BrickBase

A Lego Catalog System for Syncing and Updating Lego Information for Warehouse Management

Minke Bohlmeijer
Cas ten Have
Koen de Jong
Julian van Santen
Erjan Steenbergen

Supervised by:
Claudenir Morais Fonseca
Tiago Prince Sales

`{m.w.bohlmeijer, c.j.tenhave, k.j.dejong, j.vansanten, e.j.steenbergen}@student.utwente.nl`

`{c.moraisfonseca, t.princesales}@utwente.nl`

UNIVERSITY OF TWENTE.

Electrical Engineering, Mathematics & Computer Science

November 2023

Abstract

In front of you lies the result of the BrickBase project, in which a system for syncing and updating LEGO information was created. It was developed for a client who operates a LEGO reselling foundation. The project was offered in the *Design Project* module, part of the graduation modules of the Bachelor of Technical Computer Science at the University of Twente. The goal of the project was:

"to develop a catalog system that contains all official LEGO items and their metadata, which keeps itself up to date with the release of new items and integrates with the current system"

As a result, a system known as BRICKBASE has been developed. The system was written as an additional component of the existing software. This document highlights the design process, the development results, the testing of the product, a product manual and a recommendation for maintaining the software in the future by other developers.

A group of people working together only becomes a team with INTEGRITY

Contents

Chapter 1

Introduction

In 1932, Ole Kirk Christiansen began making wooden toys in his workshop. This marked the start of *The Lego Group*. In 1949, his company started making early versions of the plastic bricks known today [1]. Fast forward to today, Lego is still a wildly popular toy brand among children, but also among adults. Some adults who used to play with Lego in their childhood had emerging nostalgic feelings towards the creative outlet of building with Lego, which resulted in a group of adult Lego fans, also known as Adult Fans of Lego (AFOLs).

1.1 An introduction to the client

Located in Enschede resides Brickworkz, a non-profit foundation created by Lego community member and AFOL Peter Westenberg. The non-profit is focused on giving adolescents aged 13 to 21 with Attention Deficit Hyperactivity Disorder (ADHD) and Autism Spectrum Disorder (ASD) who struggle to adapt to a standard working environment a chance to work for a real employer. Unbrickable, a subsidiary of Brickworkz, does this by offering the adolescents a job in the Lego warehouse.

Unbrickable uses Lego reselling websites such as BrickOwl and BrickLink to offer their stock, as well as the partner program of Bol.com, known as Bol.com Plaza. They offer their stock on these platforms, which is done with the programs the websites offer. Warehouse management is not done using these platforms and is done using unoptimized[1] and proprietary software packages. The stock sold on BrickOwl does not automatically affect the stock listing on BrickLink.

1.2 The problem

The client wants a smart and optimised way to keep track of his warehouse and web store listings. Overall, the client wants a complete system that can manage everything from the start of the warehouse to the end of the pipeline. This software package is a Warehouse Management System (WMS). Warehouse Management System software packages tend to be large and contain many components. Thus, the client has split the task of developing such a system into multiple components. Previous parts of the system have been implemented by other student groups, into a software package called *BrickConnect* (also called Kulla in the internals of the software package).

1.3 The current solution

Currently, the client uses pre-made systems offered by Bol.com Plaza and other proprietary software made by others. There is no connection between these software packages, thus

[1]Unoptimized for this use case, namely the storing and selling Lego bricks

some administration still has to be done by hand. A new software stack has been partially created by previous student teams and offers partial functionality.

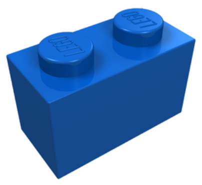
Warehouse management has been implemented by a previous group, as well as a minimal catalog system. This catalog system is insufficient for the use case, as information about Lego bricks and sets is manually added. The amount of information there exists about Lego is vast and changes over time, thus an automated solution has to be developed. The current implementation does not allow this, as the database design in place is unsuited for correctly storing Lego information.

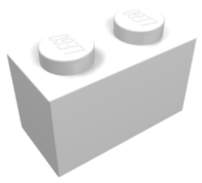
1.4 A proposed solution

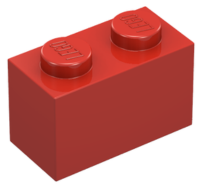
Because of the shortcomings of the old catalog system, a new system has to be designed and implemented. This new system will be inserted into the existing system, to allow other components to integrate with the new catalog database. This system will be used by, among others, adolescents with special needs. These adolescents might interact differently with a system than an average user's standard, meaning that the user interface requires a design analysis.

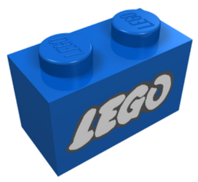
The categorization of Lego products is non-trivial. Previously mentioned databases such as Rebrickable, BrickLink and BrickOwl exist and have certain hierarchies for items and abstractions of them. They do not use the same internal structure, but there is certainly overlap in how items are categorized.

An example that highlights the difficulty in the hierarchy can be given with a single Lego piece, such as a blue one-by-two brick. A screenshot from Rebrickable can be seen in Figure 1.1a. The brick itself was pressed in the Lego factory with a mold: the form of the brick in which it was cast. There exist multiple molds of some bricks, with for example a hollow inside or with a pin. The mold of the one-by-two brick can be seen in Figure 1.1b. Multiple bricks are cast out of a single mold, for example, a red one-by-two brick seen in Figure 1.1c. Molds can also have different prints, for example the brick shown in Figure 1.1d is a mold with a print of the old Lego logo. In the next chapter, we outline the requirements for the software product in more detail and how the program handles this hierarchy.


(a) Blue one-by-two brick


(b) Mold of the one-by-two brick


(c) Red one-by-two brick of the same mold



(d) A one-by-two print variation

Figure 1.1: LDraw models of one-by-two bricks

Chapter 2

Product Requirements

Before the design phase of the project, requirements were identified. These requirements were put into user and system requirements, using the MoSCoW method. The MoSCoW method sorts requirements based on the importance of a feature, where the priority of a feature resides on a sliding scale from must, should, could, and won't, with must as the highest priority. These requirements help determine which features will be in the Minimum Viable Product (MVP), which will be the first functional product.

2.1 User Requirements

The user requirements have three roles: a **user**, an **admin**, and a **developer**. A user is a normal user of the software with basic rights to the system. An admin has all rights to the software. A developer is someone who will work on the *BrickConnect* software. Internally in the software, a user system along with permission settings already exists. For this reason, the BrickBase software component will not include a new user management part, but will instead use the existing system.

Functional user requirements

- As a user, I want to have access to a public catalog within the BrickConnect environment;
- As a user, I want to search for items in the catalog within the BrickConnect environment;
- As a user, I can view different types of catalog items in different views;
- As a user, I can view the information on items within the BrickConnect environment;
- As a user, I can make *edit* suggestions on parts in the catalog;
- As a user, I want to be able to add items from the BrickBase catalog to *batches* in the previously built system;
- As an admin, I want to review item change suggestions;
- As an admin, I want to approve/deny suggestions made by (other) users;
- As an admin, I want to view a changelog with all changes to the BrickBase system;
- As an admin, I want to see the status of an importer;
- As an admin, I want to start importing data with the click of a button.

Non-functional user requirements

- As a user, I want the loading of a catalog page to take no longer than one second;

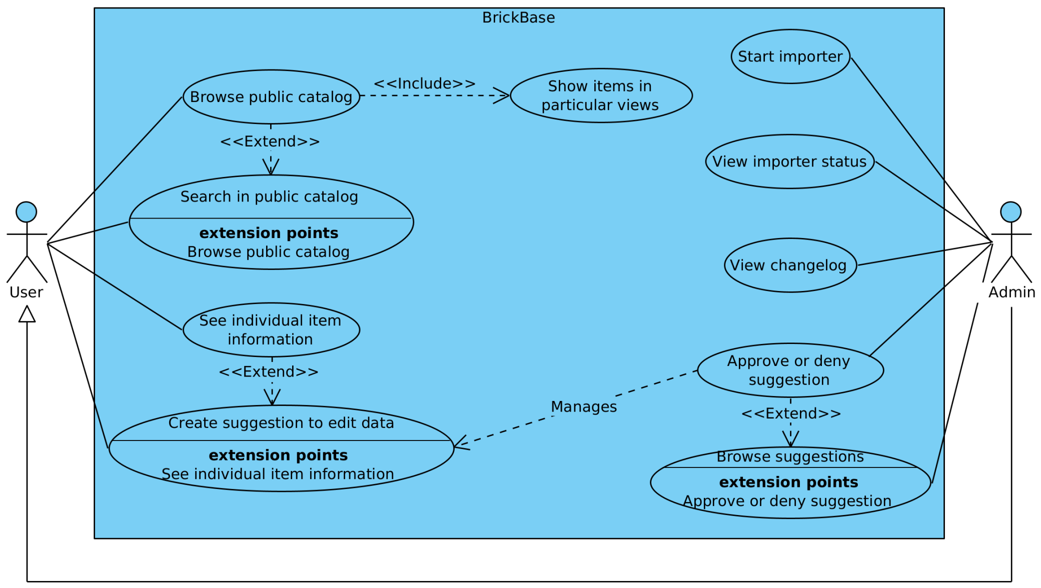


Figure 2.1: Use Case Diagram of functional user requirements

- As a user, I want the application to be user-friendly for
- As an admin, I want to take data from outside sources lawfully;
- As an admin, I want the importer to complete in a maximum time of two hours;
- As a developer, I want a clear overview of all API endpoints;
- As a developer, I want there to be good documentation in the back-end code;
- As a developer, I want the system to integrate with the current existing BrickConnect software.

2.2 System Requirements

Below is a list of system requirements, ordered according to the MoSCoW model. The Minimum Viable Product is seen as sufficient when all *must* requirements have been implemented (all requirements in section 2.2). A full overview of the system requirements ordered by priority can be found in the MoSCoW table, see appendix A.3.

Must

The requirements marked as *must* are needed to create a system with which the client can work. Failing to complete these system requirements means the product will miss key components.

- The system must have a *public* catalog that includes at least all Lego items as seen on Rebrickable;
- The system must be updated using the online sources once per day;

- The system must allow suggestions to be made by users to the *public* catalog;
- The system must allow admins to approve suggestions made by users;
- The system must persist the changes accepted by admins when updating the catalog;
- The system must be able to digest information and automatically process the data from 2 sources: Rebrickable and BrickLink;
- The system must offer a link to BrickOwl for all items.

The *public* catalog will be the main focus of the BrickBase program. Pulling data from the mentioned three external sources and updating the database with this information is the overall goal of the project. User suggestions must be able to override the imported data, in case the data is incorrect or otherwise different from that in the local catalog. Because the *batches* component of BrickConnect will at some point use the BrickBase catalog, the system must be developed as part of the existing BrickConnect software (see section 4.4 on what this component is and how integration is handled).

Should

- The system should integrate with the existing projects/batches component;
- The system should have fast queries on the database;
- The system should have a changelog, which logs the changes made to any items either by users or by the importer;
- The system should be able to offer a link to a suggestion to revert changes shown in the changelog.

As mentioned previously, the BrickBase catalog should be integrated into the existing projects/batches structure. This is not a must for this project, but it is kept in mind while developing BrickBase. Furthermore, speed is a factor. The database needs to be optimized to prevent slow query commands. The intricacies of the Lego ecosystem, along with its relations, show a slippery slope to unoptimized ways of storing data. The existing system is a prime example of this: loading all items from the current catalog takes about thirty seconds. Especially when BrickBase is integrated into different parts of Kulla, computations might grow in complexity. Hence, it is important to keep all code that processes catalog data optimized. Finally, changes made to data should be kept in a changelog. This way, admins of the system can check why data has changed and make changes where necessary.

Could

- The system could filter on parts that have an element with a specific colour;
- The system could support more advanced queries based on brick attributes;
- The system could be able to export catalog data to `.bsx` files for use with the BrickStore offline management system.

A search for specific Parts will be implemented in any case, but an optional filter setting might also be added. The catalog will show (among other things) Parts, but not Elements.

Elements are Parts with an applied colour, so filtering out all Parts that do not have an Element with a certain colour was marked as an optional feature. Finally, the software package BrickStore is used by the client. This software package is an offline LEGO management suite, which can import and export `.bsx` containing warehouse information. The client indicated it would be nice if they could import and export this data format in the BrickBase catalog as well. If there is time, the feature may be implemented.

Won't

- The system won't give an average price of the piece based on sell prices from the sources;
- The system won't allow different sources to be added by the end user;
- The system won't allow the chosen sources to be changed or disabled;
- The system won't have a private catalog for non-LEGO items;
- The system won't allow suggestions to be made by users for the *private* catalog;
- The system wont include AI image search systems, such as *Brickognize*;
- The system won't offer a warehouse management system that keeps track of the number of items in the warehouse;
- The system won't integrate with Product Information Management (PIM) systems;
- The system won't offer a versioning system for information retrieved from the aforementioned online sources.

The initial wish of the client was to have a public and a private catalog. The private catalog would be private per user who would have access to it. The user would be able to add products to their database and optionally ask admins to add certain items to the public catalog. Due to time constraints and a rearrangement of priorities, this idea has been scrapped. The structure of the software might support an implementation of such a feature in the future.

The integration of AI-recognition tools such as *Brickognize* have been discussed. *Brickognize* can recognize LEGO pieces in a picture and give back an identifier for it. After discussing with the client, this feature would make more sense in the warehouse management component. This way, it will be easy to register incoming bricks and connect them to items in the warehouse.

