UNIVERSITY OF TWENTE.

BRICKWORKZ
CREATE YOUR FUTURE

# Design Project

## BrickSyncer 2.0: Syncing Bricks to Stores for Bricks

Radu Roznovăț (s3164322)
Tudor Matei (s3184609)
Sebastian Dorobanțu (s3047504)
Ewout van Dijk (s3178811)
Boaz Voort (s3135136)
Alisa Biclesanu (s3164314)

**Supervised by**: dr.ir. V. Zaytsev

April 17, 2026

# Abstract

Brickworkz (Unbrickable) is a social enterprise that operates an online LEGO store that sells individual parts, custom sets, and corporate gifts worldwide. Besides selling LEGO, the organization focuses on providing employment opportunities for young people facing social challenges, that prevent their development. Currently, managing multiple large LEGO inventories across multiple online platforms such as BrickLink and BrickOwl is highly manual and employs ready-made and outdated software.

As part of the "Design Project" course at the University of Twente a system called BrickSyncer 2.0 has been developed with the goal to build a more resilient and integrated application to function as a synchronization tool between marketplaces and inventory. Its purpose is to simplify inventory synchronization and traceability, reduce errors and enable their remediation as well as making the overall workflow more straightforward and robust.

Using a agile methodology, a team of 6 students conducted the required domain research in order to gather and analyze the requirements of the system. This requirements elicitation procedure combined with the difficult communication with the product owner ended up constituting the biggest design constraint. Said requirements were used in developing a favorable design for the software system. The design makes use of available company infrastructure while staying decoupled enough to allow for continuous development. The decision still allowed for the uploading, synchronization to the stores, traceability and reversibility of stock changes resulting from both direct user interaction with the system and asynchronous stock changes in the internal company database.

The design was implemented in a dockerized environment using ASP.NET as a backend framework and Vite + React as front-end framework to a reasonable level of success given the given time and requirement elicitation constraints. At the end of the project, the vast majority of the functional and non-functional requirements have been met and the implementation is deemed a working product.

# Contents

# Chapter 1

# Introduction

Brickworks (Unbrickable) is a social enterprise that mainly employs young people facing challenges such as ADHD, autism or other neurodivergent conditions, in order to help them gain work experience. Their business consists of reselling LEGO parts, as well as making custom LEGO corporate gifts. Their mission is to help youngsters by providing jobs, so at later age they will be able to fully integrate in society. In Enschede the company has a warehouse where they store Lego bricks and do order picking. They also have an office in the same building to effectively provide services and do administrative work.

To support the way of working at Brickworkz various software is used to automate tasks and support employees in doing their tasks. The landscape of the software used is highly dynamic and multiple systems are outsourced while other systems are hosted in-house. Moreover, many of the systems are either outdated or hard to operate and requiring lots of manual operation. For this reason, Peter, the owner of the company, decided to create a design project proposal for a software system tailored to the use case of the company to fix parts of this problem. Enter BrickSyncer 2.0, a system made to syncronize inventory changes from the local wearhouse of Brickworks onto multiple online stores for bricks in a reliable, observable and reversible way that can be used with a very low learning curve.

This report details the design and implementation of BrickSyncer 2.0, chapter 2 introduces the domain of the system outlining the different systems which require to be understood preliminary, how the world of selling bricks works and all the preliminary knowledge for understanding the system. The design of the system starts with chapter 3 which details the process of requirements elicitation, analysis and validation, laying out the base for the system's specification which is the root of the design decisions for the design of the system which is explained in chapter 4. After the design of the system, the report continues in chapter 5 to an explanation of the testing strategy and results. Before overviewing the process of the systems inmplementation in chapter 6 which highlights the organizational details of the implementation before discussing the results of the project. Concluding, the system as well as the process as a whole is evaluated in chapter 7.

# Chapter 2

# Domain Analysis

The second-hand market for LEGO bricks has grown into a complex, globally-distributed ecosystem where enthusiasts and small businesses alike buy, sell, and trade pieces outside official retail channels. Understanding this ecosystem that Unbrickable operates in is paramount to having the necessary information for making the proper design decisions involved in creating BrickSyncer 2.0 .

This chapter starts by introducing the domain in which BrickSyncer 2.0 operates. The data formats, marketplace APIs, and organizational structures that define this ecosystem are examined in the subsequent chapters. Afterwards, the evolving software infrastructure at Unbrickable that forms the context for the system's design is explained before the conclusion highlights the main design considerations resulting from this analysis.

## 2.1 Domain Introduction

Outside of official retail channels, the second-hand market for Lego bricks has evolved into a sophisticated global ecosystem. With many diverse brick types, colors, and wear conditions, this community, organized around enthusiast networks that largely transcend geographical boundaries, has developed a relatively open and thriving marketplace facilitated by open-source tools and commercial platforms coexisting.

To accommodate the heterogeneity of available inventory, the community has adopted BSX files, an XML-based format acting as the standard for describing brick attributes and characteristics, having as a main tool for creating this file BrickStore. At this time, two dominant platforms exist for selling bricks: BrickOwl and BrickLink. Despite their market dominance, these platforms have not achieved complete standardization. Instead, they have each established their own conventions for representing brick attributes, resulting in inconsistent data formats across platforms. Consequently, sellers and automated systems must manage multiple attribute representations to maintain accurate inventory across different marketplaces.

While the secondary Lego brick market operates at significant scale, it remains a niche enthusiast practice. This positioning is reflected in the varying levels of maturity across APIs and software solutions within the ecosystem, where inconsistent documentation and implementation standards occasionally present technical challenges for integration efforts. This market gap resulted in the need for BrickSyncer 2.0

## 2.2 BSX files

BSX files are the main data-representations for bricks in this domain and are used with most of the systems in the second-hand-Lego world. A BSX file contains crucial information about one or more bricks or brick combinations. Every detail of a brick is described in this XML-like file in which the items are batched together with some additional information regarding the whole batch. Each tag in the BSX file has a descriptive name that contains the relevant information. The most important section of this file is the inventory section in which all items are placed with their attributes. An overview of the most important attributes and a short description is given in Table 2.1

While the structure of BSX files seems to be a convention across the domain, the content varies depending on the system using them with some systems using them for describing order details such as address, name, etc. This, however, is out of scope of this system. For the rest of this report, BSX files are treated as containers of stock changes only. Moreover, even the attributes may have different representations for their values such as different codes for different colors. As BrickSyncer 2.0 only handles BSX files from Unbrickable a single format needs to be accounted for. As indicated by the product owner, this format is the one resulting from BrickStore,

| Attribute | Description |
|---|---|
| ItemID | Identification number of the item |
| ItemTypeID | Letter value that describes whether the item is a 'part' or some other type |
| ColorID | Identification number of the color for this item |
| ColorName | The corresponding name of the ColorID |
| ItemName | Descriptive name of the item |
| ItemTypeName | The corresponding name of the ItemTypeID |
| CategoryID | Identification number of the category to which the item belongs |
| CategoryName | Corresponding name of the CategoryID |
| Status | Letter value describing whether the item is set to 'Include', 'Exclude' or 'Extra' |
| Qty | Quantity of the item |
| Price | Price of the item |
| Condition | Letter value describing whether the item is 'New' or 'Used' |
| Remarks | External notes on the item |
| Completeness | Completeness of the Lego set: 'Complete', 'Incomplete' or 'Sealed' |

Table 2.1: Overview of BSX file attributes

an open source application that allows for the creation of BSX files.

## 2.3 BrickLink

BrickLink is one of the two dominant marketplaces in this ecosystem. Their system organizes stocks into multiple inventories, each containing a list of Lego items and having a unique BrickLinkId returned by the API upon creating an item. This id needs to be used when modifying or deleting inventories. An inventory is what BrickLink uses to display a specific item the company sells [1]. Each inventory can be modified, a new inventory can be created, and specific inventories can be deleted by calling the BrickLink API. To authenticate the caller of the BrickLink API, BrickLink uses an OAuth-like system. A unique pair of oath attributes, key, token, timestamp, etc. is sent with each request to the API to fulfill the required API call. The Item Ids in a BSX file are the same ids BrickLink is using as `inventory_id`. A `color_id` in the request maps to a specific hash color code available in another part of their API.

## 2.4 BrickOwl

The second marketplace which has to be synchronized is BrickOwl. Their system is organized similarly but by using BO Lot Id instead of BrickLinkId [2]. Lots in the

inventory can be queried, updated, and deleted by means their API. Authorization is done by specifying the unique access token in the request url. This token can be created from the web interface of the store. Notably, BrickOwl also uses their own identification for colors which is different from the one provided in the BSX files. While a API endpoint exists for translating between BrickLink colors found in the BSX file and BrickOwl colors, this doesn't work for all colors.

## 2.5 Unbrickable

Unbrickable sells different types of Lego pieces on the BrickLink and BrickOwl marketplaces. New and used items are in the company's warehouse and are for sale in both BrickLink and BrickOwl. The company has their own warehouse and warehouse tracking systems which track internal stocks. In response to orders from the marketplaces these stocks need to be updated.

The parts are stored per type of the Lego piece. The type which is the most frequent is a 'PART' while other types are 'MINI FIGURE' and 'SET'. To communicate internal stock and update existing inventories on the sales platform, they BSX Files. These BSX Files are manually created by employees at the company using BrickStore to reflect the changes in the warehouse.

### 2.5.1 PED

The Product-Entity Database is a database containing the current inventory of Unbrickable. It acts as the single source of truth for BrickSyncer 2.0 . At the time of writing, the PED is still under development. The following information is the most up-to-date information available at the time of writing as the domain analysis of the PED system continued until 3 weeks before the end of this project with facts about it's functionality changing continuously.

The PED is of very high importance for the BrickSyncer 2.0 system as this is the source of all the changes that need to be synced. In order to do so, the PED also keeps a change log of all inventory changes where a change in the attribute of an item, identified by the `PEDItemId`, is represented by the old and new value as well as the FieldName which represents which attribute was changed. The primary key of a PED item is a composite key made up of the following columns Finish ts

### 2.5.2 Software Environment

As introduced in this section, the status quo increased the demand for a more elaborate and robust system. The old systemname the system was a command line interface in which users could do some basic actions to synchronize their storage to both BrickLink and BrickOwl which was installed on a laptop in the office of the

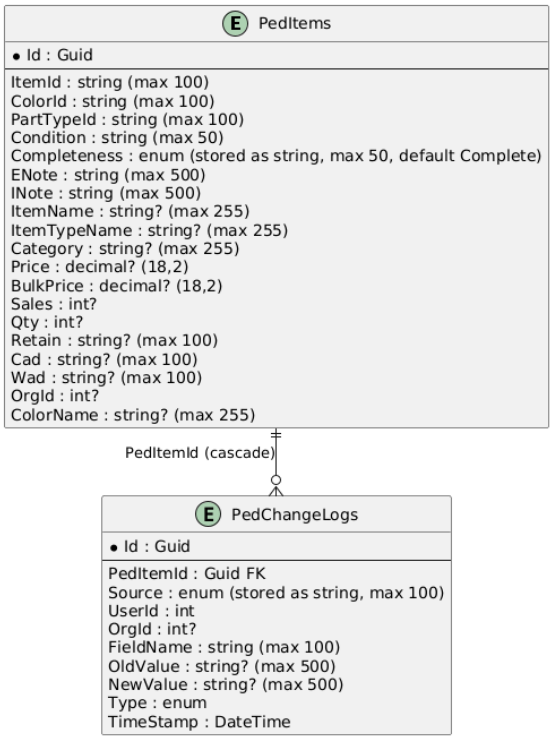



Figure 2.1: PED database schema

company. BrickSyncer was insufficient as there were a lot of problems updating the stock. The program could easily fail after synchronizing a BSX file, many times the stores had to be manually checked and updated because there were bugs in the system and the system did not provide an undo functionality or overview of all changes. Moreover, it also lacked the functionality of scheduling the updates.

Other than syncing, Unbrickable used the administrative software of `Monday.com` for storing and managing internal stocks as well as other systems. At the time of this project, Unbrickable was going trough a major refactoring by replacing this system with the PED implemented in PostgreSQL as well as another warehouse management system developed in parallel by another team doing a design project. All of these new systems are to be developed in deodorized environments and use ASP.NET as a runtime.

## 2.6 Conclusions

Resulting from the preceding domain analysis is a lot of domain specific conventions that BrickSyncer 2.0 needs to be designed to operate with such as BSX Files. All of the information about the domain at large is very useful in designing the system.

Moreover, design of the system is closely tied to the PED which still does not exist resulting in the need to implement a solution for testing. Despite the design and implementation of the PED being out of the scope of the project, a minimal version

of the PED with very similar functionality was implemented by the team using the information available. A ERD of this implementation can be seen in Figure 2.1. It's operation is designed to estimate the operation of the PED-to-be using the very scarce provided information.

Likewise, due to the ongoing development in the new software systems of Unbrickable, a decision was taken to design the system as modular and decoupled from the other company systems as possible. This is both to ensure that the system can easily adapt to future changes in requirements and to allow for implementation of the system to be as independent from the team internal to the company.

Lastly, regarding the stores. Different representations of some attributes such as color will need to be handled and as a result of the stores using different unique identifiers to represent stocks, these will also need to be tracked internally.

# Chapter 3

# Requirements

Requirements engineering is the discipline, or art, of finding out what the system one is building needs to do, expressing those needs in a structured way, resolving conflicting views and needs between stakeholders, and turning those needs into formal, unambiguous requirements that can then act as the foundation for the design of the system. It is often overlooked, yet it is absolutely crucial to ensuring a successful development process and an end result that actually solves the problems it was meant to solve.

This chapter describes this process as it unfolded during this project, covering the process of acquiring, reviewing, refining, and finalizing its requirements. The elicitation section details the approach that was used to arrive at an initial list of requirements, as initially given by the client, both directly and indirectly. The analysis section explains how conflicts and ambiguities were resolved in this initial collection of requirements, while the specification section focuses on the process of formalizing and prioritizing the requirements. Finally, the requirements are validated highlighting the iterative nature of this process and how the team confirmed, revised and refined the requirements to arrive at a final list.

## 3.1 Requirements elicitation

The process of requirements elicitation started from the very first meeting with the client. Initially, an unstructured conversational approach was used for requirements gathering interviews. It was quickly determined that a more guided, better structured methodology would be more suitable to keep the sessions focused and extract relevant, measurable criteria. For the subsequent meetings the team prepared highly detailed agendas, with agreed-upon topics and pre-formulated questions prepared, to ensure the maximum amount of concrete, in-scope requirements were extracted. This was done in an iterative manner - after each meeting, the team would debrief and discuss the newly elicited requirements, and prepare questions and goals for the next meeting.

One point of difficulty during the phase of requirements elicitation was the client's limited availability for meetings, and the fact that meetings with other stakeholders within the company were not permitted. Seeing as the sole stakeholder that interviews were conducted with was the company's CEO, the perspectives that requirements were gathered from were somewhat limited. This also introduced somewhat of a delay in communications, since the client often needed to discuss points with his software team before giving concrete answers to some technical aspects, instead of the team being able to discuss these matters directly with them in a joint interview.

After a few meetings, due to the client's availability, the process of requirements elicitation shifted towards primarily email-based communication. One of the main sources of requirements obtained during this time is a sizable email the client sent in response to a list of questions, detailing multiple aspects of the (sub)system to be built, as well as the larger system it fits in. Attached to said email were also a number of internal diagrams depicting the architecture and functionalities of various internal systems. Unfortunately, a significant part of the email and diagrams relied on internal naming schemes and notions that were not known to the team, and/or contradicted information gathered during previous meetings or from other sources. Efforts to clear up these uncertainties were not entirely fruitful, mostly due to the vast amount of ground covered by these materials. Despite this, the requirements gathered so far proved sufficient to continue to the next step in the process, namely requirements analysis.

## 3.2 Requirements analysis

During the requirements analysis phase, several informal documents were drafted in an attempt to gather all the gathered requirements, regardless of scope, category, and contradictions, in one place. This would then allow for easy comparison between the requirements, consolidation of requirements with significantly overlapping scope, and resolution of any conflicts between requirements.

Multiple rounds of reviewing and modifying requirements took place, with many

requirements needing follow-up questions or meetings with the client to clarify ambiguities or resolve conflicts where reasonable assumptions could not be safely made. The end result was a coherent, if unstructured and informal, list of requirements that together comprised the functional and non-functional aspects of the system to be built.

The analysis was done through the prism of the high-level system goals that were formulated by the team after the initial rounds of requirements elicitation, and through open discussion with the client. These goals served as the main direction of project, with all lower-level requirements having to facilitate or embody them in some way. They are as follows:

The main purpose of BrickSyncer 2.0 is to reduce manual input required from the employees of Brickworkz to synchronize inventory across multiple marketplace platforms. This has to be implemented while also reducing the possibility of human error, through intuitive design and reliable error correction possibilities.

The main problem that Brickworkz is facing is that the current syncing system they use is a command line tool that requires technical knowledge to operate. This leads to a complex workflows which in turn lead to long training times for new employees. Furthermore, this complicated workflow leads to a high number of human errors, which is detrimental to the efficiency of Brickworkz in its operations.

BrickSyncer 2.0's main goal is to prevent these inefficiencies through the development of an intuitive interface while also having numerous safe-guards in place that prevent errors, or at the very least offer reliable methods of undoing or fixing them.

## 3.3 Requirements Specification

The main focus of the requirements specification phase was to arrive from an unstructured list of requirements to a clear set of formal, properly formulated requirements that could then be used as a base for the design phase. To this end, it was decided that the best approach was to formulate the requirements according to the SMART criteria (Specific, Measurable, Achievable, Relevant, Time-bound) to ensure no problems arise due to requirements that are ambiguously formulated, not achievable or not necessary.

During this phase, internal meetings were held during which the team analyzed each requirement and its scope; some were split into multiple smaller requirements (For example REQ-001 and REQ-002), many requirements were re-formulated to avoid confusion due to ambiguous wording, and some were scrapped altogether.

In order to ensure both the timely delivery of an MVP and that no essential functionality is left out, it was decided to use the MoSCoW system for requirement prioritization - all requirements, functional and non-functional, were put into one of four categories: Must, Should, Could and Won't. The Must requirements are non-

negotiable and are essential for an MVP release. Should requirements are highly desirable and should be implemented for a full release; could requirements are desirable but not necessary for achieving the system goal, and won't requirements are out of scope.

