Design Report

Warehouse Management Web Application for Brickworkz

Mara Teodorescu

Natasa Tudorache

Tania Mincu

Rian Sood

Aadi Malhotra

Yash Sakore

Supervisor: Dr. C. Morais Fonseca (Claudenir)

April, 2026

Department of Computer Science

Faculty of Electrical Engineering, Mathematics and Computer Science,

University of Twente

UNIVERSITY OF TWENTE.

BRICKWORKZ
CREATE YOUR FUTURE

Contents

1. Introduction

Brickworkz is a Dutch social enterprise that sells individual LEGO elements and custom sets worldwide, while providing meaningful work opportunities to young people aged 13–21 with conditions such as autism, ADHD, or giftedness. Their current operations rely on a combination of Make.com and Monday.com, tools that, while functional, are not tailored to the specific workflows of a LEGO parts warehouse. This results in fragmented processes, a steep learning curve, and increased risk of human error, particularly for the young employees at the core of the organisation.

To properly support daily operations, Brickworkz needs a web-based system that is built around its own way of working rather than adapting to generic tools. A custom solution would allow order handling, warehouse processes, and administrative tasks to be combined in one clear and consistent environment. This would make the work easier to understand, reduce mistakes, and shorten the time needed for training new employees. By tailoring the system to the users and processes at Brickworkz, the application can better support both operational efficiency and the organization's social mission, while also leaving room for future improvements.

2. Project Planning, Organization and Approach

In this section, we discuss how we developed the project and used our proposal agreed with our client to formulate an approach to making the deliverable. First we discuss the different project phases as we sought out to follow. The methodology used during these phases of development is also emphasized upon.

2.1 Project Phases, Milestones and Timeline

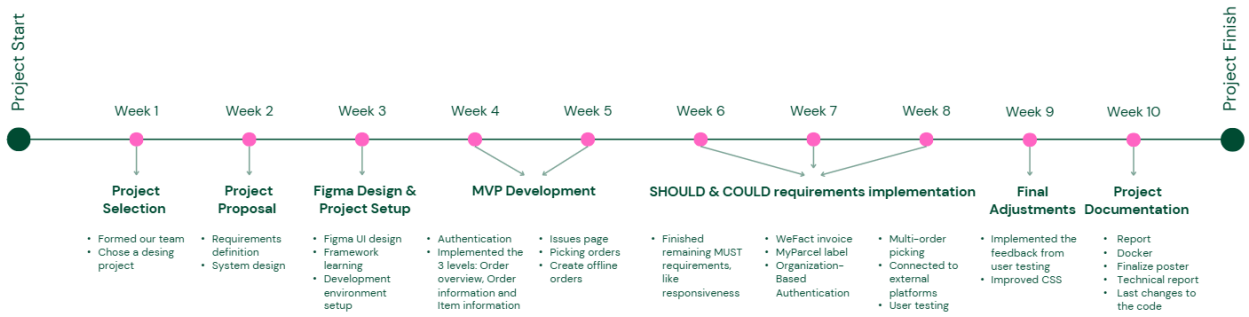

Figure 1: Development roadmap

The project was carried out over a period of 10 weeks, as can be seen in **Figure 1**. The first week was dedicated to project selection, where we formed our team and chose the Brickworkz project. In week 2, using the information gathered from our first client meeting, we wrote the project proposal, defined the requirements and created system diagrams. Week 3 was dedicated to project setup, during which we familiarized ourselves with .NET and Nuxt, created the initial Figma UI designs, and configured our development environment. During weeks 4 and 5 we developed the MVP, implementing core features such as authentication, order overview and picking orders. Weeks 6, 7 and 8 were spent implementing the SHOULD and COULD

requirements, including WeFact invoice integration, MyParcel label generation and organization-based authentication. In week 9 we focused on final adjustments, incorporating user testing feedback and connecting to external platforms. Finally, week 10 was dedicated to project documentation, including the technical report, finalization of the poster and last code changes. We were able to stick to our proposal for development as will be followed in the methodology below.

2.2 Development methodology

Throughout our project, we held weekly internal meetings to monitor progress, discuss challenges, plan upcoming tasks, and divide work among team members. WhatsApp served as our primary communication platform for day-to-day messages given its ease of use. For client and supervisor communication, Tania was responsible for contacting the client, sending emails regarding meeting scheduling and any questions or requests we had, while Mara handled all communication with the supervisor, keeping them informed of our progress and raising any concerns when needed. Weekly meetings with both our client and supervisor ensured we remained aligned with project expectations throughout the project.

For version control and collaborative development, we used GitLab to manage source code and maintain a clear record of changes. We tracked all tasks using GitLab Issues, assigning labels to each issue to indicate the area of the system it concerned such as *backend*, *frontend*, *database*, *design*, or *api*, making it easy to filter and understand the scope of each task at a glance. As shown in Figure 2, issues were either open or closed, giving the team a clear view of overall progress at any point in time. To keep the repository organised, each issue had a corresponding branch named after it, following the format [issue-number]-[functionality], so the purpose of every branch was immediately clear. This is illustrated in Figure 3.

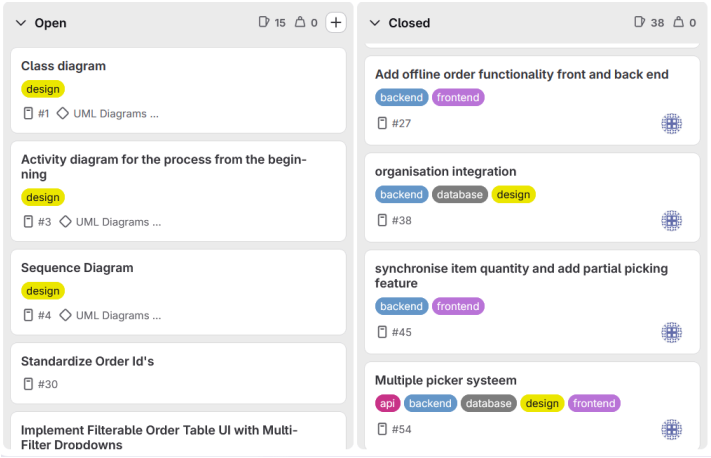

Figure 2: GitLab issues

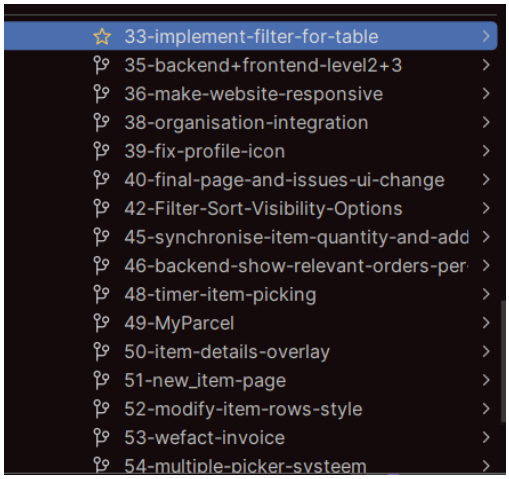

Figure 3: Branch naming convention

3. Technical Architecture and Design Choices

In this section, we discuss the main design of our project and the technical implementations made for the product. We first mention the design choices made and why we made them then give an overview of the tech stack used. The tech stack of the backend and front end is emphasized on and we finally explain the Docker container we used.

3.1 Global Design Overview

In this section, we discuss the design and diagrams we use to describe our main system. We have chosen a Use Case diagram to show actors' interactions with the system and summing up functionality and an Entity Relationship diagram which describes our database structure which helps us understand how our system manipulates and stores data.

The Use Case diagram presents the main actors of the system and the interactions they have with the application. In this project, the main actors are New Organization, Admin, Expediter, and Picker, together with external systems such as Marketplace, Scheduler, MyParcel, and WeFact. The diagram shows the main system functions, including organization registration, login, order management, picking activities, issue handling, inventory management, marketplace synchronization, and fulfillment tasks such as shipping label and invoice generation. Please open the diagram with Draw.io using this link: Use CaseDiagram.

Figure 4 illustrates the database schema of the system. At the center of the schema is the *orders* table, which stores high-level information about each order, including service type, order date, financial details (such as subtotal and total), payment and currency information, shipment data, and overall status. Each order is associated with exactly one organization, while an organization can have many orders (one-to-many relationship). The *order_items* table extends this by representing the individual items within an order, meaning that one order can contain multiple order items, while each order item belongs to exactly one order (one-to-many). It also captures details such as quantity, remaining quantity, and status, and tracks which users have claimed or picked specific items.