The client has also asked if it would be possible to develop the system with a Product Information Management (PIM) structure. A PIM system can in theory integrate with web stores such as Bol.com and Amazon. This would open up the possibility of easy synchronization of the warehouse to these web stores. After researching the structure of a PIM system, it was decided not to adhere to this structure. The reasoning communicated to the client can be found in Appendix C.

2.3 Priority changes during the design phase

During the design process, the requirements have changed slightly compared to those of the initial proposal. For example, the request for a public/private catalog distinction has been moved from a *must* to a *won't.* The development of a separate private component was determined to be a large task, which would leave no time for other requirements such as good documentation and a solid implementation. Failing to meet that requirement would harm the reusability of the code. The client agreed that it was a better idea to set up the code structure to make it compatible with a private catalog in the future, but not to implement it.

Furthermore, some requirements marked as a *should* have been moved to a *must.* After discussing with the client during the design phase, the importance of integrating with the current system became more apparent. Not building the new component into the existing system would hinder the integrability. Connecting two different platforms would require API connections that are arguably more complex than integrating the catalog system in the current codebase. Extensibility is an important part of the project, and integrating the codebase was determined to be the only viable way to make this happen.

All requirements regarding the suggestion system have also been moved to a *must.* In principle, the catalog will be filled with data imported from outside sources. The client indicated that data imported from these sources might contain incorrect information or might need extra metadata added by users of Kulla. By adding a suggestion system, it becomes possible to add and edit data to the system. It is also important that edited data will not be overwritten by the importing system, thus this is also added as a *must* requirement.

Chapter 3

Global Design

In this chapter, the global technology choices will be explained, along with the choices for certain technologies. In section 3.1, some information is given about how the front end will be built. In section 3.2, the back-end technology will be discussed, following a database technology analysis in section 3.3.

Because the new component must fit into the existing software stack, the options in technology choices are limited. To keep everything compatible, the software must extend the Kulla suite. The choices made by previous development teams had motivations to select the current in-use technologies, but some of these choices are not in line with what the current development team would have chosen. Some of the choices do not match well together, such as the prepared SQL statements and the use of an ORM framework. The current technology stack is explained below, along with any additional technologies that will be added by BrickBase.

3.1 Front end

For the front end, the existing technology will be used as well. The front end is developed with Vue. Vue is a JavaScript front-end framework for building user interfaces. This framework is open-source and well-suited for the use case. Unfortunately, again the multiple iterations of this project have resulted in added dependencies in the code that overlap. The initial design of the application was created with `Materialize-CSS`. This CSS framework does not have built-in Vue components, but it can be used with Vue. Materialize offers pre-designed elements such as buttons, sidebars and more.

Two years ago, during the development of Kulla Warehouse Storage Application (KWaSA), the framework `BootstrapVue` was added. `BootstrapVue` is a Vue framework with pre made components that have integrated CSS styling applied. These components make it simpler to integrate design components with Vue. Because both CSS frameworks offer classes that have overlapping names, components from `BootstrapVue` break down. This means the newly added software will minimize the use of `BootstrapVue` to avoid problems.

3.2 Back end

The first developers that started writing Kulla chose a web server built in Python, using the Flask framework. Flask is a microframework, which means it comes with only the bare minimums for setting up a web server. It is quite useful for simple applications that offer an API but is not well-suited for big applications that make use of other components such as authorization and Object Relational Mapping (ORM) tools. The ORM tool `SQLAlchemy` was added along the development cycle by another team, but this is not natively part of Flask. This is in contrast with a full web framework, such as Django for Python.

3.3 Database

The database software in use at the moment is MySQL. This database application is still seen as one of the most solid Relational Database Management System (RDBMS) applications in use. Before adding anything, the database contained around 90 tables, added by various groups. These tables all contain different entries, not all relevant to our part of BrickConnect. There are some tables meant to implement a catalog system in place but these tables do not form a coherent and compatible structure to store the Lego data that is retrieved from external sources. Furthermore, the code that has been written previously contains next to no documentation, making it hard to determine what the code is supposed to do. To maintain stability in the project and not mess with hidden dependencies, a new database structure will be introduced into the system that is modular and can be connected to existing software by future developers.

Non-relational databases – A consideration

It was considered to use a non-relational database as well. A database platform like *MongoDB*, which stores all data in the form of XML documents, was considered. Using a non-relational database would allow more flexibility in the data stored, as any metadata can be added to any data field without having to create a new table structure. For this to work, however, a new system would have to be introduced. A new database would have to be used and the connection between existing components will be more complex. For this reason, we will reuse the same database schema as the existing system.

Object Relational Mapping

Object Relational Mapping tools offer a developer-friendly way to create database objects and simultaneously reference them from the code. It is a *mapping* from *objects* in the programming language to a *relational* structure in the database. This allows these object models to be referenced from anywhere in the code, without having to write plain SQL statements.

By itself, Flask does not offer any ORM tooling. The second group introduced the ORM framework `SQLAlchemy` to Kulla, which they used to create their models and database tables. Generating the database tables directly from `SQLAlchemy` results in errors, due to overlap with existing tables created with plain SQL. The migrations need to be manually fixed because of the overlap with the first database models. Because of this, the new catalog structure will be using new `SQLAlchemy` data structures that interact minimally with the current incomplete database schema.

Chapter 4

Detailed Design Description

In this chapter, the design choices made during the project are highlighted. The front-end UI design and code structure are explained in section 4.1. The database design is explained in detail, with complementary diagrams to give an overview of the connections between the data. This can be found in section 4.2. Finally, the code structure of the back-end code is explained, along with logic data and sequence flows, this is done in section 4.3.

4.1 Front end layout

The front-end was written in the existing *BrickConnect* application. This meant a substantial part of the front end was already in place. The structure of the front end, as depicted in Figure G.3, shows only the directories and files that were either added or edited.

BrickConnect has a navigation bar on the left, the navigation tool used to access other pages. To accommodate the BrickBase pages `navbar.vue` was updated to include the four main pages and `index.js` was edited to incorporate the BrickBase routes. Additionally, a new component, `Paginate.vue`, was added, which is used to add pagination to tables. This was added on a higher level rather than within `brickbase/components` as pagination is used in other pages we have changed outside of the BrickBase context.

BrickBase

Brickbase added four pages accessible with the side navigation bar, with the catalog being the most pivotal addition. The catalog consists of tables for all categories of items in our database and an overview of all the items. Following the Vue philosophy, these tables are modular components that can be found in `brickbase/table/` which makes them easily adaptable if more categories need to be added.

Within the catalog, the user can click on the info button for an item to open a sidebar with more detailed info. This allows the user to navigate to an info or an edit page. This edit page allows the user to make a suggestion or add a relation for an item. The *More Info* page shows all the related items and for parts, it will show the elements.

The changelog page uses a table to show all the logs and gives the option to go to the changed object. If the log shows an added relation or a changed colour, which does not have info pages, the changes will be shown when the users select the info button. The changelog is only accessible to admins, as the changelog shows information about the changes to *BrickBase* as a whole. It is possible to create a suggestion to revert a change shown on the changelog panel.

The suggestions page has a table which shows suggestions made by users that can be accepted or rejected by admins. Initially, the page will only show the suggestions that still need a decision to be made, but it is possible to view suggestions that have already been accepted or rejected. This made sense design-wise, as the admin would first only want to see suggestions that have not been accepted or rejected yet.

Lastly, there is the *Controllers* page, which uses a table to show all the past data imports. It will show if a controller is busy, has finished or failed and what it currently does.

4.2 Database Design

Due to the complexity of the Lego production hierarchy, many iterations of the database structure were created. Big milestones in the design process are included in this report under Appendix E. During the design phase and even during the start of the development phase the database design kept changing. Many iterations were made in the online design tool Lucidchart, which was used to sketch and review the design of the database. During the development phase, it was discovered that the proposed design seen in Figure E.1 was incompatible (or at the least uncomfortably different) with the imported data received from external sources. The abstractions of *molds* and *parts* are not given clearly by the external tools used to create the imports. Thus, the abstractions of *molds* were dropped from the final product.

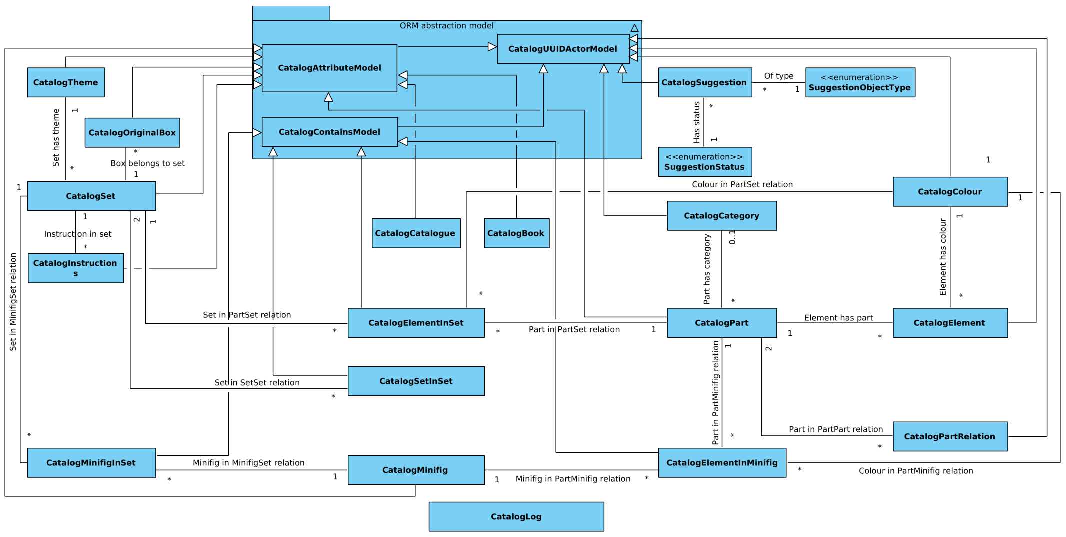


Figure 4.1: Class diagram of the ORM database model

The final design was realised using the SQLAlchemy tool. All classes except the `CatalogLog` class are implementations of one of the three abstract models: `CatalogUUIDActorModel`, `CatalogAttributeModel`, and `CatalogContainsModel`. The `CatalogUUIDActorModel` is the base class, which every other class implements. Both `CatalogAttributeModel` and `CatalogContainsModel` are also implementations of the `CatalogUUIDActorModel` class. This main class contains fields storing editing metadata, such as who created and who edited the data. It also specifies a timestamp field to keep track of when an item has been created and the last time it has been edited. See Figure E.3 under Appendix E for a bigger print.

4.3 Back end structure

As is the case with the front end, the new component is fitted into the existing software stack for the back end as well. Appendix G shows the code structure: all added files are shown in green with a plus, and files that have been changed are shown in yellow with a dot. The back end is divided into three sections: `models`, `queue` and `routes`. Next to that, new database migrations were added to the existing database.

Models

In `models`, all new database tables and relationships for the solution were defined using the ORM framework `SQLAlchemy`. As shown in Figure 4.1 all models inherit from `CatalogUUIDActorModel`, `CatalogContainsModel` and `CatalogAttributeModel`. The three abstract models are defined in `core.py`.

The package `catalog` contains all tables for the different types of Lego items, so these include, among other things parts, sets, minifigures and relations between different items. The package `sources` contains `controller.py`. In this file, there is a table defined with which we can keep track of the controller's status. In `suggestions`, the definition of the table `CatalogSuggestion` is made, including an enum which describes the `SuggestionObjectType` and the `SuggestionStatus`.

Next to all the definitions of tables and their relations, all classes contain a function `parse`. This function is used for parsing database items to a dictionary such that it can be sent used in `routes` for API requests.

Queue

The package `queue` contains the functions that fetch all information from the external sources Rebrickable and BrickLink. Different fetches are all called in `task.py`.

Import flow

Every item is processed in 3 steps as can be seen in Figure F.1. First, all items are fetched from the external source, then they are compared to the items already existing in the database. This comparison returns 3 lists: new database items, updated database items and items that have to be deleted from the database. These lists are then put into the function `DatabaseExport`, which creates, updates or deletes the items that are in the lists.

The comparison function first checks if the new data is found in the old data. The new data that is not found in old data or is different in old data is checked using the Rebrickable ID. Depending on if an item with the ID exists or not, the item is created or updated. There is also a check if the old data is found in the new data. If it does not exist there, it is deleted.

The sequence diagram in Figure F.2 shows a different importing flow: it shows how parts are imported. First, it requests all items from Rebrickable with an API call, and from BrickLink using a file download. The Rebrickable Application Programming Interface (API) also returns an external ID to BrickLink. This external ID combines and adds extra information from BrickLink that can not be found on Rebrickable. Once this information is combined, it is compared to the existing data and then added to the database. What you can also see in Figure F.2 is that the importer also requests the category link between the Rebrickable ID of the category and the ID of the category in the catalog. The `CatalogPart` model includes a foreign key to the category and this needs to be linked correctly in the database.

As explained above, the categories should be added to the database first to correctly fetch parts. Therefore the order of fetching items is important. First, the items that do not need any other items in the database can be fetched, namely categories, colours, themes, books, catalogues and minifigures. Once these are fetched, parts can be fetched using categories and sets can be fetched using themes. With sets done, instructions and original boxes can be fetched. Parts are used for elements in combination with colours and part relations. Last, all inventories are fetched, as these need parts, colours, minifigures and sets.

Executing the importers periodically

The import infrastructure mentioned above runs monthly at 3:00 at midnight. The entire operation takes approximately 1 hour. The import flow is executed on Celery, a tool which runs tasks along the server application and picks up jobs. Celery is often used in sending emails to many users asynchronously, but in this case, it is used to import Lego parts.

