IMPLEMENTING REMOTE ACCESS SMART TAG PRINTING FOR KIMBERLITE

Implementation process of a new feature for business

Alex Ihonen & Rajat Rijal
Bachelor’s Thesis
Spring 2025
Degree Programme in Information Technology Engineering
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences

Degree Programme in Information Technology Engineering

Author(s): Alex Ihonen, Rajat Rijal

Title of thesis: Implementation of Remote Access SmartTag Printing for Kimberlite

Thesis supervisor(s): Pasi Mustonen

Term and year of completion: Spring 2025

Pages:

This thesis was developed for a new way to approach the flea market business with a feature helping an existing platform called Kimberlite, which provides a platform for customers to engage in flea markets currently in the Ostrobothnia region. Currently operating in Järkikirppis Oulu, the flea market establishment provides a digitalizing feature to modernize the flea market business. Table owners gain a platform to market their products and increase visibility for all the different tables and their contents. Kimberlite also offers a SmartTag system to prevent issues like traditional tagging frauds or missing tags on items. Kimberlite also features a program to scan items in a studio to provide presentable images and descriptions of the items being sold with the help of artificial intelligence.

The thesis project is initiating a new way for Kimberlite to sell tags with better efficiency. With a remote access feature, customers can order tags before bringing them to their tables. As most of the backend of the project has already been established, this project will focus on the front-end implementation and explaining how it works with the existing system.

TABLE OF CONTENTS

Abstract 2

Glossary 5

Introduction 7

1 Kimberlite 8

2.1 How Kimberlite works 8

2.2 Problem Statement 9

2.3 Solution Statement 11

2 Implementation 14

3.1 Requirements 14

3.2 Design 14

3.2.1 Styling 15

3.3 Frontend 17

3.3.1 React Frameworks 18
3.3.2 Routes 21
3.3.3 Validation and Erro handling 22
3.3.4 Tailwind 23

3.2 Backend 24

3.2.1 Objectives 25
3.2.2 Core Technologies 27
3.2.3 Backend Structure 27
3.2.4 Data Handling 28
3.2.5 API Endpoints 29
3.2.6 File Upload Mechanism 31
3.2.7 Storage Mechanism 31
3.2.8 Validation and Error handling 33
3.2.9 API 34

3 System Architecture 37

4.1 QR Scan and Request Handling 37
4.2 Product Upload via Form Submission 37
4.3 Data Status and QR Code Generation 37
4.4 Printing with Kimberlite Machine 38
4.5 Passkey System 39

4.6 Templates Engine....40
4.7 Container & Deployment....41

4 Testing....42

5.1 Phase 1....42

5.1.1 Problems Faced During Testing in Phase 1:....43

5.2 Phase 2....44

5 Discussion....45

6.1. Comparison between the total time taken by old system and new system....46

6.2 Security considerations....47

6.3 Future Implementation....47

6.3.1 Payment Method....47
6.3.2 Check in and Checkout System....48
6.3.3 Security Measures....48
6.3.4 Status of the Items....48

6 References....49

GLOSSARY

Term	Definition
Fast API	It is a modern high-performance web framework for building APIs with Python. Designed to create web applications.
HTML (Hypertext Markup Language)	It is the standard language used to create and structure content on the web.
Jinja templates	It is a web templates engine for Python used to generate dynamic HTML content in web applications.
JavaScript (JS)	It is a programming language used for creating function-based programs.
QR Code (Quick Response)	Are barcodes which store text, URLs or other machine-readable information. Commonly used for links to websites or authentication.
Tailwind	It is a framework that provides predefined classes to build custom designs directly in HTML.

Swagger	Swagger is a backend API documentation interface
Multer	Is a node.js middleware for handling multipart/form-data.
CORS	Cross-Origin Resource Sharing is a security measure to control access from foreign domains.
NODE.js	Is a runtime environment which allows developers to run JavaScript code on the server side to build scalable and fast network applications.
Express framework	Minimal web application framework for Node.js
Figma	Is a web application for creating and designing purposes

INTRODUCTION

In January 2025, both authors of this project started their internships at Alvidiotech. After a successful internship period, in April, we were presented with an opportunity to work on our thesis project with the systems and products we were already familiar with.

This project is about improving and extending already established systems by the company. Working with their existing features to provide a new tool of operation in the existing market alone and offer a project opportunity to work with a real business. The product this project will revolve around is called Kimberlite, and it is an already published and working product run by Alvidiotech in Oulu.

Currently as it exists, customers enter the flea market with products tagged with traditional hand-written tags. If they would like to add them to the Kimberlite platform, they would have to notify the staff and let them know which products they would like to be added to the webpage. After that the staff would collect the items and start scanning them one-by-one, tagging and placing them on the customer's table.

The task was that the project would extend its features further by introducing a new way of handling customers' goods. We came up with the idea together with our project supervisor to implement a web application with smart-tag printing feature.

The user would pick their product and set a description and a price with their own device, then list it to the queue using the Kimberlite interface and next time they visit the store, they can print out the smart tags from the system with their table card they receive from the flea market store. This will increase the efficiency of listed items on the website and save work hours. This project was initiated and made for Alvidiotech, Oulu.

1 KIMBERLITE

Kimberlite is a service and platform providing web-application. In its current state, it is being utilized in flea markets and other second-hand stores online as a digital marketing site for increased visibility. More visibility means more customers find the products they need. Inventory information is updated automatically and instantly by the Kimberlite users and staff, and new products are visible on the site within seconds. Kimberlite comes with its own hardware and software which is provided and maintained by the service provider.

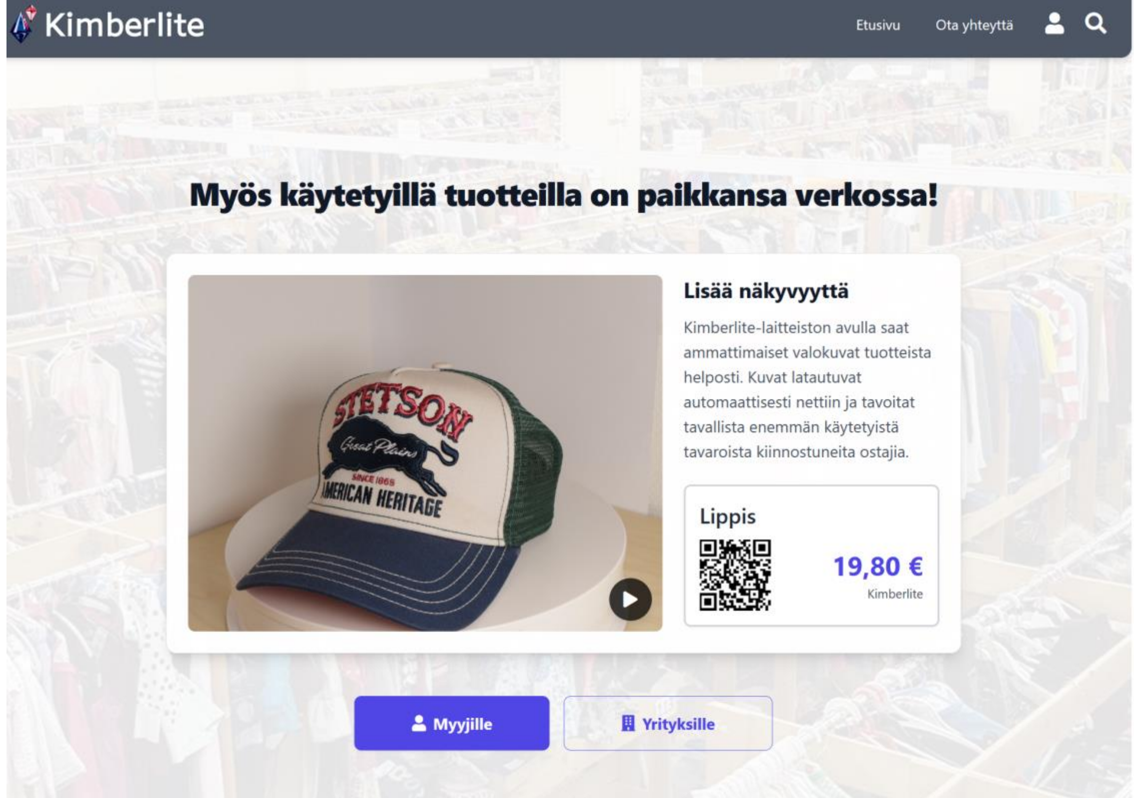


Figure 1 Kimberlite Frontpage[(5)](5)