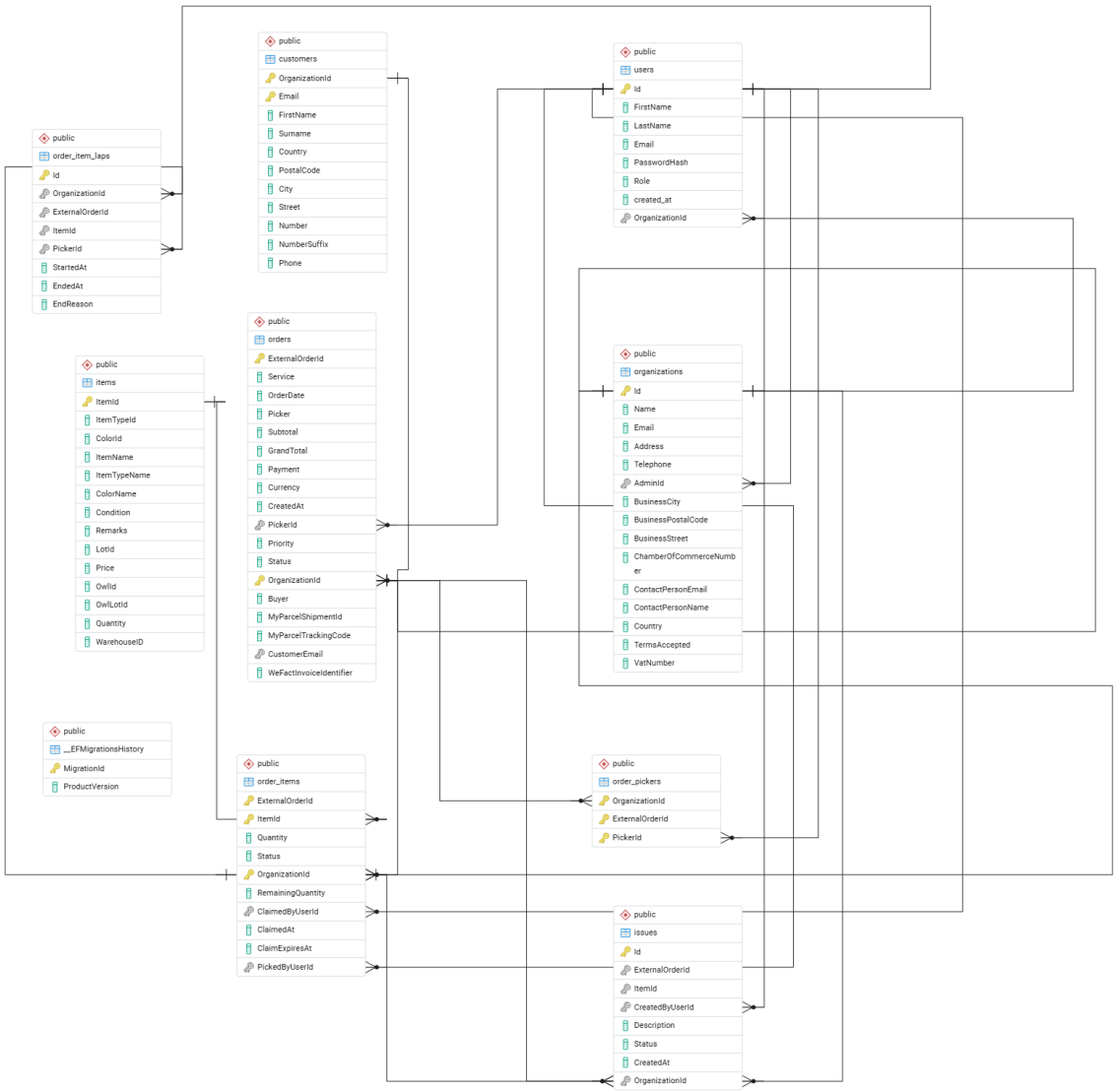

Figure 4: Entity Relationship Diagram

Product-related data is maintained in the *items* table, where each item can appear in multiple order items across different orders (one-to-many from items to order_items), while each order item refers to exactly one item. This separation allows inventory to be managed independently from orders. The picking process is supported through the *order_pickers* table, which models a many-to-many relationship between users and orders: one order can be assigned to multiple pickers, and one user can act as a picker for multiple orders. Additionally, the *order_item_laps* table logs detailed picking activities, where each lap is associated with one order item and one picker, while an order item and a picker can each have multiple laps over time (one-to-many relationships).

To handle exceptions, the *issues* table records problems related to order items. Each issue is linked to exactly one order and one item, while an order or item can have multiple associated issues (one-to-many). Furthermore, each issue is created by a single user, while a user can create multiple issues (one-to-many).

The system supports multiple organizations through the *organizations* table. Each organization can have many users, customers, orders, items, and issues (one-to-many relationships), while each of these entities is linked to exactly one organization. The *users* table stores system users and includes a role attribute, with each user belonging to one organization, but potentially participating in many orders (via picking or issue creation). The *customers* table represents end customers, where one organization can have many customers, while each customer belongs to exactly one organization and can be associated with multiple orders (one-to-many from customers to orders).

Overall, the schema integrates order management, inventory tracking, user management, and operational workflows into a cohesive structure, with clearly defined multiplicities ensuring data integrity and reflecting real-world relationships. These one-to-many and many-to-many relationships enable flexible yet structured tracking of orders, items, users, and processes, supporting efficient warehouse operations across multiple organizations.

3.2 Tech stack and Development Frameworks used

The design of this project is based on a modular layered structure. The system was divided into three main parts: the frontend, the backend, and the database. This separation makes the application easier to develop, maintain, and extend, since each part has its own responsibility.

The frontend, developed in Nuxt, handles the user interface and user interaction. The backend, developed in ASP.NET Core, manages the application logic, processes requests, and communicates with the database. It is further structured into controllers, services, selectors, and models, which helps keep the code organized and easier to manage. The database, implemented in PostgreSQL, stores the main system data such as users, organizations, orders, items, and issues. By splitting the project into these layers, the system becomes more structured and scalable. This design provides a clear foundation for the implementation and supports future development.

For every full-stack project, we need a methodology to deliver the best possible product. The below table sums up the tech stack we have used. In order to explain the tech stack we have used for our multiple processes, we will use the below table.

Backend	ASP.NET core Web API	This C# based service was used to develop the product and be the backend API point, handling routes, URLs and all logic relating to our product.
Database	PostgreSQL	Used a postgres database running on localhost to ensure we could continue development irrespective of having a database server running somewhere. We used the schema mentioned in 3.1.

Frontend	Nuxt.js (Vue)	This vue based Javascript framework allowed us to do all our web development and front end related logic and used a stable preset file structure which managed our routing easily.

Table 1: Tech stack summary

Using this tech stack, we used the MVC (Model,views, controllers) structure to build our project from scratch. The model is a class in our backend which shows the structure of our database and every object of this class from the database is treated through the EF (Entity Framework) core model which is an ORM (Object relational mapper). This ORM uses models from our backend and manipulates and maps the data of an object from these models and makes changes to the database, without having to write SQL commands for every action. EF core creates migrations for the models made in the models directory in the project and when we update the database with these migrations, the database is updated, again without having tedious SQL commands. The controllers are API endpoints managed by the backend of ASP.NET core which carry the data from front end to backend where our services can do the CRUD operations in the database while the front end views are provided by our Nuxt webpages. We also use hubs for direct client server connection from the backend, skipping API endpoints for some functionality which requires real time information relaying. This is a lot of information and therefore **Figure 5** shows how data is manipulated from one part of our stack to another. Through sections 3.3 and 3.4, it is better explained how the components' connectivity is maintained.

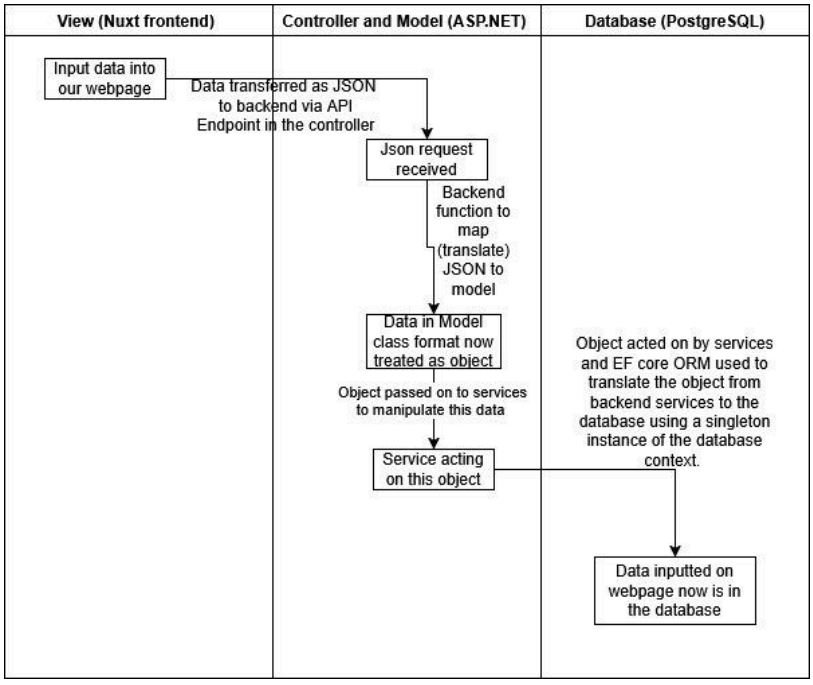

Figure 5: Data flow of our system (POST request)

3.3 Backend

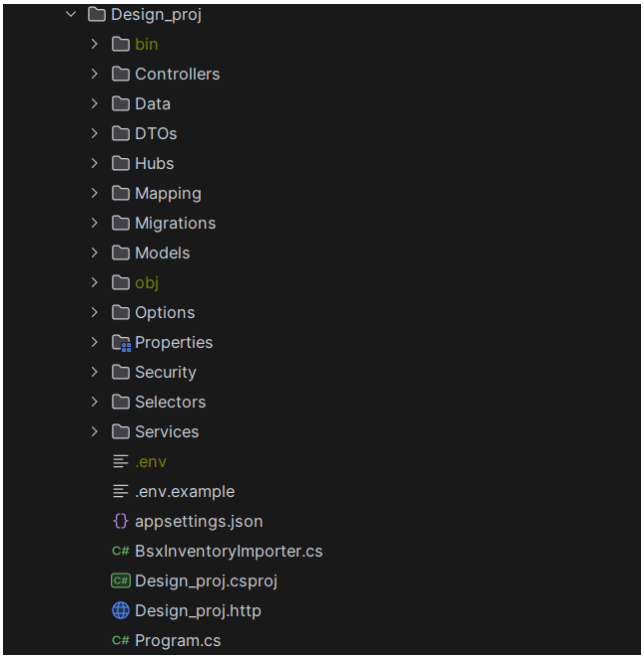

Figure 6: Structure of our Backend

ASP.NET Web API core was used for the backend of this product. C# is an object oriented programming language and .NET is a cross-platform development framework used for web, mobile, desktop, games, and cloud services. We were instructed by our client to use .NET so we could integrate easily with the system that they have at hand. The backend is structured across multiple directories as shown in **Figure 6**, each responsible for a distinct part of the application. Important directories will be explained along with some code snippets and a better description regarding their tasks and connectivity.