The end result of the requirements specification phase was a Product Requirements Document. This is a document that details the curated, specified and prioritized list of requirements, as intended to be used for the design phase, ready for the review of the client. This document can then be used as a platform for discussion and review with the stakeholder, and iterated upon. The finalized list of requirements is as follows (it should be noted that this list was achieved after multiple iterations of validation):

**Functional Requirements**

| ID | Requirement Description |
|---|---|
| **MUST HAVE (Critical)** | |
| *Non-negotiable requirements essential for launch.* | |
| REQ-001 | The system must synchronize inventory attributes (e.g., quantity, price, remarks, comments) from the inputted BSX file to the Product Entity Model |
| REQ-002 | The system must synchronize inventory attribute changes (e.g., quantity, price, remarks, comments) from the Product Entity Model to external marketplaces |
| REQ-003 | The system must support BSX file upload, parsing, and validation as the primary input for inventory updates and synchronization. |
| REQ-004 | The system must allow for the creation of new product entities and update existing product entity attributes inside the PED based on uploaded BSX files. |
| REQ-005 | The system must maintain an audit a log of all synchronization actions, attribute changes, and system events. |
| REQ-006 | The system must autonomously convert internal data (necessary for syncing) to platform-specific formats for BrickLink and BrickOwl without the need for user intervention. |
| **SHOULD HAVE (Important)** | |
| *Important but not vital; can be delayed if necessary.* | |
| REQ-007 | The system's interface must include a dashboard for synchronization status, queues, and system activities. |
| REQ-008 | The system should support scheduling of synchronization tasks and batch processing. |

| ID | Requirement Description |
|---|---|
| REQ-009 | The system's interface must include log viewer the logs mentioned in the above requirement |
| REQ-010 | The system must support undoing changes from the above mentioned audit log |
| REQ-011 | Upon uploading a BSX file, the system must offer a way to select which item and which of the attributes of each item in said file to sync and which not to sync. (option of ignoring certain attributes or items) |
| REQ-012 | The system must allow users to assign and manage batch IDs and project IDs for groups of changes during upload. This information should remain in the system to ensure traceability and read from the system at a later time. |
| REQ-013 | The system's interface must include BSX file visualization as part of the BSX file upload workflow. |
| REQ-014 | The system must provide an upload preview interface displaying all changes in the BSX file and clearly indicating new items and attribute changes before confirmation. |
| REQ-015 | Upon detection of attribute changes that need to be syncronized, the system should place these changes in a queue that awaits for approval from the user before these end up syncronized. |
| REQ-016 | The system's interface must allow for the configuration for webstores (API keys, store selection, polling settings) |
| REQ-017 | The system's interface should allow for choosing whether setting an item's quantity to 0 should delete it from the databases |

## COULD HAVE (Desirable)

*Desirable functionality to include if time/budget permits.*

| | |
|---|---|
| REQ-018 | The system could support synchronization with additional marketplaces beyond BrickLink and BrickOwl (e.g., future platform integrations). |
| REQ-019 | The system must offer a way to visualize all of the PED contents. |

## WON'T HAVE (Out of Scope)

*Agreed out of scope for this release.*

| | |
|---|---|
| REQ-020 | Direct implementation of a full order management system (only indirect updates via product entity attribute changes). |
| REQ-021 | Warehouse digitization and physical inventory management processes. |

| ID | Requirement Description |
|---|---|
| REQ-022 | Full product catalogue development and catalogue enrichment features. |
| REQ-023 | Development of unrelated modules outside the synchronization domain. |
| REQ-024 | The system could provide advanced inventory insights and analytics dashboards. |
| REQ-025 | The system should provide manual editing of product entity attributes through the interface when necessary. |
| REQ-026 | The system could offer a way to set custom stock modifiers per external marketplace. |
| REQ-027 | The system could offer a way to set custom price modifiers per external marketplace. |

**Non-Functional Requirements**

| ID | Requirement Description |
|---|---|
| **MUST HAVE (Critical)** | |
| *Non-negotiable requirements essential for launch.* | |
| NFR-001 | **Reliability**: The system should be able to tolerate a complete unexpected restart without loosing data. It should be able to recover synchronization status of all the platforms and databases autonomously upon regaining power. |
| NFR-002 | **Usability**: The system shall provide an interface that allows for all requirements to be fulfilled without the need of programming knowledge or command line tool involvement. |
| NFR-003 | **Security**: The system shall use secure communication protocols (HTTPS), authentication, and encrypted storage of sensitive data such as API keys and user credentials whenever customer data is involved and marketplace APIs allow for it. |
| NFR-004 | **Scalability**: The system shall support storing up to 150000 entities. |
| NFR-005 | **Maintainability**: The system shall be modular and well-documented to allow easy updates, debugging, and future extensions. |
| NFR-007 | **Traceability**: The system shall maintain comprehensive audit logs of all actions, changes, and synchronization events for monitoring and recovery. |

| ID | Requirement Description |
|---|---|
| NFR-008 | **Compatibility**: The web application shall function on major modern browsers (e.g., Chrome, Edge, Firefox) and standard cloud environments. |
| NFR-009 | **Compliance**: The system shall comply with relevant data protection regulations (e.g., GDPR) and best practices for data handling. |
| **SHOULD HAVE (Important)** | |
| *Important but not vital; can be delayed if necessary.* | |
| NFR-010 | **Maintainability**: Marketplaces should be implemented using an abstract interface contract that allows the UNBRICKABLE team to easily implement new marketplaces into the future without the need for a redesign of the rest of the system. |
| NFR-011 | **Technologies** : The backend of the system must be written in .NET. |
| **COULD HAVE (Desirable)** | |
| *Desirable functionality to include if time/budget permits.* | |
| NFR-012 | **Extensibility**: The architecture could allow easy integration of future modules and additional platform connectors. |
| NFR-013 | **Portability**: The system should be deployable across different cloud providers with minimal configuration changes. |
| **WON'T HAVE (Out of Scope)** | |
| NFR-014 | Native mobile application support in the initial release. |

## 3.4 Requirements Validation

The requirements validation phase was the final stage of the requirements engineering pipeline. During this iterative process, the requirements as defined in the product requirement document (PRD) were presented to the client for review. This was done for two purposes. The first one was simply confirming that the requirements that had been extracted, analyzed and formalized are indeed correct and embody the client's vision for the product. The second one was adjusting the priority and distribution of the requirements within the MoSCoW framework. This is an act of compromise between staying true to the system goals and the broader purpose of the project, and ensuring that producing an MVP is possible for the given time frame.

Thus, multiple meetings were held in which the client was asked to confirm if the

requirements are accurate. In addition, he was asked to distribute the requirements in each of the MoSCoW categories. Constant discussion and bi-directional feedback between the team and the client ensured that no crucial requirements had too low a priority, and that not too many requirements crowded the Must sections such that an MVP becomes unfeasible.

The end product of the requirements validation phase, after multiple rounds of validation, was a complete, ordered list of requirements that are formulated according to the SMART criteria, match the client's needs and vision for the final product, and are accurately and realistically categorized per priority. Based on this list, a project proposal was drafted. This document outlines all the important aspects of the project, including, among others, the problem definition and project motivation, scope of the project, technical definitions, requirements, planning and organization, deliverables, test strategy and risk analysis. It describes the team's planned trajectory for the process of developing the product, ensuring both transparency for the client and a pre-defined "game plan" for internal use. Additionally, it acts as a contract between the client and the team, representing the cornerstone of the then-upcoming design phase. The client's agreement on this project proposal represented their acknowledgment that the rest of the development process would rest upon this foundation, and that changes to the requirements would not be guaranteed to be possible from this point onward.

This project proposal was presented to the client, and and agreement was reached. This signified the end of the requirements engineering phase, and the start of the design phase. Additionally, the proposal was presented to the project supervisor. Apart from feedback on the writing and formatting of the document itself, there were no urgent points of feedback.

# Chapter 4

# Design Of The System

This chapter presents the design of the BrickSyncer 2.0 system. Firstly, the chapter covers the architecture of the system followed by an overview of the system's sub-systems and their interactions through a concrete behavioral analysis that is linked to the requirements defined. The structure of the system is also presented together with how the chosen structure helps in implementing the required behavior, and the reasoning behind certain decisions. Lastly, the chapter concludes with a detailed front-end description that showcases the motivation behind interface choices.

## 4.1 The system

In order to properly explain the design of the system it needs to be properly defined. The boundaries of the system arise from the goals of the system and interaction with the domain of deployment as further explained in the previous sections. The Syncer sits in between the PED, the user and the stores acting as an intermediary. The PED, as the single source of truth, stores all of the required data about the items and their current state. In most cases, the Syncer communicates to the ped to query the data regarding the items. It also sends data to the PED when talking about BSX files being updated. While the system also interacts with the user, that interaction is further elaborated in 4.3. Interaction of the Syncer with the stores is straight forward, the Syncer updates the stores inventories when a synchronization action happens. The boundary of the system can be illustrated using the C4 diagram shown in Figure 4.1

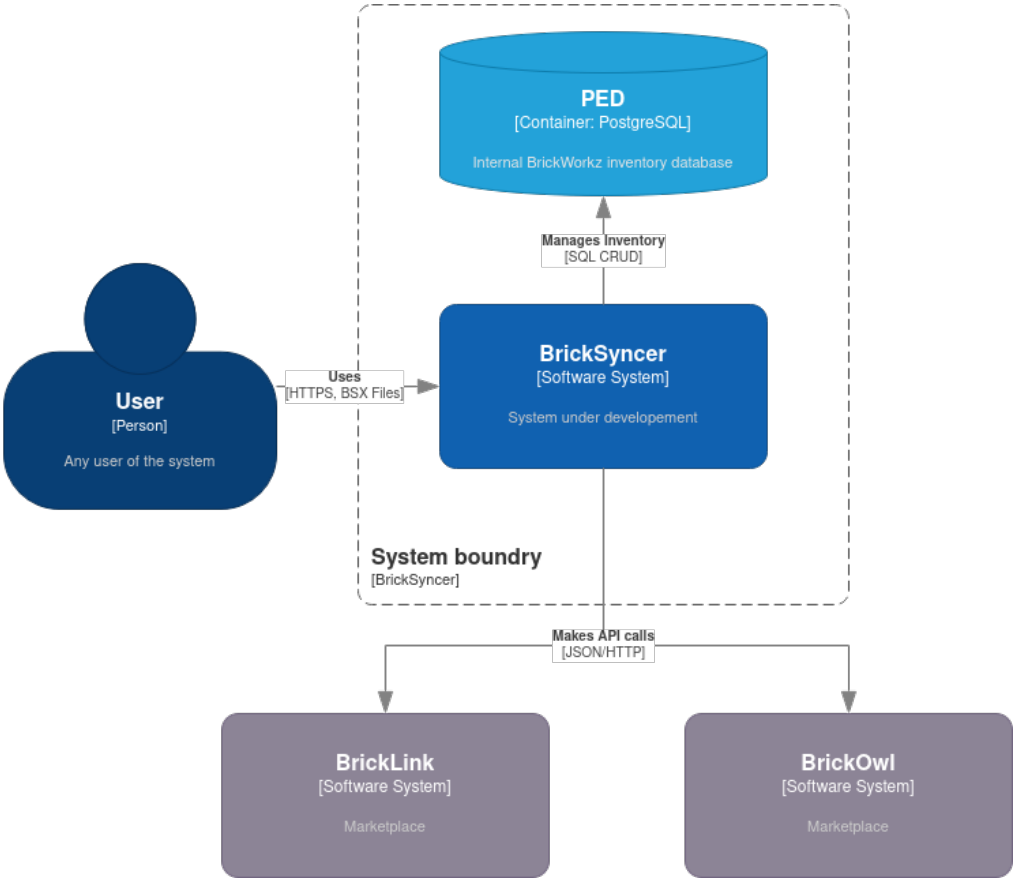



Figure 4.1: System Context diagram

## 4.2 Global design

With the boundaries of the system presented, this section delves further into the high-level design of the system mainly the global architecture, tooling and framework

choices.

### 4.2.1 Architecture

As a result of the direct interaction with the user which needs to satisfy non-functional requirement 2 the system is further divided into a front-end and a back-end. And in order to satisfy non-functional requirement 5, the following high level architecture was chosen:

The front-end and back-end are built as separate services as this closely follows the main principle of separation of concerns. The front-end is only responsible for user interactions whereas the back-end implements the business logic which is required for each interaction to work. Furthermore, this separation also allow future scalability as maybe in the future Brickworkz wants a different front-end such as a mobile application or distinct web framework. This would integrate seamlessly in the existing architecture satisfying non-functional requirement 12. Besides scalability, this separated architecture offers clear maintainability advantages as debugging is straightforward between the two services.

The back-end is further split into a layer architecture. BrickSyncer 2.0 uses a presentation, domain and persistence layer architecture as presented in [3]. This pattern allow for a clear separation of concerns as each layer has only one clear functionality [3]. The presentation layer handles the API endpoints that the front-end and store syncing functionality can use. The domain layer implements the logic of the system and the persistence layer handles the database communication such as communication with the PED or the internal database of the system. As some of the required functionality such as logging requires the storage of data not available in the PED, a local database was necessary. This architecture closely follows the maintainability, scalability and extensibility design principles present in the non functional requirements, as it improves readability, bug tracing and separation of concerns. Furthermore, by having a layer architecture testing becomes streamlined as tests can be targeted for specific functionality and better verify requirements and user flows.

### 4.2.2 Technology stack

BrickSyncer 2.0 follows a containerized architecture using Docker. This is further split into a client-server architecture. The client is mainly the browser that interacts with the front-end container. The server is the back-end container that represents. As shown in Figure Figure 4.2 BrickSyncer 2.0 also contains a database container that is only accessible through the back-end. All containers are orchestrated with the help of docker compose, and the container communication between the front-end and the back-end is done through an Nginx reverse proxy which maps the URLs to the specific containers Figure 4.2.

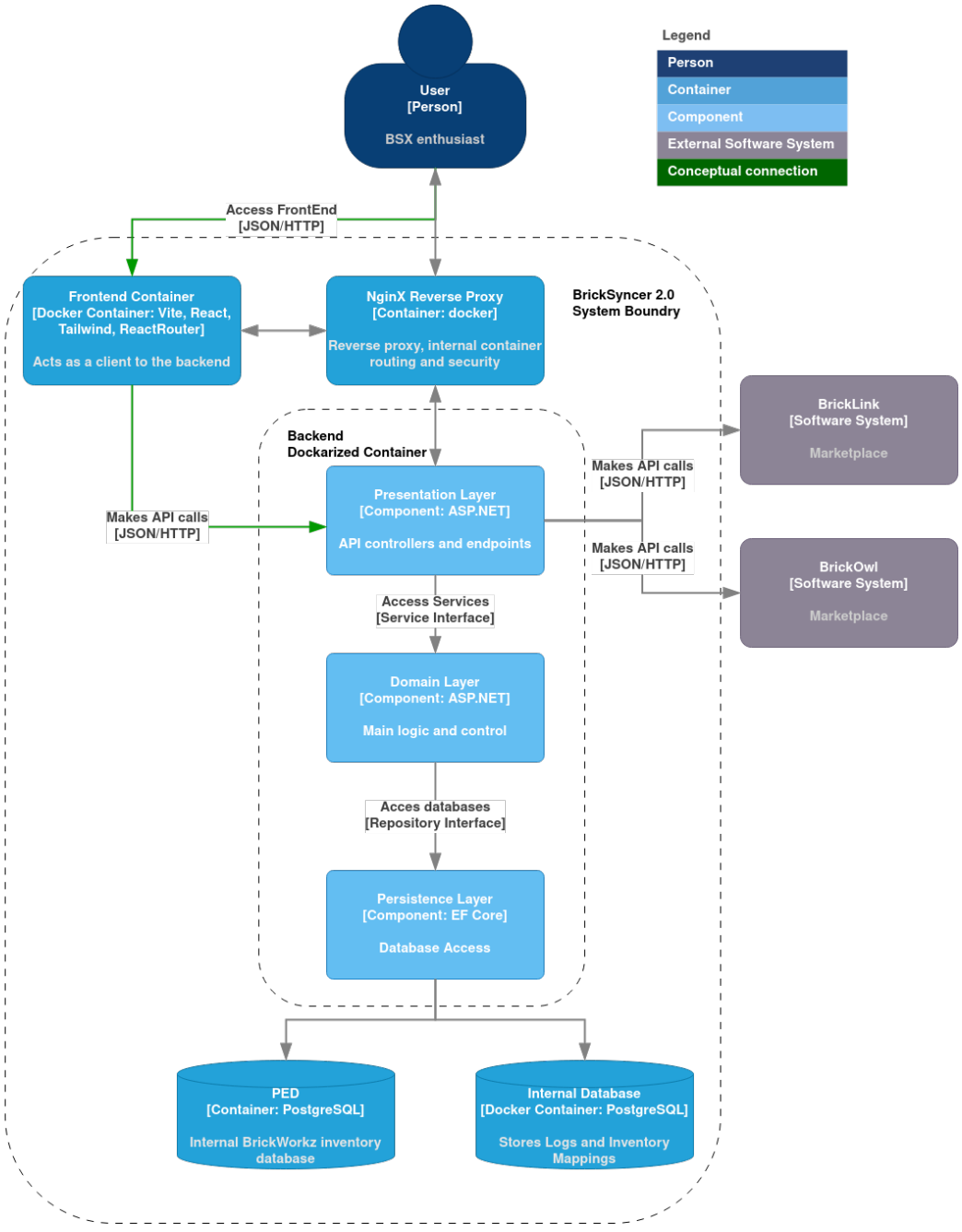



Figure 4.2: Global System Architecture

Green connections actually happen through nginx.

BrickSyncer 2.0 was setup as a fully dockerized environment. This approach still respects the scalability and extensibility design principle as their existing architecture is already composed of a docker infrastructure. Thus, integrating the BrickSyncer 2.0 system into the existing Brickworkz infrastructure would be straightforward. This approach perfectly abides by the developer experience & maintainability principles as Docker ensures the system runs on any machine while also improving the developer experience for future maintainers. This choice of infrastructure was discussed with the client and later approved.

### Front-end

Having consulted with the client, the conclusion was that BrickSyncer 2.0 would benefit from using the ReactJS framework single page application using React Router rather than NuxtJS. This design choice clearly improves developer experience, as there is no middle-ware server present as opposed to heavier front-end web frameworks. This decision further sustains future maintainability because code is easier to trace and understand.

Improving the front-end tech stack even further Tailwindcss and ShadCN UI libraries were used. Tailwindcss allows for rapid writing of CSS code while ShadCN offers reliable user-interface components that are easily integrated and tested. These two technologies allowed quick prototyping of UI, while also receiving recurring feedback through useful 3rd party integrations such as Figma. The approach of using these two technologies clearly falls inline with the usability design principle as, our client could easily decide what is suitable and what needs changing, in order to fit the requirements of BrickSyncer 2.0.

### Back-end

The back-end was straight forward in terms of frameworks and tools, as Brickworkz requested that the .NET framework be used resulting in non-functional requirement 11. BrickSyncer 2.0 uses C# and the ASP.NET framework for its server-side functionality. In line with the consistency and data integrity design pillars, the system uses a reliable database migration tool called EF Core, which ensured consistency across the development and production persistent environments. This is also important for the developer experience as it prevents database drifts during development, and acts as a version history for the database schema.

### Database

The database of choice was PostgreSQL as this is the current technology Brickworkz uses. This choice further sustains the maintainability and extensibility aspect of this system, because it simplifies future integration efforts.

**Environment**

Having built BrickSyncer 2.0 with a containerized architecture, differentiating between a development environment and a production environment was straight forward. The development environment was optimized for the developer experience this includes auto reload of the application on code changes for both front-end and back-end and Adminer service which allows developers to preview the contents of the database. The development environment was optimized for development efficiency not production ready optimizations. On the other hand the same architecture was constructed easily using Docker for the production environment, by optimizing build images and removing unnecessary services like Adminer. This flexibility between environments was why BrickSyncer 2.0 was containerized. Furthermore, Docker clearly aided in improving the developer experience by increasing efficiency and readability, while also ensuring scalability and extendability of the system.

### 4.2.3 Engineering Practices & Tooling

To further emphasize the maintainability and extensibility aspect while also targeting the developer experience certain developer tools were used inside the front-end React architecture. BrickSyncer 2.0 was developed with Prettier, a tool that automatically formats tangled front-end code. This allowed us to have fewer discrepancies and formatting differences which improved overall code quality and readability. Furthermore, the system also used ESLint which is a JavaScript linter that enforces best practices and prevents problematic code patterns. The use of these tools, dramatically improved developer experience while also ensuring a clean and readable environment for future maintainers.

## 4.3 Behavioral Modeling

The behavior of the system was designed using a top down approach. As the system works mainly as an intermediary between other systems the system was designed as conceptually asynchronous. With all the actions of the system being triggered by a asynchronous event outside the system. In that regard, the starting point was defining the use cases of the system and determining the flows of information trough the system.

The events that could enact our system into action arise from the main user stories and are split into two categories. Actions produced by the user and changes in the PED. Some of the use-cases are not objectively asynchronous, as the system does have to poll for the changes in the ped and spring into action independently when a sync is due. However as both of these still require a previous external-to-the-system event, they're still conceptually thought of as asynchronous. The user has multiple use cases: uploading a BSX File, viewing and undoing logs and viewing

and scheduling syncs. The PED upon a change triggers the creation of a sync that should be added to the sync queue displayed to the user.

These use cases and their relationships are better ilustrated by the diagram portrayed in Figure A.2. Having an overview of the high-level actions to be taken by the system, a grouping into subsystems easily emerges which can also be seen in the diagram.

Further, by expanding the main use cases, the flows of information in the system can be further outlined.