2.1 How Kimberlite works

Currently, Kimberlite is operating on Järkkirppis at Tehtaankatu 1 Oulu. Kimberlite has its own hardware which is called Kirppiskamu. Kirppiskamu has a camera setup, QR printer, power relay, access point, rotating table, tablet QR reader,

automatic lightning system, back walls, computer and screen. The whole setup is placed in the corner of the shop. The current system is run on a Linux-based machine, and initial setups run through a terminal. It requires staff assistance to use it, and customers must get to the front desk of the store and ask the staff to use it. Once staff are available, they can go to the kirppiskamu and take pictures from the kirppiskamu and upload the items online with required data in the input fields. Once the item is online, the printer will print the tag itself. The whole process for a single item takes around 30 seconds per item.

Figure 2. First demo of Kirppiskamu

2.2 Problem Statement

Despite the platform's digital nature, the current process for generating and printing QR codes is marked by several inefficiencies that hinder scalability and operational efficiency.

- The product upload process is manual. Vendors are required to upload each product individually, which becomes increasingly time-consuming and impractical when dealing with large inventories.
- The system is heavily reliant on staff support.
- End-users often need assistance from employees to complete the QR code process, contributing to increased labour costs and reducing system autonomy.
- The time required to generate and print a single QR code is approximately 30 seconds.

In high-volume scenarios, such as handling thousands of items, this leads to significant delays and creates operational bottlenecks. The dependency on manual labour also hinders the scalability of the solution. As user numbers increase or larger product batches are processed, the inefficiencies become more pronounced, resulting in extended wait times for vendors.

Given a target of processing 50,000 items per month per vendor, the current system lacks the necessary efficiency and automation capabilities to meet such demand. Therefore, a more scalable and automated approach is required to improve overall performance and reduce reliance on human resources.

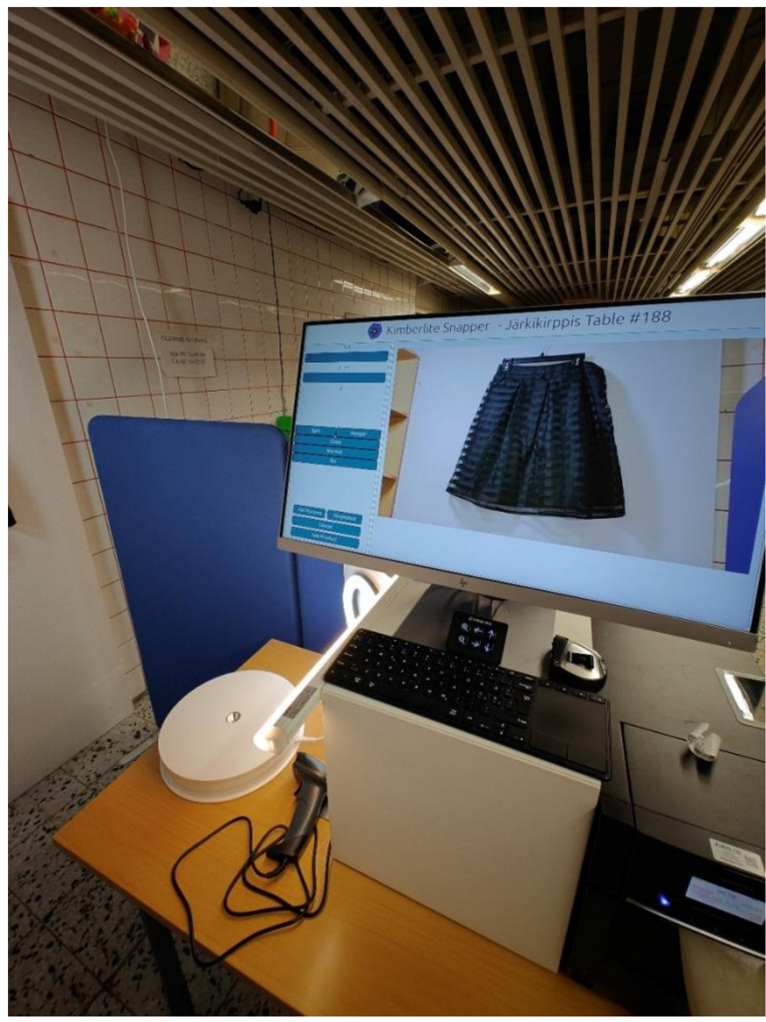


Figure 3 Kirppiskamu controls

2.3 Solution Statement

To address the problems mentioned in chapter 2.2, this project proposed the development of the Remote Access SmartTag Printing System web application alongside the Kimberlite web-app.

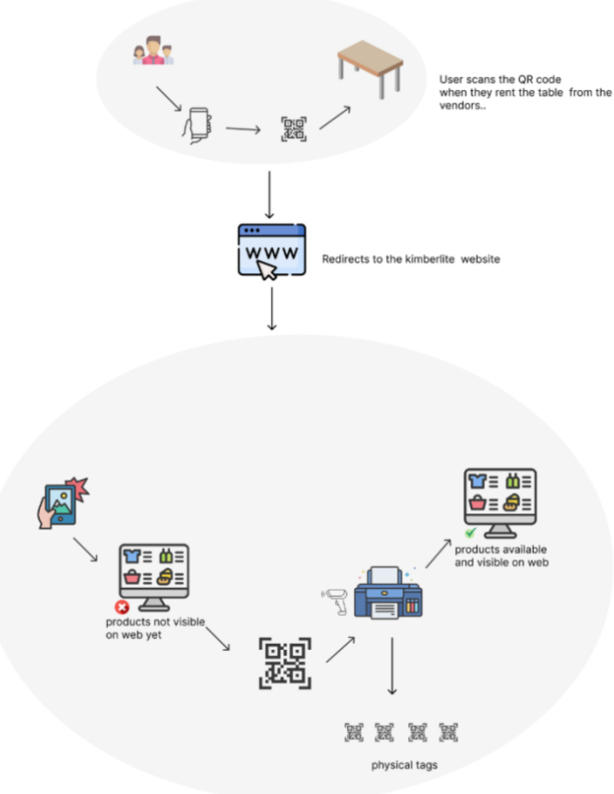


Figure 4 Representation of Kimberlite Web-app

In the current Kimberlite system, when a user rents a sales table from a vendor, they are provided with a table card containing a QR code and a unique passcode. This passcode allows the user to log into the Kimberlite website, where each table is individually identified and registered with its corresponding card. Once logged in, the user can upload product images along with all necessary details to the platform.

However, the uploaded products remain hidden from the public interface until the QR code is printed and activated. The system automatically generates a single QR code for the batch of uploaded products, which is required for use with the Kirppiskamu self-service terminal. This self-service terminal enables users to independently print their QR codes, thereby minimizing reliance on staff and reducing operational bottlenecks.