3.3.1 Controllers

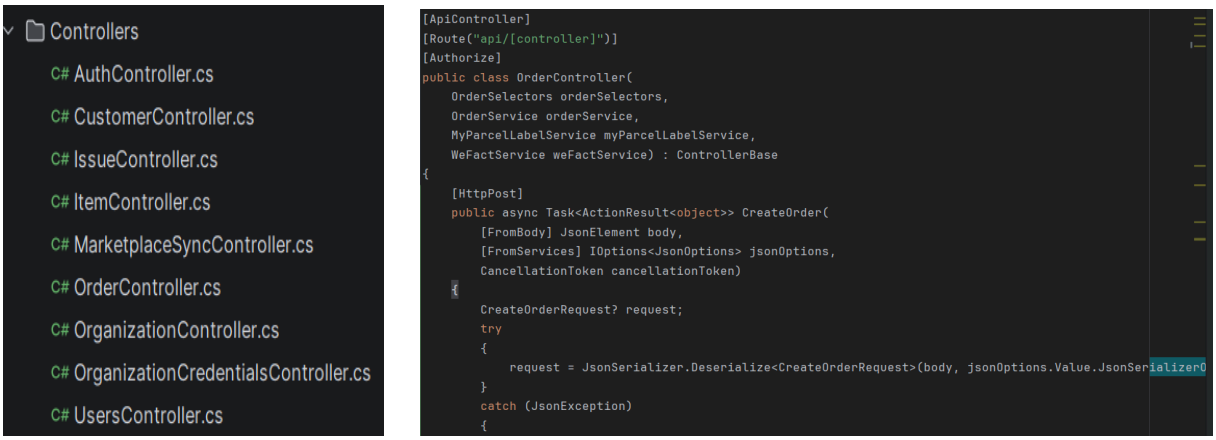

Figure 7: a) Controllers and 7 b) Example of a controller (OrderController)

Controllers are the main routing system containing API endpoints and managing the data to and from the frontend. The above controllers contain API endpoints for each functionality as mentioned in the name of the controller. The example of the Order controller is given above, showing how each controller contains a service and how some controllers, such as this one, also contain selectors. These services and selectors contain functions which manage the data to and from the front end. The *[controller]* in the route above the class always corresponds to the lowercase name of the controller, which in this case, is order. The *[authorize]* deals with JWT (JSON Web Token) authentication to ensure security and organisation separation. Above each method, the type of HTTP request is mentioned as shown in the image. This part also contains any parameters that can be passed in the URL. These methods make use of the aforementioned services and selectors in the body of the method. Moreover, the controllers also manage authentication and security along with helper functions from classes in the Security directory. Hubs help for direct server-client communication which is important for real-time data transmission which is necessary for the functionality of Multi-Order picking (Requirement 1 of MoSCoW Could requirements).

3.3.2 Models

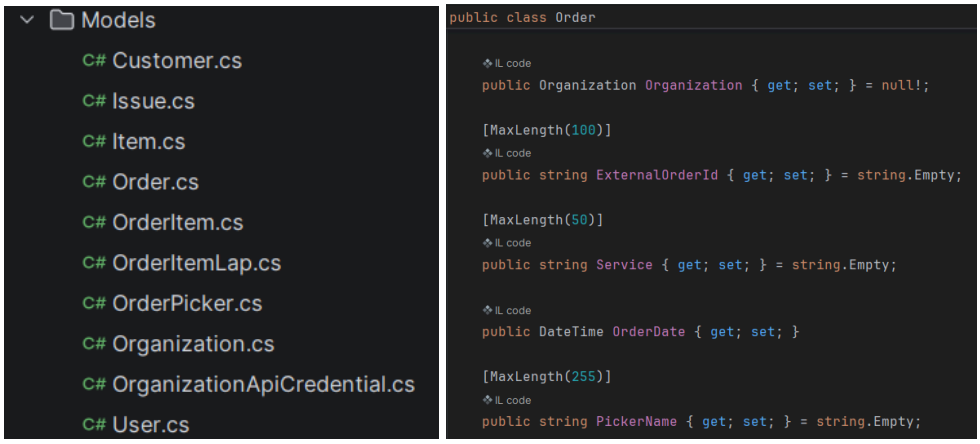

Figure 8: a) Models and b) Example of a model (Order)

The Models directory contains the skeleton of our database and the EF core ORM uses instances of these model classes to perform CRUD operations on the database after receiving data from our controller. Each model class is found in the database and an example of the order class is shown alongside our model structure. These models are applied using the migrations present in the migrations directory as seen in **Figure 6** which are added by running EF core[1] commands. These models are translated to and from JSON objects using the Data Transfer Objects (DTOs) in the DTOs directory for frontend communication via controllers. The Data directory has an AppDbContext for all tables to be managed in the database.

[1]dotnet ef database update - this command applies migrations | dotnet ef migrations add [MigrationName] - adds migrations if you make changes to a model class.

3.3.3 Services and Selectors

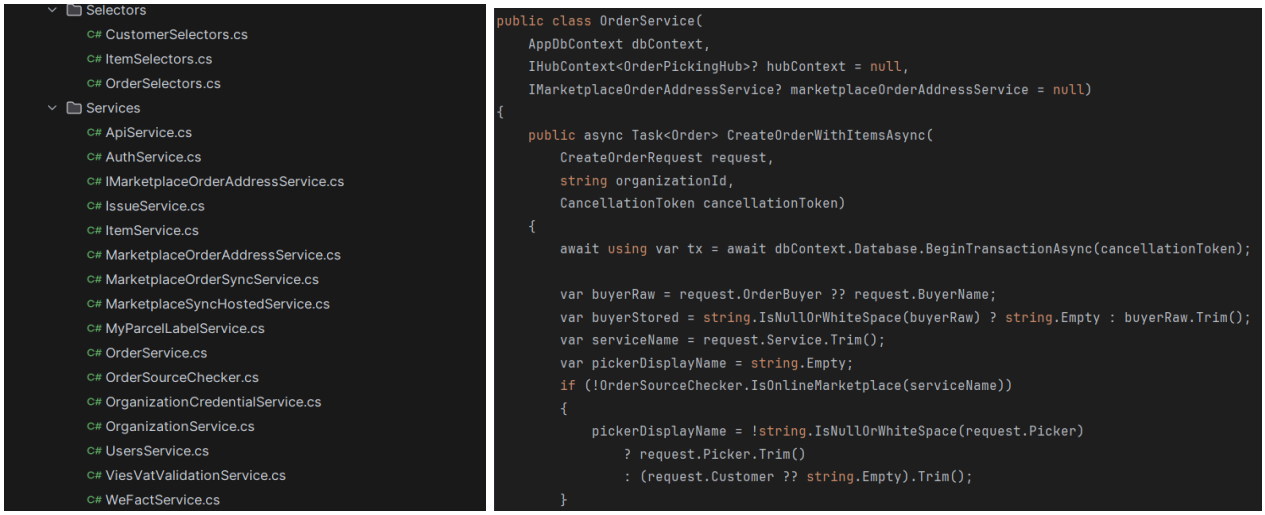

Figure 9: a) Services and Selectors and b) Example of a Service (OrderService)

The services and selectors directories contain the core methods which manipulate data to and from the controllers. These services contain additional functionality and offer an easy way to use them in the controllers to divide the tasks per controller, with each having a service as mentioned above. Each service uses the AppDbContext for a singleton instance of the database to perform CRUD actions as the service requires. The async Task shown in **Figure 9b** deals with creating an order and this is called in the Order controller to create an order in the database. This task receives parameters for creating an order and the CreateOrderRequest corresponds to a DTO class which converts JSON to model notation so that the service can act upon this using EF core and create an order in the orders table in the database. This request can also be seen in **Figure 7b** and therefore this is how our backend connects multiple methods from different directories to make our system's backend function smoothly.

3.4 Frontend

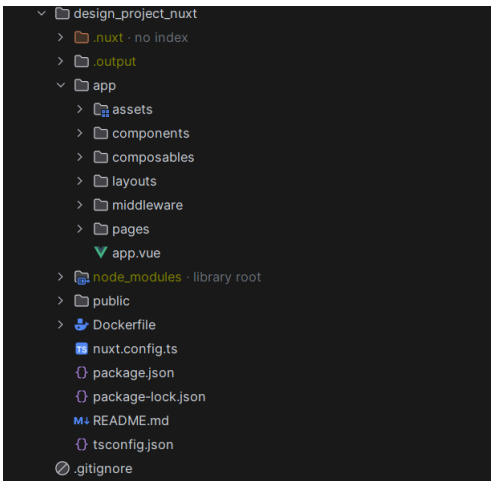

Figure 10: Structure of our frontend

The front end of the project is made using Nuxt.js which is a javascript web development framework. Our frontend is extensive and follows a strict structure because of Nuxt having a specific routing system and multiple file naming conventions. Nuxt is a vue.js based front end framework and therefore all files are .vue except a few typescript files in middleware and composables directories used for security. Important directories will be explained with some code snippets.