### 4.3.1 Upload flow

The upload flow represents the materialization of requirements 1, 2 and 4. As also in lustrated Figure A.1, it starts with the user uploading a BSX File. The BSX File needs further to be processed and returned for the user to visualize (requirement 13). The user than needs to assign batch and project ids to the upload for requirement 12 and in order to satisfy requirement 14 the system also queries the ped and returns the current items in the ped this allows the front-end to display a "before and after" view of applying the changes in the uploaded BSX File. At this point, the user also has the option of selecting which changes to upload (requirement 11) before confirming the upload which gets sent to the backend. At this point, the backend needs to create a log of the changes in order to ensure requirement 5 is satisfied after which it updates the ped. This PED update will further result in the syncer putting these changes in the sync queue which is explained in the next subsection, Subsection 4.3.2.

### 4.3.2 Sync Flow

The sync flow is the core informational flow realizing functional requirement 2. This depends on 2 independent events.

Firstly, for there to actually be a change that needs to be synced, the PED needs to have been updated. This can happen both due to a BSX File having been uploaded using our system, or by a different external system having updated the PED. In order to ensure both decoupling of the system from the PED and the upload source agnostic handling of the syncing, a key design choice was made to keep uploading of changes and the creation of the sync queue separate. The sync queue is actually just a subset of all the changes that happen in the ped containing only the changes that have not been synced, thus functional requirement 15 is also satisfied. In that regard, more information about it will be provided in the **logging section**.

Secondly, the user needs to schedule the syncing (requirement 8). To enable that, the front-end also provides an interface according to functional requirement 7 the design of which is further explained in Section 4.6.1. Once the scheduled time for

syncing is reached, the system must syncronized the changes to the stores. In order to do so, the changes must be translated in the store specific format. This is both to fulfill functional requirement 6 but also as there wouldn't be another way for this to be implemented. At this point if an item needs to be updated, as explained in 2.6, the specific id that the store uses to refer to the item needs to be determined before the transaction with the store. Conversely, if the item was created, that id needs to be stored for later reference after the transaction with the store.

Following the transaction, the changes need to be marked as synced. The reasoning for this is two-fold: Firstly, this removes the item from the sync queue and updates the status in the interface for this sync. Secondly, this contribuites to the fullfilment of non-functional requirement 1 as if the interaction would fail, this item would still not be marked as synced and the operation would be retried. This interaction is also ilustrated in Figure A.3

### 4.3.3 Logging Flow

The logging system is a design choice resulting from multiple requirements such as non-functional requirements 1 and 7 but also functional requirements 10 and 15. It's purpose is to provide tractability, observability and reversibility to the system.

In order to achieve the relaibility of the system, every action processed by the syncer is in some way logged. The structure of these logs that ensures they contain all the information required to repeat the transaction if it fails is. The structure of how these logs are structured is further explained in the next section. As the upload and syncing systems function asynchronously, synchronization actions need to be confirmed as successful to ensure they are retried. This is further refereed to as "confirming the log".

As previously explained, the actions that the syncer system needs to track may also result from changes to the PED that are external to the system. To create logs for these actions the system periodically polls the changes table of the PED creating the logs for the changes that are newer than the last polling time which do not result from the syncer. This last requirement can be validated by checking the source field of the change.

Undoing changes is another requirement of the system (requirement 10) that is part of the logging sub-system. Reversibility of a log without "rewriting the history" which would damage the traceability and observability of the system was achieved by creating a inverse action to the action being undone. For example by returning attributes to their old values or deleting created items, through a new inverse log.

## 4.4 Structural Modeling

The following section expands upon the behavioral overview of the system explaining more in depth the design of the core components that enable the expected functionality as explained in the previous section. This section aims to explain how the back-end of the BrickSyncer 2.0 system is structured and provide reasoning for the developmental choices, while also showing how these choices complement and aid in the construction of the required behavior.

This section consists of two subsections. First, the structure of the back-end architecture is described, including the responsibilities of its main layers and components. Second, an overview is provided of how this architecture supports the behavioral requirements defined in the previous section.

The back-end is split into a layered architecture. As stated previously, this allows for a high level of modularity within the system, which leads to a better developer experience and easier maintainability from the Brickworkz developer team. The following sections will describe each of the layers in depth.

The structural overview of the system starts with the persistance layer by explaining the data necessary for the operation of the syststem, further, the domain layer explains the way this data is handled to ensure operation

### 4.4.1 Persistence Layer

The persistence layer handles database interaction and configuration. This is the layer in which the system defines the actual database logic which the repositories inside the domain layer reference. As such, the layer contains **repository classes** that implement the defined abstract operations inside the domain layer. A diagram showing the conceptual mapping of this can be seen in Figure 4.3.For instance, the BsxRepository class from the persistence layer implements the methods defined in the IBsxRepository interface that is defined in the domain layer. This class contains the actual code need to query the database. As stated previously, this design choice aims to decouple the database (persistence layer) from the actual business logic and API definition. Thus, this leads to high modularity, which allows for potential replacements in the persistence source such as migrating to a new database provider.

Furthermore, this layer holds all the relevant database configuration files. In the case of the BrickSyncer 2.0 system database connectivity is implemented using the .NET Entity Framework (EF) tooling. Therefore the persistence layer also contains relevant configuration files that define how EF should create the tables and how to map domain entities to database entities.

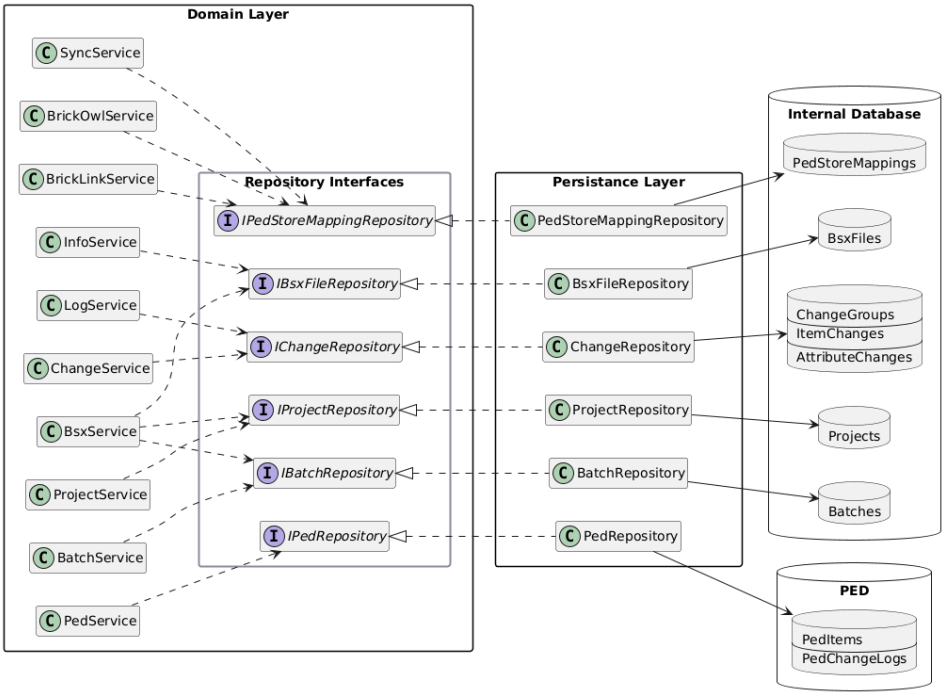



Figure 4.3: Repository interfaces dependency graph

### Changes

As explained in previous sections, the main flows of information involve propagating stock changes that come either from the PED or from the BSX File . Moreover these changes need always be logged to ensure readability. To ensure this the system couples changes and logs into the same data structure named ChangeGroups. As the only information that needs to be logged are these changes themselves the ChangeGroups function both as an internal data object for changes that need to be applied and as logs for reversibility. Consequently, they need to store all the information needed to ensure that they can both be reversed and reapplied in case of failure. A single change, such as one coming from a BSX File file, may change multiple attributes of multiple items. Moreover these attributes may be numeric or non-numeric.

Taking into account all of these things the database structure depicted in Figure 4.4 was designed where one ChangeGroup relates to one change, which has all the information about the change such as the BsxId relating to the BSX File that generated it [1] as well as the scheduled sync time and a isSynced boolean representing it's sync status.

A ChangeGroup may have multiple ItemChanges of different types all of which are related to a item in the PED using the PEDId and which have one or more AttributeChanges.

[1] A diagram for the table storing BSX file ids can be found in Figure 4.6

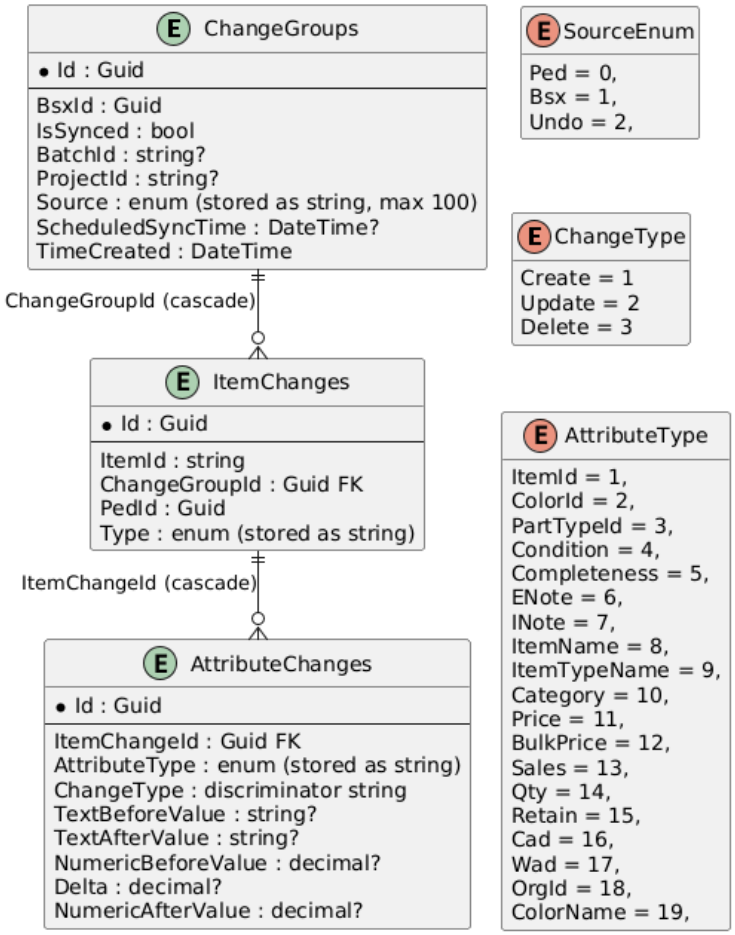



Figure 4.4: Changes ERD

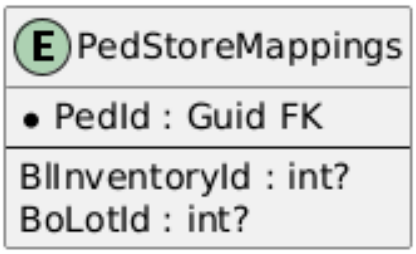



Figure 4.5: PedInventoryMappings

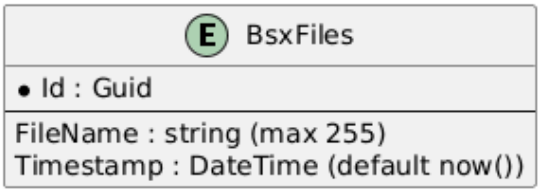



Figure 4.6: ERD: Table for storing bsx files

**PEDInventoryMappings**

As explained in Subsection 4.3.2, the stores use their unique ids for referencing inventory which needs to be tracked as part of the system to ensure later reference. This is also stored in the database using a simple table depicted in Figure 4.5.

### 4.4.2 Domain Layer

The domain layer is the most complex layer as it is the layer that contains the actual business logic. After receiving a user request and validating it, the presentation layer provides the data to this layer for processing. The main goal of this layer is to keep dependencies to a minimum while decoupling the logic from any other external APIs, databases or any other services. This allows for easy extensibility and modification of existing business logic without major breaking changes in other dependencies of the system. Conceptually, the domain layer is made up of 3 main components:

- **Entities:** The core domain entities of the system. These classes model real-world objects and contain the data structures used within the domain such as logs, stores or BSX File
- **RepositoryInterfaces:** Interfaces that define how the domain layer interacts with the persistence layer. These abstractions allow the business logic to remain independent from database implementation details and act as the main connector between the layers.
- **Services:** Services represent the logic of the system. Service classes organize domain operations, enforce rules, and interact with repositories when data access is required. Each class being responsible for one specific functional property of the system.

Moreover, in coordinating between all other systems and sub-systems the different representations of the entities that are needed are implemented using Data Transfer Objects (DTOS). These objects define request and response models used by services and controllers. Conceptually, services use the interfaces to communicate information about entities that are represented by DTOS.

### 4.4.3 Presentation Layer

The presentation layer builds the BrickSyncer 2.0 API interface. This mainly refers to defining API routes that the front-end can interact with. These routes are defined using **controller classes** whose goals are to control the flow of data between the user interface and the domain layer. This flow of data contains two main responsibilities: Parsing of the received data from the JSON format supplied by the front-end into

a internal representation such as a readable entity or class and validation of the request.

The main reason for choosing to have a presentation layer and not interact directly with the business logic, is decouple business entities from user sent data. Furthermore, the presence of this layer allows future developers to easily introduce additional validation rules without requiring modifications to the business logic.

## 4.5 Back-end Interaction Overview

While the previous subsection described the structural organization of the back-end, it does not yet show how these individual components interact in order to provide the expected system behavior described.

Therefore, the following subsection will provide an overview of the interactions between controllers, services and repositories, while emphasizing the system behavioral flows. These flows will be explained with the help of class diagrams to better showcase the relationships between layers and separation of responsibilities which are the basis for the structural choices.

### 4.5.1 Upload Flow

The upload flow contains two key interactions. First, the user needs to be able to upload the BSX File and see exactly what changes that file will produce. Secondly, the user should use the information received from the first step and be able to alter the the changes that will be produced by the BSX File by editing and manipulating the values inside the file before sending it back.

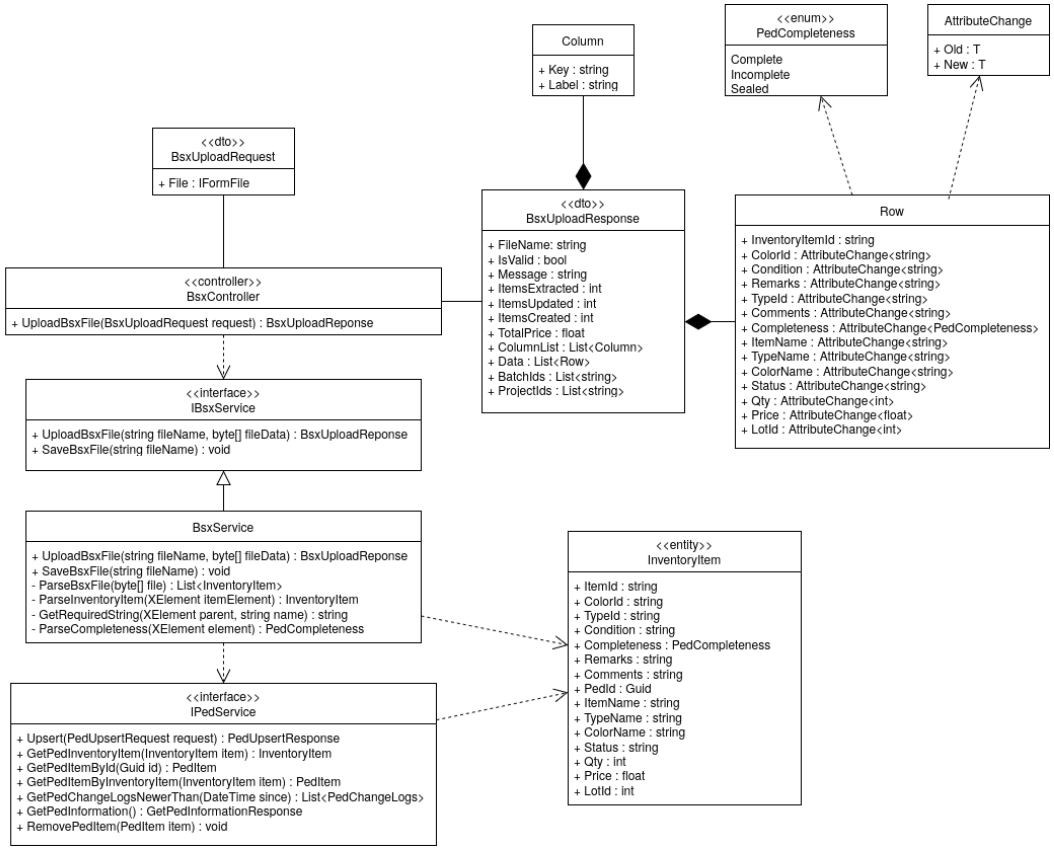



Figure 4.7: BSX file upload class diagram

As shown in Figure 4.7, the `BSXController` receives as user request through the form of the `BsxUploadRequest` object. This object contains all data necessary for the `BSXService` to process the user request. In this specific case, the `BSXController` does some initial validation on the data, such as file validation, before delegating the upload processing logic to the BsxService through the `IBsxService` interface. Interfaces are widely used inside the BrickSyncer 2.0 system as they provide abstraction of the implementation and allow the presentation layer to remain independent from the concrete service classes details. The system can then use the DTO object together with the `IPedService` interface to retrieve existing PED items for comparison. Notice how by separating the data transfer object from the actual business entities such as `InventoryItem`, the response can easily be modified to fit client needs with no changes in business logic.

The second part of the upload flow begins with the `ChangeController` in 4.8. By receiving the `ChangeRequest` DTO with the modified BSX File data and using that for processing, the design avoids directly exposing domain entities. Within the domain layer, the `ChangeService` aggregates all the BSX File modifications into a single `ChangeGroup` entity. This object design ensures all BSX File uploads are treated as a single entity, atomically. This decision was taken because the client wanted full undo functionality for BSX File uploads, which represent the main interaction inside the system. By using this approach, any other operations

that use such an entity such as syncing the change or logging it can easily operate on this single object, rather than having to reconstruct relationships between individual changes. This design choice bases the control of syncing and logging around BSX File uploads, which falls in line with the main requirement 1.

The use of a hierarchical structure (`ChangeGroup` → `ItemChange` → `AttributeChange`), was a design choice that closely respects that functionality of a BSX File upload. This is because a BSX File upload can contain multiple item changes that each can change a various number of attributes. This design component reflects the natural structure of the BSX File data. Basing the syncing and logging functionality of the same `ChangeGroup` objects is a logical decision as it allows synchronization and logging operate on a single consistent data structure. This approach mainly avoid duplication of logic for representing "changes" inside the BrickSyncer 2.0 system. Therefore, the `ChangeGroup` object was built to represent changes resulting from a file upload, PED detected changes or results of undo actions.

### 4.5.2 Sync Flow

The synchronization functionality builds directly on the `ChangeGroup` unit presented during the upload flow and in 4.8. A `ChangeGroup` contains all the necessary information for the synchronization of the current state to the marketplaces. The main controller that enables synchronization is the `ChangeController` as it manipulates `ChangeGroup` entities. More precisely the controller allows for the scheduling and unscheduling of `ChangeGroup` objects. The main service that implements this functionality is the `LogService`. Notice how both `ChangeService` and `LogService` handle operations on `ChangeGroup` entities. The `ChangeService` focuses on creating changes, whereas the `LogService` manages changes, including scheduling, synchronization, tracking and undoing.

The actual synchronization logic, which involves sending requests to external BrickLink and BrickOwl marketplace APIs, is done through a background worker. Figure 4.9 presents the `StoreSyncWorker` service in detail and how it interacts with the `IStoreService` interface. The main reason for designing a background worker is to ensure the synchronization is decoupled from the rest of the BrickSyncer 2.0 system, and this individual component is only concerned with `ChangeGroup` entities and getting them to the stores. The background worker architecture allows the BrickSyncer 2.0 system to poll the `ChangeGroup` database and detect items that have to be synced, at a configurable interval.