The process has been designed to streamline efficiency. Users can retrieve their QR codes directly at the printing station by scanning the code generated on the

Kimberlite website. This reduces the number of manual steps involved and allows the QR code printing process to be completed independently by the user.

The proposed self-service model significantly enhances operational efficiency by automating both the QR code generation and printing processes. As a result, the solution allows vendors to manage a high volume of products without requiring additional staff. Furthermore, the user experience is improved through a reduction in wait times and increased autonomy, allowing users to complete the process without external assistance.

One key advantage of the new system is that it requires only a single machine to perform both the QR code generation and printing tasks. This simplification reduces the need for multiple devices and eliminates unnecessary procedural steps. In addition, the system's independence from staff interaction contributes to reduced operational costs. Users can print an entire batch of QR codes instantly, without having to wait for assistance, which is especially beneficial during peak usage periods.

Overall, the implementation of the self-service QR code printing system supports scalability, enhances user satisfaction, and increases overall efficiency. It represents a significant improvement in the manual, staff-dependent approach currently in use, offering a more practical and sustainable solution for high-volume operations.

2 IMPLEMENTATION

The proposed Remote Access SmartTag Printing System was built with a modular and scalable architecture that ensures good performance, easy maintenance, and smooth deployment. The system follows a typical client-server model and is developed using modern web technologies. Below is a breakdown of the main components used in the system: frontend, backend, database, deployment, and other supporting tools.

3.1 Requirements

The current method of scanning and printing labels for products is a time consuming and inefficient method. This project’s main goal is to aim to develop a new feature to address this by giving the tools for the user to scan and print labels themselves with initially a very simple frontend and basic backend skeleton to simulate testing. Chapter 3 focuses on the tools used to implement this.

Developing a feature which requires user interactivity must naturally be easy to use and have wide accessibility with different devices. As of today, many current users have a smartphone and will most likely use that device to capture images, scan QR-codes and use the Kimberlite service. Therefore, our primary focus is to develop smartphones as the primary device.

3.2 Design

We designed the features with the same respective Kimberlite logos and style, using the colours and styles from the design documentation.

Figure 5 Kimberlite logo prism

3.2.1 Styling

We started designing the application initially with Figma, which is an online mockup tool to design and draw mockups. With Tailwind components (see chapter 3.3.4) we were able to import from its existing library all the colour styling required. Then we created within an array all the components to be readily available for assets to be used.

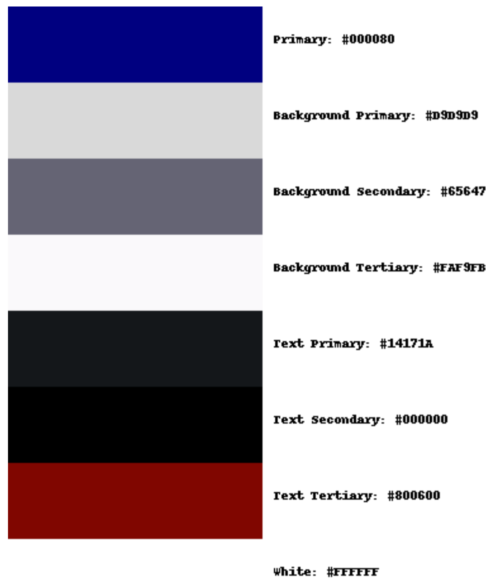


Figure 6 Colours used as tailwind.css components.

Going through the colours we have chosen, Primary blue colour is found in button and title components. Background primary and secondary upload components found in Figure 7. Background tertiary for the base white colour contrasting under all the assets.

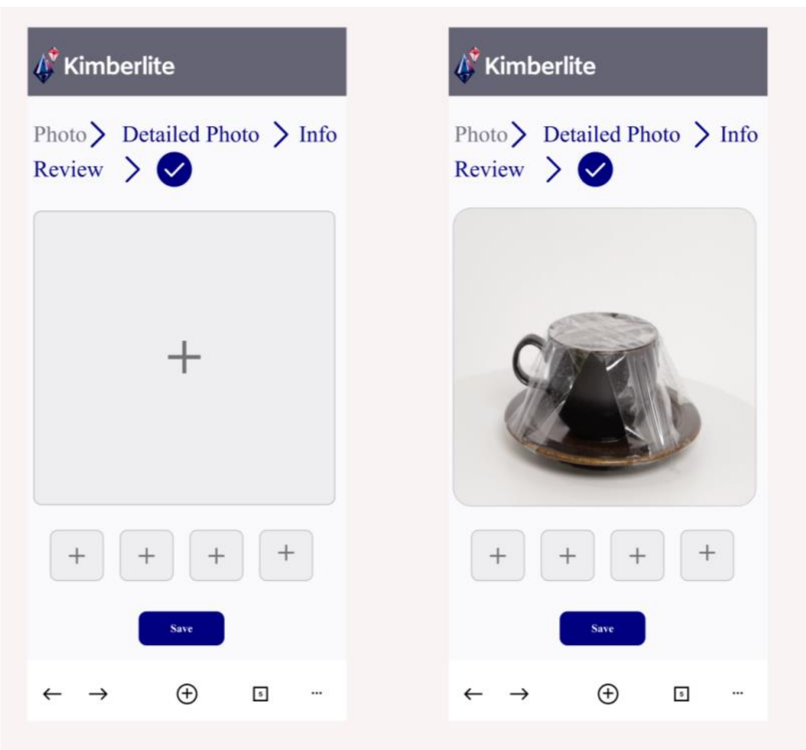


Figure 7 Mockup of upload page

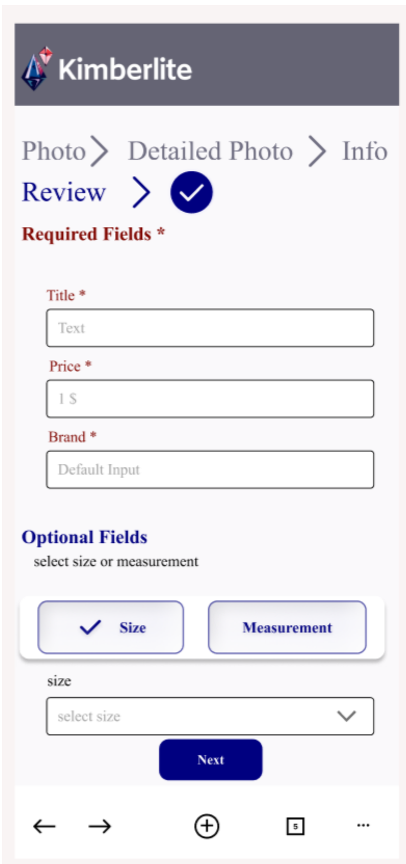


Figure 8 Input fields to describe the product

Talking about the design approach, in the beginning a visual mockup was created to guide the layout and user interface. However, during the development phase the focus was shifted more towards the functionality of the system, making the User Interface simple and easy to use. Once all the features work smoothly, the initial design will be implemented in the future update.

Figure 9 Simplified UI

3.3 Frontend

The frontend is responsible for rendering and ensuring properly working components and functionality of the user interface. Designing the frontend was done together with the commissioner of this project, with initially a simpler approach. Main requirement that the user interface that can be built and rendered with most devices, but foremost for smartphone devices which is going to be the common device using this application. Like all frontend user interfaces, it all begins with choosing a Framework.