3.4.1 Pages

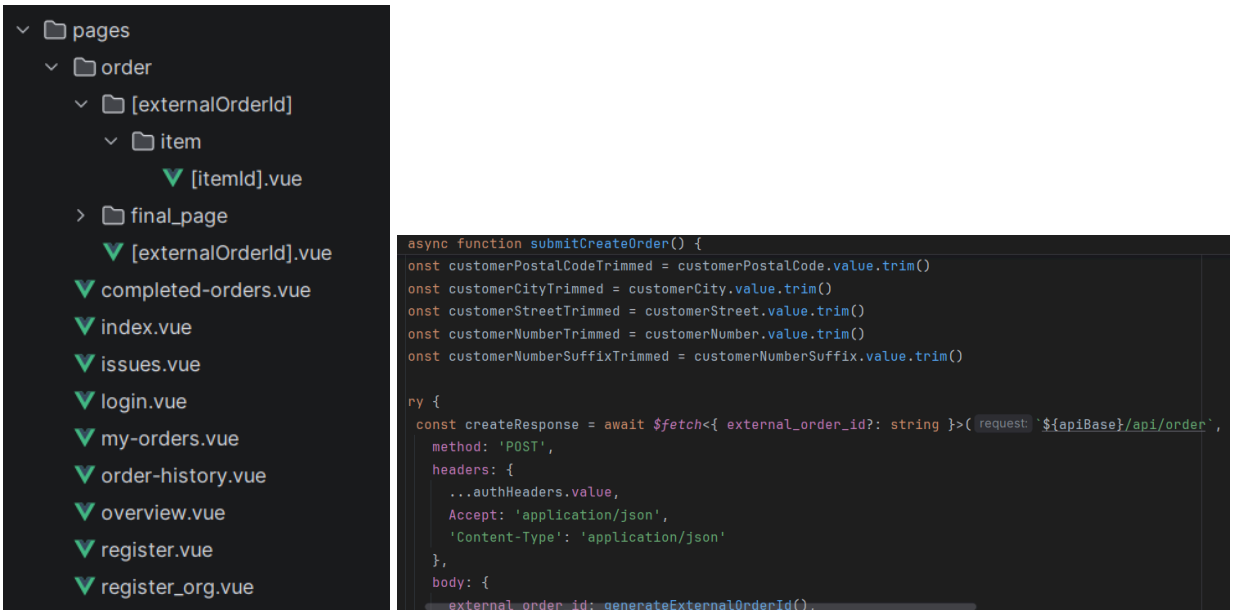

Figure 11:a) Strict structure of our pages directory and b) POST request from component CreateOrderModal of overview.vue to backend

The pages directory contains all the webpages of our product. There are strict structures and naming conventions to be followed. For example, index.vue is the page that you always land on at first, even when the URL does not contain index.vue. The other pages however, use the name of the vue file. The URL for overview.vue uses a /overview suffix. The other structural feature to be noted is that to go to the page of an individual order, you go via the directory order -> *[externalOrderID].vue* which corresponds to that order and the square brackets store the order ID, making it a unique URL per order. The example URL to view details for an item for example is *order/ORD-20260403124459-532/item/117* with 117 corresponding to *[itemId].vue*, therefore uniquely recognising an item part of an order.

Some of these pages use CSS styling files from the assets directory for frontend design styles. These pages also use default.vue [2] from the layout directory. Pages that need a common component such as header or sidebars do not need to be repeatedly coded in each page since the layout keeps it common over multiple pages. In **Figure 11b**, we can see how the pages make API requests to our backend. The *submitCreateOrder* function uses the POST request from **Figure 7b**

[2] Strict naming convention where pages use this file in layout directory to keep all components common for those pages.

to create an order in the database showing how we connect with our backend. Further we will show how components from the components directory interact with our pages.

3.4.2 Components

Figure 12: a) Structure of components directory and b) Component of order/[externalOrderID].vue

Components follow a structure too for simple routing. The vue files under the components/order directory correspond to the *order/[externalOrderID].vue* page in the pages directory. To invoke these components, we use the simple syntax shown in **Figure 12b** to display this component on the order page. This style of design makes it easier to know the different components of a webpage and especially conditional components can easily be moderated using this feature of the frontend framework.

3.5 Docker

Docker was used to containerize the project and create a consistent development environment. The application was configured with three main containers: frontend, backend, and database. These containers are managed together through Docker Compose, which defines how the services are built, started, and connected.

To run the system, the environment configuration is first prepared by copying .env.docker.example to .env and adjusting the values if necessary. After that, the full application can be started with the command docker compose up --build. This allows all services to start together in one step and ensures that the application runs in the same way on different machines.

Using Docker reduced manual setup and helped avoid environment-specific configuration problems. It also made the project easier to test and deploy, since all required services are packaged in the same containerized structure.

With the technical foundation and deployment environment established, the focus now shifts to the purpose of the application and how it meets the needs of its users. The following section details the core functionality of the product, beginning with an analysis of the stakeholders who interact with the system and design choices made for them.

4. Functionality of the product

In this section, we discuss the stakeholders of the product and how they interact with our system. We also discuss design choices.

4.1 Stakeholders

The key stakeholders of this project are the order pickers as primary end users, the company contact person who defines requirements and provides feedback, the warehouse manager who oversees operational efficiency, Brickworkz customers as indirect stakeholders, the project supervisor who ensures academic quality, and the student development team responsible for delivery.

Both the client and supervisor were actively involved throughout the project. The client was engaged through weekly meetings and frequent email communication, playing a key role during requirements approval and User Acceptance Testing. The supervisor will provide academic guidance, review major deliverables, and give feedback to ensure the project meets the required standards. After each major phase, results will be reviewed with both parties to maintain alignment and allow for timely adjustments.

4.2 Design choices and System Interactions

4.2.1 Organization login and register

For all stakeholders interacting with the system, the organization's login screen is the first page they land on. This screen allows users to select the organization they want to log into and also gives new organizations the ability to register themselves. This functionality stems from an additional requirement that was added later on in the development life cycle to allow multiple organizations, including hobbyist sellers and registered companies, to use this platform for their warehouse management. This corresponds to requirement **C** in the Additional Requirements.

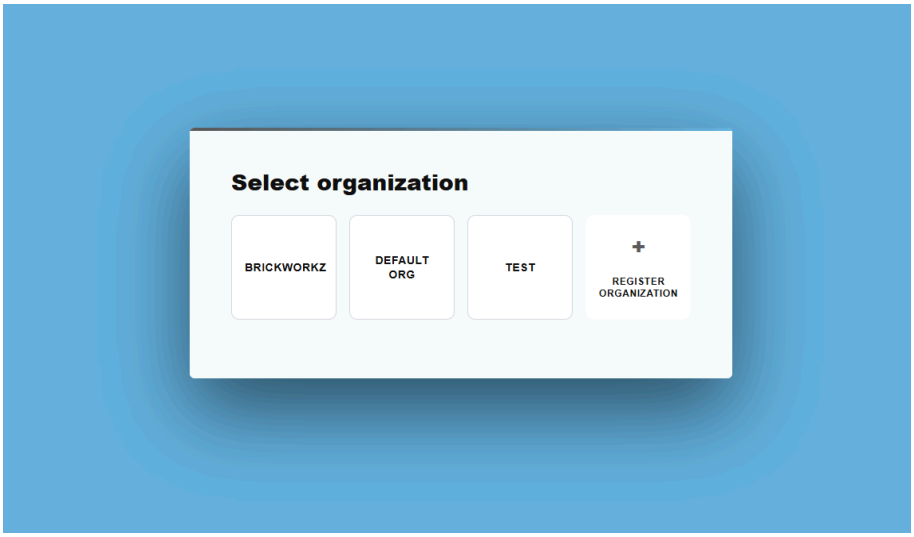

Figure 13: Organization login and register page

4.2.2 Order Overview screen

When logged in as an admin, the user lands on the order overview page (requirements A and B of the MUST requirements). This page allows users to look at a table of all orders along with customizable columns for more information. Additionally, the order overview page allows users to sync the system with their online marketplaces at will and create offline orders. The order overview screen also contains a variety of filtering and sorting options to support further customization of the orders. In conjunction, there are tabs to aid order management by enabling one-click filtering and separation of orders.

When the logged in user is of the role Picker, their side bar only contains the order overview page and no other links. Additionally in the order overview table, they only see orders that have been assigned to them.

These functionalities have been added through the thinking that each person in an organization would like to format their order overview screen differently. An admin, for instance, would like to briefly see some information or bifurcate orders based on their internal procedures. A picker on the other hand, might not want to see information such as order value or platform because they are not essential to the picking procedure.

We believe our implementation will support multiple working principles across organizations.