The `IStoreService` interface was a mandatory design step as it defines a modular approach for integrating new stores into the synchronization platform as the client requested in functional requirement 18. This interface defines what a functional store needs to implement in order to work with the syncer. Figure 4.9 shows the conversion from the internal business entities such as PED Item and `ChangeGroup` to marketplace specific formats, using API specific resources. The actual conversions are done in `BrickOwlService` and `BrickLinkService` respectively, as each stores

has its own internal resource mapping rules.

Lastly, the sync queue is comprised of `ChangeGroup` entities that have a scheduled sync time. Therefore, the `StoreSyncWorker` is able to poll `ChaneGroup` objects and check if they are scheduled. If the sync date of a `ChangeGroup` has passed the current server date, then a sync action will commence, and items will be synchronized to the two marketplaces.

### 4.5.3 Logging Flow

The core logging functionality is based on the `ChangeGroup` entities. This is the same entity presented in the upload flow and in figure 4.8. This means that every logged action that happens in BrickSyncer 2.0 system contains all relevant information about the change performed, more exactly it contains the items affected, attribute level modifications and group metadata such as batch, project, timestamp, source and synchronization state. As logs are a structural component, this design choice allowed the system to use logs for operations (undoing and synchronization) rather than simple observation. A `ChangeGroup` represents a structured way to store a change without having to reconstruct the original change data. This design choice ensures consistency between what is actually logged and synced.

One important detail about the logging flow is the source of a `ChangeGroup` entity. The source indicates if a `ChangeGroup` was generated by the BSX File upload (`SOURCE = BSX`) or by a detected change from the PED (`SOURCE = MANUAL`). Its important to understand how the detection of changes in PED is structured. A background worker just like the syncing one, polls the PED at a fixed interval, and checks if the last `ChangeGroup` corresponds to the last `PEDChangeLog` entity. If not, then all unaccounted logs are accumulated and placed into a single representative `ChangeGroup` entity inside the BrickSyncer 2.0 system. This is the basis for creating a `ChangeGroup` from outside the BSX File upload flow. The inclusion of a `Source` field in the entity allows the system to distinguish between internally generated and externally detected changes. The main advantage of this design is that regardless of source, the `ChangeGroup` entities are all processed uniformly because they have the same structure.

Lastly, the undo functionality is implementing by creating the reverse of a `ChangeGroup` entity. Due to the hierarchical design of the entity, getting the inverse operation of is trivial. Furthermore, instead of altering or deleting exiting `ChangeGroup` entities, BrickSyncer 2.0 generates a new `ChangeGroup` that represents the inverse. This design decision ensures the system is fully traceable and essentially not rewriting history. This implementation is possible mainly because of the attribute level precision of a `ChangeGroup` which stores both the previous and the after values.

## 4.6 Design of the Front-End

The front-end of the system was developed with the intention of providing the system's users with an intuitive flow, while closely following a complete implementation of the system's functional requirements. A main focus discussed with the stakeholder, starting with the first phase of design, was creating a system that minimizes the risk of user error. The current BrickSyncer 2.0 system was described as user-error-prone and difficult to operate by new or unexperienced staff, with a significant amount of time required for employees to learn its functionalities. Thus, the key goal was to reduce the learning curve by implementing a logical, easy-to-follow flow and a clear depiction of the main system functionalities: upload bsx, sync changes, managing and monitoring logs.

The main tool used for supporting the front-end design process was Figma. The team operated under an iterative approach, creating and discarding many prototypes in the initial phase, while following the requirements list and consulting the client. Because of this approach, the implementation of the front-end was straightforward. Only once the Figma design was agreed upon by the team and client, the implementation was started. This strategy didn't only ensure that efforts are not lost to rewriting front-end code, but it also made the actual implementation easier.

### 4.6.1 Interface

By following an iterative strategy for designing the front-end, the team could notice inconsistencies in the system's requirements. After every design iteration, the requirements were revisited and adjusted together with the client. Even though the Figma design was used as a means of communicating the practical understanding of the system in between teammates, the client had difficulties expressing feedback on the interface in a way that was productive to the development. Limited input from the client resulted in a high degree of freedom in shaping the interface. While this allowed for a large degree of creative exploration, it was also counterproductive due to the fact that the team had to rely on indirect feedback and assumptions based on constantly changing information.

#### Dashboard

The dashboard (figure 4.10) is the entry point of the application and it provides a layout of cards displaying various system information, such as: total PED entities, last BSX File upload, last/next sync. The scope of this page is to review the current state of the system in an efficient manner, with respect to REQ-007.

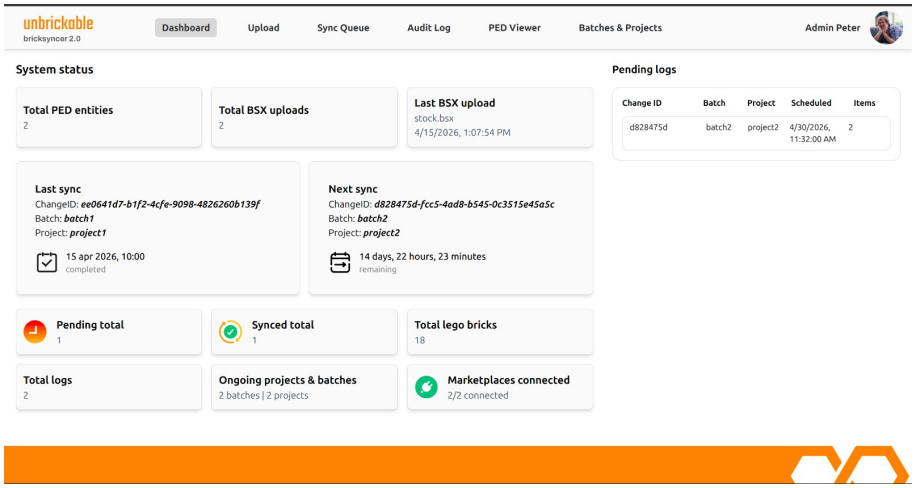



Figure 4.10: Dashboard

Constructing the layout of the dashboard page required a rigorous series of feedback analysis. The client couldn't describe what type of information is essential for getting a quick overview of the system. Overcoming this design challenge meant prototyping various possible dashboards. Deciding what information is telling of the system's state encouraged the team to rethink the scope of the BrickSyncer. The team concluded that the primary focus was displaying information about the current state, and information that points out if any recent user-errors have been made.

Tracking the current state of the system is made possible through cards such as:

- Last sync card (change group ID, batch ID, project ID and date of completion) - tracks the information of the last synced change group
- Next sync card (change group ID, batch ID, project ID and scheduled date) - tracks the information of the next change group that is going to sync
- Last BSX File upload (name of last uploaded BSX File, date and time of upload) - tracks the information of the last uploaded BSX File

Tracking user-errors through cards is done indirectly by following the numerical values of card such as: Total PED entities, Total BSX Fileuploads, Pending total ,Synced total ,Total lego bricks ,Total logs ,Ongoing projects & batches

Tracking system failure is done through checking "Marketplaces connected".

When logs are awaiting confirmation in sync queue, a list of "pending logs" is displayed in the dashboard as well, in a compacted manner, on the right side.

### Upload BSX - Path

The upload BSX Filepath consists of 4 pages: upload BSX, preview BSX, edit BSX, confirm BSX. This is one of the core features of the system, thus close attention was

given to ensuring the flow is natural for the user, and editing the item attributes is intuitive. This flow implements multiple requirements.

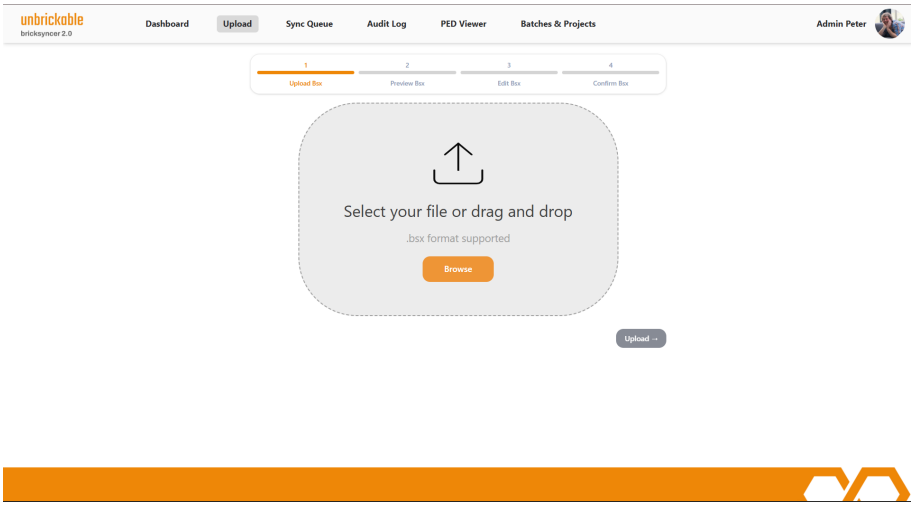



Figure 4.11: Upload BSX

The upload BSX File page (figure 4.11) allows users to select a .bsx file from their local system, and upload it, in respect to REQ-003.

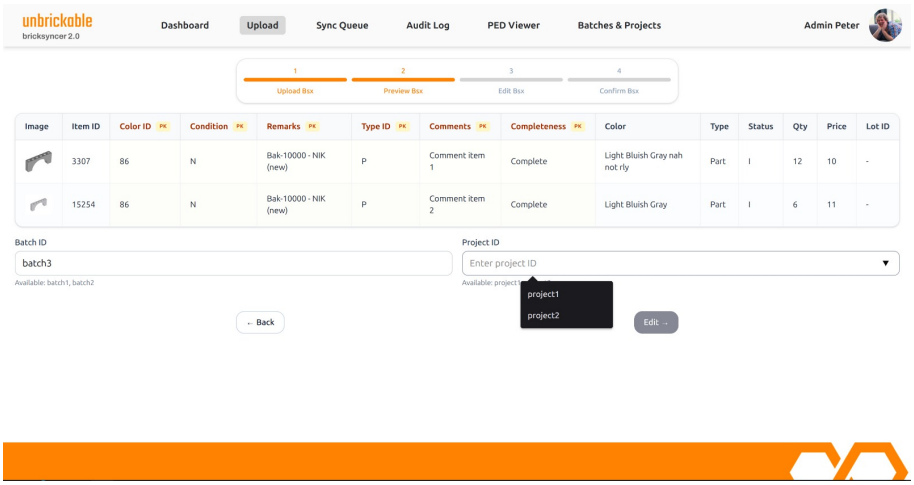



Figure 4.12: Preview BSX

Visualizing the BSX Files serves the purpose of ensuring the correct BSX File was uploaded. This page (figure 4.12) embodies REQ-013 and REQ-014.

Before moving to the select attributes phase, the user must input a batch ID and project ID. These ID's will be attributed to the change group, and already existing ID's can be viewed and selected from the drop-down prompt, according to REQ-012.

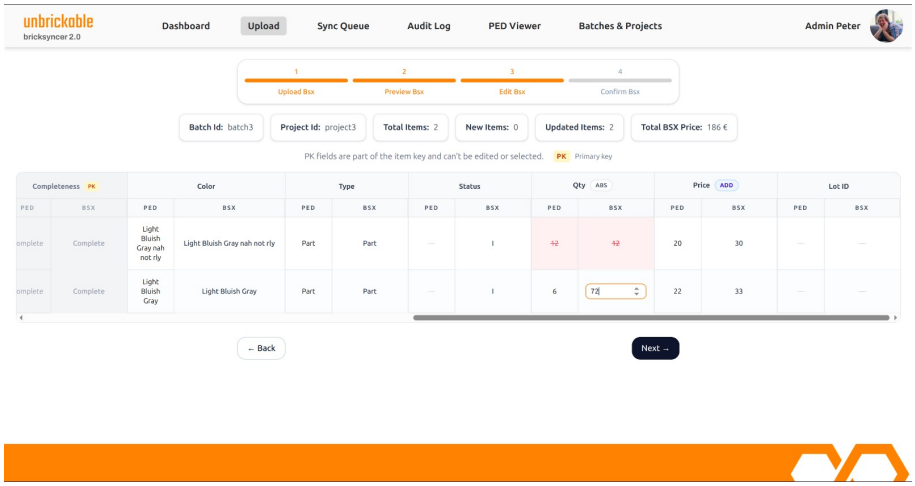



Figure 4.13: Edit BSX

In the edit BSX File page (figure 4.13) the user selects what attribute columns, item rows or individual attribute cells to sync. Information cards in the top bar display information meant to identify the change group (project and batch ID), together with the numeric changes uploading the file would result in (total items, new items, updated items, total BSX File price). This page is the implementation of REQ-011.

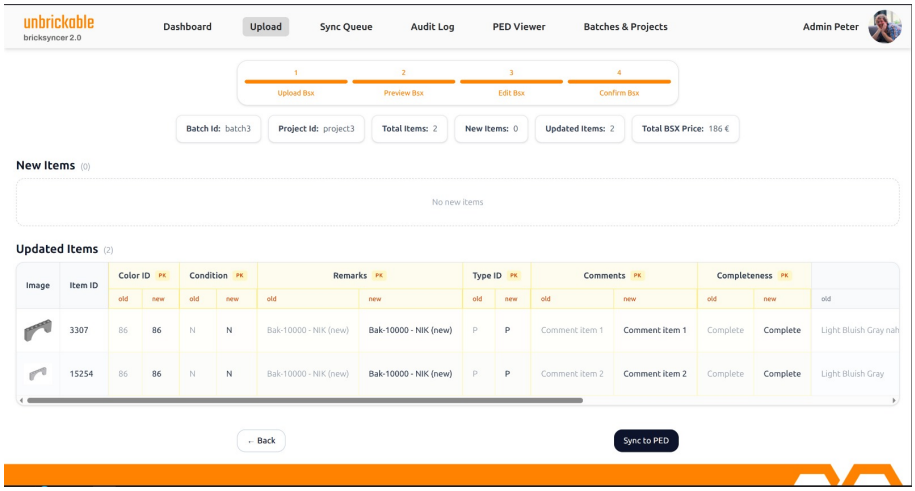



Figure 4.14: Confirm BSX

Lastly, the confirm BSX File page (figure 4.14) displays the individual item attribute changes one last type before the users confirms the sync to PED action: REQ-001, REQ-004.

### Sync Queue

This page (figure 4.15) displays the logs that are waiting to be synced, and it provides the user with the option of scheduling a time and date of sync. This is the

implementation of REQ-015, REQ-002.

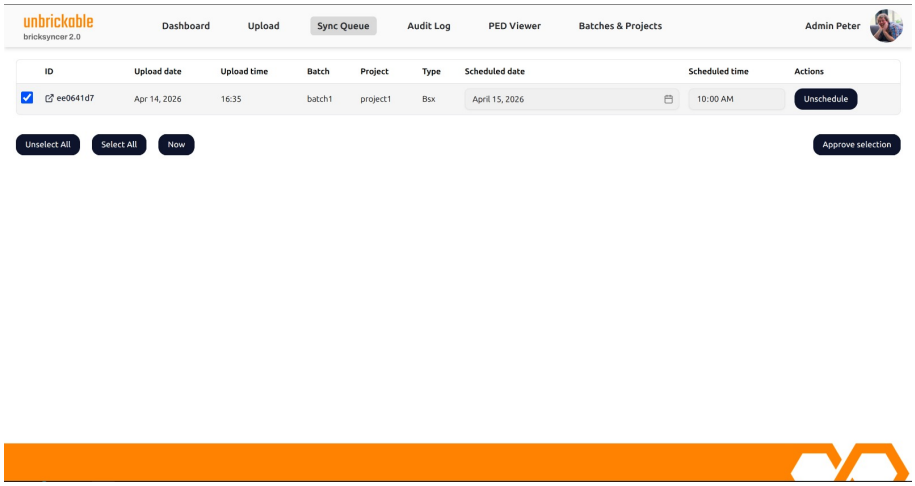



Figure 4.15: Sync Queue

”Scheduled date” and ”scheduled time” are pop-ups that allow for the user to change the sync date and time, in accordance to REQ-008. The ”unschedule” functionality sets the date and time to null, placing the log back into a waiting-for-sync state.

The ”now” functionality immediately syncs all selected logs, by changing the scheduled time and date to the current time and date.

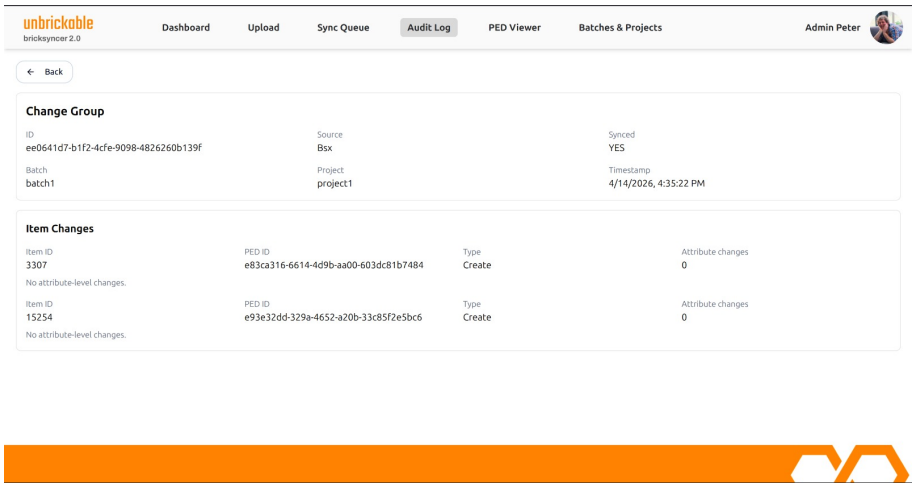



Figure 4.16: Change log information (redirect from Sync Queue)

Individual log information together with individual item changes related to the change can be viewed by clicking the icon on the left of the changelog ID (figure 4.16).

### Audit Log

In the audit log page (figure 4.17) the user can view a history of all the changes. This includes all BSX File uploads regardless of whether they have been synced to marketplaces or not, as well as external changes whose PED change logs have been converted to internal change logs and synced to the internal change log.

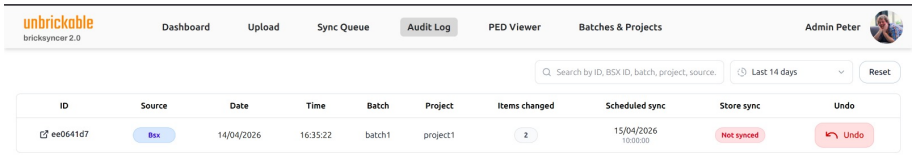



Figure 4.17: Audit Log

An ”undo” functionality is available for all logs (except undo logs), as per REQ-010. Undoing a log results in the system calculating a reverse of the change that was applied. That change is synced to the PED and further treated like any other change, being able to be scheduled to external marketplaces.

The audit log functionalities are described in requirements:REQ-005, REQ-009

### PED viewer

The PED viewing page (figure 4.18) provides a convenient way for a user to view all the items of the PED. Although this was a could requirement REQ-019, this functionality proved very useful for development purposes, and it was subsequently polished and kept in the final product.