Frameworks is technology with a pre-built collection of tools and libraries that help develop our product more efficiently. Popular frameworks like React, Vue, Angular, or Flutter are found in the most commonly used applications(6). To achieve our product to have the widest accessibility, we wanted to use similar popular technologies which already are known to be working on most devices,

rather than developing. Next chapter will explore the choice of framework for the frontend of this project.

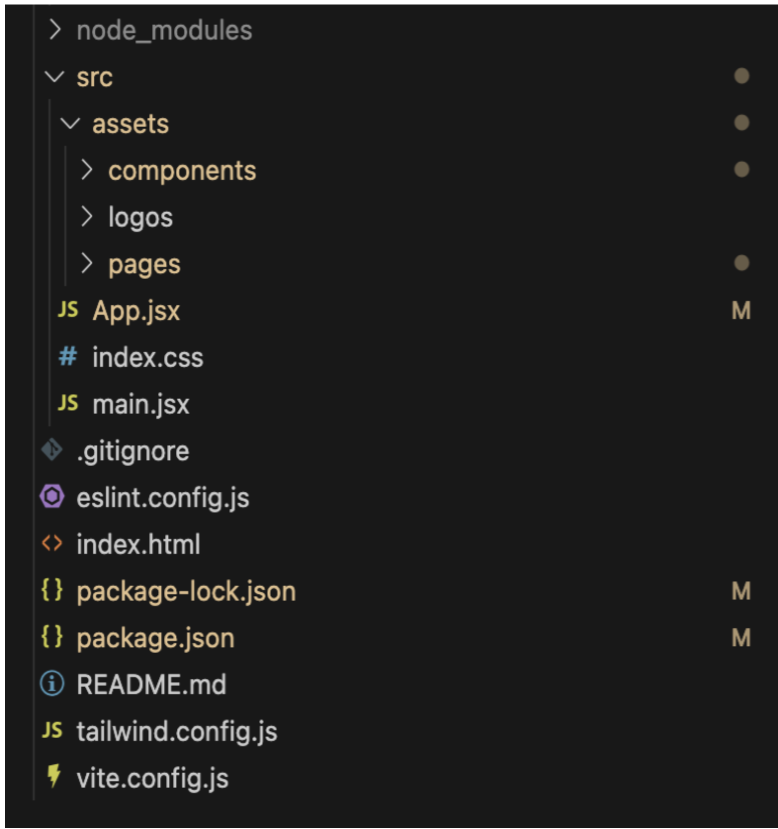


Figure 10 Folder structure of frontend

3.3.1 React Frameworks

React Frameworks is a JavaScript library for building user interfaces (UI) developed by Meta. React is widely used with many applications such as Facebook, an application found in most smartphones today, therefore an easy choice for this project(12).

React offers a component-based approach to building the application together with JavaScript(2). Components act like formulas to define which elements we want to use.

```json
{
  "name": "kimberlite-frontend",
  "version": "0.0.0",
  "lockfileVersion": 3,
  "requires": true,
  "packages": {
    "": {
      "name": "kimberlite-frontend",
      "version": "0.0.0",
      "dependencies": {
        "@tailwindcss/vite": "^4.1.7",
        "react": "^19.1.0",
        "react-dom": "^19.1.0",
        "react-router-dom": "^7.6.1"
      },
```

Figure 11 React Virtual-DOM

The special part about using React is the Virtual-DOM (Document Object Model) which enables all the specific changes we make in the application to apply fast and efficiently. It does so by comparing the old version of the application and only applies to specific changes made.

Finally, React also hosts a rendering API. It’s initialized with createRoot imported from the react-dom/client and finds the root element from the index HTML where its then rendered(3).

```html
index.html > ...
1   <!doctype html>
2   <html lang="en">
3     <head>
4       <meta charset="UTF-8" />
5       <link rel="icon" type="image/svg+xml" href="/vite.svg" />
6       <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7       <title>Vite + React</title>
8     </head>
9     <body>
10      <div id="root"></div>
11      <script type="module" src="/src/main.jsx"></script>
12    </body>
13  </html>
14
```

Figure 12 React rendering on line 10

```jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

Figure 13 React-DOM Rendering the page

When the frameworks are in place, it helps to keep the project organized and efficient while we implement the rest of the features. Next chapter

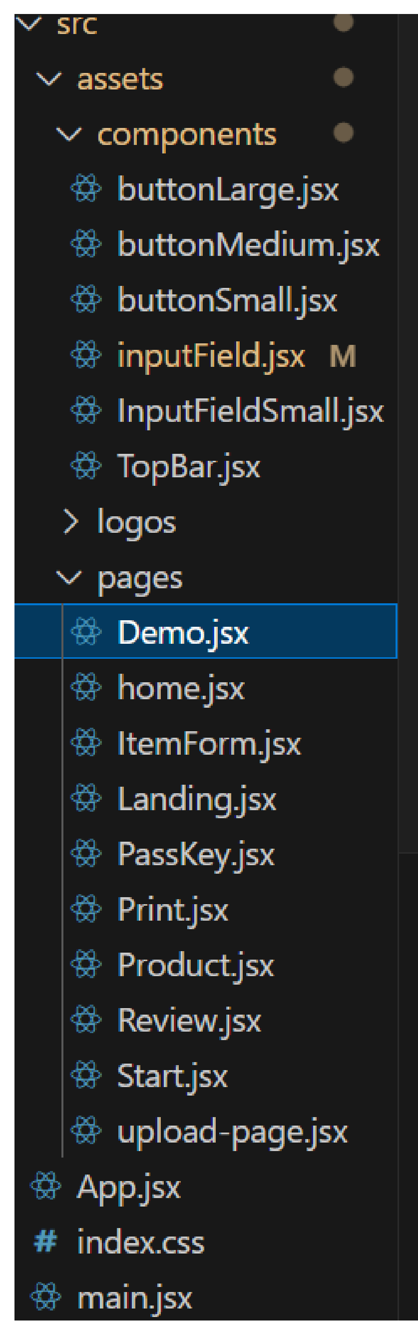


Figure 14 Frontend components and pages

3.3.2 Routes

For better user experiences we used the React Router Dom to handle all the routing and navigation between multiple pages. It allowed us to manage the navigation by defining routes that connect the URL paths to specific pages.1

All the routing is defined in App.jsx file which is in the root of the folder. <BrowserRouter> is used to enable client-side routing that allows us to navigate between different pages without full page reloads.

```
function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Landing />} />
        <Route path="/itemform" element={<ItemForm />} />
        <Route path="/review" element={<Review />} />
        <Route path="/start" element={<Start />} />
        <Route path="/demo" element={<Demo />} />
        <Route path="/home" element={<HomePage />} />
        <Route path="/upload/:itemId"  element={<UploadPage />} />
          <Route path="/passkey" element={<PassKey />} />
        <Route path="/print" element={<Print />} />
          <Route path="/item-details/:id" element={<Products />} />
        <Route path="/create-passkey" element={<CreatePasskey />} />
      </Routes>
    </BrowserRouter>
  )
}
```

Figure 15 Routing paths in App.jsx

From the figure, we can see the different routes where the route "/" is the main or home page of the website. Routes like "/page_name" is to go the specific pages. Dynamic routes like /:itemId and /:id use variables to load content based on the item or record ID.

3.3.3 Validation and Erro handling

The frontend handles errors and validates the data to ensure the data is correct. Once the data type is verified by the system, only the frontend sends it to the backend.

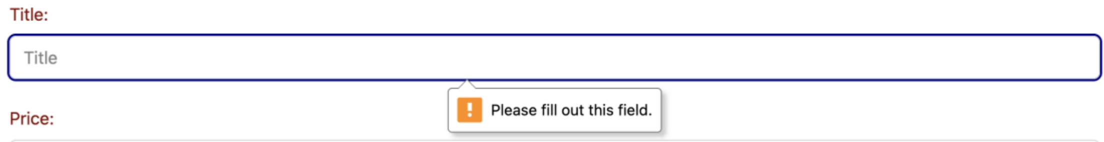


Figure 16 error handling in frontend

As seen in. the figure form cannot be submitted with empty space.