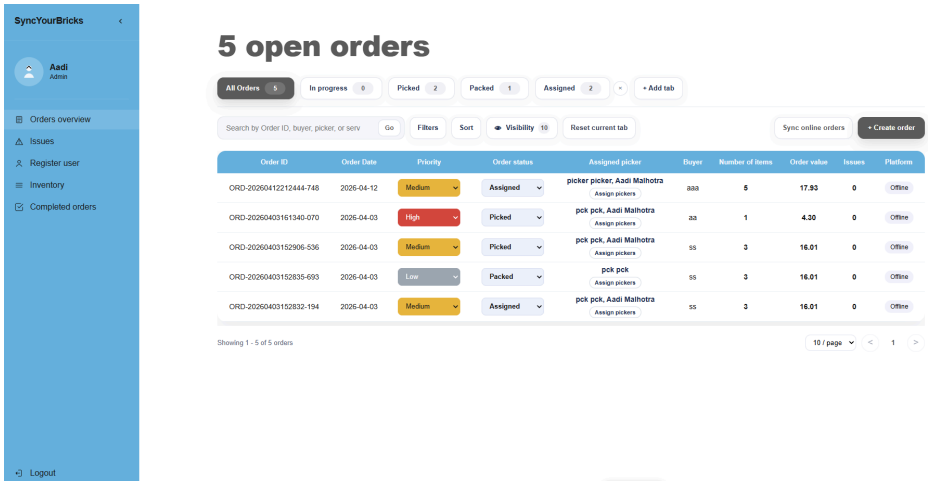

Figure 14: Order Overview screen

4.2.3 Sidebar functionality

From the main screen, the admin can navigate to different parts of the web app using the side bar. This includes viewing all order and warehouse issues, registering new users with one of three roles: Admin, Expediter, and Picker, editing inventory stocks, and viewing all completed orders. These functions encapsulate all the identified admin use cases and allows them to view their entire operation more cohesively.

Some of the sidebar functionality is kept through overlays since it allows admins to quickly change warehouse stock for a specific item and register a user without loading into a new page.

We believe these functionalities will not be a routine task, once the organization is operational, due to reliable database syncing.

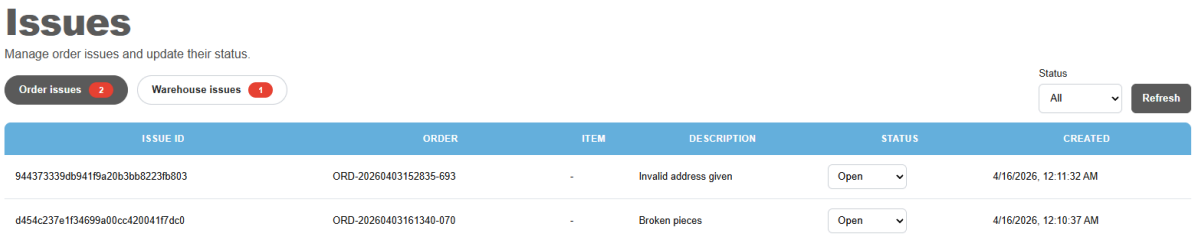

Figure 13: Issues screen

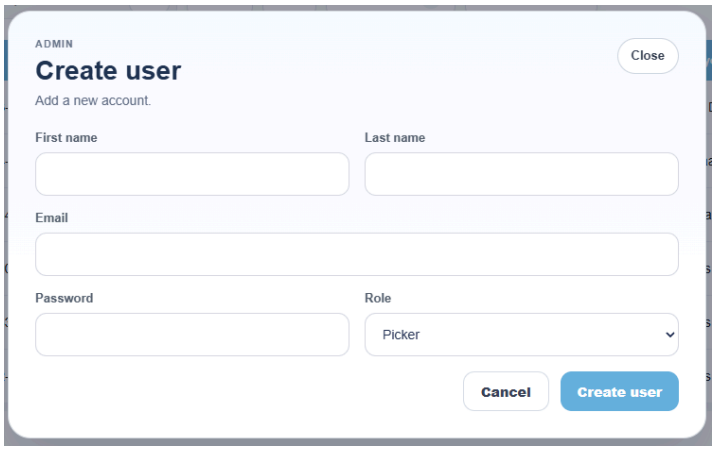

Figure 15: Create User overlay

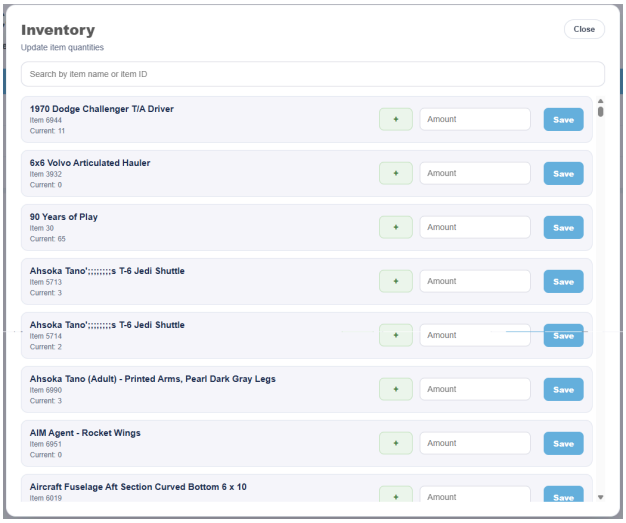

Figure 16: Inventory Update overlay

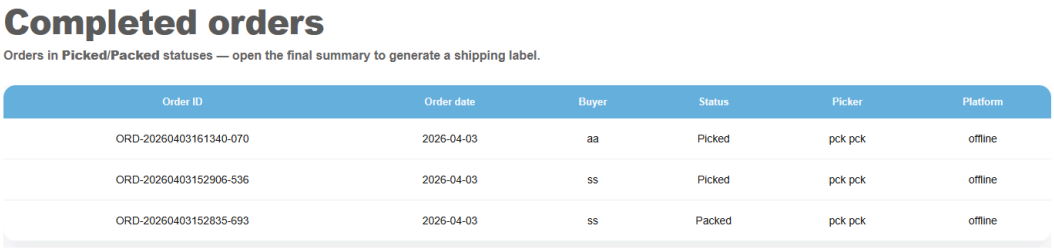

Figure 17: Completed Orders screen

4.2.4 Order screen

Upon clicking on an order from the overview screen, the user is presented with a screen that showcases all the order information (requirement D of the MUST requirements). This information includes the items to be picked, the progress of the order picking so far, and any open issues that might be associated with the order. This screen also gives users the ability to start picking, for an admin to assign pickers, and to vacate the order if the picker becomes occupied elsewhere.

This order screen is the result of multiple front end design iterations which eventually concluded on this card-based design. We believe this implementation looks clean and elegant and it presents the large amount of information on this screen in a way that is legible and easy on the eyes.

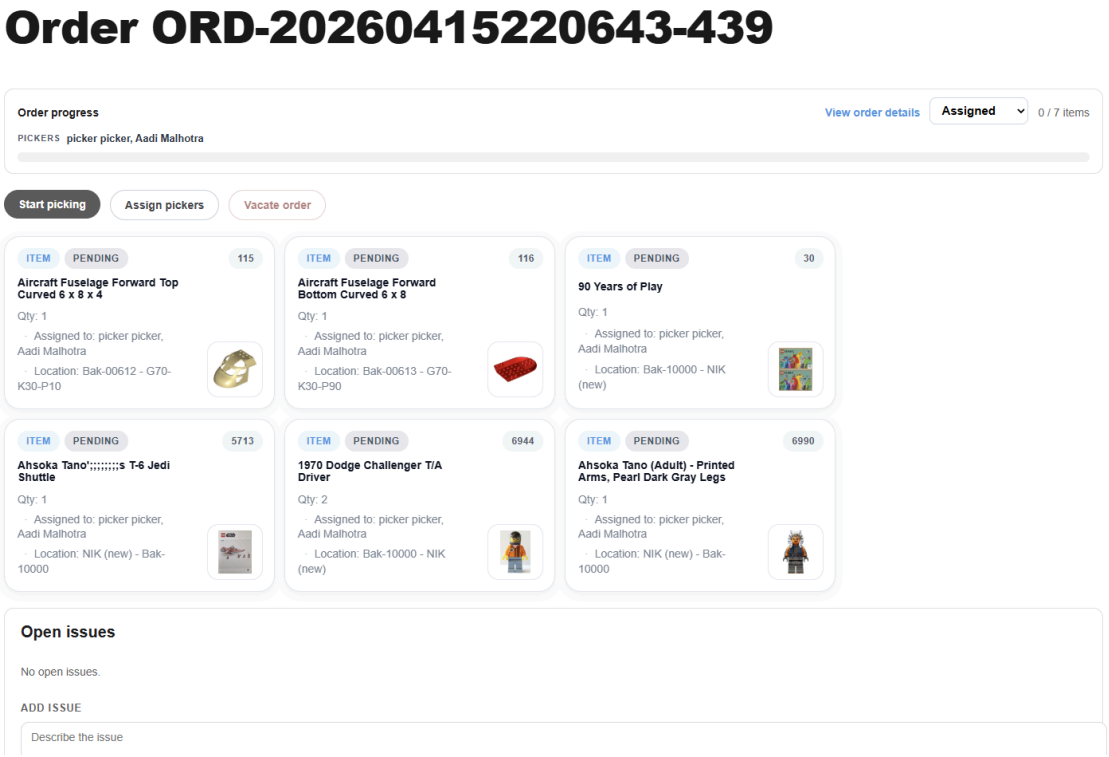

Figure 18: Order screen

4.2.5 Item screen

Upon clicking “Start picking”, the user is taken to the item page which consists of a lot of the core functionality of the system. This page displays important information (requirement F of the MUST requirements) such as order progress, item location in the warehouse, an item image, the current status of the item along with a picking timer, and any open issues for this item. On this page, the user can either mark the order as picked, not found, report that there is not enough stock in the warehouse, or raise a warehouse issue (requirement I of the MUST requirements). Additionally, if the user is a picker, there is the functionality to report an incorrect inventory quantity which raises a warehouse issue notifying the admin. The same functionality, when triggered by an admin, directly updates the warehouse quantity for the item. This page also contains a button to open an overlay which showcases all the items in the order and their current status.