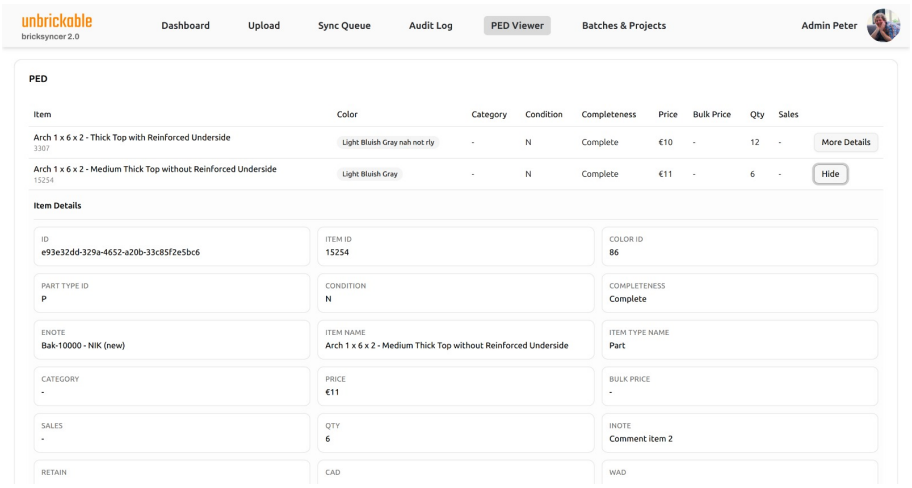



Figure 4.18: PED viewer

### Batch and projects

The batched and projects page (figure 4.19) provides an interface for viewing, creating, editing and deleting batches and projects REQ-012.

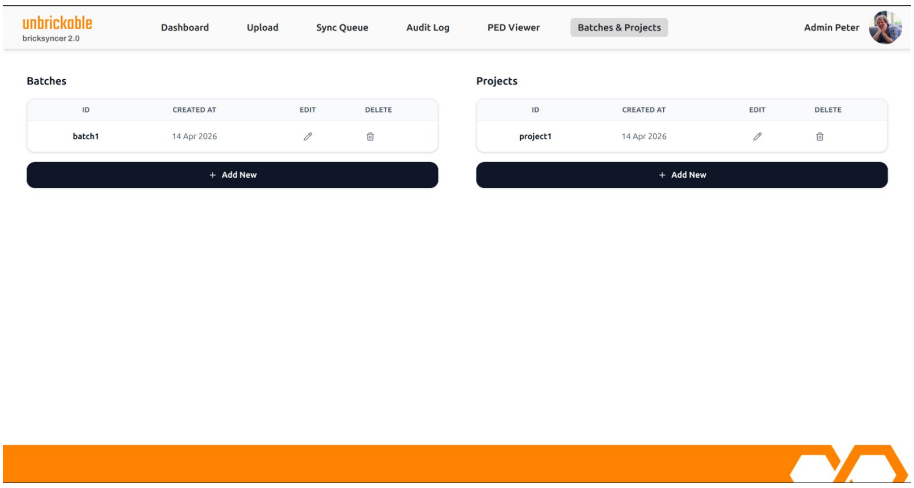



Figure 4.19: Batch and projects

## 4.6.2 Structure

This section analyzes the underlying architecture and structure of the front-end. It will first outline the high-level architecture and its philosophy, followed by the lower-level structure of the concrete codebase and how it is organized. The Tools subsection outlines the main technologies and tools that were used to build it.

As referenced in section 4.2.2, the front-end is implemented as a Single Page Application using a React, together with React Router for navigation. Tailwind CSS

and shadcn/ui were used as styling libraries, which provided the team with an easier method of developing the interface design. Figma served as a interface design tool and the main point of reference during the coding of the front-end, in terms of layout, styling and functionality.

### Architecture

The team opted for a SPA (Single Page Application) design for the implementation of the front-end. This choice was made taking into account the need for a fast development cycle, where components can be reused across the application. In addition, Single Page Applications help the system feel more responsive, since they only load once and dynamically update content. Navigation between the different parts of the application is handled through client-side routing.

The front-end acts as the client in the broader client-server architecture that the whole system is based on. The client is dockerized and runs in its own container - this decision was made to respect the principle of separation of concerns, and it additionally helps with portability and ease of deployment.

All communication with the backend happens through the API endpoints exposed by the backend, through HTTP requests. This centralized approach ensures that reliability and scalability demands are met, in addition to being widely supported and standard in all modern browsers.

### Code Structure

The front-end codebase is modular and supports reusing components, while trying to follow the best practices of React. The folder structure is meant to be scalable and readable, while it groups apart different functionalities.

The frontend/src/pages folder (figure 4.20) contains all views of the application: dashboard, audit, upload bsx, sync queue, ped viewer, and batch & project viewer. View-specific components sit together with the view, for example: dashboard and it's two types of cards `SyncCard` and `SystemStatusCard`.

Reusable components are found in the frontend/src/components directory (figure 4.21). Some of the components are more generic, and they are used in multiple parts of the system, such as Button and Notification, while others are specialized such as BsxTable or StatCard. These are custom-built components. Components from the frontend/src/components/ui folder are based on shadcn/ui. These components provide pre-built, accessible, and customizable interface primitives that serve as the foundation for the application's design system.

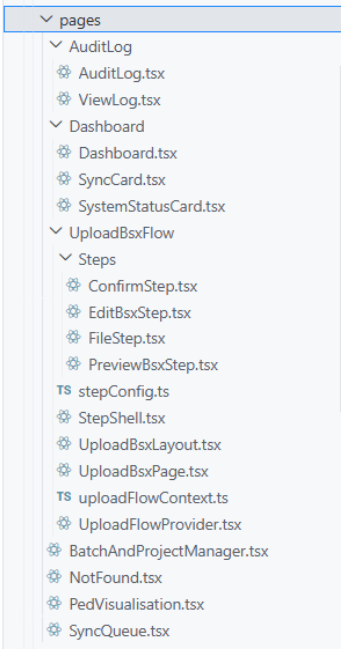



Figure 4.20: Pages

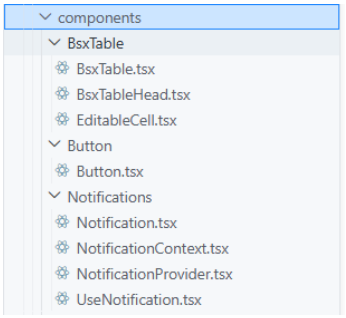
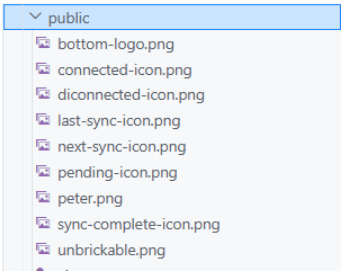
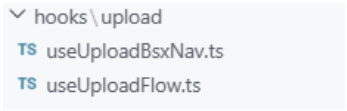



Figure 4.21: Components

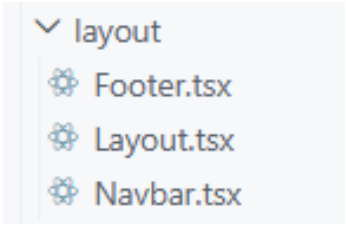
All logos and icons are in frontend/public (figure 4.22a). The icons are of the type .png and they are written in kebab case. Hooks are defined in the frontend/hooks directory (figure 4.22b). Layout of all pages in frontend/layout (figure 4.22c).




(a) Logos




(b) Hooks




(c) Layout

Figure 4.22: Frontend structure overview

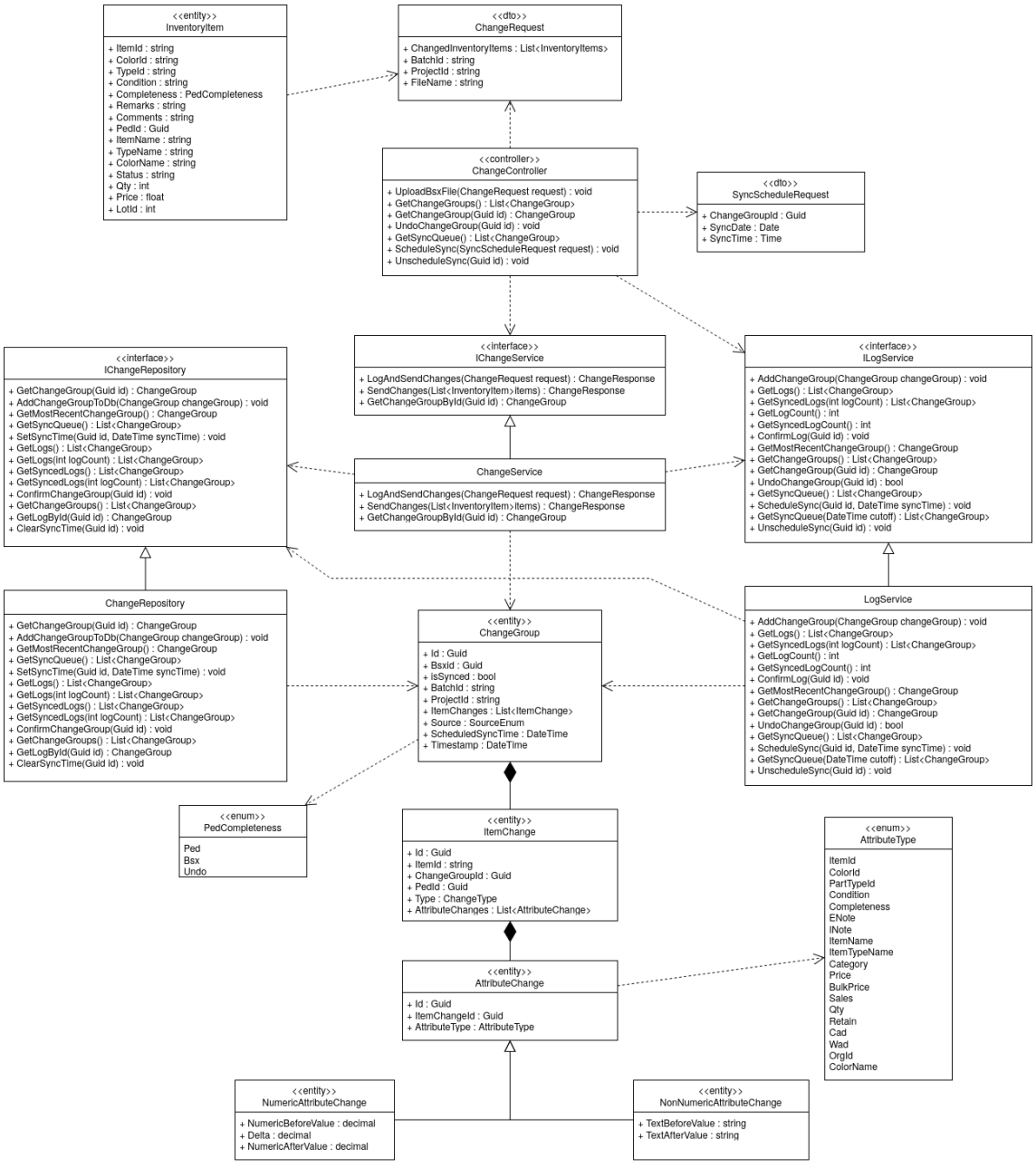



Figure 4.8: Upload flow class diagram 2

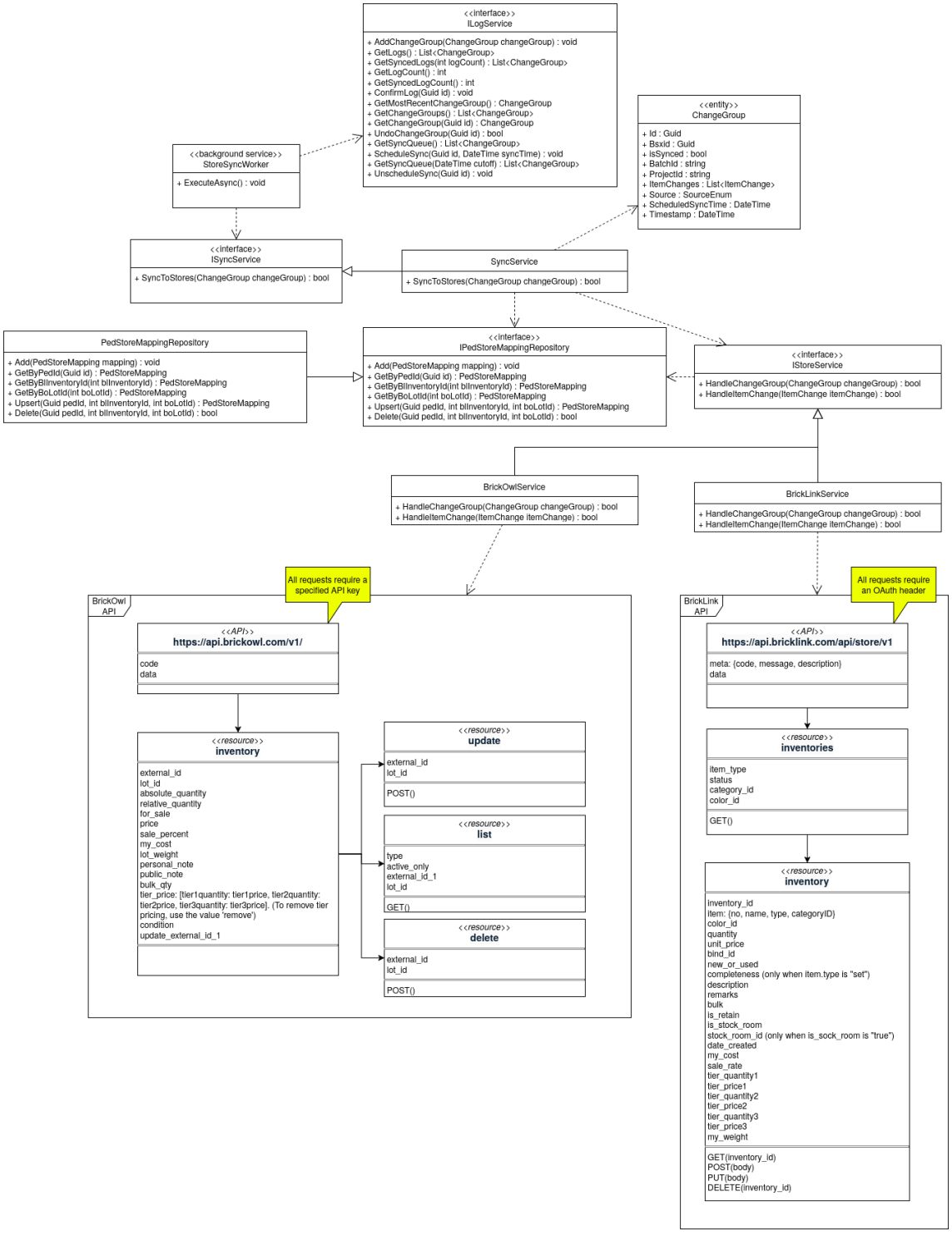



Figure 4.9: Sync flow class diagram

# Chapter 5

# Testing

## 5.1 Test Plan

### 5.1.1 Testing Objectives

The primary objective of the testing phase for BrickSyncer 2.0 is to ensure data integrity, system reliability, and seamless synchronization between the PED, the internal database and external marketplaces (BrickLink, BrickOwl). Given the system's goal of reducing human error, the testing strategy heavily emphasizes fault tolerance, data persistence, accurate parsing of BSX Files, correct error handling and robust handling of external API interactions.

### 5.1.2 Scope of Testing

Emphasis was put on testing Backend business logic (C# / ASP.NET), including BSX file parsing, change calculation, PED database transactions, log database transactions, and synchronization between aforementioned databases, as well as Integration between the React frontend, ASP.NET backend, databases and marketplaces (stores) synchronization.

### 5.1.3 Testing Strategy

Our system, by its nature, is a tightly coupled and highly impure one. It is thus impractical to try to test it purely on an input to output basis, as one would test a pure function, since for any given input to a subsystem we can expect different outputs in one or several of the other, either internal or external, systems to occur based on the states of the various other subsystems at play. Therefore, to still test the system as if it were pure and its subsystem were independent, it was decided

that mocking, i.e. simulation, of repositories would be used. The repositories are the interfaces to the persistence layer, which contains the database interactions.

Mocking these repositories allows for controlled response from the different storage and subsystems. For each service or subsystem that is under testing, all its dependencies are mapped out. In C# this is commonly known as dependency injection, which is a software design technique achieving Inversion of Control (IoC). Dependency inversion is a key factor to build applications that are loosely coupled, since it allows for higher level abstraction, without dependence on lower level implementation details [4].

According to this common design principle, the services are then divided into input and output dependencies, per test case. A given dependency may act as an input for a test case and as output for another. So, by mocking the inputs and outputs, returning either a fixed value or a range of values based on other values, the service can be tested for their functionality independently. The output dependencies are simply checked for their expected outcomes, to find out whether the test succeeded. For every service, different combinations of inputs and outputs will be checked to broaden coverage of the testing.

The mocking strategy comes in use for the integration testing level of the system, which is one of the four levels of testing that occurs in the backend. This is further described in section 5.1.4. The Integration testing covers the main use of the system. To reach the specified testing objectives, Unit testing will be used to test the functionality of individual methods inside the different services of the BrickSyncer 2.0 system. This will add to increase correct handling of data items and errors. 2 services required notable changes in the implementation, the first one being the StoreService. This test file required a wrapper called the RestClientWrapper in order to test the MOCK HTTP response received after running a method inside the StoreService. System testing will test the cohesive functioning of the system. Lastly also User Acceptance Testing will be conducted to receive input from external individuals from the client company and help discover invisible deficiencies in the system for the project team. These levels will be further explained in section 5.1.4.

### 5.1.4 Testing Levels

To ensure comprehensive coverage, the system will be evaluated across four distinct testing levels. These are unit testing, integration testing, system testing, and user acceptance testing.

#### Integration Testing

This section expands upon the general mock testing strategy defined in section 5.1.3. The integration tests for BrickSyncer 2.0 cover testing the API endpoints consumed by the frontend, as well as reliable communication and data exchange across all

Table 5.1: UnitTest Coverage

| **Service** | **Test Coverage** |
|---|---|
| BatchService | Retrieve (all) mapped batches; Handling batch non-existence; Deleting and updating batches |
| BsxService | File parsing and PED matching; Null PED match handling; Optional field parsing (ItemTypeName, ColorName, Status, Qty, Price, LotID); Decimal and completeness parsing with invariant culture; String field trimming; Malformed XML handling; Missing/whitespace-only required fields; Invalid completeness values; Database upload aborting |
| CatalogService | Validate itemID when color ID or Item ID is empty/non-existent; Handling valid part and existing itemID |
| ChangeService | Managing empty inventory item list; Existing PED item updates with response aggregation; New item creation; Multi-item batch processing with count aggregation; Exception propagation |
| LogService | Add, get, or undo a ChangeGroup; Config log using ChangeGroup repository; Retrieving synced logs |
| PedService | Creating new PED items with change logs; Updating existing items with field-level change tracking; No-change detection; Null field handling; Primary key-only requests; Database failure handling; Retrieving and mapping PED items |
| ProjectService | CRUD projects; Obtain mapped projects with field-level change tracking; Handling non-existent projects |
| StoreService | CRUD BrickLink inventory endpoint; CRUD BrickOwl endpoint with different attributes; Retrieving correct BOID from BrickOwl API with correct formatting |

services that are found inside the backend (e.g. LogService, StoreServiceDatabase). Rigorous testing of the ”happy path” is performed as well as edge cases with malformed payload, for instance.

In the system 4 main paths or flows were discovered. In order to efficiently cover all these flows of the backend, a template was made containing possible inputs, outputs and checkpoints for each flow. In addition, the required mock structures were defined including their required methods. This helped to achieve all possible combinations that were needed to have a proper testing coverage.

**Upload Flow** This flow can be divided into two parts. The first part covers the uploading and parsing of the submitted BSX file. Before the second half is reached, the user is first allowed to select which items to keep from the BSX file. The second half covers the processing of the resulting ItemChanges, which are received by the ChangeService from the frontend. Separating these two parts helps to test more thoroughly the code implementation for the upload flow, since the frontend is excluded from the equation. Figure 5.1 shows a diagram with a high level overview of the Upload flow with regards to its services and repositories.

For the first part, mocks are needed for both the BSX file repository and the PED repository. The first mock requires no functional mock as only BSX file writing is done there. The BSX mock therefore becomes a verification or assertion mock and is labeled as an output. The PED mock, is an input mock, since it needs implementation that allows it to already contain the items that will later be processed. The input for the system is a BSX file and the output is found in both the `ChangeGroup` and BsxFiles repository. Several different BSX files are used for input, including existent and non-existent PED items in the BSX file items, but also BSX files with invalid values. The resulting `ChangeGroups` are checked for among others the correct attribute changes, but also whether their type is correct, i.e. Create, Update or Delete.

