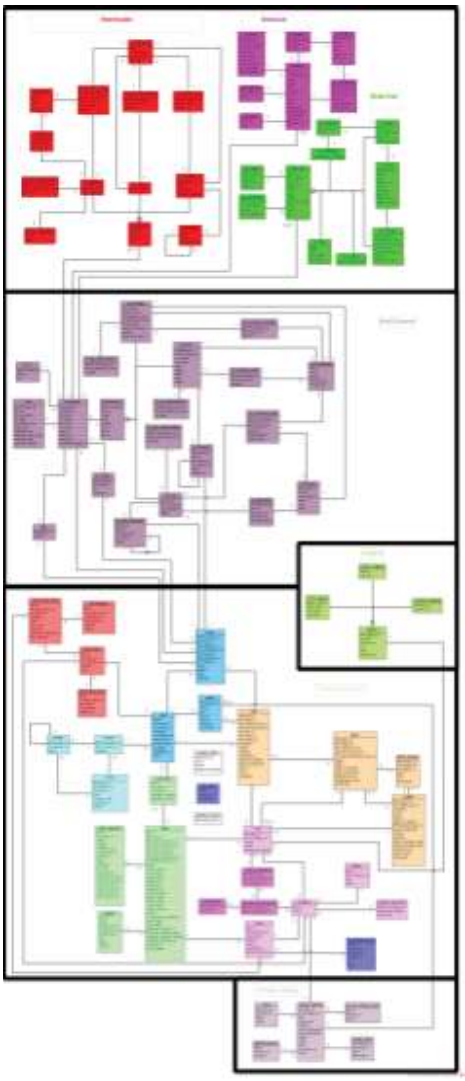

Figure 1. Overview of complete diagram

For the Bricksyncer, only certain parts of the database are relevant. The BrickConnect structure, seen in figure 2, is one of them. It contains general information about the Lego parts like images, categories and themes. It also contains the conversion table which connects Brickconnect items to the identical items on the platforms.

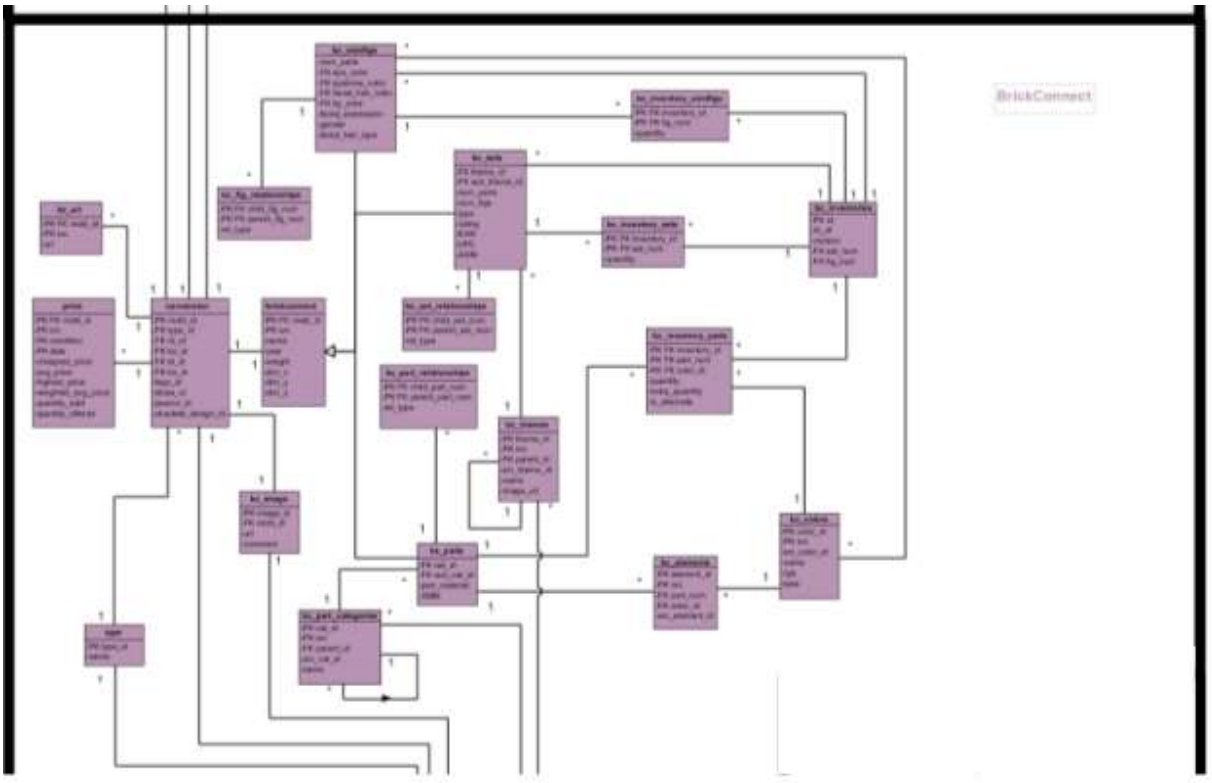

Figure 2. Brickconnect catalog database structure

The other part of the database relevant for the Bricksyncer is the Project structure seen in Figure 3. These tables contain the inventory of the warehouse and the stock sold on the platforms. It also stores the orders placed on the platforms.