The item screen has been designed to be quite compact and incorporates important visual cues such as colour coded buttons to reduce frictions in the picking procedure.

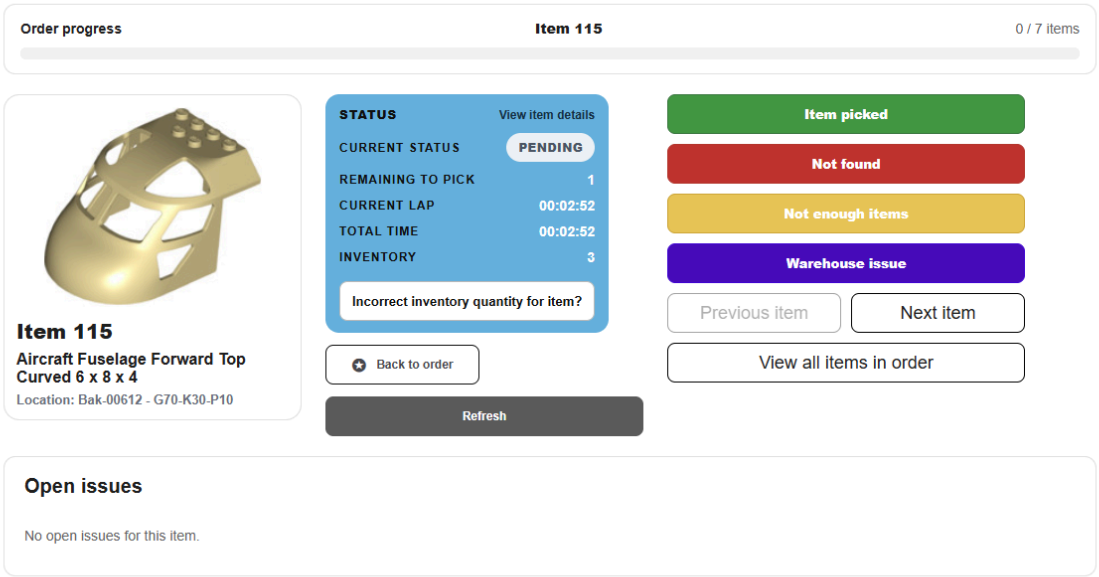

Figure 19: Item screen

4.2.6 Final Order screen

In accordance with requirement A of the additional requirements, when an order has been completed and all items have been marked, the user is taken to the final order page. This page contains a summary of the order which contains the pickers involved, the total time taken to pick the order, as well as a breakdown of the statuses of all the items in the order. Additionally, it contains a card with the important item details which can be filtered according to status. Any user also has the functionality to mark the order as completed and to create an issue at this stage of the order.

For an admin or expediter specifically, they have the functionality to mark the order as packed, and to generate a shipping label and an invoice for this order.

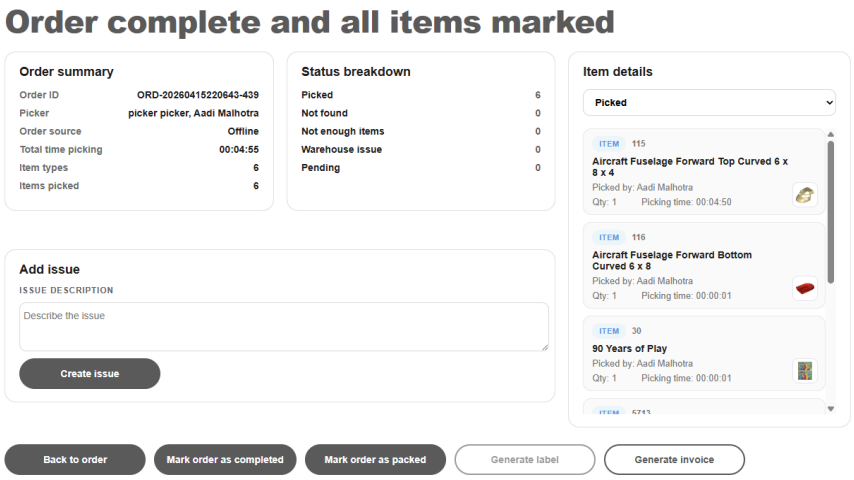

Figure 20: Final Order screen

To view more of our design choices and functionality please view our User Manual where we explain how to use our system as any stakeholder of our product. We now move on to how we tested our functionality with Unit tests and their results and the interface usability with user tests and their results.

5. Testing

To verify both the technical correctness of the system and its usability in a real-world context, testing was approached from two angles. Unit tests were written to validate the behaviour of individual components and API endpoints, ensuring the core logic of the application functions as expected. In addition, a user testing session was conducted with actual Brickworkz employees, observing how they interacted with the system in practice. The following sections provide an overview of both testing methods, followed by a discussion of the results and the changes made to the system as a result.

5.1 Unit tests

Unit tests are organised into three categories: Controllers, Selectors, and Services, covering the core logic of the application across all major functional areas.

The Controller tests cover a broad set of API controllers, including Auth, Customer, Issue, Item, MarketplaceSync, Order, Organization, OrganizationCredentials, and Users. These tests verify that endpoints handle both valid and invalid inputs correctly, return appropriate HTTP responses, and enforce authentication and authorization where required.

The Selector tests focus on the CustomerSelectors class, ensuring that database queries return the correct data. In particular, tests validate that methods such as retrieving customers are properly scoped and respect organisational boundaries.

The Service tests form the largest portion of the suite and provide detailed coverage of the business logic layer. The services tested include AuthService, IssueService, ItemService, MarketplaceOrderAddressService, MyParcelLabelService, OrderService, OrderSourceChecker, OrganizationCredentialService, UsersService, and WeFactService. These tests ensure that each service performs its intended operations correctly, handles edge cases, and responds appropriately to error conditions such as missing data or failed external integrations.

5.1.1 Results

```
  Design_proj.UnitTests test net9.0 succeeded (4,8s)

Test summary: total: 104, failed: 0, succeeded: 104, skipped: 0, duration: 4,8s
Build succeeded in 6,8s
```

Figure 21: Test summary

The figure above shows a summary of our backend test coverage. The coverage percentages are not particularly high, mainly due to the structure of the backend. A significant portion of the codebase consists of Migrations, DTOs, and Model classes, which primarily define data structures or database schemas rather than containing executable business logic. These components typically do not include testable logic and are therefore not directly covered by unit tests.

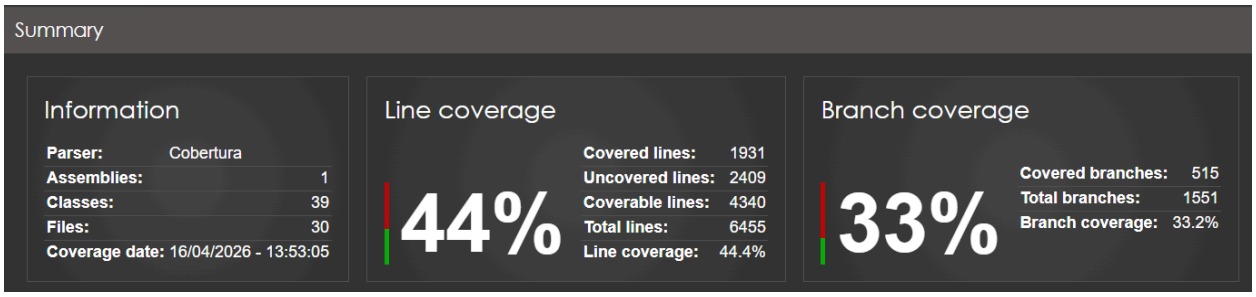

Figure 22: Summary of backend test coverage

5.2 User testing

A user testing session was conducted at the Brickworkz office with three employees. Each participant was given a set of tasks to complete while being encouraged to think out loud, allowing us to observe difficulties as they arose in real time. A short debrief followed each session, covering questions about overall usability, clarity of labels, and whether the order picking flow felt logical. The user testing plan that we used can be found in the Appendix (User Testing Plan).

All tasks were completed successfully across the three participants. The overall usability rating averaged 4 out of 5, with participants noting that the interface was clean and well-structured. Several specific points of feedback were collected during the session.

Navigation and orientation were the most common sources of confusion. After creating an organisation or a user account, participants were not automatically redirected to the main page, which left them uncertain about whether the action had succeeded. Multiple participants also got

stuck in the filter tab, as leaving it reset all filters without warning. The SYB navigation menu was suggested to be replaced with a hamburger menu to reduce visual clutter.

The order creation flow caused some confusion, particularly around the distinction between buyer name and first/last name fields, and around finding the item list and place order button within the modal. One participant initially mistook the items section for an order overview.

The picking flow was generally found to be logical, though participants suggested that items should be displayed in a defined order and that it should be possible to see multiple items at a glance during picking. It was also suggested that the "Start picking" button should automatically assign the order to the picker, reducing the number of steps required.