For the second and last part, both the `ChangeGroup` and PED repositories are only output mocks. The incoming list of `ItemChanges` represented by the `ChangeGroup` is the input for this part of the flow, and we check in the outputs that the contents reflect what is expected from the `ItemChanges`.

**Store Sync Flow** This flow aims at viewing and approving sync items. It contains figure 5.2 to help the reader understand the flow better, since this is the most complex flow. It covers logging, syncing and the API interaction with the stores. A mock is required for the PED, since it should be able to return items as syncing pulls the values from the PED. The `ChangeGroup` mock must allow for returning the Sync queue, which reflects the PED items, and accept scheduling for these `ChangeGroups` inside the Sync queue. Lastly the Ped Inventory Mapping mock should support both setting and retrieving mappings.

The setting of the sync schedule dates leads to the items being synced by the sync

worker. The test suite inside this Story sync flow confirm the data are correct in the ChangeGroups, the correct items are send to the stores using the real APIs and that correct mappings show up in the PED store mapping mock. These methods inside the test suite cover scenarios such as syncing new and existing items to store, but also deleting them. On top of these are edge cases such as invalid store IDs.

**ChangeLog Sync Flow** This third flow cover the syncing that occurs between the external PED ChangeLog and the ChangeLog inside BrickSyncer 2.0. When a user views logs in the front-end interface, the backend will retrieve logs from the `ChangeGroup` repository, which contains logs that are synced from the PED change log which occurs when the PED is changed by systems outside of BrickSyncer 2.0. A `LogSyncWorker` periodically checks the PED for updates and places these inside the local system. To test this flow, a mock PED is required that is able to return hard=coded logs that are not yet present in the `ChangeGroup` table. The `ChangeGroup` table mock is then verified as the output.

**Audit Log Flow** Lastly the Audit Log Flow, is a flow that covers the viewing and undoing logs in the Audit Log. It uses the same mocks as described in the above ChangeLog Sync Flow, except that the PED mock is now the output and the `ChangeGroup` mock is the input. Tests in the testing suite for this flow focus on undoing different types of logs, which impact the PED. For instance an update log is reverted such that the PED is updates with the opposite deltas of the initial update. Edge cases are non-existing `ChangeGroups`, numeric null values and updating extinct PED items.

### System Testing (End-to-End)

Up to this point, the testing of both individual methods inside the code implementation, as well as the main flows inside the system have been described. However, there remains a need for testing of the system as a whole. Testing the entire workflow (uploading, syncing, undoing, syncing, etc...) from the user's perspective. Testing the full life cycle of the system, to ensure the a cohesive functioning.

To ensure this correct functioning of the system as a whole, the team is using the system like real users would. The setup for system testing is made up of using the Dockerized system from the latest stable branch. No specific flows were written on paper. This was chosen to increase the efficiency of testing as well as increasing the randomness. Issues and branches are created in case any bugs or unexpected results come forth.

#### User Acceptance Testing (UAT)

With User Acceptance Testing, the last phase of testing approach is reached. In order to validate, with real business users, that the system meets the agreed-upon requirements and is ready for real usage. This shall be conducted before the system goes live and after the functional testing. It covers bug tracking and end-to-end workflow testing.

For BrickSyncer 2.0, the team established two in person meetings with the client to perform User Acceptance Testing. The first meeting will allow for time afterwards to still fix bugs and update small changes. After that meeting and the implementation of any updates to the system, a last confirmation meeting is held. Additionally to these two in person meetings, several online meetings with the client are held throughout the development the system and also allow for user testing.

### 5.1.5 Test Environment and Tools

#### Environment, Frameworks and Responsibility

Testing will be conducted in a localized, Dockerized development environment that mirrors production but connects to sandbox accounts for both BrickLink and BrickOwl and a test replica of the PED made by the team.

xUnit was used for testing C# and API logic in .NET, and scalar was used for manual endpoint testing and mocking requests.

Unit testing was conducted continuously, as logic was implemented. Integration testing was done collectively after each module was done, while system testing was conducted in a cross-testing manner, with members testing specifically for functionalities they did not implement, in order to minimize developer bias.

## 5.2 Testing Outcomes

The tests results for all four testing levels are described below. These results help to give a clear overview of the successful working of the BrickSyncer 2.0 system and improve the confidence of the reader in the stability of the system. Some relative unimportant results have been omitted or described shortly, to reduce the amount of text required to present the testing results. Appendix B.1 goes into detail about the use of AI in formulating the tests.

### 5.2.1 Unit Tests

Unit tests were defined after all the functionalities of each service were defined and implemented. The tests were run continuously throughout the further development of the code to remain a stable functioning of the system and to avoid gaps in logic. The ChangeService required better handling of edge cases, the team for instance found out that when the attribute list of the ChangeGroup is empty, was not handled as we expected. The team ended up refusing the ChangeGroup to be created.

When running the tests in the stable version of the MVP, all 112 Unit Tests passed for the 8 different services, see figure 5.3. 10 of these Unit tests were API tests for the stores.

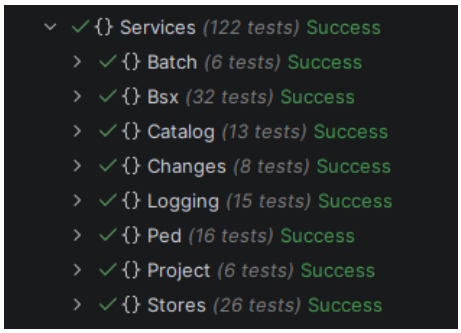



Figure 5.1: Unit Test results

On top of these results, the coverage of the Unit tests are presented in figure 5.4 below. For the service an 87% code coverage was achieved.

The remaining uncovered percentages primarily consist of functionality that is difficult to test in isolation. For instance, this include error-handling branches that depend on rare external failures. Furthermore, certain edge cases that involve asynchronous operations were excluded, as these case were thoroughly tested throuhg the intergration tests that target the specific flows.

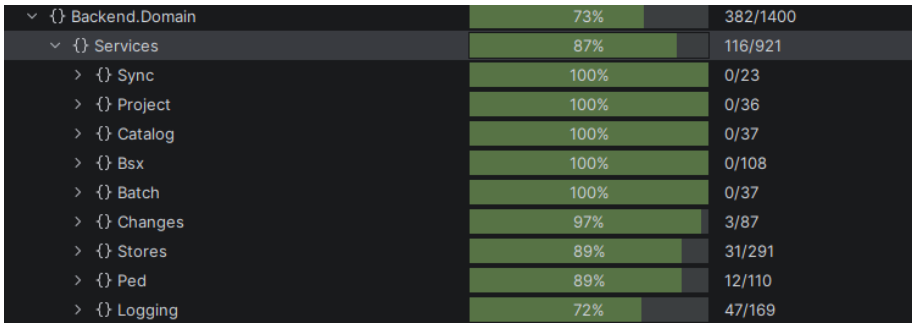



Figure 5.2: Unit Test Coverage

### 5.2.2 Integration Tests

The integration tests, formulated according to the strategy described in Section 5.1.3, resulted in four different classes containing their own test methods for their respective flow they are testing. Each of these flows is described below with their respective results.

#### Audit Log Flow

As a result, 3 main flow tests were formulated that focus on viewing and undoing regular ChangeGroups. Additionally 6 edge case tests covered flows were ChangeGroups were non-existent, numeric null values and other edge cases.

*Results* Upon executing these 9 tests our system managed to pass 7 of them. 2 Edge cases required re-analysis of the system to bring forth solutions to these edge cases. Yet after re-evaluation of these edge cases it was found that these edge cases were not realistic to happen and were therefore removed. This was due, since both edge cases would require an input that was not allowed by a previous part in the pipeline of the system, namely an empty ChangeGroup. As described in Subsection 5.2.1, these empty ChangeGroups are rejected such that they won't enter into the system. Therefore the team ended up with just 7 test methods for the Audit Log Flow.

#### ChangeLog Sync Flow

4 different tests were conducted to cover this flow, including regular syncing, handling invalid values and ignoring ChangeLogs that originated from our own system.

*Results* Some of the tests failed, for instance by checking invalid values in the ChangeLog. After updating the code, all tests ended up succeeding.

#### Store Sync Flow

The largest flow in our system required the most extensive test methods to ensure proper results. This flow tests the syncing of items, i.e. ChangeGroups from the sync queue, to the stores were the items are sold. For the regular flow, the ChangeGroup repository is filled with ChangeGroups that are not synced and require the worker to trigger the syncing process. Along the way, the correct values must be confirmed in the repositories to reduce errors that remain on a high level, when tests will fail and help pinpoint bugs faster. Finally the stores are evaluated to return their expected values upon interaction through the API.

9 different tests were born from the formulation of the brainstorming for potential issues in any of the milestones in the flow. One test uses a mock API to test the main flow without interacting with stores. Next to this test, 8 live tests interact with

the stores covering potential issues such as invalid store IDs, PED items missing and only 1 store succeeding. To increase the clean interaction with the stores, each live test ends with clearing any items from the stores that were created in the respective tests.

*Results* After a few iterations of running the test set with 9 methods, the system ended up handling all 9 cases successfully. A major issue that arose while testing was the case were one of the stores failed and the other succeeded. An improvement to the implementation was required to prevent labeling the ChangeGroup as synced when not all stores have succeeded. Another noteworthy bug that was found, was the completeness attribute used in the BrickOwl Implementation. Due to a difference in naming conventions between both marketplaces, invalid condition values were attributed to BrickOwl. A code review resulted in using a switch statement to link similar conditions and containerizing distinct conditions to enable expected behavior for both marketplaces.

Running this test set takes over a minute due to the asynchronous nature of the tests brought forth by the API interaction and scheduling time of the syncing.

#### Upload Flow

In the last flow of the system the flow of uploading a BSX file to the repositories are covered. Once the BSX file is uploaded, it is processed by the BSX controller and subsequently submitted to the change controller that handles the changes and stores them in the repositories.

8 tests were specified that covered the regular flow as well as unusual cases that might cause the system to break. For instance negative price and quantity in the BSX file and numeric values that exceed the threshold.

*Results* The test set for this last flow ended up all succeeding. The different edge cases were handled well by the system and successfully cover a large part of the code implementation that handles the BSX uploading. The successful Unit tests cover most of the left over code.

### 5.2.3 System Tests

Rigorous manual system testing was conducted by all team members, covering all the expected usage scenarios, as well as edge cases such as malformed or invalid inputs. No major issues were found, increasing our confidence in the system as well as the strong combination of both Unit and Integration testing. All major communication paths of communication had already been tested during integration testing, and all internal logic had already been tested by the Unit tests.

One notable issue found by one of the team members, was that changing either the price or the quantity in the BSX file upload interface caused this updated value to

become string inside the system. This issue was resolved, as well as other similar small bugs. The result was a system that achieves at minimum the MUST HAVE requirements specified in Section 3.3.

### 5.2.4 User Acceptance Tests

User Acceptance Testing was conducted in collaboration with representatives from BrickWorkz. Feedback was overwhelmingly positive; The users reported that the React-based interface provided a significantly more intuitive and error-resistant workflow compared to their legacy command-line tools. The intuitive interface reduces the need for technical expertise and saves valuable working hours by an increase of efficiency when using the system.

Although the system worked as expected and was received positively, still some remarks were made by the client in the first meeting. Below a list is given of some of the feedback that was received and implemented by the team after the first meeting.

1. Primary Key (PK) Columns should not be editable, but still shown
2. Let the `/api/change` route reject POST requests if the PK in one of the items is not present
3. When editing the uploaded BSX file for the quantity, price or other numeric attribute, the interface should have a button that toggles between absolute value or simply adding on top of PED
4. Allow for removing or resetting a schedule time in the Sync Queue
5. Display the PK label to columns on both preview and edit color of an item
6. When the sync queue is empty, the view should be improved, for instance by textual information
7. Delimitate columns better in the edit flow
8. Pressing the image to select a row, next to the ID

In the second meeting, the newly updated system was shown and some last hidden features were explained to the client. Helping the client to better understand BrickSyncer 2.0 and allowing for ease of use. No further bugs were found.

# Chapter 6

# Process

This chapter provides an overview of the execution, documentation and final outcomes of the project. It begins by explaining the team's organizational strategy, detailing the roles of individual team mates, and the internal and external communication methods. Then it moves on to describe the method in which Scrum was performed, and the choice of using Github Projects as a project management tool. Following, the timeline of the project is described in Section 6.2 highlighting the processes of requirements engineering, design and implementation. The documentation section explains how this project is documented, covering both in-code and static documentation, and motivating the choices that were made during its writing. Finally, the results section analyzes the end result, clearly stating what goals were achieved, which features were implemented and which not, and the reasons and motivations behind them. It also covers future work, quantifying the future integration and development work the client will need to undertake following the hand-off, and why each aspect of it is necessary, as well as the current limitations of the system.

## 6.1 Organization

The first organizational step the team took was creating a team-contract, where all members agreed on matters such as: the use of AI, expected response time, roles and accountability. Decisions regarding coding standards and conventions were made, and the team agreed on adhering to the best practices of using version control. The organizational platforms were chosen during the same meeting, together with the desired communication channels. The main purpose of establishing this contract early on was to encourage accountability. This ensured everyone knew where they could start contributing.

### Roles

The division of roles was done considering the strengths of each individual. Alisa had the scrum master role and was responsible of organizing daily stand-up meetings, keeping track of assigned tasks and stories, making a planner for the week, and supervising the storyboard, product backlog, and sprint backlog. Tudor was team-lead, and he was in charge of ensuring team-members adhere to their roles, communicating with the client, submitting assignments, checking deadlines, and tracking documents. Sebastian was the team's tech lead, he was in control of the system's design, code quality and implementation. Ewout was charged with ensuring that the report is written and updated as the system is being built, and it's quality is meeting the team's standards; this made him the team's report lead. Boaz, the meeting lead, took minutes and notes for each meeting, and prepared the meeting agendas. Radu was the test lead, and he was responsible with creating a testing strategy, and ensuring test coverage suffices.

### Team communication

The communication channels agreed upon by the team were Discord and WhatsApp. Discord was used to facilitate online meetings, announce important updates, schedule meetings, store access links and track resources and references. WhatsApp was used for quick, informal communication.

No specific structure was followed in the organization of in-person meetings, due to the fact that the team members had conflicting schedules. On average, the team met 2-3 times a week in person. For every day in which the team did not meet in-person, online meetings were held for the daily stand-ups. Thus, progress was being closely monitored at all times.

In the team contract the expected reaction time was established to be of 2 working days.

### Client communication

Communication with the client was done through emails, Microsoft Teams meetings, and sometimes, even by phone calls. Weekly video call meetings were held every Thursday or Friday. In the first stages of the project these meetings served as requirement elicitation interviews, and during the design and implementation phases they turned into client review sessions. During the rest of the work week, the team communicated their questions to the client through email. When immediate answers were needed for system design questions, communication switched to phone calls, with the encouragement of the client.

### Scrum

To effectively manage collaboration, the team agreed to follow the scrum methodology. This organizational framework allowed for an iterative implementation strategy, where tasks are tracked and assigned daily, and goals and progress are monitored weekly. The team attended daily stand-ups, in which everyone was updated on the individual progress of members, their blockers and what tasks they plan on working on next. Weekly scrum-planning meetings were attended at the start of the week, to choose what tasks from the backlog should be ”brought out” and implemented during the upcoming sprint. In a similar manner, during sprint review sessions the team reviewed the work finished in the past sprint, and estimated how much workload remains, thus setting the pace.

### Github projects

At the first meeting, the team chose Trello as its project management tool. Upon further research on the topic of collaboration tools, a migration was made towards Github Projects. Using this platform, the team developed an efficient strategy for managing and tracking progress, which indirectly sped up the development process.

The most important ”views” (view = customizable layout that allows tasks and issues to be visualized and managed in different formats) of our system were: product backlog, current sprint and organizational (non-code) (figure 6.1).

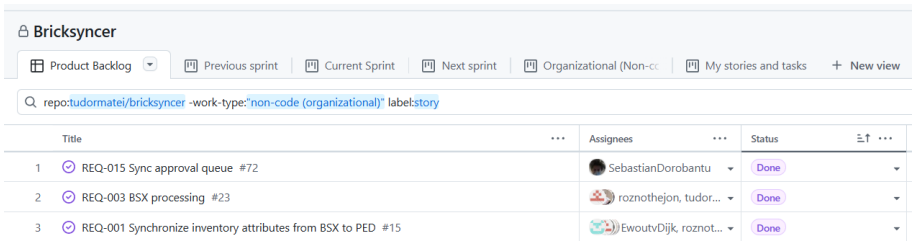



Figure 6.1: Github Projects - views

Issues ”live” in the backlog until the scrum master bring them to the current sprint, under the ”todo” tag. Team members assign themselves to the issue during the stand-up, announcing they are taking over the implementation. They assign it the ”in-progress” tag, and a branch is created to implement the feature on. Upon finishing the task, the team member moves it into ”testing”. After testing is done with at least another team-mate (or in the case of important features, more/all team-mates), the task can be moved into done, and the branch is merged into main. Sometimes, incorrectly implemented issues are re-opened and moved back into ”todo”.

Two labels were created for separately tracking the two organizational aspects of the project: non-code-related tasks (internally referenced as ”organizational” tasks), and code tasks. Organizational tasks are related to interface designing, diagram making, document writing, report writing, or the making of presentation slides.

Code issues follow a more intricate structure. Functional requirements were turned into stories (figure 6.2), and placed in the product backlog.

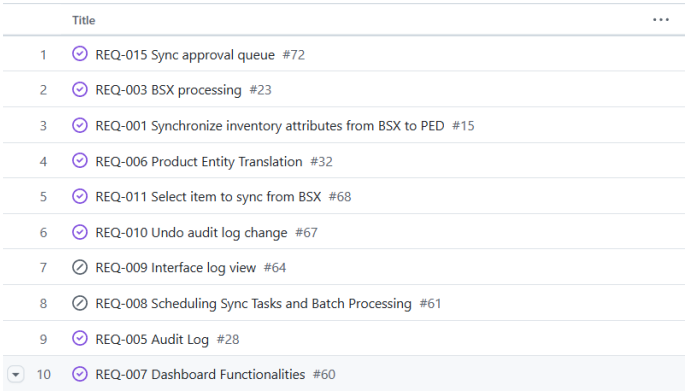



Figure 6.2: Github projects - stories

Story issues are broken down into task issues (figure 6.3), which detail the concrete implementation of the parent story.

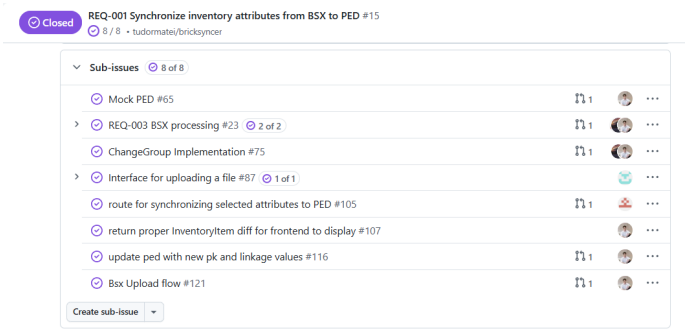



Figure 6.3: Github projects - tasks

The description of issues is written using a template that lists the sub-tasks and

tasks, together with the acceptance criteria. (figure 6.4).