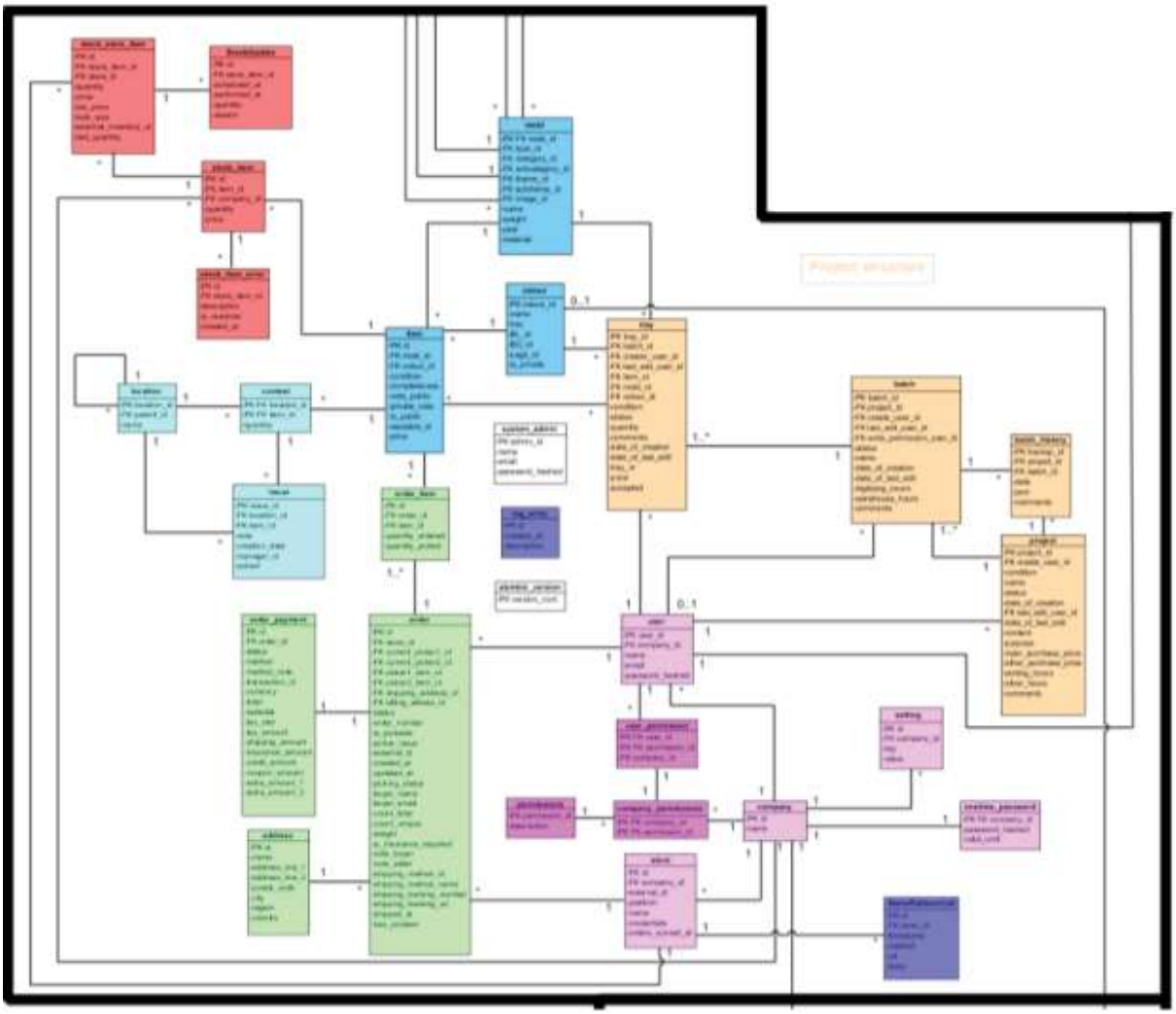

Figure 3. Project database structure

More information on each table can be found in section B of the appendix, database design description.

B. Database design description

The database tables used by the Bricksyncer with a description of every column can be found in the attached document.

C. API Documentation

For the API calls made by the frontend that use the SQL queries and have route v1, documentation can be found in `docs/bricksync/syncer_api.md`. For API calls made to v2 routes using SQLAlchemy models, general documentation can be found in the JSON API specification (https://jsonapi.org/). Additional data model documentation can be found in the `docs/datamodel` folder.