Visual feedback was highlighted as an area for improvement. Participants noted that order statuses would benefit from colour coding, and one participant appreciated the red indicator on the Issues tab, showing that small visual cues are effective.

Other suggestions included making package size selectable during label generation, turning issue creation into a popup rather than a separate page, and ensuring that item details are displayed more prominently on the final picking page.

5.2.1 Results

Following the user testing session, a number of improvements were identified and implemented. These changes targeted usability, visual clarity, and correctness of behaviour across the application.

Order creation improvements addressed several points raised during testing. The redundancy between the buyer name field and the name fields was eliminated. Item quantity input was simplified to a single control, removing the duplicate arrows alongside the plus and minus buttons. Required field validation was added, so that attempting to place an order without filling in all required fields now triggers a clear error message. A bug was also fixed in which the search functionality did not work when creating an offline order.

Navigation and redirects were corrected based on direct feedback. After creating an organisation, users are now redirected to the organisation selection page. After creating a user account, users are now redirected to the main page. Both of these resolved the disorientation participants experienced during testing. Creating a new user was also converted into a popup, consistent with the suggestion to reduce full-page transitions for simple actions.

Order and item viewing was improved so that items can be viewed from any item page within an order, rather than only being accessible through the start picking flow.

Filter and tab fixes addressed several issues with the filtering system. Sort labels were updated to be contextually accurate — priority sorting now displays "high to low" and "low to high" rather than the generic "A–Z" and "Z–A", and date sorting shows "earliest to latest" and "latest to earliest".

Mobile responsiveness was improved across the application to better support use on phones, which participants indicated would be the primary device during picking.

6. Conclusion

To conclude our project, we are very happy with our system produced and received good feedback from our client. In our conclusion, we discuss our success with the fulfilled requirements, discuss unfulfilled requirements as future work and finally reflect on the project as a result of our team effort.

6.1 Fulfilled requirements

Looking at the Requirements list, all Must requirements were fulfilled in the final system. The core functionality of the application was fully implemented, including the complete order-picking workflow, order and item overviews, issue reporting, status handling, offline order creation, responsive design, and navigation between overview, order, and item levels.

In addition, all Additional requirements were also fulfilled. The system supports the final summary page for completed orders, navigation back from that page to items, multi-organization support, login with different organization admins, registration of admins, pickers, and expediters, and inventory or item-status related management features.

The remaining gaps were only in some Should and Could requirements. The only unfulfilled Should requirement was filtering items by category to create a more efficient picking route. The unfulfilled Could requirements were optimized warehouse routing, warehouse bin scanning, proof photos, keyboard shortcuts, and multi-language support. These features were not implemented because they were considered optional improvements rather than essential project functionality.

6.2 Future Work

Future improvements of the platform should include multi-language support to make the application accessible to a wider audience. This can be implemented efficiently using POEditor, a tool that manages translations in a centralized cloud environment rather than hard-coding them into the files. By connecting POEditor to the project through its API or GitHub integration, the team can automate the synchronization of language files.

Another addition would be the implementation of a pathfinding algorithm to optimize how items are picked within the warehouse. Currently, the system lists items for collection, but it does not account for the physical layout of the building. Moreover, we would like email authentication in the future when creating a user and functionality to therefore reset a user's password. We would also like functionality of adding items to the order even after an order is created, such as in Bricklink and Brickowl, where it is possible to add batches of new items after an order is placed online as well. The final page of an order currently connects a user to the marketplace order page too but we would like functionality where if you update the status of the order, it updates on the marketplace too.

For more easy integration with our system, the client has requested us to create a github repository with issues for newer functionality that they want to implement. The client was positive with our work and would like to build on our project in the future for when they actually use the system. The client would also like to integrate our project in the future with a Product

Entity Database (PED) which stores information and better syncs their items in the warehouse with the online marketplaces.

6.3 Reflection on challenges faced and Teamwork

The project was a team effort and would not be delivered with great success if we did not function well as a team. Every project has its challenges and we had a very few of our own. One of our big ones was that in the beginning, we had begun our development in Django as we agreed in our proposal. We completed one week's progress with our order overview page and our connectivity between front end, backend and database established with Django and had to therefore move everything from Django to .NET. As a result, another challenge we faced was that not everybody was well versed with .NET so we spent about a week learning this but after this challenging week, we began development very quick and stuck to our schedule. We also received a lot of additional requirements during the course of the product development and we embraced these challenges satisfying all these Additional Requirements.

In terms of individual contributions to the team, we are very happy with our collaboration and contribution. Most of us worked on every aspect of the system but some people more on backend and more on front end. The team wanted to learn complete full stack development and therefore we did not split for only front end or only back end. The below Table shows our individual contributions:

Aadi Malhotra	JWT Authentication, creation of User functionality(backend), Order Issue creation(backend) and VIES[3] check for VAT
Mara Teodorescu	Overview Page and filtering functionality, responsiveness on some pages and MyParcel label generating services
Natasa Tudorache	Item page with updating statuses, timer of picking an item and full order and WeFact invoice generation services
Rian Sood	Organisation integration for database, Multi-picker order and Integration with online marketplaces(Bricklink and Brickowl)
Tania Maria Mincu	Order overview page, Images display for each item, progress bar for each order and create user popup (frontend)
Yash Sakore	Mobile front end design and responsiveness, Issue page(frontend) and final page of the order summary

Table 2: Contributions of the Team

The team was happy with the way the work was divided and chose the components what they want to work on, on their own branch and git issues as mentioned. To conclude our project, we would also like to thank our client for the opportunity to work on a fun and challenging project and for their complete guidance. We would also like to thank our supervisor for his feedback throughout the module. We are excited for the future and hope that the system can be soon integrated into our client's workflow.

[3] VIES - Vat Information Exchange System

Appendix

Link to the source code

Here is a link to our Gitlab repository - https://gitlab.utwente.nl/s3200183/designproject.

AI Statement

During the preparation of this work the authors used ChatGPT and Claude in order to Generate simple code which was reused across multiple controllers and services for backend and CSS for front end alignment and ease-of-use. After using this tool/service, the author(s) reviewed and edited the content as needed and takes full responsibility for the content of the work.

Requirements

We have added the requirements in the appendix due to the increasing number of pages for our page limit and referred important requirements in the main body.

MoSCoW Must Requirements

A. The system must show an overview of all open, pickable orders.
B. The system must show details about every order in the order overview such as order id, priority, issue with an order and status of an order.
C. The system must allow the user to open an order from an overview of orders.
D. The system must show a small overview of all items part of an order which includes the preview, location, description, colour and number of each item.
E. The system must allow the user to open an overview for every single item from the overview of a single order opened previously.
F. The system must present a detailed overview for each item, containing the item name, location, description, category, associated image, the quantity required for picking and the current inventory level in storage.
G. The system must allow users to introduce a ticket and comments for issues with items.
H. The system must allow users to introduce a ticket and comments for issues with orders.
I. The system must allow actions with an order and item to mark an order with its status such as picked, in progress or not picked.
J. The system must calculate the remaining item quantity after the user enters the number of items picked.
K. The system must be responsive on each device.
L. The system must allow easy switching between all three levels - Order overview, order level and item level.
M. The system must allow a user to create an offline order by adding items to that order.
N. The system must show in the order overview:
    a. Order number / ID
    b. Number of item lines

    c. Order status (e.g. “In picking”, “Completed”)
    d. Number of lines with an issue (if applicable)
    e. Notes field (optional free text field from the sales portal)
    f. Assigned user (order picker)
    g. Priority (optionally assigned manually)

O. The system must allow navigation between item lines (next / previous).

MoSCoW Should Requirements

A. The system should allow users to pick orders from the overview page and assign themselves to the chosen order.
B. The system should allow easy integration with MyParcel.-
C. The system should allow the user to assign priority to orders.
D. The system should have a clear column structure in both the overview page and order page.
E. The system should have visual indicators for issues and priorities.
F. The system should have rules created for pieces to make the user aware of confusing pieces.
G. The system should filter items in an order based on its category to create an efficient route.
H. The system should allow a user to time their order per item, if the user chooses that option.
I. The system should support integration with WeFact to generate and view invoices.
J. The system should show picking progress within an order (e.g. line X of Y or progress bar).
K. The system should support external product links (e.g. BrickLink / BrickOwl) from the item screen.
L. The system should support mobile swipe navigation on item-level screens.

MoSCoW Could Requirements

A. The system could allow multiple orders to be picked simultaneously.
B. The system could determine an optimized picking route to reduce walking distance in the warehouse.
C. The system could allow scanning of warehouse bins to confirm picked items.
D. The system could allow orders to be visible only to users with the appropriate role or assignment.
E. The system could allow the administrator or planner to view and manage all orders.
F. The system could allow order pickers to view only orders assigned to them or the global queue.
G. The system could allow users to change the sorting of open, pickable orders in the overview page.
H. The system could allow users to filter by columns such as status, number of lines, assigned user, notes.
I. The system could require users to take a photo of the items in an order to serve as proof of correct picking.
J. The system could support keyboard shortcuts for desktop picking.

K. The platform could support multiple languages, allowing users to switch the interface language.

MoSCoW Will Not Requirements

A. The system will not have an option to change the colour layout of the front end.
B. The system will not have the functionality to display a map with the location of the item.

Additional Requirements (added later in the project)

A. If an order is completed, the system should display a final page with a summary of it.
B. The system should allow going back to items from the final page.
C. The system must support registering organizations and private persons, each with their own orders, users and information.
D. The system must allow logging in with different organization admins.
E. The system should allow admins to register order pickers and expeditors.
F. The system could allow admins to modify inventory, using the +/- buttons.
G. The system could allow switching of item statuses with syncing of item quantity.