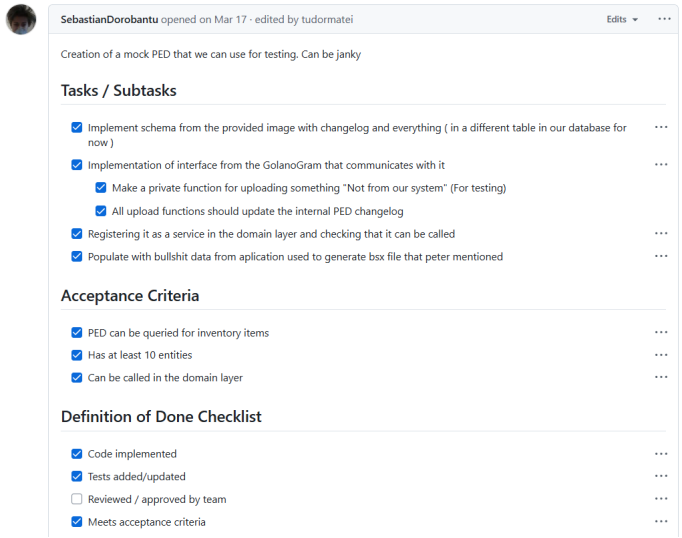



Figure 6.4: Github projects - issue description

The fields created to help categorize issues are: status, work type, sprints, priority, start date and end date. ”Status” (figure 6.5) represents the stage of the task, and it tracks progress: todo, in progress, testing, done, cancelled, blocked. ”Work type” is of two types: code and non-code and it works as discussed above. ”Sprints” is an interative field used for visualizing the sprint specific workload in the three views: previous sprint, current sprint, next sprint. ”Priority” sets the importance of the issue on a scale from ”low” to ”critical”. ”Start date” and ”end date” set the expected duration of implmentation.

The labels ”status”, ”work type” and ”sprints” are the only labels used consistently and continuously during the course of development, with the other three being used for short periods of time.

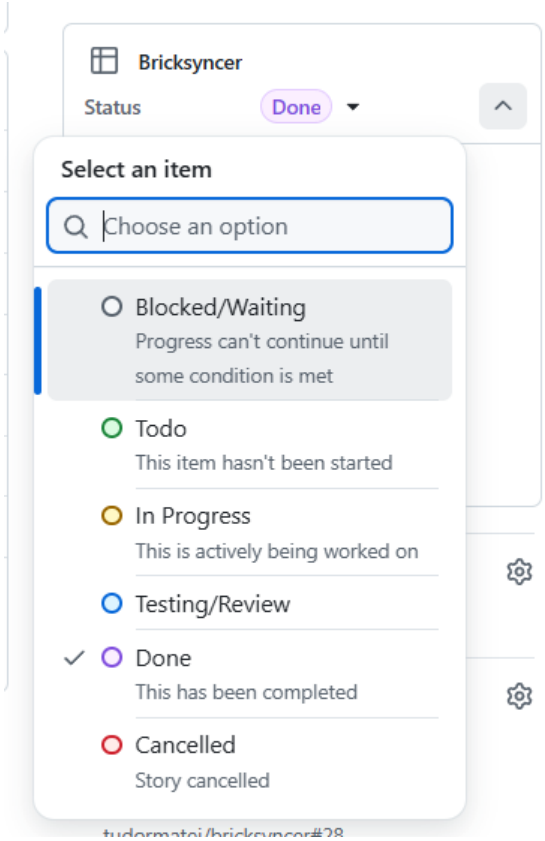



Figure 6.5: Github projects - status field

#### Overleaf

Overleaf served a dual purpose for the team. It was used as both a real-time collaborative text editor, and the main document repository. This way, the team kept all meeting reports, together with the project proposal (containing the system's requirements), final report and team contract in a single place.

## 6.2 Timeline

### 6.2.1 Requirements engineering phase

As noted in chapter 3, the first phase of the project was that of requirements engineering. This phase was intended to last approximately two weeks, but it drew out significantly longer than expected, lasting approximately 4 weeks. The iterative cycle of requirements specification and validation happened relatively fast internally. The bottleneck, in this case, was mostly a number of misunderstandings between the team and the client regarding some technical details, as well as limited opportunities for meetings.

### 6.2.2 Design phase

The design process started immediately after the project proposal was approved, around week 5 of the module. The proposal now acted as a foundation for the rest of the project, and high-level designs were drafted of the system.

Design was initially done from a behavioral standpoint, with use case, activity, and sequence diagrams being important tools in this phase of the design.

After several meetings in which the design was presented to the client and refined, structural design started. In this phase, the team drafted initial designs of the main structures and entities of the system, slowly building towards a complete, detailed structural design of the system, which could then be implemented.

### 6.2.3 Implementation phase

Actual development work began around week 6 of the module. Although the implementation phase started at the time, design work continued in the background to adapt to various issues that only then came to light. The team tried to avoid this situation, but shifting requirements and unclear technical details regarding the PED made it difficult. Regardless, implementation work continued at a good pace, speeding up around week 8, when the foundational details of the system were already in place.

One issue the team ran into during this phase was the lack of API keys for the stores and credentials for the company's database, on which the translation tables were located. Once they were received, further delays ensued because some of the details were wrong. Once those issues were resolved, store implementation work started.

Most of the basic functionalities were working at the end of week 8. Feedback was received from the client on an early prototype, including additional requirements - some of which were feasible and ended up being implemented. By the end of week 9, a completely working MVP was ready, and development work ceased except for minor fixes and clean-up work.

## 6.3 Documentation

The project is documented in two main ways: the first one is via in-code comments that are meant to clarify the purpose of each method and class, and the second one is the comprehensive, structured documentation that is provided along with the code.

### 6.3.1 In-code documentation

The in-code comments were written at the time of development. According to the agreed-upon coding conventions, the code should be self-documenting. That is, the naming of the services, classes and individual methods should reflect their purpose and usage, as to allow for easier development and maintenance, especially within a team where not all members work on the same pieces of code. However, in practice, due to the complex nature of the project and the architecture, it was found that the use of explicit naming conventions was not enough to guarantee total clarity. This is also due to the fact that although the name of a method may explain its immediate scope and purpose, it does not provide any insight into its larger rationale, where exactly it fits into the system and why it was made the way it was.

To address these shortcomings, it was decided that each team member would document their code via extensive comments. These comments should explain the arguments and output of a function, where and when it should be used, and, in case it is not obvious, why the function exists in the first place.

### 6.3.2 Standalone documentation

Although these in-code comments are helpful to maintainers and developers during the act of actually coding, they are not convenient for gaining a larger understanding of the system as a whole and how it all functions together. To that end, it was decided that a standalone full documentation is necessary. This documentation was written after all major design decisions and implementation had taken place. It acts as a manual to whomever may need to understand how the system works, how it is meant to be used, and how to maintain and expand it.

The standalone documentation consists of the following sections:

**Architecture:** Explains the overall system design and how the different components interact. It covers the frontend, backend layers, database integration, and how services communicate through Docker.

**Development:** Provides guidelines for developing within the project. This includes the development workflow, running the application locally, coding practices, conventions, tools used by the team, and how frontend and backend changes are structured.

**Database:** Describes the PostgreSQL database setup, schema management, and migration workflow using Entity Framework. It also explains how to inspect and interact with the database using Adminer.

**API:** Documents the backend API structure and how the front-end communicates with the backend. This includes routing conventions, request flow through backend layers, and how to test endpoints using Scalar.

**Testing:** Explains how the project was tested, and how to run the actual testing commands. It also covers the testing strategies used to ensure reliability and code correctness.

**Deployment:** Explains how the project can be deployed using the existing Docker setup.

## 6.4 Results

The final deliverable of this project is a functional containerized web application that fulfills the system goals as defined in the chapter 3. To this end, all the "Must" functional requirements are fulfilled, as well as most "Should" and some "Could" requirements. The non-functional requirements are all fulfilled, except for NFR-001 and NFR-002 , which are partly fulfilled.

### 6.4.1 Functional Feature/requirement status

Throughout the implementation process, prioritization of the features and requirements was done primarily based on the MoSCoW categories. Despite this, when it came to sorting features with the same category, priority was given to the features that were deemed most important to fulfilling the system goals. Thus, it was the case that some requirements deemed less important were de-prioritized in order to assure the timely delivery of the MVP.

Table 6.1 depicts the status of each major functional feature, and the requirements associated with it.

#### "Must/Should" Functional requirements that were partly implemented

**REQ-016 - Store configuration interface:** Initially, the interface was supposed to include the functionality of configuring web stores. In practice, this would have meant having the ability to change the API keys. However, due to the stores' differing authentication methods and the requirement's non-urgent nature, this interface was not implemented anymore. This decision was taken in consultation with the client. The store credentials are stored in an environment variable and can be modified there. This also avoids unnecessarily exchanging sensitive information, reducing the system's attack surface.

Table 6.1: Functional Features Traceability & Status Matrix

| **Core System Feature** | **REQs** | **Scope** | **Status** |
|---|---|---|---|
| Marketplace Sync | 002, 006 | Must | **[Implemented]** |
| BSX Input & Pre-Processing | 003, 011, 013, 014 | Must/Should | **[Implemented]** |
| Logging & Audit System | 005, 009, 010 | Must/Should | **[Implemented]** |
| Internal Database & (PED) | 001, 004, 019 | Must/Could | **[Implemented]** |
| Sync Control & Scheduling | 008, 015 | Should | **[Implemented]** |
| Seller Dashboard | 007 | Should | **[Implemented]** |
| Configuration & Meta-Data | 012, 016, 017 | Should | **[Partial]** |
| Store configuration wizard | 018 | Could | **[Not Implemented]** |

**REQ-017 - Retain setting interface:** This requirement refers to an interface element that would allow toggling ”retain” behaviour for the stores and PED. When enabled, items would have their quantity set to 0 instead of being deleted. This is now controlled via an environment variable. This behaviour is currently only implemented for the PED, and not for the stores, due to time constraints.

**REQ-006 - Internal translation:** The translation between the BrickLink and BrickOwl standards was supposed to be implemented based on translation tables to be provided by the client. These tables were provided very late in the implementation phase and did not provide the full functionality needed for this translation, and so the team was advised by the client to only provide empty methods for translation, to be implemented at a later date. Despite this, an alternative translation method was found, making use of BrickOwl's API. This method works for all color Ids and the vast majority of part Ids, but some Id's appear to be missing, and cannot be translated using this approach. Thus, this requirement is considered as being partly implemented.

#### ”Could” Functional requirements

**REQ-018 - Future store expansion:** Initially, the possibility of an interface that would facilitate the setup of new stores was discussed. The client emphasized that it was neither expected nor necessary, and that they would instead prefer the ability to implement new stores directly in code at a later date. This requirement was marked as ”could”. Due to the design of the system and the vast differences between external stores, such a setup wizard indeed proved impossible to implement.

### 6.4.2 Non-functional Requirements status

Table 6.2 depicts the agreed-upon non-functional aspects of the system, the associated requirements, and their current status.

Table 6.2: Non-Functional Requirements Traceability & Status Matrix

| **Quality Attribute & Focus** | **NFRs** | **Scope** | **Status** |
|---|---|---|---|
| Architecture & Maintainability | 005, 010, 011, 012 | Must/Should/Could | **[Implemented]** |
| Compatibility & Portability | 008, 013 | Must/Could | **[Implemented]** |
| Security & Compliance | 003, 009 | Must | **[Implemented]** |
| Performance & Scalability | 004 | Must | **[Implemented]** |
| Observability & Logging | 007 | Must | **[Implemented]** |
| Usability & UX | 002 | Must | **[Implemented]** |
| Reliability & Recovery | 001 | Must | **[Partial]** |

All but one non-functional requirement have been fulfilled. It should be noted that requirements 003 and 009 are implicitly fulfilled by the simplicity of the system - the security requirement is fulfilled because there is simply no sensitive information exchanged or stored to encrypt, the API keys are statically stored in a local .env file and never leave the back-end, and with the system being strictly internal, HTTPS between the back-end and front-end and authentication were not necessary. HTTPS and secure authentication are used in all communications with the stores.

**Requirement 004** actually relies very little on the BrickSyncer system, with the main database, the PED being external. All the internal databases use PostgreSQL, whose capabilities vastly outmatch the maximum load this system could ever expect. Given that there are no bulk operations, and the volume of stored data is minimal, the requirement of being able to store up to 150.000 entities is easily fulfilled.

### Partially implemented Non-functional requirements

The only non-functional requirement that is not fully implemented is NFR-001, which refers to reliability and resiliency against crashes, power loss and other such events. It is not complete in the sense that the system in its current state cannot guarantee complete fault tolerance. However, the system does provide a framework for implementing it.

The design of the system was built around resiliency. The isSynced attribute of the changegroups was initially thought of as a resiliency measure - it would only be set true after a successful sync was confirmed. At startup, the system checkes for failed syncs and unconfirmed changegroups, and syncs them if any were found. To support this functionality, the store service implementations include reliability features, such as tracking which items were successfully synced and which failed. A shortcoming of the implementation of the system does however come in the lack of the ability to detect *partially* successful syncs which can result in some API calls being made twice. These features were not implemented due to the limited time available, and the team's focus on fulfilling the functional requirements first. However, these points are documented and the client has been made aware of them, and are thus going to be implemented by their internal software team according to their needs.

### 6.4.3 Future work

Although almost all requirements are fulfilled and the end product is a functional tool that can be deployed, work remains to be done by the client.

#### Integration work

In its current state, the system makes use of a "mock PED" - because the PED was not complete at the time of development, it was determined that the best course of action would be to implement a PED replica according to the designs provided by the client. This is not technically in the scope of the project, but it would not have been feasible to develop and test it without a working database.

To facilitate integration, the PED replica was decoupled from the rest of the system as much as possible. The PED repository provides an interface for interacting with it, and that is the only point of contact between it and the rest of the system - replacing it would require only implementing the PED repository to work with another database. It was discussed and agreed upon that this integration work would be done by the client after the completion of the project.

One possibility that was discussed was that of the client deciding to use the replica PED instead of their in-house developed PED, depending on the features and functionality the replica provides. As of the final client meeting, it has not yet been decided if this will happen.

In order to facilitate the deployment of the project, a separate "production" docker-compose file was provided to the client along with the project and the development setup. This file runs the project without the logs and development aids such as hot-reload, database and API front-ends, etc.

**Further development**

As discussed above, some functionalities will require further work by the client to be fully implemented or customized. The following areas are expected to undergo further development work on the client's side after the end of the project:

**Remove related sync items from sync queue:** Currently, if two changegroups related to the same items are queued for sync, and one of them gets synced, the stores will be updated with the latest information, but the other changegroup will remain in the sync queue. The client may wish to implement a removal mechanism that "cleans up" related entries from the sync queue, so that it does not become cluttered over time.

**Adding new stores:** In the future, it is likely that the client will expand their operations to external marketplaces other than BrickLink and BrickOwl. In that case, they may make use of the modular architecture of the system to add synchronization to these stores as well, without needing to rebuild any part of the system.

**Implementing proper translation and validation:** Currently, the system's translation from BrickLink to BrickOwl Ids works most of the time, but not always. This has been discussed with the client, and the current level of implementation exceeds what was agreed upon. Still, the client will need to implement reliable Id translation themselves, probably using their internal translation tables. Additionally, they will need to enable and complete the validation features that are currently disabled.

**Duplicate item detection:** During the final feedback meetings, the client expressed their concern for the scenario in which a user uploads a BSX file containing existing items, but makes a typing error in the comment field of the item, such as a stray whitespace. Because the comment is a primary key in the database (a design choice made by the client), this would create a duplicate item. For preventing such errors, the client requested a feature that would check the PED for items with similar primary keys, but different comments, and alert the user of such occurences. Unfortunately, this request had to be denied, because the very late stage of development at the time would have made it unfeasible to implement. Still, the client may wish to implement this feature themselves in the future.

### 6.4.4 Known limitations

Although the project successfully fulfills most must and should-class functional and non-functional requirements, there are a few known limitations and edge cases.

These do not represent design flaws or bugs, but rather aspects that have not had the time to be fully polished.

**Undoing delete logs:** Currently, undoing delete-type logs is not possible. This is due to the fact that delete-type changelogs do not store the values of the deleted items, and thus do not contain the information necessary for them to be re-created in order to undo the action. This decision was taken because storing this data would have introduced a significant amount of duplication of PED data due to the fact that the PED is external. Having a single centralized database would have enabled this functionality, among others, with much greater efficiency. Still, this feature can be implemented by the client without minor design changes if desired.

**Syncing logs with deleted items:** Due to the fact that the sync queue is not being ”cleaned up” of outdated entries (entries referencing non-existent or already synced items) after syncs, users can try to sync changelogs referencing items that have since been deleted. This results in an invalid request to the stores and is not properly handled. Ideally, this would be resolved by removing those entries from the sync queue.

**BrickLink bulk create not used:** The BrickLink API offers the functionality of creating items in bulk. This has the advantage of creating a large number of inventories with a single call. While this feature is not essential for functionality, the fact that the system currently makes one call per item could theoretically lead to reaching the daily API limit of 5000 calls if a very large number of items are created.

**Internal Id translation:** As discussed in the previous subsection, the current temporary BrickOwl Id translation method does not work for all Ids.

# Chapter 7

# Evaluation

In this chapter, an evaluation on the organization of the project is provided, by reflecting on the different project phases, and the collaboration inside the team and with the client. Insights into the communication with the client, and into the team dynamics are expressed and analyzed, while also showcasing the stakeholder feedback and interactions.

## 7.1 Project phases

The development lifecycle involved various challenges, that were presented into detail in the previous chapters of the report, together with the associated design choices. While the team successfully delivered an MVP of the Bricksyncer 2.0 system, different choices could have been made in order to make the process easier. The following section evaluates the three project phases: requirements engineering, design and implementation, in a reflective manner.

### 7.1.1 Requirements engineering phase

As described in Subsection 6.2.1, the requirements engineering phase was hindered by several misunderstandings in between the client and the team regarding certain technical details and terminology, alongside limited meeting opportunities.

One of the early issues encountered in this phase was a misunderstanding in vocabulary. The software team that developed the client's legacy system created an internal terminology that describes their software. Even though BrickSyncer 2.0 is not based on the previous existing system, the client was still using terms specific to the old system's in his lexicon. The client never mentioned the existence of a nomenclature, and until the team cleared up this misunderstanding, the progress was being slowed down. An example of such a term is the "PED", which turned out to be a conventional database, as was deduced after 3 weeks. The takeaway of this experience is to first clarify the client's terminology early on in the project.

While weekly meetings would have theoretically sufficed, the received feedback often drifted in scope, not fully clearing up the confusion. Moreover, materials were often received from the client the very night before a meeting, giving the team very little time to understand and incorporate them, and email communication during the rest of the week tended to be very sparse. All these factors led to a slower and more frustrating experience than would have been ideal or expected, at least during the phases where client involvement was most important.

### 7.1.2 Design phase

The design phase started after the project proposal was approved by the client. During this phase, the team first worked on defining the intended behavior of the system and subsequently its structure.

As the name of the module would suggest, the design phase was the phase where most effort was spent. Emphasis was put on finding smart, elegant and efficient solutions to problems rather than just getting something to work. This goal has been largely met, with especially much thought being put into the `ChangeGroup` system, which is closely tailored to its purpose. Rather than simply storing an item's old and

new version, which would have been much simpler but very inefficient, the current system stores strictly what is needed for achieving all the desired functionalities, in a highly organized hierarchical manner. This strict organizing gave confidence to the team by establishing a solid foundation.

Throughout this phase, the main challenge has been navigating the pre-existing PED design. It would have been more elegant to simply have one central database, with one change log through which everything passes. However, this assumption could not be made, which once in a while resulted in frustrations inside the team. An internal change log had to be designed, which became the `ChangeGroup` system. To account for the possibility of changes occurring from sources other than the BrickSyncer 2.0 , a separate service had to be written just to keep these two change logs in sync.

Unexplained ambiguities also did not help the progress. For example, as was discovered around week 6, the name of the BSXId attribute of a PED entry does not stand for "BSX identifier", but for "BSX Item Data", which is a reference to an entirely separate table containing a collection of information about BSX items, which the team was not aware was available, and had not accounted for. Such discoveries introduced constant changes in design, hindering progress. The team questioned the proper communication of the client once more. Despite this, a complete, high-quality design was achieved in time for implementation.

### 7.1.3 Implementation phase

The implementation phase started with the implementation of the mock PED and all the entities that were necessary for the exchange of data between different services. This laid a solid foundation for the rest of the services, as these could be divided between team members, who would need only to use the already existing common infrastructure. This was a significant boost for the progress and was a great relieve for all members.