```
if (price === "" || isNaN(Number(price)) || Number(price) <= 0) {
  alert("Please enter a valid price greater than 0.");
  return;
}
```

Figure 17 validation of data type

Here is another example of how the frontend handles data. From the figure we can see that price cannot be empty space, number should be a valid number and greater than 0.

3.3.4 Tailwind

The most widely used web interfaces are written with HTML/CSS. HTML defines the structure and content elements of the web page, and CSS controls the appearance and layout of those elements. Tailwind is a CSS framework for building modern interfaces applying single-purpose classes called components directly to HTML(11).

```
"@tailwindcss/vite": "^4.1.7",
```

Figure 18 Tailwind version used in project

```
<button
  onClick={onClick}
  className=" min-w-[130px] min-h-[52px] bg-primary  rounded-xl px-4 py-2 text-white "
>
```

Figure 19 Using Tailwind components

The visual result of this would be a button that has a blue background, displays white text, is padded for clickable space, and has rounded corners. Traditionally you would have to first create the element in the HTML folder and then have a separate file for CSS to define all the class styling. With Tailwind, when it's directly implemented in the HTML file, eliminating a separate folder for CSS.

Figure 20 Tailwind components on start page

Simply, we create a separate folder to contain all the components necessary for the frontend and then export them to be utilized in the main rendering file. It increases the clarity of the project and makes it easier to make changes to the code.

3.2 Backend

Implementing the backend had to follow the same principles as designing the frontend. We won't have access to and use the pre-existing backend used with Kimberlite since it is not required for the development of this feature. Additionally, at the time of this writing, the Kimberlite backend was not available for us to use.

We developed a backend which enables us to create simple tests for use-case scenarios. First, we started building the backend system using Node.js with

express Frameworks(1). It is also supported by FastAPI, which is a high-speed Python web framework and it also supports QR code scanning, image capture, metadata entry, and web-based uploads. Using FastAPI improves speed and supports async operations, which is useful when handling many users at once. FastAPI hosts all the required elements we need and is compatible with many devices.This system offers a simple file-based CRUD API that supports image uploads, passkey management, and item storage along with providing interactive API documentation through Swagger(10).

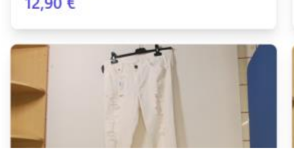
This backend is designed in a way that does not require a database for now. This backend system was created for this project and for testing purposes only. In this system all data persistence relies on local JSON files and uploaded files, i.e. pictures are stored in a dedicated directory i.e. Kimberlite/backend/uploads/pic.jpg inside the project file.

3.2.1 Objectives

This system supports full CRUD operations, allowing the user to create, read, update, and delete items. Every item has also the possibility to include uploading and storing multiple pictures per item.

Figure 21 Image upload page

To integrate the first level of security in this system, passkey management not only secures a customer's items behind a passkey but also gives us some tools to manage specific tables.



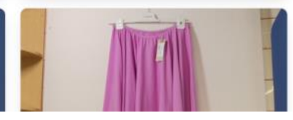


Figure 22 Table #188 in Kimberlite

With every backend, API documentation is important when building and developing the system. FastAPI provides us with default API documentation software, but we found the interactive and intuitive use of Swagger to make it clearer. Swagger provided us with an easy way to understand and test endpoints. Additionally, we use cross-origin access is enabled via CORS to ensure a smooth communication with our frontend applications(7).

3.2.2 Core Technologies

This backend uses several core technologies to make the system simple, flexible and easy to test

- Express.js: This is the main framework that handles routing and middleware. It helps to create RESTful APIs quickly and with minimal setup.
- Multer: Middleware to process multipart/ form-data file uploads. It handles file uploads, i.e. pictures(13).
- UUID: Generates unique identifiers for each item.
- Swagger: Provides an Open API-compliant API documentation which is very useful for developers.
- CORS: It allows cross-origin requests, so the frontend and backend can communicate without any issues.
- Node.js: Handles reading and writing JSON files for data storage and persistence(8).

  This technology was chosen for its lightweight and high performance. And it enables developers to test without a database.

3.2.3 Backend Structure

The backend consists of:

Index.js: This is the main file that contains all the routes, definitions, and logic.

data.js: It stores all the data that is provided by the user from frontend in Json format.

/uploads: This is the folder that stores all the pictures provided by users locally.

Passkey. Json: This stores passkey string in json format.

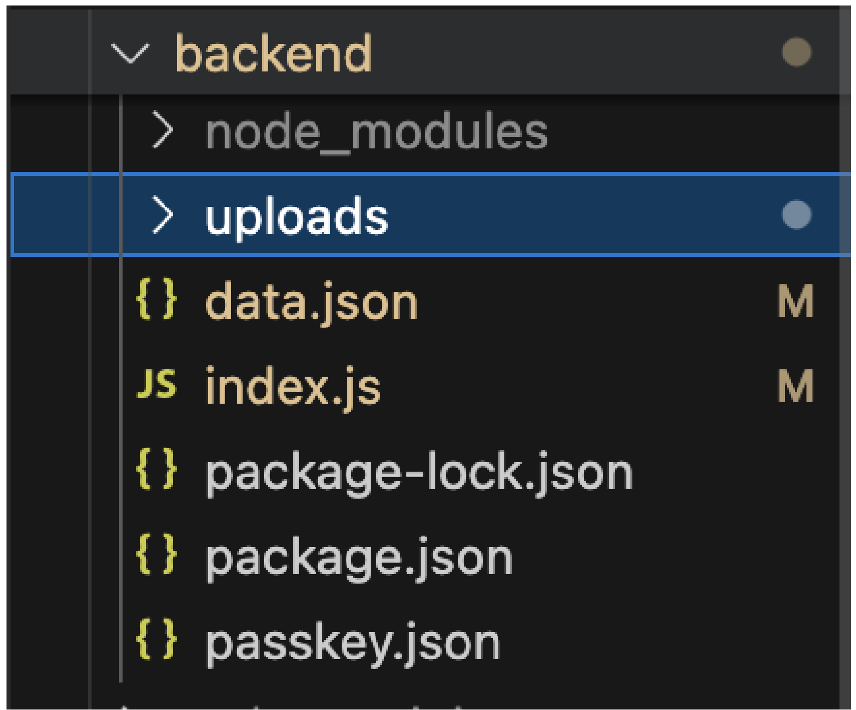


Figure 23 Backend structure

3.2.4 Data Handling

This backend saves all data in data.json and passkey.json files locally instead of a database. There are two types of data handled by the system.

1) Items Data (data.json)

Each item is stored as an object with these fields:

Id: a unique ID is created for each item using UUID.

title: The name of the item.

description: Details about the item.

price: price for the item.

pictures: An array of picture file paths.

All the items are saved together in data.json as an array.

2) Passkey Data (passkey.json)

This file stores a single passkey used for simple validation.

```
1  {
2    "passkey": "passkey"
3  }
```

Figure 24 Storing passkeys in passkey.json

This is the format for passkey in json format. This system saves a passkey when it is created or updated, loads it when it's whenever validation is required and returns an error if no passkey is stored.