User Testing Plan

Brickworkz Order Picking Platform

1. Test Objectives

This user test aims to evaluate the usability and completeness of the Brickworkz Order Picking platform at its current advanced stage of development. The session will cover the full application flow as it exists today.

Specific goals:

- Identify usability issues in the order picking flow
- Verify that key features are intuitive without prior instruction
- Collect qualitative feedback on the overall user experience
- Detect bugs or unexpected behaviour encountered during use

2. Scope

The following areas will be covered during testing:

Area	What will be tested	In scope?
Login / Authentication	Login with valid credentials, error handling	Yes
Order overview	Viewing the list of orders, filtering relevant orders	Yes
Order detail	Opening an order, viewing items	Yes
Order picking flow	Picking items one by one, marking as complete	Yes
Issues	Creating issues for both order and item, viewing them in Issues overview	Yes
WeFact Invoices	Generate WeFact invoice for an order	Yes
Mobile responsiveness		Yes
BrickLink / BrickOwl integration	The connection with the external platforms	Yes

3. Participants

Tester profile: Brickworkz employee's, some have technical knowledge others don't.

#	Name / Alias	Role	Date
1		Tester	
2		Tester	
3		Tester	

Recommended: 3–5 participants. One team member should act as facilitator and one as note-taker during each session.

4. Test Tasks

Participants will be given the following tasks in order. Tasks should be read aloud to the participant. Do not explain how to complete them — observe how they attempt it.

#	Task	Success criteria
1	Create an organisation with admin credentials. Follow the steps that appear on-screen. Remember the admin credentials. **Note**: At step 3 of 'Create organization', select the first option.	User creates an organisation without any assistance.
2	Sign in with the admin credentials.	User signs in with the credentials they created.
3	Register a new user with the role set to **Picker**. Credentials do not have to be remembered.	User identifies where a new user can be registered and creates one without any assistance.
4	Create an order. Enter the first and last name, along with all required fields.	User creates an order and fills in all the fields that are necessary.
5	From the table, select the order created in **step 4**.	User identifies how to view the order details.
6	View the details of the order and then close the popup.	
7	View the details of an item from the order and then close the popup.	User identifies how to view the item details.
8	Start the picking process. You can choose which items are found or not found. At the end mark order as completed.	User finds the button used to start picking and goes through all items that need to be picked until they reach the completion of an order.
9	Go to "Completed orders" tab and select the recently completed order.	

#	Task	Success criteria
10	Find where you can generate a label for this order but do not generate one.	
11	Generate an invoice for this order and open the resulting PDF. Verify that the details match the order.	
(The following steps require an organization that already has users, multiple orders with different status, items in each order)		
12	Log out of the current account.	User identifies where the log out button is.
13	Select the organisation: FILL IN ORGANISATION	
14	Sign in with these credentials: FILL IN CREDENTIALS	
15	In the “Orders overview” tab, navigate through the available sub-tabs.	
16	In the “Orders overview” tab search in the table using any input you want.	
17	In the “Orders overview” tab **filter** based on: Status - Picked Priority - Medium Issues - Yes **Note**: You may filter the results based on your preferred criteria.	
18	In the “Orders overview” tab **sort** based on any field you want.	
19	Change the columns visible in the “Orders overview” tab.	
20	Clear all filters and options applied in the previous three steps. **Note**: Avoid resetting them manually.	
21	Go to an open order assigned to you. Navigate to an item page and create a warehouse issue. Go back to the order page. Create an issue on the order page.	
22	View these newly created issues in the “Issues” tab.	
23	Change the status of one of them to “Resolved”.	

5. Metrics & Evaluation

For each task, record the following:

Metric	Description
Task completion	Did the user complete the task? (Yes / Partial / No)
Errors	Any mistakes, confusion, or wrong paths taken
Assistance needed	Did the facilitator need to intervene?
Comments	Verbal observations or think-aloud notes

6. Session Structure

Each session should last approximately 25-30 minutes and follow this structure:

Phase	Duration	Description
Introduction	2 min	Welcome the participant, explain the think-aloud method, clarify what the user testing is for .
Tasks	15 min	Guide participants through tasks. Observe and take notes. Avoid giving hints.
Debrief	10 min	Ask open-ended follow-up questions (see Section 7).

7. Debrief Questions

Ask these after the session to gather qualitative feedback:

- What was the most confusing part of the app?
- Was there anything you expected to find but couldn't?
- On a scale of 1–5, how easy was the app to use overall? Why?
- Were there any steps where you weren't sure what to do next?
- Did the order picking flow feel logical to you?
- Was any text or label unclear or confusing? If yes, which ones and why?
- Did you encounter anything that didn't work as you expected?
- Was it easy to find the features you needed?
- Did any task feel too long or unnecessary?

8. Recording Results

Use one copy of this document per participant. Fill in the metrics table (Section 5) during the session, and add a brief summary after. Compile findings across all sessions to identify recurring issues.

After all sessions, prioritise issues found:

- Critical — blocks task completion
- Major — significant friction or confusion
- Minor — small UX improvements

User Manual – SyncYourBricks order management platform

Welcome

SyncYourBricks is a warehouse and order management system used to handle orders from different marketplaces and organizations. It helps teams manage orders from the moment they arrive until they are shipped and invoiced.

This guide explains how to use the system in a simple, practical way.

---

1. Getting Started

1. Open the application in your browser.
2. Select the organization you belong to or register a new one (if you are an admin).
3. Enter your username and password. If you are a picker or an expeditor, the admin should give you the credentials. If you are the admin, log in with the credentials you provided when registering your organisation.
4. Click Log in.
5. Everything you now see in the system depends on the organisation you selected.

---

2. User Roles (What You Can Do)

Your access depends on your role:

Admin

- Full access to all features
- Manage orders and assign pickers
- Update item and order status
- View and edit issues
- Register users
- View and edit inventory
- Configure API keys
- View all completed orders
- Generate invoices and shipping labels
- Create offline orders
- Pick orders

Expediter

- Manage orders and assign pickers
- Update item and order status
- View and edit issues
- View all completed orders
- Generate invoices and shipping labels
- Create offline orders
- Pick orders

Picker

- Pick orders
- Update item and order status
- Report issues with items or orders
- Create offline orders

---

3. Main Dashboard (Order Overview)

After logging in, you will land on the **Order Overview** page. This is your main workspace.

What you can do here:

- View all active orders
- Search for specific orders
- Filter orders (e.g., by status, date, or customer)
- Sort orders (e.g., newest or oldest first)
- Customize which columns you see
- Save your preferred layout

Tips:

- Use filters to quickly find orders you are working on
- Save your layout if you always work the same way

---

4. Working with Orders

Viewing an Order:

1. Click on any order in the overview.
2. You will see:
    - Order details
    - Items in the order
    - Current status
    - Assigned pickers
    - Any issues

Creating a New Offline Order (in store order):

1. Go to the Order Overview.
2. Click Create Order.
3. Fill in:
    - Customer details
    - Items
    - Quantity
4. Place the order.
5. The newly created offline order will be displayed in the order overview.

Assigning Pickers (Admin and Expeditor):

1. From the order overview, select the picker from the dropdown in the “Assigned picker” column.

Pickers will now see the order in their workflow.

---

5. Picking Process

This is the most important part of the warehouse workflow.

1. Open your assigned order*.
2. Check out the order details by clicking “View order details”.
3. Click “Start Picking”.
4. For every item in the order, click on the colorful buttons:
    a. “Item picked” if you have found the item in the warehouse and picked it
    b. “Not found” if you did not find the item in the warehouse
    c. “Not enough items” if there are not enough items in the warehouse (also enter the number of items you have picked)
    d. “Warehouse issue” if you want to report an issue

5. Once every item has been picked, you will be taken to the final page to confirm and mark the order as packed.
6. The admin and expediter can also generate invoices and shipping labels at this stage. They will be downloaded as PDFs.

*It is also possible to pick an order with multiple pickers. In this case, different items from the order will appear in each picker's workflow.

---

6. Handling Issues

Sometimes problems happen during fulfillment. The system allows you to report them easily.

Reporting an Issue

You can report issues for:

- An entire order (from the order page)
- A specific item (from the item page through warehouse issue)

Viewing Issues

- Expediter and Admin users can see all reported issues in a separate page. They can be filtered by status.
- Issues are tracked until they are resolved.

---

7. Marketplace Orders

Orders can come from external marketplaces. You do not need to create these manually.

They will:

- Appear in the system automatically
- Be synchronized with marketplace data
- Move through the same workflow as manual orders

---

8. Personalizing Your View

You can adjust your order overview:

- Show or hide columns
- Change sorting order
- Apply filters
- Save your preferred layout

This helps you work faster and focus only on relevant information.

---

9. Registering users

The admin can register users with different roles (Picker/Admin/Expediter) from the “Register user” tab from the sidebar.

---

10. Inventory

The admin can view the inventory and edit the item quantities from the “Inventory” tab from the sidebar. There is also a search functionality that allows admins to find items faster by searching by item name or item id.

---

11. Summary

SyncYourBricks helps your team:

- Manage orders from multiple sources
- Assign and complete warehouse picking tasks
- Track issues clearly
- Handle shipping and invoicing
- Work together in real time

If you follow the workflows in this guide, you can move orders smoothly from arrival to completion.