Routes

In `routes`, all API routes are defined. The package `catalog` defines all routes for requesting the items in the database. Next to this, there are also some different routes created. These routes can be found in Appendix H.

`Controller.py` contains the route where a controller can be started. A controller updates all data in the database by importing it from external sources. It also contains a route such that the controller can be started using celery. `Suggestion.py` contains all routes that have to do with creating, accepting, and rejecting suggestions. With `log.py`, all logs from the controller can be requested.

4.4 Integration into existing software

Next to creating a catalog, the client also requested if the new catalog could be integrated with the existing warehousing system, such that items of the new catalog could be added to the warehouse. Due to time limitations and the scope of the project being the catalog itself, it was decided to adjust the existing code and not rewrite it completely.

Existing software

The existing software that needed to be changed consisted of a warehouse overview and a project part. The warehouse overview shows everything that is at that point in the warehouse. It uses a tree form to show the exact location of the parts. How a location is shown in the system can be seen in Figure 4.2. The warehouse overview also has an edit function, with which the quantity of an item at a location can be changed, a new item can be added and items can be deleted from the warehouse.

The project part is used to add new bricks to the warehouse. A project exists of multiple batches and in each batch items out of the catalog can be added. These projects can be seen as boxes full of Lego items. These items have to be sorted out before they can be added to the warehouse.

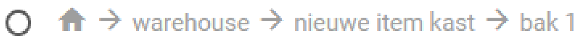


Figure 4.2: Location of an item in Kulla

Once the items have been sorted and added to batches. The items can be filled into the warehouse. While filling, the system shows where the items can be found if there are any already in the warehouse. Otherwise, it will show where in the warehouse there is space for new items.

Back end

In the back end, there were some changes made to facilitate the new catalog in the existing system. First, some database migrations were made, such as the `UUID`s and the colours of the items that could be added to batches and the warehouse itself. As not only parts, but also, for example, minifigures and sets should be able to be added to the warehouse, it was decided not to create a foreign key to one table, but use the `UUID`s of every item in combination with the colour. If an item has no colour, or is, for example, a set, the colour linked to it is the `[No colour/any colour]`.

To be able to then add catalog items to the warehouse, the existing API routes were adjusted, such that the `UUID` and colour combination could be added to the batches and warehouse.

Next to that, there was also a route created with which you can search for any `UUID` and it will return all information of this item. It is possible to request multiple items at once, as it then returns a list of the items to the front end, no matter what type it is. In the front end, this is used to be able to show the information and picture of an item.

Front end

In the front end, the catalog view was imported into the existing system so that all items in the catalog could be shown and added to the batches and warehouse. Once someone chooses an item and wants to add the item, the front end will send an API request to the back end which includes the `UUID` of the item and the colour of the item.

When someone wants to fill the item to the warehouse or wants to see the item in the warehouse, the front end sends an API request to the back end, with which it requests the information using the `UUID`. Although not all types of items, for example, sets and parts, have not all the same columns, the information that is needed is there to show what item it is, which colour it has and a picture of the item.

Chapter 5

Testing the Product

The aim is to comprehensively cover API calls, database structure, and relations. This project has previously had four design project groups work on it, and currently, there is a severe lack of testing in the back-end and the front-end. The goal is to create a framework that is easy to build upon for possible further projects. This test plan does not cover the other parts of the Kulla software. As well as creating the framework, we planned to put that framework to use and write system tests for our code.

5.1 Test Plan

API tests

Postman is used to keep track of and test all API queries created for BrickBase. Postman can run a test after a response is received and test if the response is as expected. For example, when fetching for the tables, the response should have a *count*. This way, the pagination can be accurate and this can be tested with Postman.Swagger has been used to document most of the API.

System tests

We planned to create a solid framework for testing so that future groups can easily add new tests to their code. Additionally, we wanted to write tests for our back end.

We have created a working framework for testing in the back end, but due to time constraints, it was not possible to create extensive tests for our code. We and our client found that it was more important to finalise the product, instead of creating tests in the last two weeks.

5.2 Test results

Although the initial aim was to implement extensive testing into the project, the final result did not have the number of tests as first planned. A test framework was introduced in the back-end, but not all components were tested with this route. The other tests were conducted as planned.

API tests

API tests are conducted on all endpoints mentioned in the Swagger documentation. This is done using Postman, a program for creating and testing API requests. The program can load data and run tests on the received payload. Below, a test for the route `/v3/brickbase/parts` is supplied. The test requests all items from the database and first checks if the response body has HTTP code `200` and has a JavaScript Object Notation (JSON) body.

```
var data = pm.response.json().results
const count = pm.response.json().count

pm.test("Response must be valid and have a json body", () => {
     pm.response.to.be.ok
     pm.response.to.be.withBody
     pm.response.to.be.json
})

pm.test("Response should have 'results' and array size of 'results'
    should be equal to 'count', as all items are requested from the
    database", () => {
    pm.response.to.not.be.error
    pm.response.to.have.jsonBody("results")
    pm.expect(data.length).to.equal(count)
})
```

Listing 5.1: Example test for the `/v3/brickbase/parts` route

Executing this test on the staging version of BrickConnect results in the following test results:

- PASS Response must be valid and have a JSON body
- PASS Response should have 'results' and the array size of 'results' should be equal to 'count', as all items are requested from the database

Chapter 6

A Dive into the Source Code

Building a system for warehouse management is complicated. Such a system needs many modules and components that should form a whole. The existing Kulla system has a big and complex codebase, with some files that have grown to thousands of lines of code. Functions are not documented at all and some have dangling comments that do not explain what is going on, but instead are only used to disable lines of code. BrickBase aims to step away from this paradigm, as it is unmaintainable for large projects. It does this by providing clear documentation for each function and class in the back end and the front end.

6.1 A primer on code quality

One of the main requirements of the BrickBase system is good documentation. The codebase of the previous system is not well-documented and is difficult to navigate. To provide a solid codebase going forward, BrickBase will use strict coding and documentation structures that provide documentation of each function and do not include comments that do not provide more context.

6.2 Back end code

```python
def request(self, url: str) -> dict:
    """
    Request data from Rebrickable API
    :return: the response from the GET request as Python JSON
    :raises RebrickableException: If a connection to the Rebrickable API can not be made
    """                          All functions have reStructuredText formatted Docstrings
    try:
        request = requests.get(
            self.BASE_URL + url,
            headers={'Authorization':'key  '+self.API_KEY}
        )
        return request.json()
    except Exception as e:
        raise RebrickableException('Could not connect to Rebrickable API') from e            Custom exceptions are used to indicate clear error messages
```

The code snippet pasted above shows a function taken from the back end of BrickBase. The function `request(self, url: str) -> dict` is part of a class that runs an HTTP request to BrickOwl. Python does not have a standard way of writing documentation in its Docstrings, unlike other documentation systems such as JavaDoc. There are propositions, such as PEP 287 that propose the use of formatting Docstrings in reStructuredText format. This format of formatting text is somewhat similar to Markdown and is supported by the editor *PyCharm*, which is the code editor suggested by the documentation written by previous developer groups. Thus, the codebase for the back end of BrickBase uses reStructuredText inside the Docstrings.

6.3 Back end API endpoints

The API endpoints for all BrickBase components exposed by the back-end are defined in `kulla/routes/brickbase`. All routes are defined using the Flask router and reuse the permissions as implemented by a previous group. An example of a route can be seen below, it contains the route for an administrator to clear all logs.

```python
@app.route('/v3/brickbase/logs/clear/', methods=['POST'])
@permission_required(audience=AUTHORIZATION_AUDIENCE, \
    allowed_permissions=[7])
def clear_logs(user_permissions: UserPermissions):
    local_clear_log()
    return {
        'status': 'ok'
    }
```

The use of Flask allows concise function bodies with little overhead. The routes directly interact with the ORM model connected to the database, allowing for a nice development experience and again concise code.

6.4 Front end code

The front end is written in Vue version 2, an open-source component-based framework for building web user interfaces. It allows developers to write reusable components that can execute JavaScript to get user data from sources such as an application back end. The ideology of Vue (as with many front-end user interface libraries) is to write smaller components that can be combined to create one big single-page application.

Unfortunately, the previous developers of the project did not pick up on this ideology. The existing code base contains a couple of components that have all parts of the existing routes in one template.

Despite the previous developers not following this philosophy, we have attempted to do so for BrickBase. Within the `brickbase/` directory there is a directory for all the tables and the table components. All the tables make use of the same basic `StandardTable.vue` component, which results in better coherency between the pages and less duplicate code. There are also components specific

Another important component we have used is `Images.vue`, which takes a URL as a prop and a width and shows a square image using the URL. This is used for all components which need images and if a URL does not exist it will replace it with a “no image available” picture.

Chapter 7

Project Evaluation

7.1 Teamwork evaluation

Team organisation

At the start of our project we decided to work with a Scrum-inspired approach by breaking down the project into sprints and using Trello to divide and manage tasks. Initially, in the first few weeks, we mostly worked together on the database design and the client proposal. However, as we moved into the implementation phase we divided into two groups, one team specialising in the back-end and the other in the front-end.

The front-end team comprised Erjan, Julian, and Minke, while the back-end team consisted of Cas and Koen. Despite the division into separate teams, discussion persisted between the teams on what was possible to fetch from the back-end and the design of the front-end. An example of this collaboration is best illustrated by the implementation of the batches, which were made by a previous design group and still used the old catalog. Cas and Erjan worked together closely to fix this issue, where Cas improved the back-end and Erjan worked on the front-end. In general, there was a lot of collaboration between all members of the group.

In addition to our task division for the implementation, we also assigned team members to communication roles. Julian was the designated contact person for the client and Erjan was the contact person for the supervisors. All communication was promptly shared with the other team members in our WhatsApp group. And we did not just communicate well online, we also met and worked physically almost every day.

Finally, this report was written by work delivered by all group members. Each group member worked on the parts they saw most relevant to their work in the code. The writers of each chapter can be roughly seen in the enumeration below:

1. Introduction: Julian and Minke;
2. Product Requirements: All group members, Use Case Diagram made by Julian;
3. Global Design:
   (a) Front end: Erjan and Minke;
   (b) Back end: Cas and Koen;
   (c) Database: Julian;
4. Detailed Design Description:
   (a) Front end layout: Erjan and Minke, front end code structure by Erjan and Minke;
   (b) Back end structure: Cas and Koen, time sequence diagrams by Cas, back end code structure by Julian and Cas;
   (c) Integration into existing software: Julian and Koen;
   (d) Database Design: Julian and Erjan, ORM database diagram by Julian;

5. Testing the Product:
    - Front end testing: Erjan, Julian, and Minke;
    - Back end testing: Cas and Koen;
6. A Dive into the Source Code:
    (a) A primer on code quality: Julian;
    (b) Back end code: Julian, Cas and Koen;
    (c) API endpoints: Julian, Swagger API documentation by Cas and Koen;
    (d) Front end code: Erjan and Minke;
7. Project Evaluation:
    (a) Teamwork evaluation: All members;
    (b) Project internal evaluation: All members.

Overall the teamwork went smoothly and there was a good division of tasks. The team exhibited a strong sense of integrity, and our communication was consistently effective and efficient.

7.2 Project internal evaluation

Things that went well

In this section, a reflection is offered on all that went well during this project. Fortunately, the number of things that went well far outnumber the issues found during the project.

Creating a fitting database schema

Although we had to use the code stack chosen by previous developers, we were still able to pick up the pace with the stack. The database schema proposed at first was not well suited for the task, but after a couple of weekly meetings, we were able to finalize a good schema. Implementing this schema in code proved to be quite simple because we added a `catalog_` identifier to all table names. This way, we did not run into clashes with the old system.

Writing importers

During the start of the development phase, we expected the importing of data via the Rebrickable API and via the BrickLink file download to be quite complicated. Fortunately, the back-end team was able to implement this functionality incredibly fast. Once the front-end team finished their basic item view, it became apparent that the import of data worked and over one million items were loaded in the database.

Building UI with available resources

Because of the troubles with `MaterializeCSS` and `BootstrapVue`, building the UI was not always trivial. Fortunately, the front-end team found a workflow to quite efficiently build the UI. The table component from `BootstrapVue` worked without modifying

Issues

In this section, we will discuss problems we have encountered during the project and how we dealt with them.

Initial task

After the first meeting with the client, we were somewhat confident in our idea of what the customer wanted. The customer was a bit suspicious after this initial meeting, as were some of the members of the group. After the second meeting, we got a better idea about the demand and wishes of the customer. We spent quite a long time drafting and changing the project requirements, which helped us in the long run. Even during the design phase, some requirements changed after meetings with the client, but this all resulted in a product we are all proud of.

Materialize

The first group to work on the Kulla application only used Vue and `Materialize-CSS`. Later on `BootstrapVue` was also added, which caused overlapping dependencies for the naming of some classes. Especially at the start of our project, this caused a lot of issues, as we could not use `BootstrapVue` like we expected and we had to learn to use Materialize. This issue was a result of having to expand an existing system, which was unavoidable. Despite these circumstances, we still managed to create a front-end that meets our requirements.

BrickLink hack

In the last week of the design project, BrickLink suffered from a hack. Due to this, we were unable to fetch data from BrickLink, which caused our importer to not work properly. This was very inconvenient, as we were busy with making separate import tables for books, instructions, and boxes. Our temporary solution was to create our elements so that we were able to test interactions with the data on the front and back end.

Finding a Supervisor

Despite our best efforts, we had a very difficult time finding a supervisor on time. A lot of emails to various CS professors were sent but we received either a rejection or no response at all. Luckily in week 4, we finally found two professors who were willing to supervise our project. Even though it was a lot to catch up on, the communication went well and they have been very helpful in the remaining weeks.