```
if (!passkey) {
  return res.status(404).json({ error: 'Passkey not found.' });
}
```

Figure 25 Whenever passkey validation fails

3.2.5 API Endpoints

There are multiple endpoints implemented in this project. Here is an overview of implemented routes.

1. /items – Create a new item without pictures

Method: POST

Content-Type: application/json

Description: Creates a new item without pictures.

Validation: Ensures title, description, and price are provided.

Persistence: Appends the new item to data.json.

2. /pictures - Upload pictures and save URLs into the item's pictures array

Method: POST

Content-Type: multipart/form-data

Description: Attaches uploaded pictures to an existing item.

Validation: Checks itemId and existence of files.

Processing: Stores files in uploads/ and updates the pictures array of the item.

3. /items – Get all items

Method: GET

Description: Returns all items stored in data.json.

4. /items/{id} - Get an Item by id

Method: GET

Description: Retrieves details of a single item by ID.

5. /items/{id} - Delete an item by id

Method: DELETE

Description: Removes the item permanently from data.json.

6. /passkey – Create or Update Passkey

Method: POST

Content-Type: application/json

Description: Saves a passkey for future validation.

7. /passkey - Get Current Passkey

Method: GET

Description: Retrieves the stored passkey.

8. /passkey/validate – Validate Passkey

Method: POST

Content-Type: application/json

Description: Checks if the provided passkey matches the stored one.

3.2.6 File Upload Mechanism

This system uses Multer configuration with local storage.

Files are named as: uuid + original extension. For example, 4dd96ae5-6f91-46f6-8100-23c9510b3f5e.jpg. The maximum number of files accepted per item is 5 and stored in the /uploads static route, allowing the frontend to access them directly.

3.2.7 Storage Mechanism

Instead of using a database, this project uses two JSON files for storing data. All the item data is saved in data.json as an array of objects. Each object has fields like id, title, description, price, and pictures. When new items are added or deleted, this file is updated accordingly. Files are stored in a separate folder i.e. /uploads and path of the pictures are saved. For storing the passkey, a separate file called passkey.json is used. This file keeps a single passkey value that can be created or changed through the API.

This approach was the best to keep the system simple and easy to test without any setup or extra dependencies. It helps to check and edit data directly from the files if needed.


```json
{
  "id": "00a4edb7-3db0-434f-9bb0-e6f371e10fb4",
  "title": "test1",
  "description": "test",
  "price": 20,
  "pictures": [
    "../uploads/4dd96ae5-6f91-46f6-8100-23c9510b3f5e.jpg",
    "../uploads/312144ad-c855-422a-9d11-2504af328c32.jpg",
    "../uploads/883afc37-e8f3-4b64-8806-8106729a8a6d.jpg",
    "../uploads/9ca62aff-1107-48a0-aef4-3f43c95a06e2.jpg",
    "../uploads/8c9d19ad-fbb9-4eac-8a84-46713205cd85.jpg"
  ]
},
```


Figure 26 Simplest way of storing data as .json

```
uploads
0c87aa03-68f0-4c32-89bf-237ca6bfafa6.jpg U
1c6e9cb0-10be-45e3-914b-62909d8434cc.jpg
3c730b0e-49bb-4d17-86ae-921c1a697a91.jpg U
3da672cb-d243-4f7e-9262-68028aa0e0fb.jpg
4dd96ae5-6f91-46f6-8100-23c9510b3f5e.jpg U
6c3282b2-f213-453f-be75-cc0c53e26a52.jpg
8ae774dc-4eaf-4618-a225-3a472cf2e92b.jpg U
8c9d19ad-fbb9-4eac-8a84-46713205cd85.jpg U
9aa28d2e-d13b-4ab9-8a2f-c08111db2232.jpg
9ca62aff-1107-48a0-aef4-3f43c95a06e2.jpg U
9f80780f-459b-4d75-beee-41c5c3ab27ba.jpg U
41c2bb16-3969-4545-b3cd-40b5a33647bb.jpg
573ae7d6-acc1-4ced-a9af-8db465950f85.jpg
609fe653-3b06-4953-a299-3ac744d7c217.jpg U
751c7b2b-8c06-456f-aaa3-b658cdfaa21d.jpg U
883afc37-e8f3-4b64-8806-8106729a8a6d.jpg U
25574bdf-9920-4910-86b8-f745654e9296.jpg U
312144ad-c855-422a-9d11-2504af328c32.jpg U
5740602d-7a80-4317-95fa-00b0577f92fa.jpg
```

Figure 27 Uploaded images stored locally

3.2.8 Validation and Error handling

This system performs basic validation.

Checks for the required fields. Ensures Price is numeric. Confirms that all the input fields and pictures are attached before submitting the form or deleting an item.

For example:

```
if (!passkey || typeof passkey !== 'string' || passkey.trim().length < 6) {
  return res.status(400).json({ error: 'Passkey is required and should be at least 6 characters.' });
}
```

Figure 28 Backend will only accept more than 5-character passkey.

```
app.delete('/items/:id', (req, res) => {
  const { id } = req.params;

  let items = readData();
  const index = items.findIndex(item => item.id === id);

  if (index === -1) {
    return res.status(404).json({ error: 'Item not found.' });
  }
```

Figure 29 Backend handling the error for delete method

```
if (!item) {
  return res.status(404).json({ error: 'Item not found.' });
}
```

Figure 30 Error handling in index.js

3.2.9 API

The project uses Swagger to document all API endpoints. Swagger automatically generates interactive documentation where all available routes, request formats, and responses can be viewed and tested in a web browser. The documentation is set up using the swagger-jsdoc and swagger-ui-express libraries. It includes details about each endpoint which are provided in detail in section 3.2.4. The Swagger UI is available at http://localhost:3000/api-docs. This makes it easier during the development and to understand how the backend works and to try out the API without writing extra tools or scripts.

Figure 31 Swagger user interface

localhost:3000/items