One obstacle that the team faced during this phase was that of inconsistent use of organization tools. Although the team had already agreed to respect the discussed organization procedure even where it seems unnecessary, some members did not always do so. This led to a significant amount of wasted effort. For example, due to inconsistent meeting attendance and not following the assigned GitHub issues closely, an implementation of the logging system had to be entirely rewritten because the person who wrote it did not follow the design and it was not useful in the context of the rest of the system.

Occasionally, mismatches in the members' schedules also introduced inefficiencies in planning and implementation, although this was not a problem whenever the organization tools and procedures were respected, as these allowed for efficient asynchronous work.

Overall, collaboration throughout the implementation phase was reasonably effi-

cient, with the devised planning framework and the close following of the Scrum methodology being a huge help.

## 7.2 Team Cooperation

As described in Section 6.1 each team member was given a role, together with a set of responsibilities. This turned out to work surprisingly well. The team experienced great efficiency in its daily tasks due to this strategy. Even though there was a clear division of tasks, a sense of shared responsibility and accountability was kept and ensured the expected delivery of the project. In events of sickness or other reasons for non-availability, the team was quick to take upon each others roles. The shared goal of delivering a proper product was an important factor in this great team cooperation. An example of the cooperation is found in the helping of writing notes by other team members when the received information was overwhelming. Additionally, team members helped to discuss the testing strategy together to prevent putting too heavy of a workload on any one individual.

The team always showed a good sense of responsibility regarding meeting deadlines. No incidents happened in this sense. When needed, the team collectively put in more effort to speed up the work pace. After the design phase was over and the team wasn't depended on client feedback as much, the team could start working in its own pace. With little time left for the implementation phase, the team started working longer days to finish the rest of the project. During this period, the team met daily in person. This resulted in a great amount of progress during a short period of time.

Apart from this overall great team cooperation, some deficiencies could be found when over viewing the whole process. First of all, due to informal appointments and unregulated meeting times, the internal meetings were sometimes inefficient, as either members were missing or arriving in a time gap of around one hour. Secondly, the remote working of some of the members, resulted in design choices not being clearly communicated. Due to this fact, at times, parts of the code needed rewriting or sometimes complete removal. Another encountered issue is found in the usage of GitHub Projects, when assigning members to specific tasks. Sometimes, team mates would forget assigning themselves to tasks and work was done twice. At times, GitHub tasks were not descriptive enough. Using clear SMART [1] task descriptions would have resulted in better overall understanding of the assigned tasks. Though, the overall work atmosphere inside the team was great and helped to give a positive attitude towards the, sometimes tedious, work that had to be executed.

[1] SMART stands for Specific, Measurable, Achievable, Relevant and Time-Bound.

## 7.3 Stakeholder Interaction

BrickWorkz's CEO, the client and main stakeholder of this project, possesses a technical background. This was both an advantage and disadvantage. In the first couple of meetings, in the requirements elicitation phase, a lot of technical details were given out by the client, without the team getting a real understanding on the broader goal of the system. This reduced meetings efficiency, and it resulted in the team crafting the system's requirements by themselves.

The client would sometimes come up with his own system design, technical implementations, and project proposal, causing confusion in our previous understanding of the system. Further in the project timeline, communication improved, as explicit approval of design choices was requested from the client. A possible way in which the team could have tackled these misunderstandings better, would be more in-person meetings.

The advantage of the technical background of the client, was the fact that in the end he had a good understanding of technical terms and structure of the system. This was a great upside and was received positively by the team, as it eased communication regarding technical details, which would have been cumbersome with someone with no such knowledge.

At the time of delivery, the client expressed his satisfaction with the final product, noting that it is a net improvement over the current solutions, and that he expects it to be fully integrated and in use within a few months. He was made aware of all the necessary implementation and development work that needs to be done, and he assured the team that this would not be an obstacle, especially given the thorough documentation and the thought that was given to designing an expandable, maintainable system.

# Chapter 8

# Conclusion

BrickWorkz is a social enterprise whose mission is to offer employment and experience opportunities to neurodivergent youth. Their operations revolve around sorting and selling second-hand Lego parts on third-party marketplaces, for which they keep a sizable stock. In order to efficiently manage this stock and keep it in sync with the marketplaces, they need specialized software. Existing solutions are complex, unforgiving and generally unsuitable for BrickWorkz and their employees.

As part of the Design Project module, the team was tasked with developing a more suitable piece of software to fill this gap. This solution should take the form of a user-friendly web app, with its core functionalities revolving around resiliency and protection against both user error and system errors, ease of use, and clarity. Thus, BrickSyncer 2.0 was born.

The final deliverable is a fully functional containerized web app that fulfills all of those goals. It is sleek, intuitive to use, forgiving and, most importantly, syncs the client's stock to external marketplaces. It is built using a modern tech stack, employs a modular, easily maintainable and expandable architecture and is highly portable.

All of the must-have and most of the should-have functional requirements have been met. The system has passed rigorous unit, integration and system testing to ensure that the highest possible quality is delivered to the client. Most importantly, following User Acceptance Testing, the client noted that the developed solution mitigated the usability issues of their old legacy system.

Throughout the project, the team faced numerous challenges. The start was slow, with misunderstandings during the requirements elicitation phase somewhat hindering process. Despite this, a solid foundation of requirements was built, striking a balance between the client's requests, technical fesability and timely delivery. During the design and implementation phases, the team successfully navigated obstacles such as an incomplete and unavailable PED, with vague specifications and difficult designs to build around. The solution was not only successful in dealing with those challenges, but, as noted by the client, might even be adopted as the main PED.

The MVP is now complete and ready for hand-off to the client. The thoughtful architecture will allow the client to transition to using this tool with minimal integration work, despite its dependency on external systems. Besides this, some minor features are still to be finished by the client's software team, a task that will be facilitated by the detailed documentation that is shipped along with the product. This documentation also acts as a guide for future developers looking to expand or adapt the system.

Throughout this module, the BrickSyncer 2.0 team successfully navigated a complex domain and solved a number of difficult challenges, eventually delivering a functional product that will help a social enterprise offer better opportunities to the young people who need it most.

# Bibliography


[1] *Bricklink store api documentation*, Accessed: 2026-04-17, BrickLink / The Lego Group, 2026. [Online]. Available: https://www.bricklink.com/v3/api.page.

[2] *Brick owl api documentation*, Accessed: 2026-04-17, Brick Owl Ltd, 2026. [Online]. Available: https://www.brickowl.com/api_docs.

[3] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Professional, 2003, ISBN: 978-0321127426.

[4] Microsoft, *Architectural principles – dependency inversion*, Accessed: 2026-04-16, 2023. [Online]. Available: %5Curl%7Bhttps://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles%5C#dependency-inversion%7D.

[5] R. Griebl, *Brickstore*, Accessed: 2026-04-17, 2023. [Online]. Available: https://www.brickstore.dev.

# Glossary

**BO Lot Id** BrickOwl Lot Id: Identifier for BrickOwl lots. ( see Section 2.4 ). 4

**BrickLink** Marketplace for bricks ( see Section 2.3 ). 11, 12, 30, 43, 45, 48, 63, 66, 67

**BrickLinkId** BrickLink inventory ID. Section 2.3 explains how this is relevant in this system. 4

**BrickOwl** Marketplace for bricks ( see Section 2.4 ). 11, 12, 30, 43, 45, 48, 51, 63, 66, 67

**BrickStore** Application used for creating BSX Files ( see [5] ). 3, 5

**BSX File** XML-like file for describing lego stock changes. Better described in Section 2.2.. 5, 6, 21, 22, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 37, 43,

**PED** Internal Unbrickable database. ( see Subsection 2.5.1 ). 6, 23, 43

# Appendix A

# Diagrams

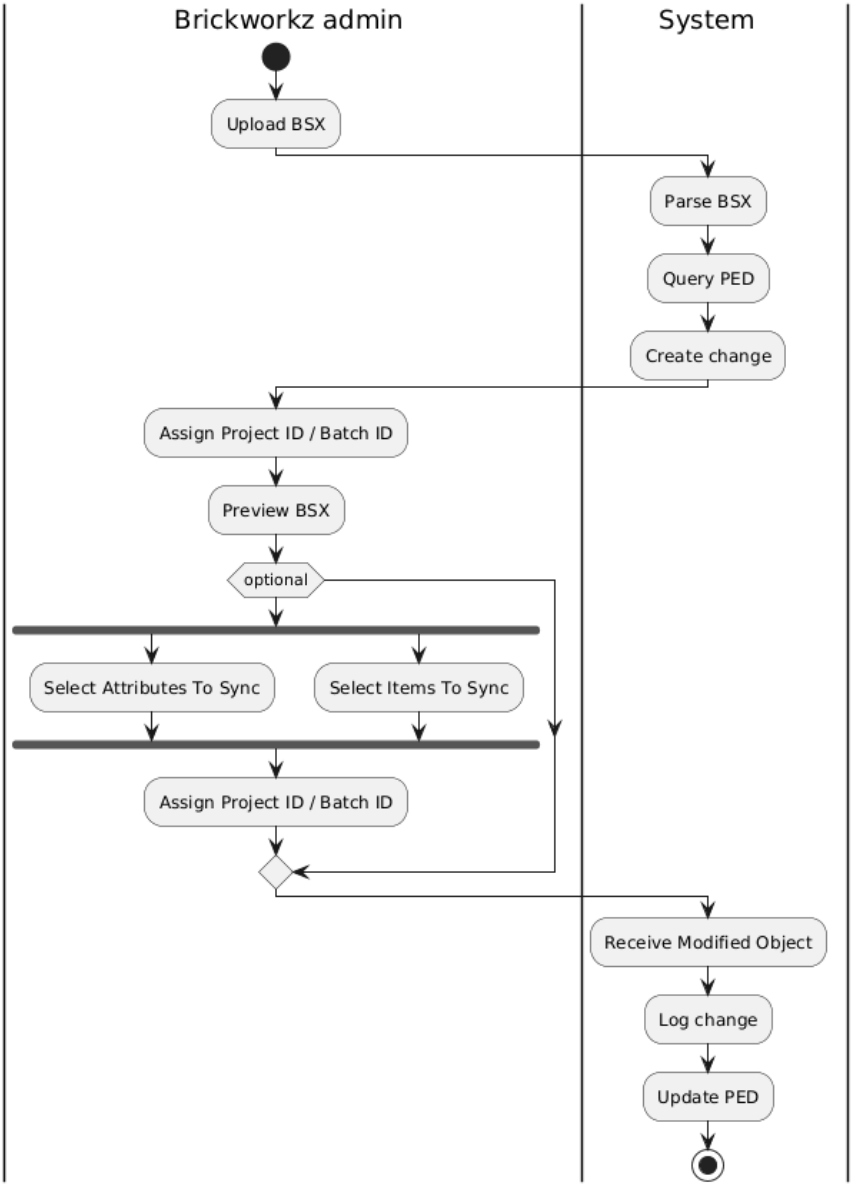



Figure A.1: Upload flow activity diagram

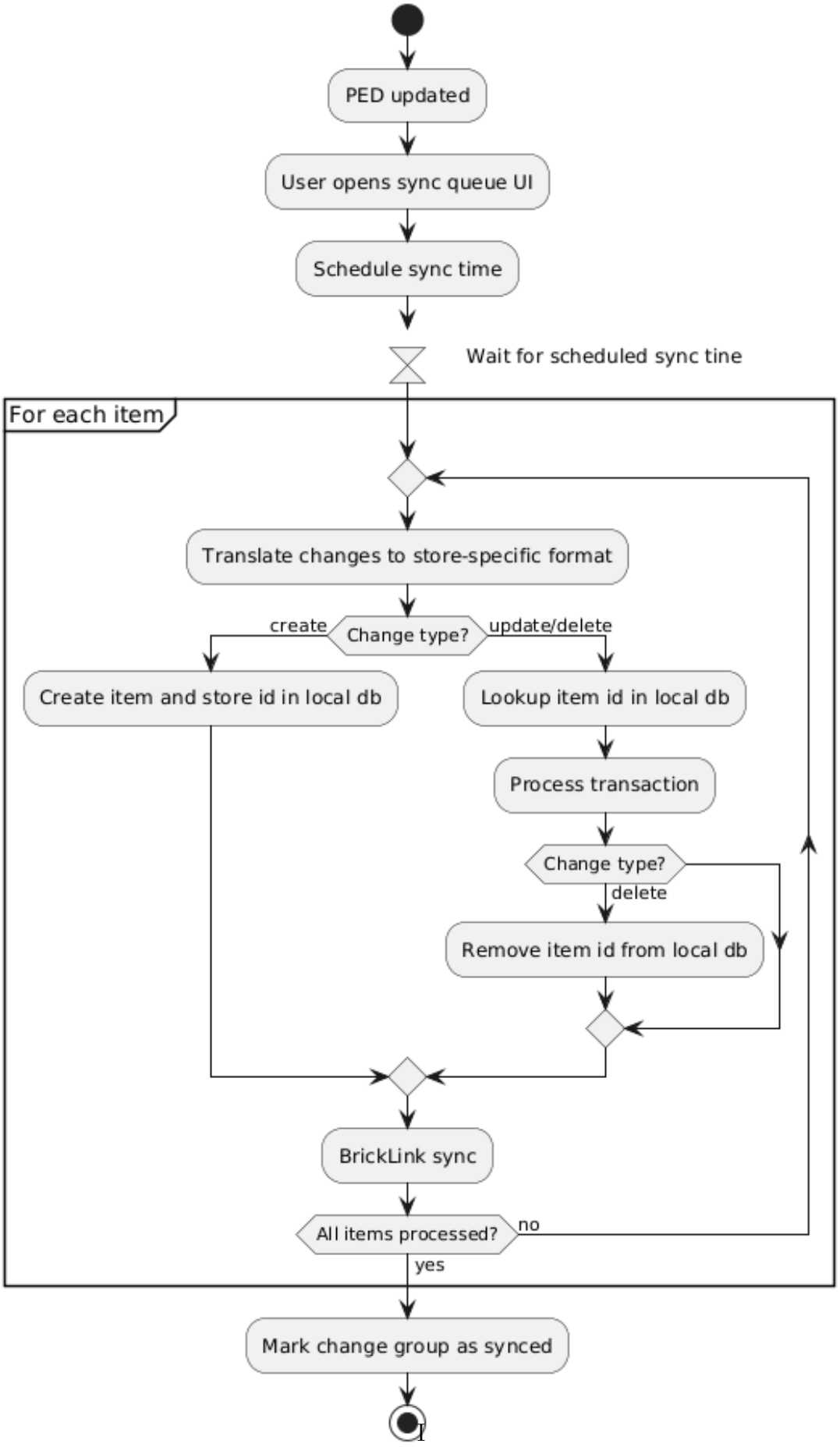



Figure A.3: Sync Flow Activity Diagram

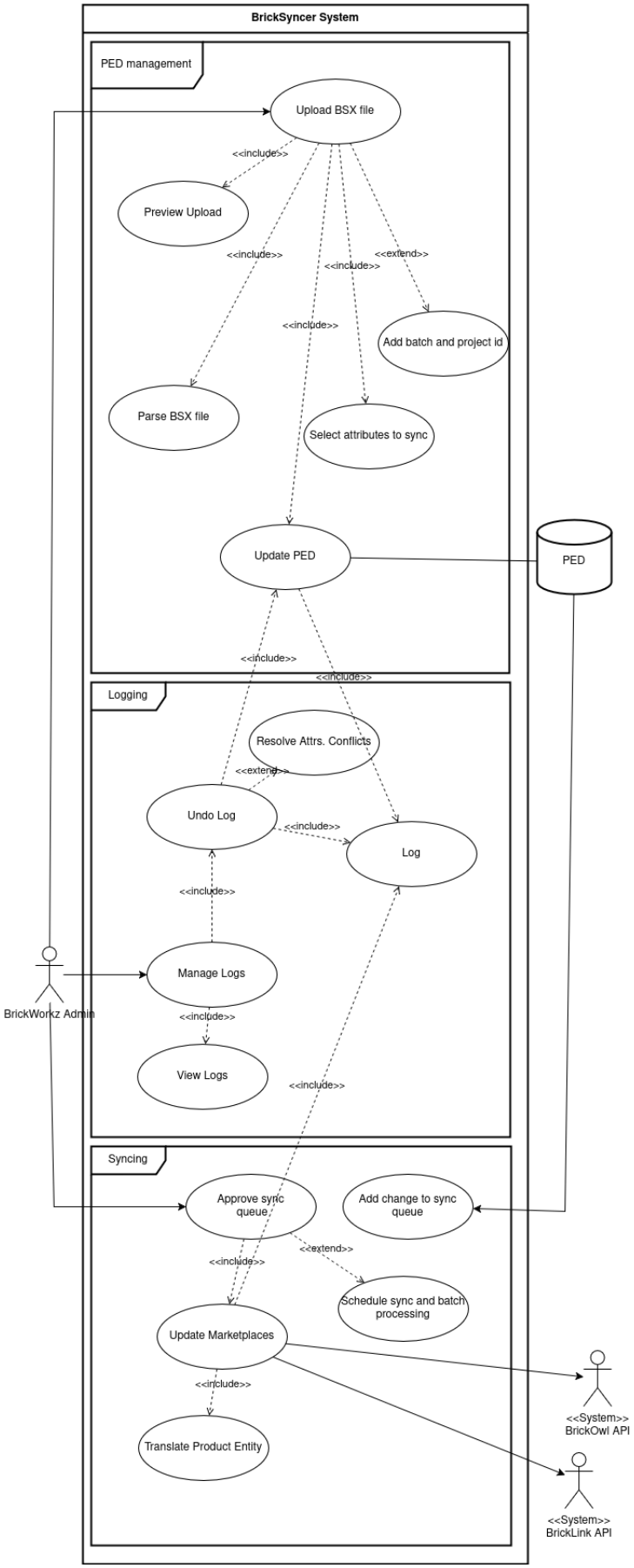



Figure A.2: Use-case diagram

# Appendix B

# AI statement

## B.1 In development

During the development phase generative AI was used sparingly to help fix syntactical errors, brain-storm solutions and explain programming concepts. In other words, no one in the team was actually familiar with C# before the start of this project. AI was used to bridge the gap of purely syntactical knowledge during development by allowing the team to apply programming experience in other languages without being constrained by the lack of .NET specific expertise and learn while doing.

## B.2 In testing

Generative AI was used to generate comprehensive integration tests. In order to create correct, coherent, and useful tests, the below workflow was used.

First, the testing strategy as outlined in chapter 5 was drafted manually. The main flows were determined, and for each flow the test cases were determined. Then, for generating the tests themselves, code-specific AI was used by utilizing GitHub Copilot and JetBrains Rider ChatGPT Plugin. Detailed prompts were written explaining to the agent both the scope of the project and of the test. In addition, it was specified what should be tested, which services should be used and which should be mocked, what actions to simulate and what outcomes to check.

After test generation took place, tests would sometimes fail for errors inside the tests itselfs. This is because the AI agents sometimes write high-coverage, coherent tests, but subsequently choose invalid values to verify the outcomes. For example, the agent would write a highly complex integration test covering several services, that tests various edge cases, respecting the instructions, but when checking a specific attribute at the end, it would check for a nonsensical numerical value. In such cases, tests were corrected manually. Once correcting the initially obvious mistakes was

done, the person generating the tests would go through the entire test and verify the correctness of the actions. If any flaws were found, they were fixed manually, or the test would be re-generated or updated when necessary. Additionally comments were written to help any reader understand the tests better. Finally, after the problems were solved, including the real implementation bugs that were found by the tests, the test is considered valid and kept.

The specific prompts and models that were used are available inside the testing code itself, in the form of multi-line comments at the top of the test files, which they were used to generate the test methods.

## B.3 In writing

During the writing of this report AI, specifically GitHub Copilot, was used to accelerate the process of designing diagrams from the code. Given access to the code and a plantUML jar file, GitHub Copilot easily created the ER diagrams from the EF core configuration files or other diagrams that describe the code. These diagrams were later reviewed and edited by the team.