Working with Legacy Code

One of our objectives of this project was to try to leave the legacy code as much as possible. Due to some requirements of the client, this was not entirely possible. The client wanted to integrate our catalog into some of the other features of the site that was made by previous groups. This was challenging, as some choices made by the previous team were in our eyes either inefficient or tricky to work with (For example editing prepared SQL statements).

We learnt to stay with their design choices and make as less changes as possible to the legacy code. Luckily, all features were successfully implemented without extensively altering the legacy code.

Testing

A significant shortcoming of our project has been the lack of testing. We have taken steps to address this by adding a testing framework to test the ORM in the back end and by testing our API calls using Postman, which is more than done by previous design groups. However, there is still an absence of tests for the front end and parts of the back end. In hindsight, it is clear that we should have put more focus on testing throughout our project, rather than leaving it for the last weeks of the development. While the client did not mind it too much, we would recommend that if any future development happens on this system testing is taken into account.

Appendices

Appendix A

Tables

A.1 Planning

Week	Planning		Deadlines	Peer review
1	🟨	🟨		
2	🟨	🟧	Proposal done	
3	🟧	🟧	Design and test plan done	Project proposal and planning
4	🟦	🟦		
5	🟦	🟦		Test plan
6	🟦	🟦		
7	🟦	🟥		Design report
8	🟦	🟥	Implementation done	
9	🟥	🟥	Presentation	
10	🟩	🟩	Poster presentation	

Legend
🟨 Proposal and introduction
🟧 Design and test plan
🟦 Implementation
🟥 Testing
🟩 Poster

Table A.1: Planning outline with legend

A.2 Risk Analysis Table

Likelihood	**Severity**				
	Negligible (A)	Minor (B)	Moderate (C)	Major (D)	Catastrophic (E)
Highly unlikely (1)	🟩	🟩	🟩	🟩 Group conflict	🟩 Server of database crashes
Implausible (2)	🟩 Not enough sources	🟩 Deprecated sources	🟩	🟨	🟨 Project not completed within time
Possible (3)	🟩 Group member gets ill	🟩	🟨	🟨 Integration problems	🟥
Probable (4)	🟩	🟨	🟨	🟥	🟥
Presumable (5)	🟩 Different structure of sources	🟨	🟥	🟥	🟥

A.3 MoSCoW table

Must	The system must have a *public* catalog that includes at least all Lego items as seen on Rebrickable;
	The system must be updated using the online sources once per day;
	The system must allow suggestions to be made by users to the *public* catalog;
	The system must allow admins to approve suggestions made by users;
	The system must persist the changes accepted by admins when updating the catalog;
	The system must be able to digest information and automatically process the data from 2 sources: Rebrickable and BrickLink;
	The system must offer a link to BrickOwl for all items.
Should	The system should integrate with the existing projects/batches component;
	The system should have fast queries on the database;
	The system should have a changelog, which logs the changes made to any items either by users or by the importer;
	The system should be able to offer a link to a suggestion to revert changes shown in the changelog.
Could	The system could filter on parts that have an element with a specific colour;
	The system could support more advanced queries based on brick attributes;
	The system could be able to export catalog data to `.bsx` files for use with the BrickStore offline management system.
Won't	The system won't give an average price of the piece based on sell prices from the sources;
	The system won't allow different sources to be added by the end user;
	The system won't allow the chosen sources to be changed or disabled;
	The system won't have a *private* catalog for non-Lego items;
	The system won't allow suggestions to be made by users for the *private* catalog;
	The system won't include AI image search systems, such as *Brickognize*;
	The system won't offer a warehouse management system that keeps track of the number of items in the warehouse;
	The system won't integrate with Product Information Management (PIM) systems;
	The system won't offer a versioning system for information retrieved from the aforementioned online sources.

Appendix B

Product Manual

Application Overview

BrickBase is a catalog system for keeping track of Lego. The system is built into the existing web-based platform BrickConnect, which means it can use the same login system and permission management. In Figure B, a brief review of how to log into the BrickConnect environment is given. This is the first step in accessing the BrickBase components. BrickConnect (also known as Kulla) is accessible through brickconnect.nl.

BrickBase has four components, accessible from the selector placed left in the BrickConnect software platform. In this manual, the purpose and the intended usage for each of these components are explained.

In Figure B, the main component of the BrickBase system is explained: the Lego *Catalog*. This component contains all externally imported and internally approved data. The user can view the items in different modes, search through the catalog and look at individual items.

In Figure B, the use of the *Changelog* component is explained, which can be used to track changes made by importers or by other users.

When data is not correct or when data is missing, users can create suggestions for edits. This can be done from the individual to-be-edited product in the catalog to edit information of such an item and from the *Suggestions* page directly to add a new item. In Figure B, more information is given.

Finally, the BrickBase catalog is filled by automatically importing Lego data from two external sources: Rebrickable and BrickLink. The status of the data importers can be found under the section *Controllers*. This is explained in detail in Figure B.

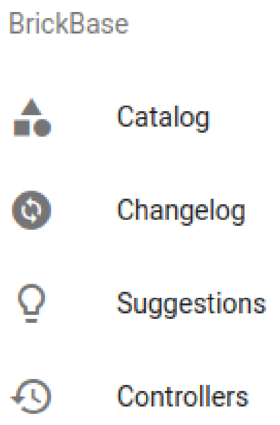


Figure B.1: BrickBase components

Permissions

Catalog

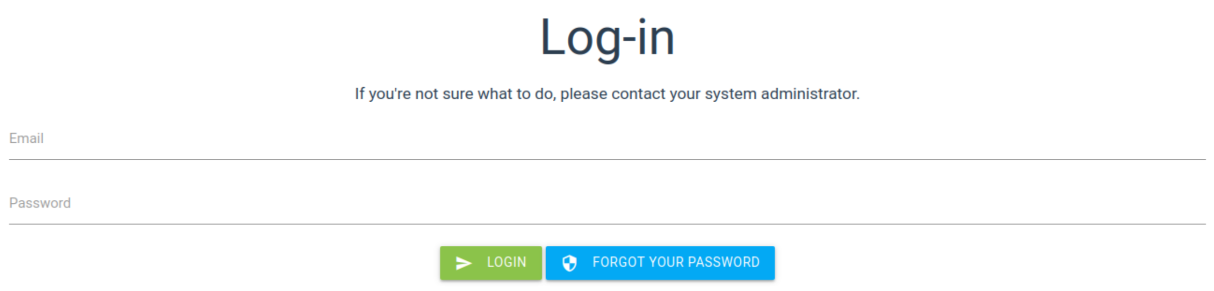
The catalog is viewable by all registered users. They can view all information and are able to create suggestions for the public catalog.

Changelog, Suggestions, Controllers

The other pages are only available for admins and DigiBrick managers.

Login

When BrickConnect is first opened, you will be greeted by a login panel. This panel is shown in Figure B.2. If you forgot your password, use the *FORGOT YOUR PASSWORD* button. A password reset link will be sent to you by email, after which you can request a new password.



Log-in

If you're not sure what to do, please contact your system administrator.

Figure B.2: Login portal

⚠ *Only an admin can create new users and assign permissions.*

Catalog

When you click on the *Catalog* tab under *BrickBase* in the sidebar on the left of the application, you are met with the catalog overview. This screen can be seen in Figure B.3. This will show you five parts, five minifigures and five sets. To search for an object in the database you can use the search bar. You can search based on the name of an item, or on the *IDs* in the external sources Rebrickable, BrickLink or BrickOwl. The overview will show the parts, minifigures and sets based on this search.

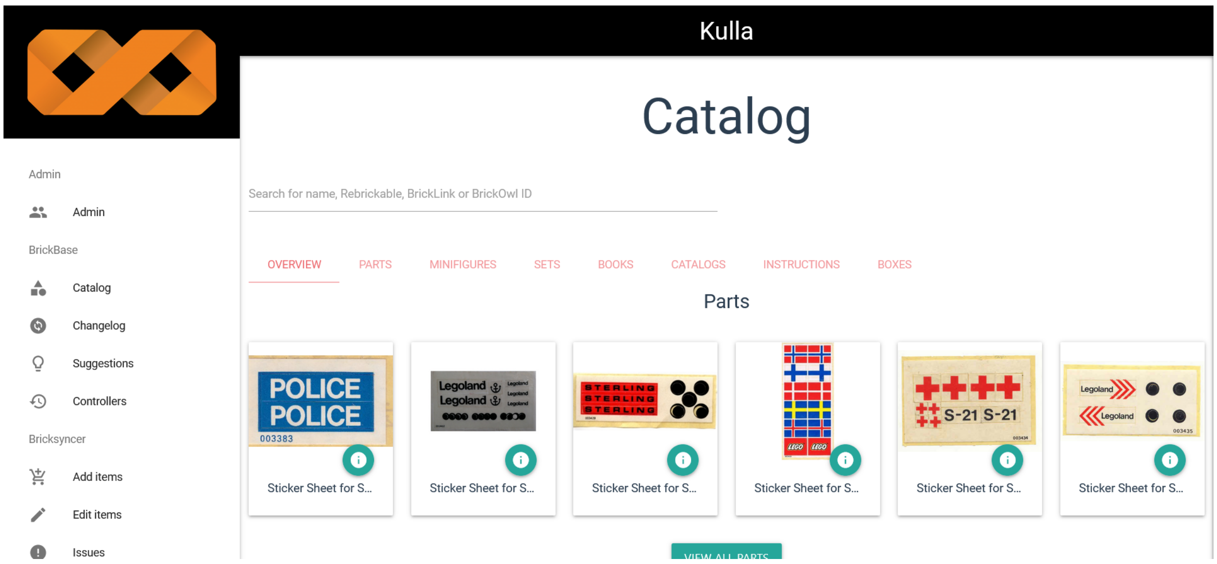


Figure B.3: Overview of the Catalog tab

Searching and Filtering

It is also possible to search on a specific category by clicking on the tabs *PARTS*, *MINIFIGURES*, *SETS*, etc.. The *COLUMNS* button gives the option to see more or fewer columns of information. This way, you can hide information from the table that is not relevant to you when browsing through the items.

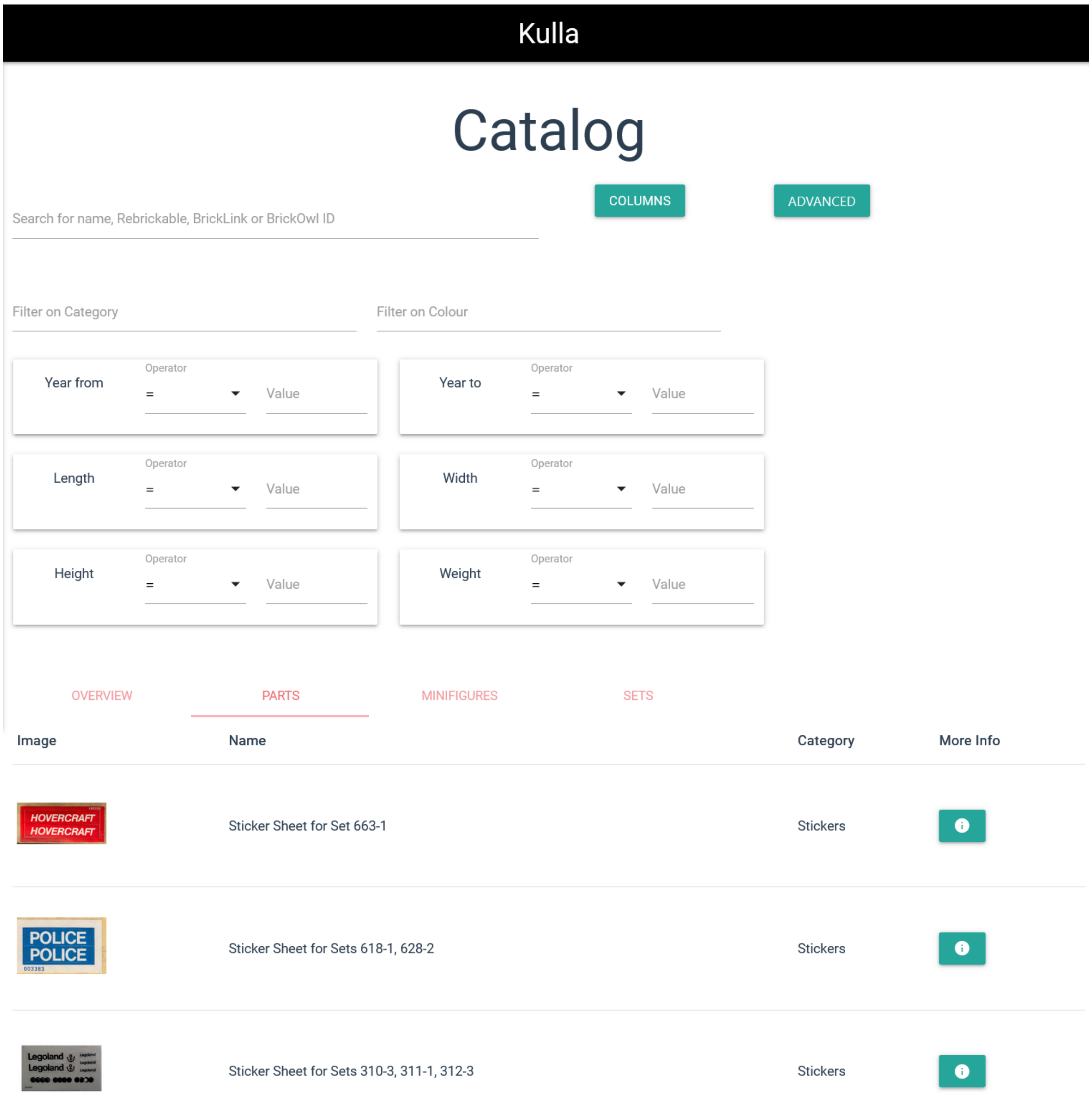
The *ADVANCED* button gives access to more advanced filters[1]. The advanced filters available for parts include the year it was first released, the year it was last released, its weight and the dimensions. In Figure B.4, you can see the available filters after clicking on the *ADVANCED* button. The filters available for sets are the same but without the year it was last released. Since these are all numerical values it is possible to filter based on if the value is less than, more than or equal. For parts it is also possible to filter on categories and colours, and for sets, you can filter based on themes.