```
[
  {
    id: "00a4edb7-3db0-434f-9bb0-e6f371e10fb4",
    title: "test1",
    description: "test",
    price: 20,
    pictures: [
      "../uploads/4dd96ae5-6f91-46f6-8100-23c9510b3f5e.jpg",
      "../uploads/312144ad-c855-422a-9d11-2504af328c32.jpg",
      "../uploads/883afc37-e8f3-4b64-8806-8106729a8a6d.jpg",
      "../uploads/9ca62aff-1107-48a0-aef4-3f43c95a06e2.jpg",
      "../uploads/8c9d19ad-fbb9-4eac-8a84-46713205cd85.jpg"
    ]
  },
  {
    id: "16de1cf5-c602-40f1-8c97-6fd002ee4dec",
    title: "test2",
    description: "test2",
    price: 20,
    pictures: [
      "../uploads/a33dbbf7-a075-45c2-84da-dbdea3f045b3.jpg"
    ]
  }
]
```


Figure 32 API endpoints for /items

Here is the list of all the API endpoints used

- POST /items - Creates a new item without pictures
- GET /items - Fetches the items
- POST /pictures - Upload pictures and save URLs into the item's picture array
- GET /items/{id} - Get an item by specific ID

- DELETE /items/{id} - Delete an item by ID
- POST /passkey - Create or update the passkey
- GET /passkey - Get the current passkey
- POST /passkey/validate - Validate a passkey

3 SYSTEM ARCHITECTURE

The process begins when a user scans the QR code located at the table or given by the shop staff where they have rented. One time passkey is created by the system which acts as a credential for the user until and unless they check out of the table. Using the passkey, which acts as credentials, users can monitor the status of the products remotely. Users will be able to add more products to the table or remove the existing ones. Once the user checks out of the table, the whole table will be deleted from the system, and a new passkey will be used for the next user.

4.1 QR Scan and Request Handling

This QR code acts as the entry point to the Kimberlite system. The user scans the code placed on the rented table. This QR code contains a unique table or vendor identifier. Once a QR is scanned, a GET request is sent to the backend which validates the request and then renders the respective user dashboard.

4.2 Product Upload via Form Submission

From the dashboard, users can take product pictures directly using their device's camera or upload images manually. Along with images, they can input all required information (e.g., name, price, description, etc.). The data is then sent to the backend through a POST API request, where it is stored in the PostgreSQL database.

4.3 Data Status and QR Code Generation

All the uploaded products remain in an inactive state initially. They are not published in the live store or frontend. After submission, the backend generates a unique batch-level QR code on the same dashboard. This unique QR code can be used later to retrieve the batch at the self-service Kimberlite Machine.

4.4 Printing with Kimberlite Machine

The user takes this generated QR code to the self-service Kimberlite Machine. The machine scans the code and sends a fetch request to the backend to retrieve all the products data. Once the request is handled, all the products will be available on the web and will be visible to everyone. The system responds with the data, and the machine prints all the product QR codes in a single batch, which is fully automated. Then, the user can place all their products on the rented table along with their respective QR code.

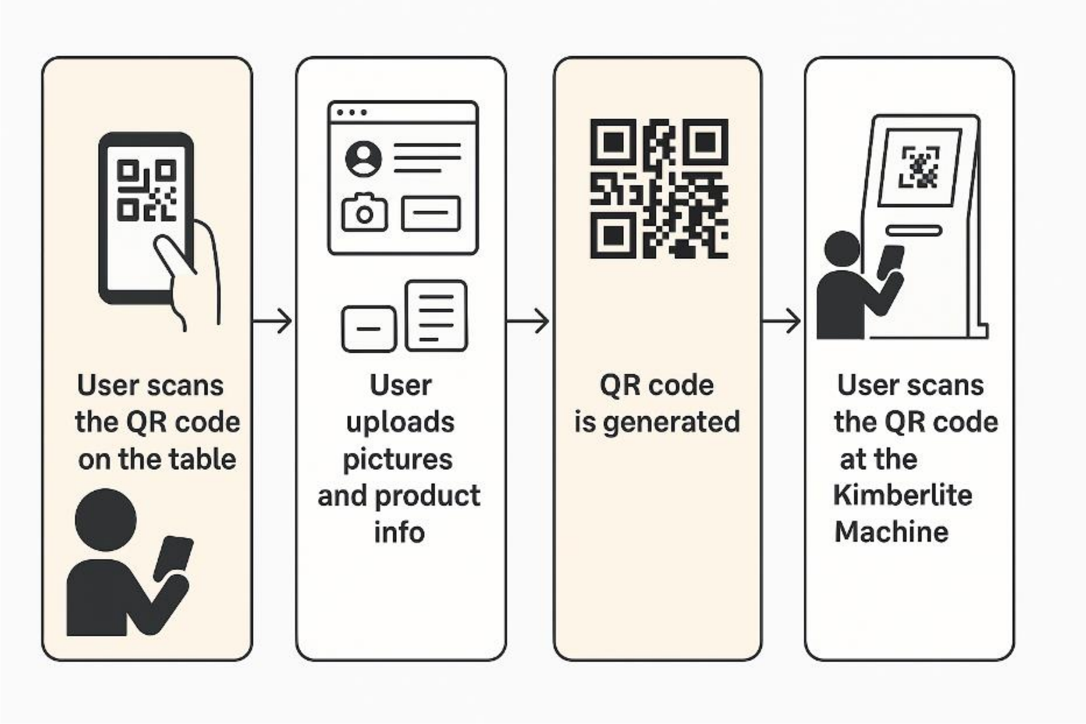


Figure 33 Printing with the web application

4.5 Passkey System

Figure 34 QR-code found at each respective table

In the Kimberlite system, a passkey is a unique alphanumeric code assigned to each table card. When a customer rents a table at the flea market, they receive a printed card that includes both a QR code and its corresponding passkey. Pass keys are unique for each table, and they can be used by the user as a login credential.

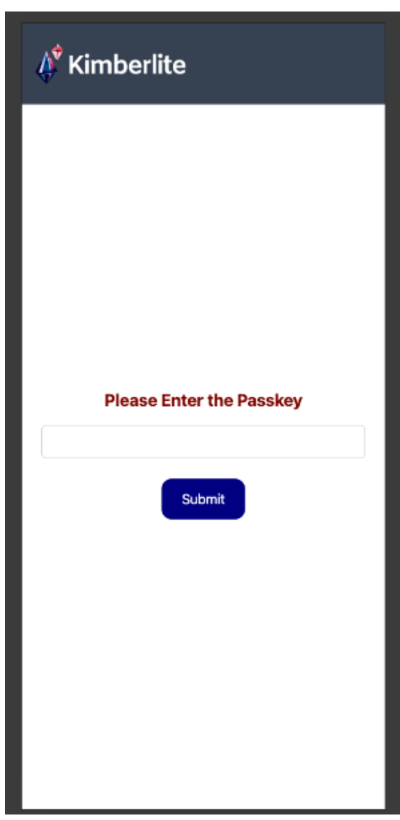


Figure 35 Accessing users table with the passkey provided with a table card.

The passkey serves two main purposes:

- Authentication: It allows the user to access the Kimberlite web application without creating a personal account. By scanning the QR code and entering the passkey, the user is granted temporary access to the system.
- Table Identification: Each passkey is directly linked to a specific table number. Once entered, the system uses the passkey to fetch and display the correct table number and its associated data in the Kimberlite system. This ensures that users are only able to manage and view products related to their own table.

4.6 Templates Engine

The system uses Jinja2 to render HTML pages from the backend which makes it easier to show dynamic data like uploaded products and products details and QR previews as well as keeps frontend and backend well connected.

4.7 Container & Deployment

To ensure consistency across all environments, the system is containerized using Docker. The entire app (backend, frontend, and database) runs inside Docker containers. Docker Compose can be used to manage multi-container setup (e.g., backend, frontend and database). It makes local development, testing, and deployment much easier. Docker ensures the same environment across different systems, staging, and production. It helps to avoid issues related to different development setups among team members, making collaboration much smoother.

4 TESTING

Testing, as with every project, is an important part of development. It helps us to verify the integrity of our features but also helps us implement them.

This system was tested in two phases.1

5.1 Phase 1

The actual backend of Kimberlite was not ready during the time of testing due to which we could not test all the possible simulations. Due to which in the beginning the Kimberlite system was tested using local storage to simulate data persistence. This approach allowed us to validate the front-end features, including form input, product listing, and UI behavior, without requiring a live server.

For authentication and access control, a predefined passkey was defined in the application to mimic the behavior of the real passkey system. This allowed for testing the logic behind verifying user input and linking it to a specific table number.