More Info

Each object has an info button which will open a sidebar with more info on the item. It will show the release year of an item, the dimensions and the weight if it is available. Most importantly, it shows the *IDs* the object has on the external sources. If you click on one of these *IDs*, the object will be opened on the respective external source in a new tab.

There is also an edit button on the bottom right of this sidebar, which will lead you to the edit page. Anyone can suggest an edit, but this edit can only be approved by someone with the correct permissions. On the left, there is a more info button, which leads to a page which shows info depending on the category of the objects.

[1] ⚠ Advanced filters are only available for sets and parts, since there are limited data on minifigures in the database.



Kulla

Catalog

Search for name, Rebrickable, BrickLink or BrickOwl ID

COLUMNS ADVANCED

Filter on Category

Filter on Colour

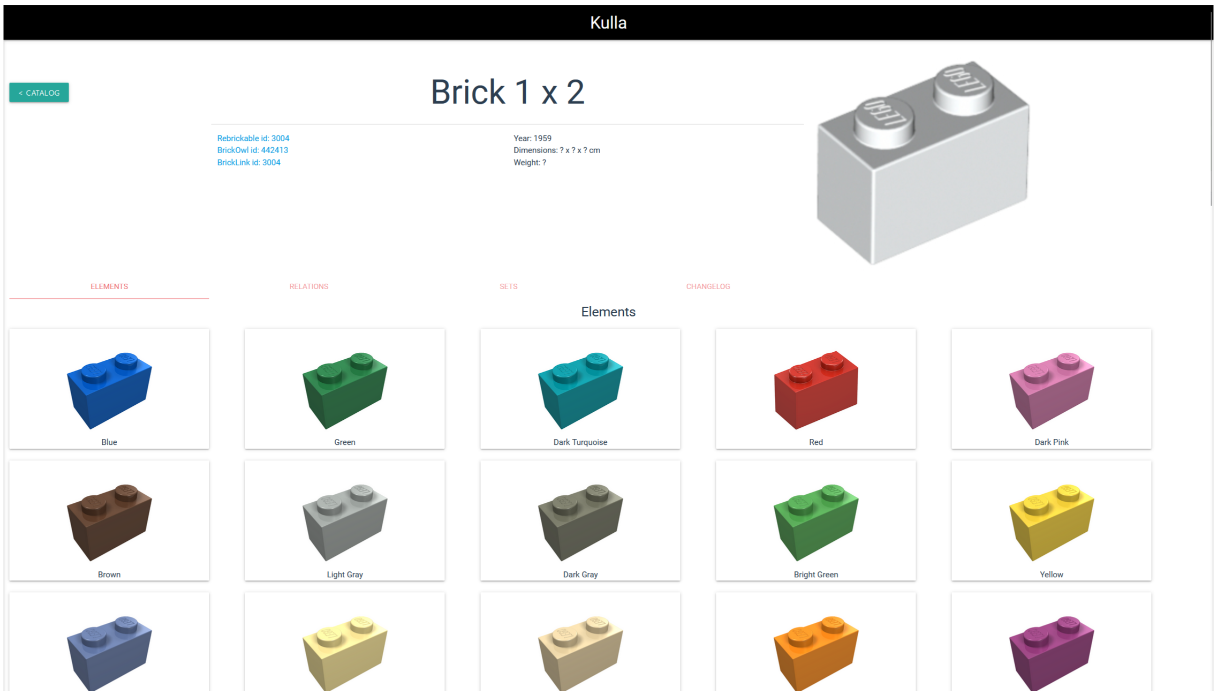
Year from	Operator =	Value	Year to	Operator =	Value
Length	Operator =	Value	Width	Operator =	Value
Height	Operator =	Value	Weight	Operator =	Value

OVERVIEW PARTS MINIFIGURES SETS

Image	Name	Category	More Info
HOVERCRAFT HOVERCRAFT	Sticker Sheet for Set 663-1	Stickers	●
POLICE POLICE	Sticker Sheet for Sets 618-1, 628-2	Stickers	●
Legoland Legoland	Sticker Sheet for Sets 310-3, 311-1, 312-3	Stickers	●

Figure B.4: Extra filter features in the Catalog

For parts, it will show all the possible colours it can have. Each shown part-with-colour combination is an *element*. This can be seen in Figure B.5. It will also show the parts it is related to, like the prints, the pairs, the alternates, the subparts, or the related molds. It will also show a list of all the sets that it appears in, but it is not possible to go directly to the info pages of these sets.


Rebrickable id: 3004
BrickOwl id: 442413
BrickLink id: 3004

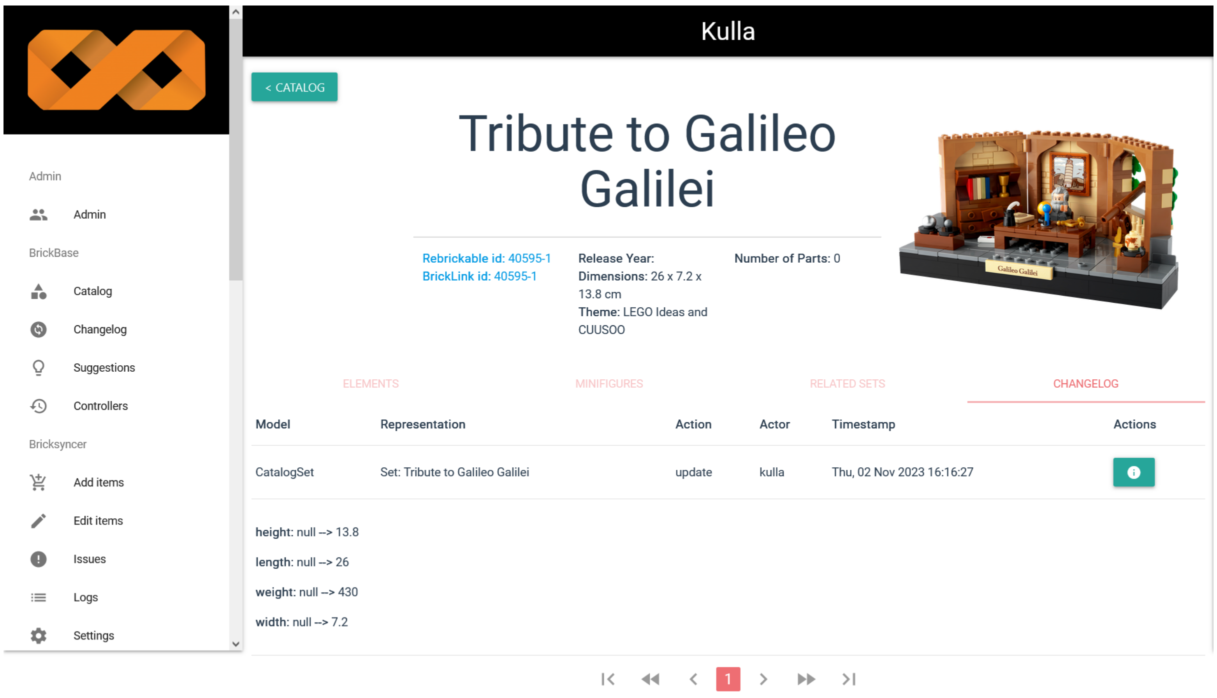
Year: 1959
Dimensions: ? x ? x ? cm
Weight: ?

Figure B.5: More Info panel for `Brick 1 x 2`

For minifigures it will show the elements which will be the items it is made up of. Often this will be legs, a torso, a head and a hair.

Lastly for sets, it will show the inventory of sets (all elements it contains), along with the quantity. If it contains any minifigures, the minifigures tab will show all the minifigures. It is possible to click on the elements or the minifigures and go to the info page for those objects. The instructions and original boxes are also shown, but those do not have an info page so there is no info button.

All info pages show a *CHANGELOG* tab, which shows a table of all the updates made to objects in the previous 24 weeks. It is possible to click on the info button of these changes to see exactly what has changed. For more information on how and what changes are stored, read Figure B.



Tribute to Galileo Galilei

Rebrickable id: 40595-1
BrickLink id: 40595-1

Release Year:
Dimensions: 26 x 7.2 x 13.8 cm
Theme: LEGO Ideas and CUUSOO

Number of Parts: 0

Model	Representation	Action	Actor	Timestamp	Actions
CatalogSet	Set: Tribute to Galileo Galilei	update	kulla	Thu, 02 Nov 2023 16:16:27	

height: null --> 13.8

length: null --> 26

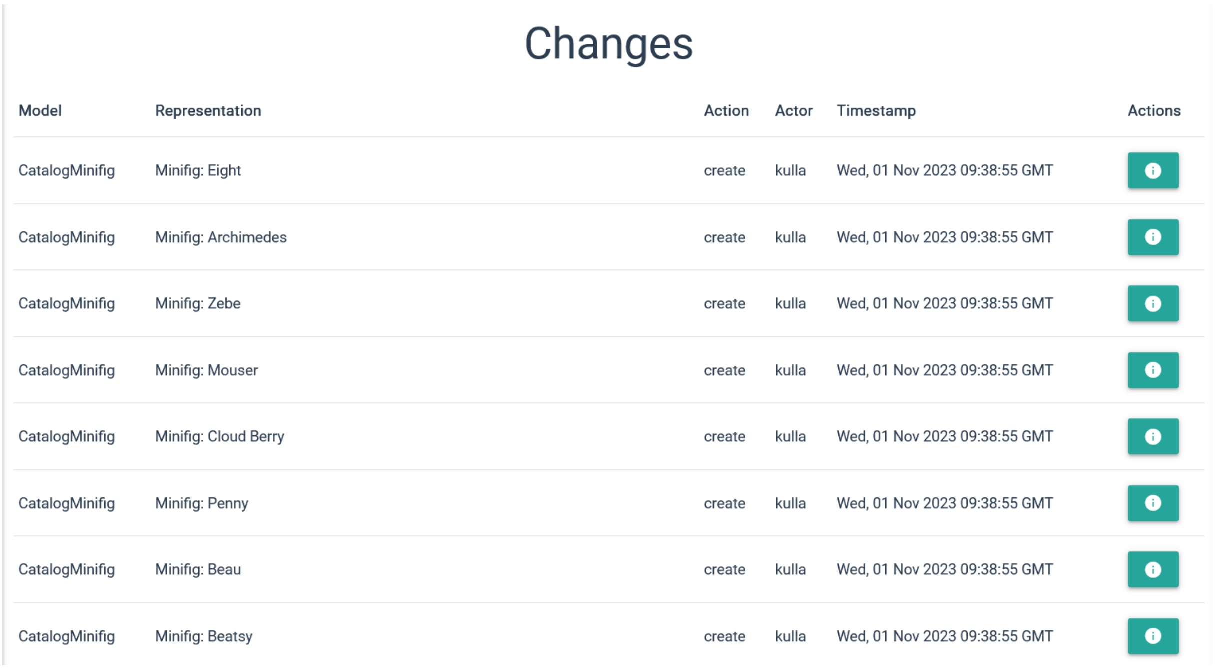
weight: null --> 430

width: null --> 7.2

Figure B.6: Changelog on set info page

Changelog

The *Changelog* shows a table with all objects created or updated in the catalog during the last 6 months (see Figure B.7). The table shows the representation of a log, including the name of an object (or a short description of it), the type of action (creation or change), and a timestamp. The logs that are older than 6 months will be deleted every day at 00:00.



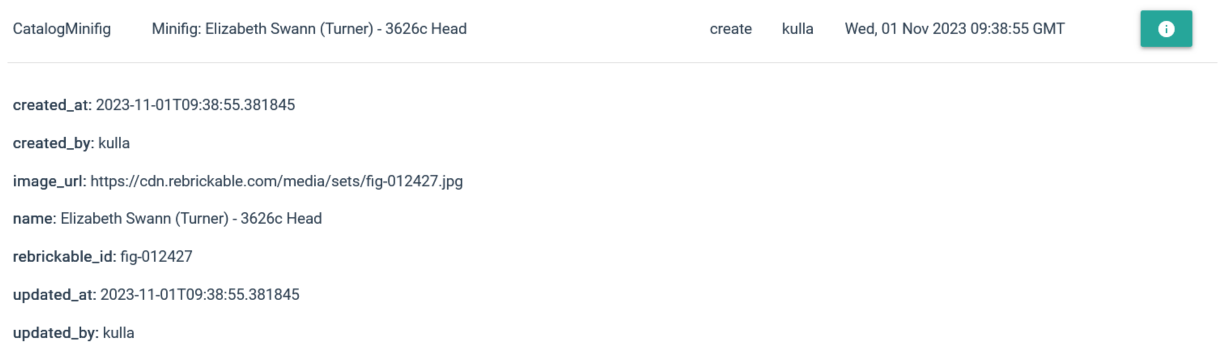
Changes

Model	Representation	Action	Actor	Timestamp	Actions
CatalogMinifig	Minifig: Eight	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Archimedes	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Zebe	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Mouser	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Cloud Berry	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Penny	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Beau	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ
CatalogMinifig	Minifig: Beatsy	create	kulla	Wed, 01 Nov 2023 09:38:55 GMT	ⓘ

Figure B.7: The main Changelog panel

You can click on the *Actions* button per entry to see its specific changes and who made them. The page will show which values have changed, or if it was an update on an item, it will open the info for that item. In the example given in Figure B.8, you can see an addition to the database. It shows the creation of a minifigure called "Elizabeth Swann (Turner) - 3626 Head". The Actor *kulla* signifies that the object was imported from the external sources automatically. The other fields show the following:

- `created_at`: time at which the item was added to the database;
- `created_by`: user that created the original item, or *kulla* if the item was imported;
- `image_url`: link location of the image of the LEGO piece (using either Rebrickable or BrickLink);
- `name`: name of the item that was added;
- `updated_at`: time at which a change was registered in the database;
- `updated_by`: user that suggested the creation/edit, or *kulla* if the item was imported.


CatalogMinifig Minifig: Elizabeth Swann (Turner) - 3626c Head create kulla Wed, 01 Nov 2023 09:38:55 GMT

created_at: 2023-11-01T09:38:55.381845

created_by: kulla

image_url: https://cdn.rebrickable.com/media/sets/fig-012427.jpg

name: Elizabeth Swann (Turner) - 3626c Head

rebrickable_id: fig-012427

updated_at: 2023-11-01T09:38:55.381845

updated_by: kulla

Figure B.8: A changelog entry for a minifigure

Suggestions

The *Suggestions* tab shows all suggestions users have submitted. This is not accessible to all users, only DigiBrick managers and above are allowed to access this tab. These permissions can be given by the admin from the admin panel, see the manual written by the first Kulla development team for instructions on how to do this.

Suggestion Status

A suggestion can have 3 types of statuses: Open, Rejected, and Accepted.

When a suggestion is open, it can be either accepted or rejected by clicking on the icon, where info about the change is then shown and then clicking on accept or reject.

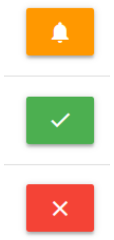
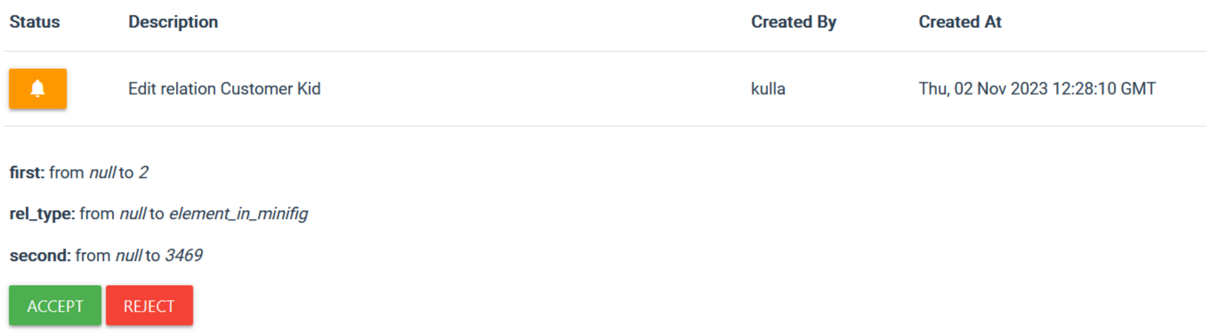


Figure B.9: Status icons from top to bottom: Open, Accepted, Rejected

Change Information

If the user clicks on the status icon, the row collapses into a card that shows all the changes of the suggestion. The cell that has changed is shown in bold, followed by the old value and then the new value. The user can accept and reject the suggestion.



Status	Description	Created By	Created At
🔔	Edit relation Customer Kid	kulla	Thu, 02 Nov 2023 12:28:10 GMT

first: from *null* to *2*

rel_type: from *null* to *element_in_minifig*

second: from *null* to *3469*

Figure B.10: An example of a suggestion from the admin view

Filter on Status

You can filter the status of a suggestion on the top left, just above the table, at the drop-down. The default is set to only show open suggestions. However, it is possible to show multiple statuses when clicking on the respective option. When choosing no status, it will show all suggestions.

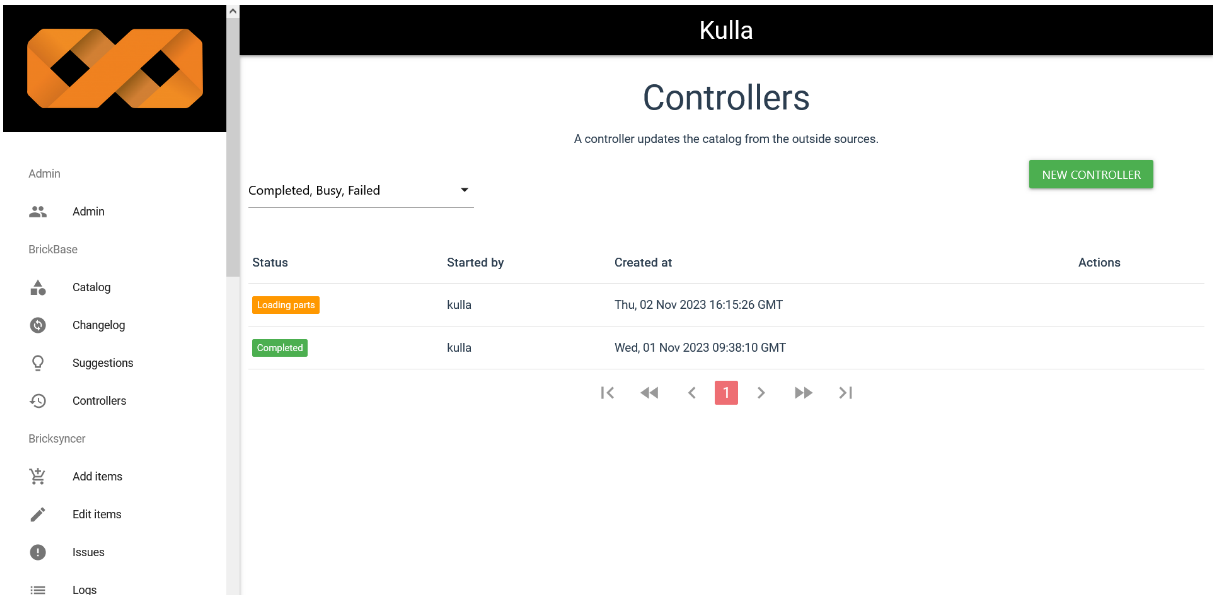
Controllers

The *Controllers* page shows all the controllers currently running, failed or busy.

As mentioned on the page, a controller updates our database from the outside sources, and any updates will be shown in the changelog. A new controller is started automatically the first day of every month at 3:00am, but if need be one can also be started manually by clicking the new controller button. Updating the database takes around one and a half hours and it considerably slows down the other parts of the system, so this is not recommended if you are planning on using the system a lot in the next hour.

A controller can have three statuses:

- Completed;
  - This is when a controller has finished updating the database and nothing went wrong
- Busy;
  - The controller is updating the database currently and it will show the current stage
  - In Figure B.11, the controller started on Thursday is loading the parts
- Failed.
  - This is when a controller could not finish updating the database. The reason for failure will be given



Controllers

A controller updates the catalog from the outside sources.

Status	Started by	Created at	Actions
Loading parts	kulla	Thu, 02 Nov 2023 16:15:26 GMT	
Completed	kulla	Wed, 01 Nov 2023 09:38:10 GMT	

Figure B.11: The main Controllers panel

Appendix C

Explanation of PIM structure for the client

C.1 A Product Information Management System

Before knowing *why* or *why not* to implement a Product Information Management (or PIM for short) in a cataloging system, some background information must be considered.

What is a PIM?

Websites like Bol.com and Amazon allow external stores to set up their store and offer their products on their website. Bol.com does this through *Bol.com Plaza* and Amazon through *Amazon Marketplace*. In theory, anyone can start a store and sell their stuff. This is quite easy to do in practice for small companies that sell a couple of products. Once a store scales up, however, it becomes harder to maintain the complete product catalog by hand. Medium-sized stores might be able to still manage everything by hand, but a store that offers many products in a big warehouse often uses software to keep track of everything inside. Such a system becomes a proper Product Information Management system if there is a translation in place to a common protocol, such that it can manage the publication of items on sites like Bol.com and Amazon. It can automate the addition of new items and process stock figures, meaning less work is needed by the store to keep their warehouse numbers accurate on external stores.

How is it implemented?

In the past, uploading content was done using the FTPS protocol for Bol.com Plaza. A similar technique was used for Amazon Marketplace. Currently, both stores rely on the use of their APIs. Using these APIs, stores can synchronize their stocks. This means that there is no one standard that both Amazon and Bol.com use, although the current infrastructure is somewhat similar.

C.2 Why it will not be implemented for now

The work described above shows some benefits of having a warehouse system set up with a Product Information Management infrastructure. Unfortunately, the implementation is platform-dependent, meaning that different implementations are needed for Amazon/Bol.com/et cetera. This is for us a reason not to implement it into our system. We believe implementing the PIM infrastructure might be its own project that can span the course of about five to ten weeks.

Another reason for us not to integrate it has to do with the scope of our current project. We are primarily focused on implementing the cataloging of Lego bricks. The intention is to index all existing bricks, meaning that only a part of the bricks might be in the warehouse. The bookkeeping of what is in the warehouse happens in a different part of Kulla. If another group will work on the warehouse management part of Kulla in the future, we suggest they pick up the link to the different stores and make it a proper Product Information Management program.

Appendix D

Previous versions of the requirements

D.1 User Requirements

- As a user, I want to have access to a public catalog within the BrickConnect environment;
- As a user, I want to search for items in the catalog within the BrickConnect environment;
- As a user, I can view the information on items within the BrickConnect environment;
- As a user, I want to make edit suggestions on parts in the catalog;
- As an admin, I want to add, edit, and delete parts in the public catalog;
- As an admin, I want to be able to review item change requests and approve/deny them;
- As an admin, I want to lawfully take data from outside sources;
- As an admin, I want to add, edit, and delete custom pieces in the private catalog;
- As an admin, I want to import and export Lego pieces with a `.bsx` file from the catalog;
- As a manager, I want to add and edit parts in the private catalog;
- As a developer, I want there to be good documentation on the system.