```
const [passkey, setPasskey] = useState("");

const handleSubmit = () => {
    console.log("Entered passkey:", passkey);
    if (passkey === "123456") {
        navigate("/start");
        alert("Passkey Accepted");
    } else {
        alert("Incorrect Passkey");
    }
};
```

Figure 36 Passkey handling function

Key areas tested:

Passkey Entry Flow: Entering the predefined passkey correctly redirected the user to the respective table view.

Data Handling: Product details were stored and retrieved from local storage, simulating how the system would behave with backend integration.

UI Interaction: All buttons, forms, and product cards were tested for expected behavior, responsiveness, and user flow.

Edit and Delete Functionality: Products can be edited and removed directly from UI, with changes reflected in local storage.

Web Flow: The entire user flow from passkey entry to table view access, to product interactions was tested to ensure a smooth and intuitive experience.

Although testing local storage has limitations, it was sufficient to verify the core functionality of the web interface. Final integration testing will be required once the backend is connected.

5.1.1 Problems Faced During Testing in Phase 1:

Since we were using a local storage for storing the data and not the actual backend, we encountered multiple problems. Together with Alvidiotech, we agreed that this to be sufficient for this time.

One of the most limiting factors in testing was the local storage capacity of the browser. When multiple product entries with large image files or multiple files at the same time, the storage limit was quickly exceeded. This caused the application to crash or fail to save new data. As a result, it became challenging to simulate real-world usage scenarios accurately. However, despite this problem, testing could still be done on a smaller scale to effectively simulate the core functionality of the system.

5.2 Phase 2

To address the issue encountered in phase1 testing, a skeleton backend was developed. The skeleton backend was good enough to handle all the required requests. The Skeleton backend was a minimal version of the main backend that could handle requests like creating, reading, and deleting. This helped to solve the problem we faced during phase 1 and testing was conducted. More details of the backend are provided in section 3.2.

Key area tested:

- All the items are successfully created and saved locally on items.json
- All the items are fetched using ID.
- Items could be deleted successfully in both frontend and json file.
- All invalid requests and data are successfully handled.
- All the routing was working properly.
- Pictures are uploaded and saved in the directory i.e. /uploads and can be seen in the front end as well.
- There is no breakage with the request and response flow from the backend. All API endpoints were tested, and all tested endpoints returned the expected response, and backend handles the requests smoothly without crashing.
- This testing verified the proper functionality of the system and the backend.

5 DISCUSSION

Working on this project gave us an opportunity to develop a real tool for a real issue in a real business environment. We approached the project with all the businesses' interests in mind. For us, the goal was to reach all the requirements set by Alvidiotech, which we did.

We achieved this with a system which will bring a feature to use for the users of Kimberlite. It will reduce the number of work hours required for the Kimberlite staff to scan and list items within Kimberlite, which the users could do more efficiently themselves. It will also make Kimberlite available for a variety of users and increase the products distribution nationally and internationally.

When we presented the final product for Alvidiotech, they mentioned that the success of this implementation is an important factor for future investors. AI image recognition (or Computer Vision) tools are being introduced in many different markets to derive meaningful information from images and other visual inputs(4). It helps to streamline workflows, and it is also very natural to use. Just like we would show objects to other people, we can now with this technology show things to computers to interpret them.

Kimberlite focuses on a simple and easy to use approach, especially for users who are not very familiar with digital platforms. The main target customers, particularly in flea markets, are often older people, who may find traditional login systems with email or phone numbers inconvenient or confusing.

To make the system more accessible, we decided to use a passkey-based access system. This enables the feature to be more accessible for users to interact with the platform without needing to create an account or remember credentials, helping promote the product and its usage in a wider range.

6.1. Comparison between the total time taken by old system and new system

In the old system, the process of obtaining product pictures and printing a single QR tag takes an average time of 30 seconds. However, this is just part of the total time for half the process. Most of the tasks are manual and highly dependent on staff availability for product data entry and printing, which is time-consuming. On average, entering product data and uploading images takes around 1 to 2 minutes per product, depending on the item and staff workload. As a result, processing a single product can take 1.5 to 2.5 minutes. For larger batches, such as 100 products, it takes hours anywhere from 2.5 to 4 hours depending on both data entry and printing. High traffic periods or limited staff availability can cause more delays, making the system unsuitable for scaling thousands of products. Overall, the workflow is time-consuming, heavily reliant on human resources, and not optimized for efficiency.

On the other hand, new system is designed to be faster, more independent and scalable for larger batch. User can handle the entire process independently using their own smartphone to capture the picture and upload the product data without the need of staff support. This eliminates the queues during rush hour and reduces the waiting time. To process 100 products, it takes less than an hour which is over 50% improvement in efficiency compared to old system.

The new system not only benefits the users (those who rent the tables) but also the shop owners. Since the system is self-managed, there is no need to hire extra staff for assisting with item uploads or tag printing. Users can handle their own tables remotely and check the status of their items online. This means fewer frequent visits to the store, just to check if something is sold. As a result, the flow inside the store becomes better with fewer table-checking customers which is very important during the rush hour.

6.2 Security considerations

The adoption of AI introduces security concerns that should be proactively addressed. For example, in this project we do not address the potential of data poisoning, but attackers could inject malicious or manipulated data into training sets, leading to flawed or biased model behavior.

There are no measures implemented for any dangerous or illegal goods either, risking a shopping customer of any such exposure. Using third-party models or suppliers amplifies the risk, as these can contain hidden vulnerabilities or backdoors(9). Additionally, biases in training data often caused by a lack of diversity—can lead to unethical or legally problematic outputs. To defend against these threats, the best practices would be continuous data validation, strict permission governance moderated by the staff, and deploying AI-driven security solutions to identify malicious items and products. Other organizations must also continuously monitor AI systems and maintain a strong incident response plan to stay resilient in the evolving threat environment.

6.3 Future Implementation

In the future, many features will be implemented according to needs and feedback from customers. Some of the most needed features that will be updated soon after the release of these products are

6.3.1 Payment Method

Adding an option for users to be able to update their Bank Account number in the system so that they will be able to receive the payment directly in their bank Account. This feature will make the payment process easier and faster. If the table is already emptied, the user should not visit the shop for their payment.

6.3.2 Check in and Checkout System

Users will be able to check-in in the system when they rent the table and check out when they do not want to rent the table anymore. This feature will ensure that the table will be available for future users for their use and streamline the process more.

6.3.3 Security Measures

With the current system, there is not much protection. System security will be updated to protect the system from the multiple threats mentioned in section 6.2, and every solution will be implemented to secure all data.

6.3.4 Status of the Items

Users will be able to monitor the status of their products within the system. They can check what products have already been sold and what is left. They will make sure that users do not have to visit the table to check for any misplaced items and can do so remotely.

6 REFERENCES

1. Express.js Tutorial. Search date 05.07.2025, https://www.geeksforgeeks.org/node-js/express-js/

2. JavaScript Tutorial. Search date 21.06.2025, https://www.geeksforgeeks.org/javascript/javascript-tutorial/

3. React Router. Search date 12.06.2025, https://www.geeksforgeeks.org/reactjs/reactjs-router/

4. IBM. Image Recognition. Search date 02.07.2025, https://www.ibm.com/think/topics/image-recognition

5. Kimberlite Homepage. Search date 18.05.2025, https://kimberlite.fi/

6. Monocubed. Most Popular Web Frameworks. Search date 10.06.2025, https://www.monocubed.com/blog/most-popular-web-frameworks/

7. Mozilla. Cross-Origin Resource Sharing (CORS). Search date 03.07.2025, https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

8. Node.js. Introduction to Node.js. Search date 28.06.2025, https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

9. Perception Point. Top 6 AI Security Risks and How to Defend Your Organization. Search date 13.07.2025, https://perception-point.io/guides/ai-security/top-6-ai-security-risks-and-how-to-defend-your-organization/

10. Swagger. API Documentation. Search date 03.07.2025, https://swagger.io/docs/

11. Tailwind CSS. Documentation. Search date 18.06.2025, https://v2.tailwindcss.com/docs

12. How React Works. Search date 18.06.2025, https://www.uxpin.com/studio/blog/how-react-works/

13. Multer middleware documentation. Search date 03.07.2025, https://www.npmjs.com/package/multer