Appendix E

ORM Design Iterations

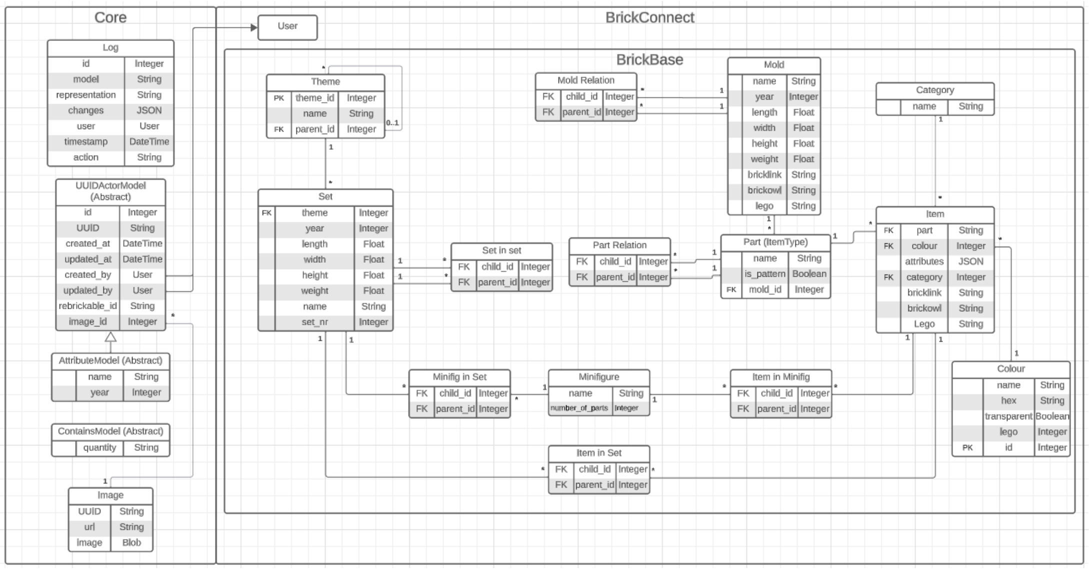


Figure E.1: First iteration of the database diagram

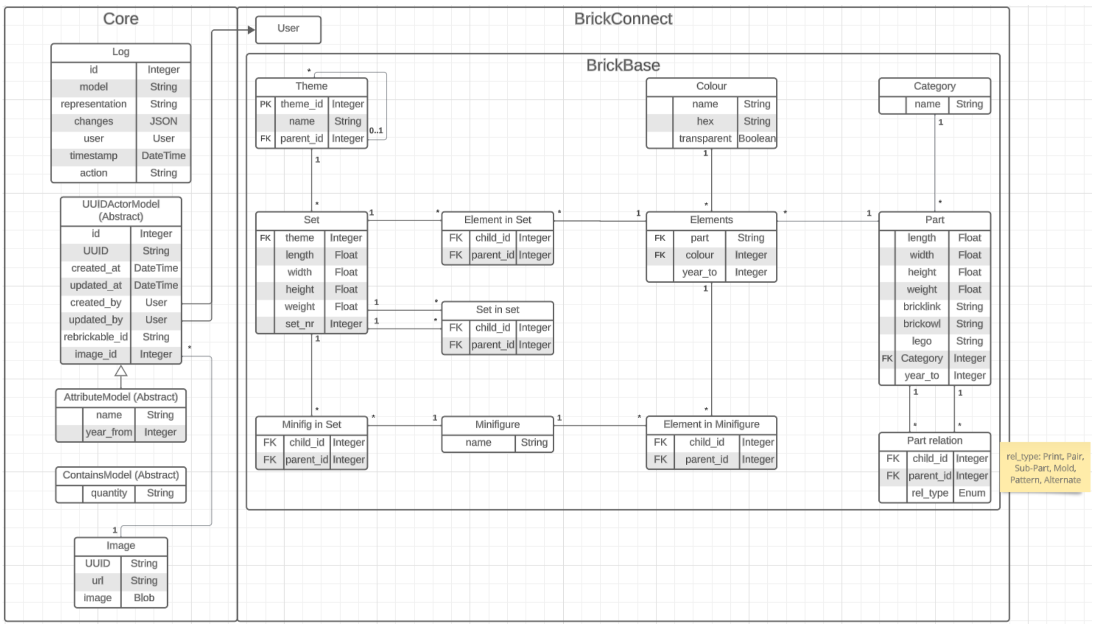


Figure E.2: Second iteration

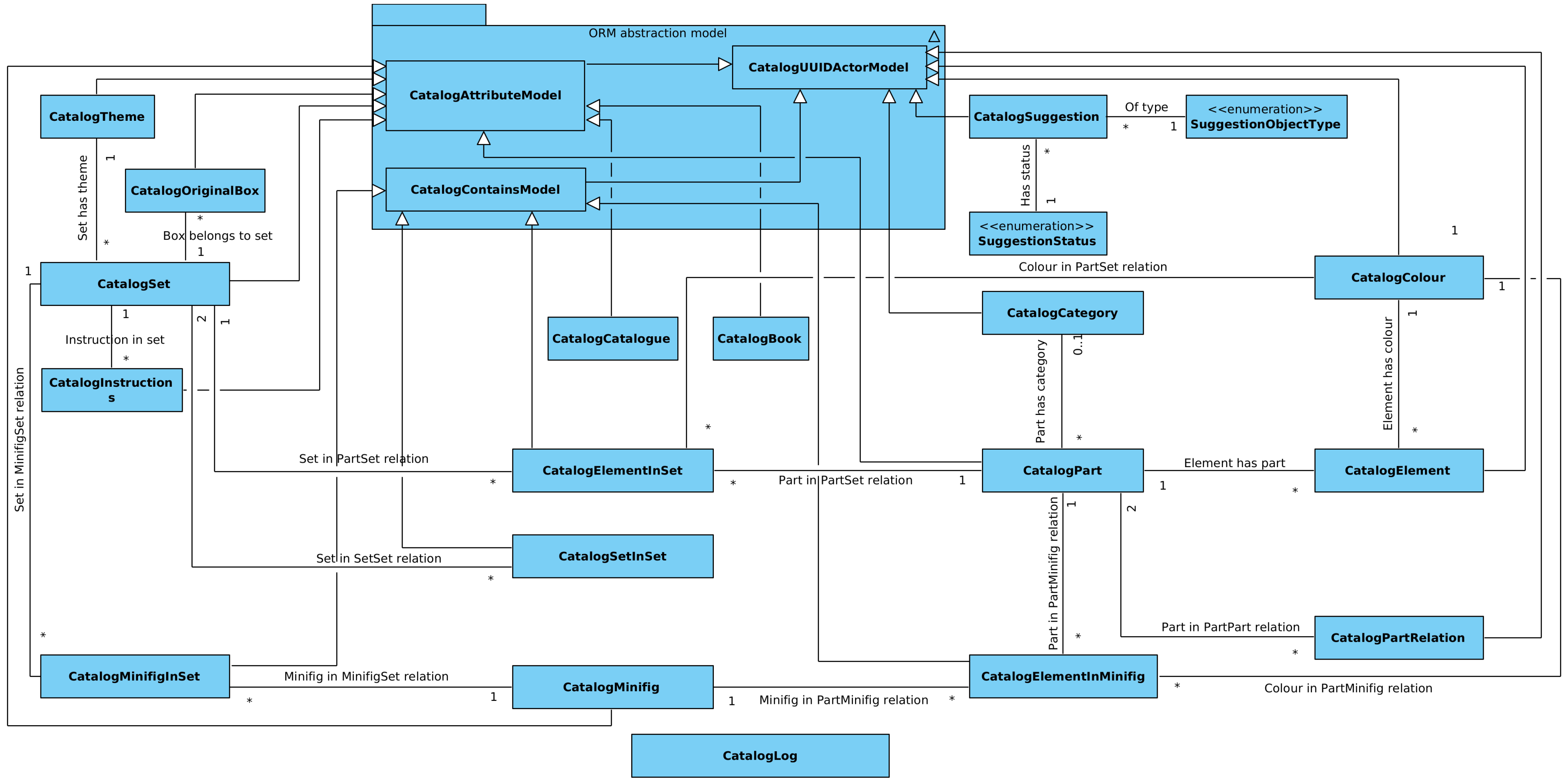


Figure E.3: Final iteration of the diagram, without variables and operations

Appendix F

Time sequence diagrams

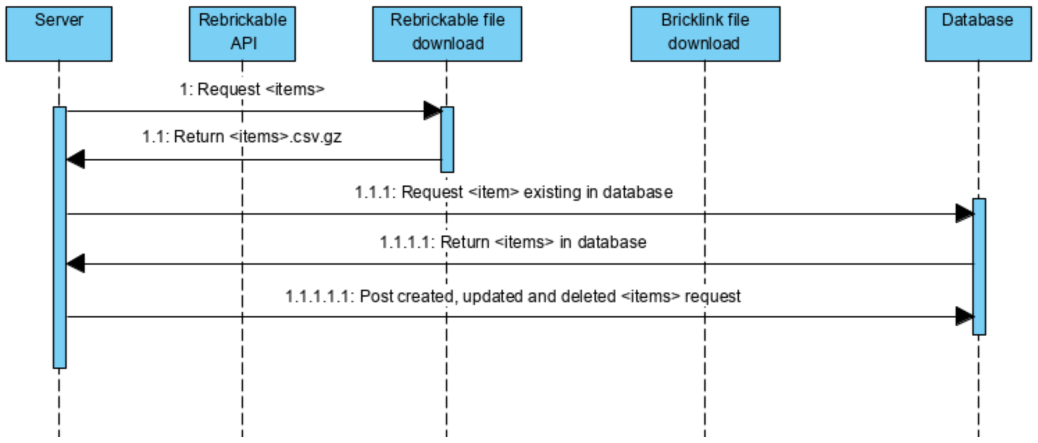

Server
Rebrickable API
Rebrickable file download
Bricklink file download
Database
1: Request <items>
1.1: Return <items>.csv.gz
1.1.1: Request <item> existing in database
1.1.1.1: Return <items> in database
1.1.1.1.1: Post created, updated and deleted <items> request

Figure F.1: Time sequence diagram Rebrickable download

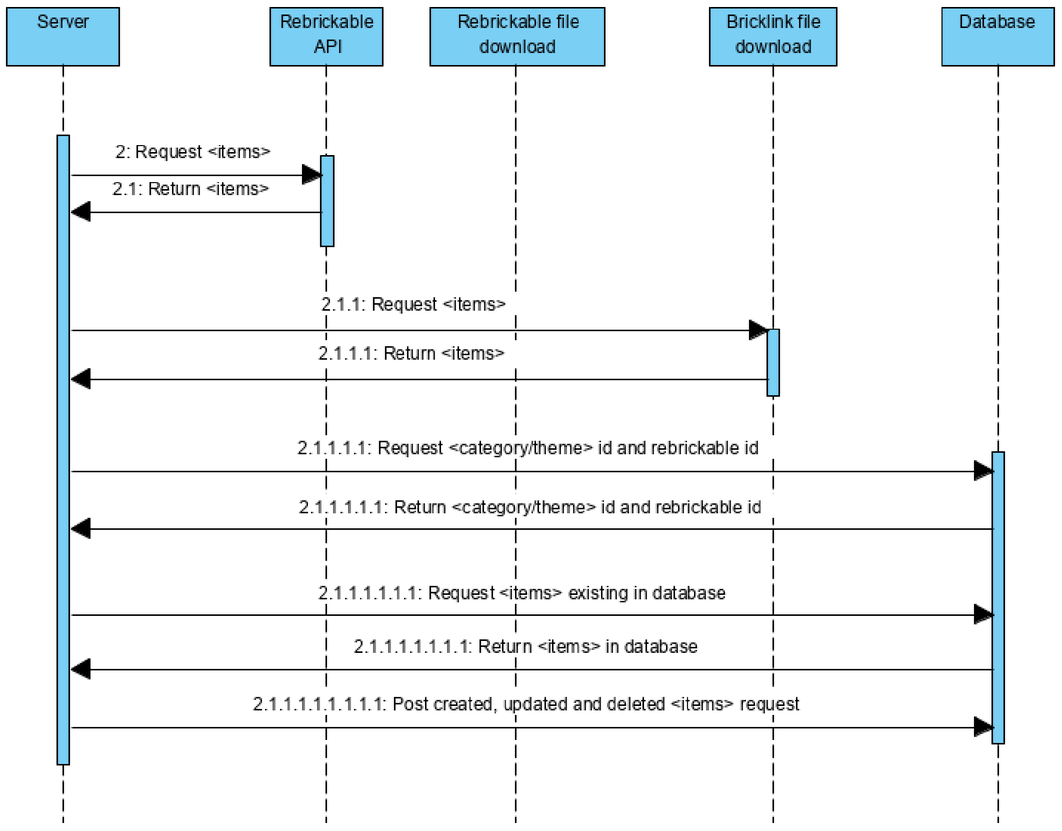

Server
Rebrickable API
Rebrickable file download
Bricklink file download
Database
2: Request <items>
2.1: Return <items>
2.1.1: Request <items>
2.1.1.1: Return <items>
2.1.1.1.1: Request <category/theme> id and rebrickable id
2.1.1.1.1.1: Return <category/theme> id and rebrickable id
2.1.1.1.1.1.1: Request <items> existing in database
2.1.1.1.1.1.1.1: Return <items> in database
2.1.1.1.1.1.1.1.1: Post created, updated and deleted <items> request

Figure F.2: Time sequence diagram Rebrickable API and Bricklink download

Appendix G

Code structure

G.1 Back end code structure

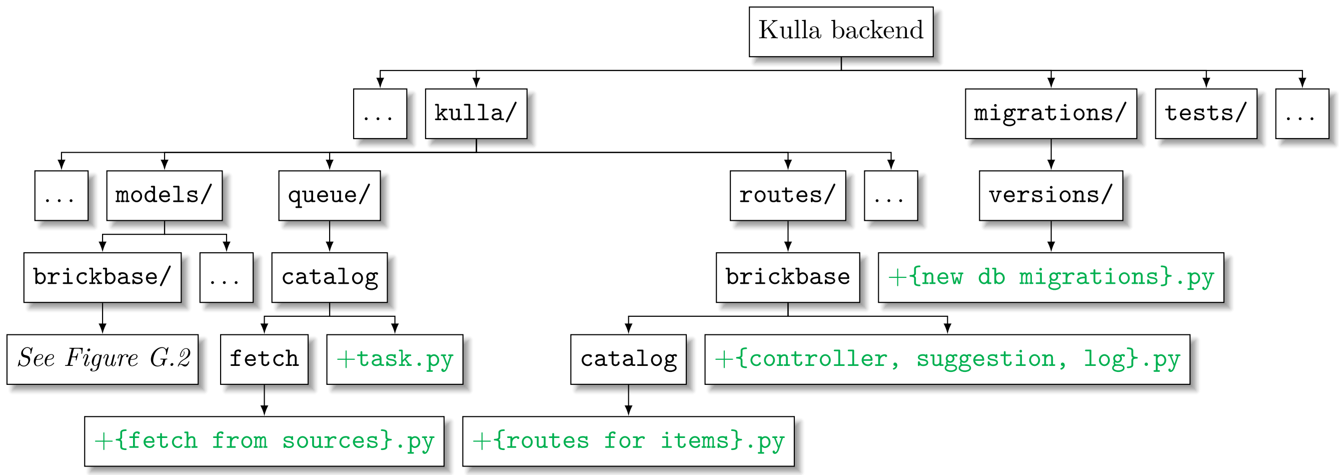


Figure G.1: Back end code structure

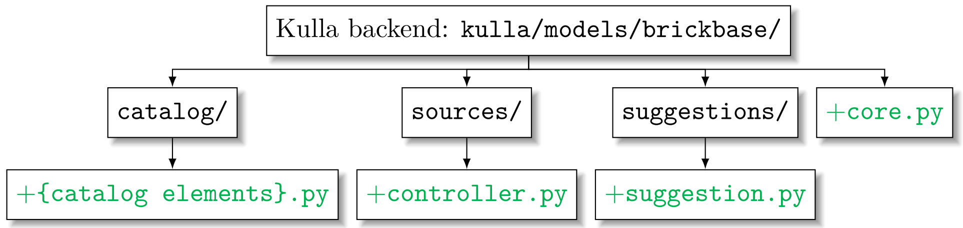


Figure G.2: Backend ORM models

G.2 Front end code structure

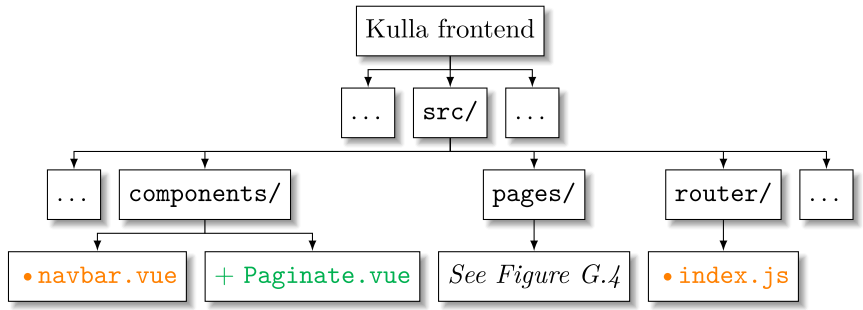


Figure G.3: Front end code structure

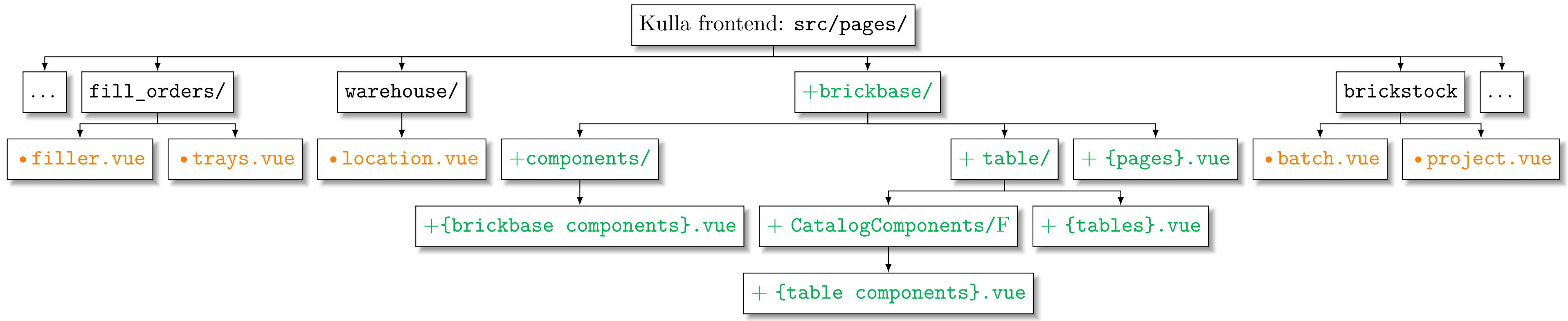
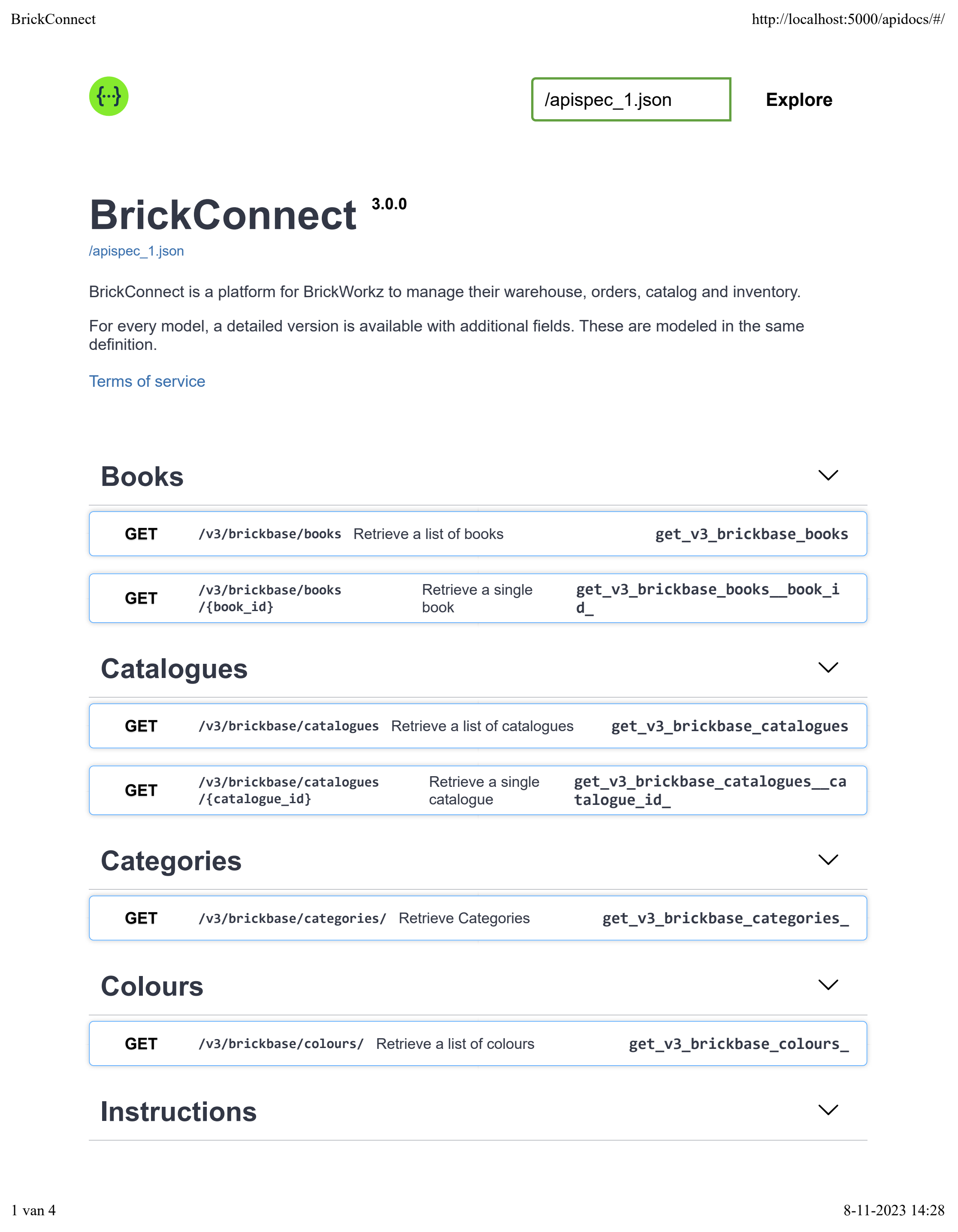


Figure G.4: Code structure within the folder pages/

Appendix H

Swagger API Documentation



BrickConnect http://localhost:5000/apidocs/#/

/apispec_1.json Explore

BrickConnect 3.0.0

/apispec_1.json

BrickConnect is a platform for BrickWorkz to manage their warehouse, orders, catalog and inventory.

For every model, a detailed version is available with additional fields. These are modeled in the same definition.

Terms of service

Books

GET	`/v3/brickbase/books`	Retrieve a list of books	`get_v3_brickbase_books`
GET	`/v3/brickbase/books/{book_id}`	Retrieve a single book	`get_v3_brickbase_books__book_id_`

Catalogues

GET	`/v3/brickbase/catalogues`	Retrieve a list of catalogues	`get_v3_brickbase_catalogues`
GET	`/v3/brickbase/catalogues/{catalogue_id}`	Retrieve a single catalogue	`get_v3_brickbase_catalogues__catalogue_id_`

Categories

GET	`/v3/brickbase/categories/`	Retrieve Categories	`get_v3_brickbase_categories_`

Colours

GET	`/v3/brickbase/colours/`	Retrieve a list of colours	`get_v3_brickbase_colours_`

Instructions

1 van 4 8-11-2023 14:28

GET	`/v3/brickbase/instructions`	Retrieve a list of instructions	`get_v3_brickbase_instructions`
GET	`/v3/brickbase/instructions/{instruction_id}`	Retrieve a single instruction	`get_v3_brickbase_instructions__instruction_id_`

Logs

GET	`/v3/brickbase/logs/`	Fetch all logs	`get_v3_brickbase_logs_`
GET	`/v3/brickbase/logs/{object_uuid}/`	Find logs related to some object in the catalog	`get_v3_brickbase_logs__object_uuid__`

Minifigs

GET	`/v3/brickbase/minifig`	Retrieve a list of minifigs	`get_v3_brickbase_minifig`
GET	`/v3/brickbase/minifig/{minifig_id}`	Retrieve a single minifig	`get_v3_brickbase_minifig__minifig_id_`

Original Boxes

GET	`/v3/brickbase/original_boxes`	Retrieve a list of original boxes	`get_v3_brickbase_original_boxes`
GET	`/v3/brickbase/original_boxes/{original_box_id}`	Retrieve a single original box	`get_v3_brickbase_original_boxes__original_box_id_`

Parts

GET	`/v3/brickbase/parts`	Retrieve a list of parts	`get_v3_brickbase_parts`
GET	`/v3/brickbase/parts/{part_id}`	Retrieve a single part	`get_v3_brickbase_parts__part_id_`

Sets

GET	`/v3/brickbase/sets`	Retrieve a list of sets	`get_v3_brickbase_sets`
GET	`/v3/brickbase/sets/{set_id}`	Retrieve a single part	`get_v3_brickbase_sets__set_id_`

Suggestions

GET	`/v3/brickbase/suggestion/{uuid}/`	Find a suggestion	`get_v3_brickbase_suggestion__uuid__`
POST	`/v3/brickbase/suggestion/{uuid}/accept/`	Accept a suggestion	`post_v3_brickbase_suggestion__uuid__accept_`
POST	`/v3/brickbase/suggestion/{uuid}/reject/`	Reject a suggestion	`post_v3_brickbase_suggestion__uuid__reject_`
GET	`/v3/brickbase/suggestions/`	Retrieve a list of suggestions	`get_v3_brickbase_suggestions_`
POST	`/v3/brickbase/suggestions/`	Create a new suggestion that changes a part of the catalog	`post_v3_brickbase_suggestions_`

Themes

GET	`/v3/brickbase/themes/`	Retrieve themes	`get_v3_brickbase_themes_`

Warehouse

GET	`/v3/brickbase/uuids/{uuid}`	Search through all objects in the catalog by uuid	`get_v3_brickbase_uuids__uuid_`

Models

CatalogAttributeModel

CatalogBook

CatalogCatalogue

CatalogCategory

CatalogColour

CatalogInstruction

CatalogLog

CatalogMinifig

CatalogOriginalBox

CatalogPart

CatalogSet

CatalogSuggestion

CatalogTheme

CatalogUUIDActorModel

User

[Powered by Flasgger 0.9.7.1]

Glossary

alternate Relation from a part to another part, where one is an alternative of the other. 34

BrickBase Catalog software built into Kulla, product of this report. 1, 6, 8, 9, 11, 13, 18, 20, 21, 30

BrickConnect Existing but incomplete warehouse software for Unbrickable, written by previous students. 4, 6, 7, 8, 12, 13, 19, 30, 31, 41, 52

BrickLink Marketplace for genuine LEGO products, similar to BrickOwl. 4, 5, 8, 15, 23, 24, 29, 30, 32, 36

BrickOwl LEGO marketplace to buy and sell LEGO Parts, Minifigures and Sets. 4, 5, 8, 20, 29, 32, 52

BrickStore Offline LEGO management tool. 8, 9, 29

Brickworkz Non-profit foundation focused on helping adolescents with special needs. 4, 53

DigiBrick Kulla platform component developed by a previous group. 30, 38

element LEGO part with a specific colour. 34, 52, 53

Flask Minimalist web application framework for Python. 11, 12, 21

integrability Ability of a system to be integrated with other systems or components seamlessly and efficiently. 10

Kulla Internal project name of BrickConnect software, named after the Sumero-Babylonian brick-god. 4, 8, 10, 11, 12, 17, 18, 20, 24, 30, 38, 52, 55

LDraw Open standard for LEGO CAD programs that allow the user to create virtual LEGO models and scenes. 5, 55

minifigure Combination of a small number of LEGO elements, can be a part of a set. 32, 34, 53

mold Hollow container used to give shape to molten or hot liquid material when it cools and hardens. 5, 34, 52

MoSCoW Prioritization technique listing requirements as **M**ust, **S**hould, **C**ould & **W**on't. 6, 7

pair Relation from a part to another part, that form a pair. 34

part Abstraction of a LEGO item that does not specify a colour. 32, 34, 52, 53

print Relation from a mold to a print. 34

Rebrickable Database containing information about all existing LEGO products. 5, 7, 8, 15, 23, 29, 30, 32, 36

set LEGO set containing elements, minifigures and other sets. 32, 34, 52

SQLAlchemy SQL toolkit and ORM framework for Python. 11, 12, 14, 15

subpart Relation from one part to another part, where one is a subpart of the other. 34

Swagger API documentation software. 18

Unbrickable Subsidiary of Brickworkz that sells LEGO bricks. 4, 52

Vue JavaScript framework for building user interfaces. 11, 24

Acronyms

ADHD Attention Deficit Hyperactivity Disorder. 4

AFOL Adult Fan of Lego. 4

API Application Programming Interface. 11, 15, 16, 17, 18, 21, 23

ASD Autism Spectrum Disorder. 4

CS Computer Science. 24

CSS Cascading Style Sheets. 11

HTTP HyperText Transfer Protocol. 18, 20

JSON JavaScript Object Notation. 18

KWaSA Kulla Warehouse Storage Application. 11

MVP Minimum Viable Product. 6, 7

ORM Object Relational Mapping. 11, 12, 15, 21, 22, 25, 46, 53, 55

PIM Product Information Management. 9, 29

RDBMS Relational Database Management System. 12

SQL Scripted Query Language. 11, 12, 53

UI User Interface. 13, 23

UUID Universally Unique IDentifier. 17

WMS Warehouse Management System. 4

List of Figures

Bibliography

[1] H. Wiencek, *The world of LEGO toys*. New York: H.N. Abrams, 1987, ISBN: 978-0-8109-1790-3.