Report on

Peer-to-Peer Storage Sharing Platform in Ireland: Addressing Temporary Storage Needs through Collaborative Solutions

Submitted by

Gnanasekar Mani (20020329)

Applied Research Project (BSIS101) submitted in Partial Fulfilment of the Requirements for the

Degree of

Masters of Science in Information Systems with Computing

Dublin Business School

Supervisor by

Eugene O'Regan

January / 2025

DECLARATION

I hereby declare that this research project titled "**Peer-to-Peer Storage Sharing Platform in Ireland: Addressing Temporary Storage Needs through Collaborative Solutions**" is my original work and has been carried out in accordance with the guidelines and regulations of Dublin Business School.

The content presented in this report is entirely my own, except where explicitly stated otherwise through proper citations and references. I confirm that this project has not been submitted previously for any degree or diploma in any other institution.

All sources of information and references used in the preparation of this research have been duly acknowledged.

I understand that any misrepresentation or violation of the above declaration could result in disciplinary actions according to the rules of the institution.

Signed: Gnanasekar Mani

Student Number: 20020329

Date: 06/January/2025

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, **Eugene O'Regan**, for his unwavering guidance, encouragement, and constructive feedback throughout this project. His expert insights and support have been invaluable in overcoming challenges and steering this work toward its successful completion. His mentorship has greatly contributed to my learning and growth during this academic journey.

I extend my heartfelt thanks to **Dublin Business School** for providing me with the platform and resources to undertake this thesis. The support from the institution, its faculty, and the academic environment have been instrumental in enabling me to conceptualize and execute this project effectively.

I am profoundly grateful to the open-source community, whose contributions and innovative tools have been pivotal in the development of this project. The accessibility of their resources and their collaborative spirit have significantly enriched my work and broadened my technical knowledge.

My sincere appreciation goes to the individuals who participated in surveys and provided valuable feedback. Their inputs have been crucial in understanding user requirements and refining the project to better serve its purpose. This project is stronger and more user-focused because of their engagement and thoughtful suggestions.

Furthermore, I would like to acknowledge the assistance of AI tools in refining certain written sections of this report; while the research and conclusions presented here are my own, the support from these tools was helpful in shaping the clarity of my final draft

Finally, I would like to thank my **family, friends, and peers** for their constant encouragement, patience, and support. Their belief in me and their words of motivation have been a source of strength throughout this journey.

ABSTRACT

The rising demand for temporary storage solutions in Ireland, driven by urbanization and increasing mobility, highlights a critical gap in accessible, flexible, and cost-effective storage options. This study aims to design and implement ***Holdhive***, a peer-to-peer storage-sharing platform, to address these challenges by fostering collaborative utilization of underutilized storage spaces (Botsman & Rogers, 2010). The research employs a mixed-methods approach, beginning with a comprehensive analysis of user needs through surveys, followed by the development of a scalable and secure web-based platform. The platform integrates cutting-edge technologies, including a SQL-backed database, Firebase Authentication for user verification, and a user-centric design interface using Python and React.js. The methodology emphasizes seamless interactions, transparency, and trust between storage providers and renters, underpinned by a robust review system. Initial results demonstrate the platform's effectiveness in connecting users and facilitating storage transactions. Furthermore, the system's modular design ensures adaptability to varying user needs and regulatory requirements. This study concludes that ***Holdhive*** successfully bridges the gap in the storage market by offering a sustainable and community-driven solution. The findings underscore the potential for peer-to-peer sharing models to address similar challenges in other domains, paving the way for future research and innovation in collaborative economies (Schor, 2014).

Keywords:

Peer-to-peer storage, Temporary storage solutions, Collaborative economy, Resource optimization, Sustainable storage practices, User-centric design, Short-term storage challenges, Ireland

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Temporary storage solutions have become a pressing need in modern urbanized societies, especially in densely populated areas like Ireland. With the increasing cost of real estate and the limitations of traditional storage services, individuals often struggle to find affordable and flexible storage options (CBRE, 2022). This has led to a growing interest in peer-to-peer sharing models, which have already disrupted industries such as accommodation and transportation (Hamari et al., 2016).

Holdhive, the focus of this research, addresses the storage challenges by leveraging the concept of collaborative consumption. By connecting individuals with unused storage spaces to those in need, the platform creates a win-win scenario for both providers and renters. The study draws on principles of trust, transparency, and economic efficiency to design and evaluate this innovative solution (Botsman, 2013).

Access ***Holdhive*** in the following Link: **http://holdhive.com:3000/**

1.1 Research Background

The increasing urbanization and mobility in Ireland have led to a growing need for flexible and temporary storage solutions. Existing commercial storage services often fail to meet users' requirements for affordability and accessibility (Eurostat, 2022). Inspired by the success of peer-to-peer sharing models like Airbnb for accommodations and BlaBlaCar for ride sharing, this research explores their application in addressing storage challenges, culminating in the development of ***Holdhive***.

For instance, Airbnb has transformed the hospitality sector by connecting hosts with unused accommodations to guests, showcasing the power of peer-to-peer platforms (Airbnb, 2023). Similarly, BlaBlaCar has successfully connected travellers with shared transportation options, reducing costs and fostering trust (BlaBlaCar, 2023). ***Holdhive*** leverages underutilized storage spaces, aiming to replicate such success in a different domain.

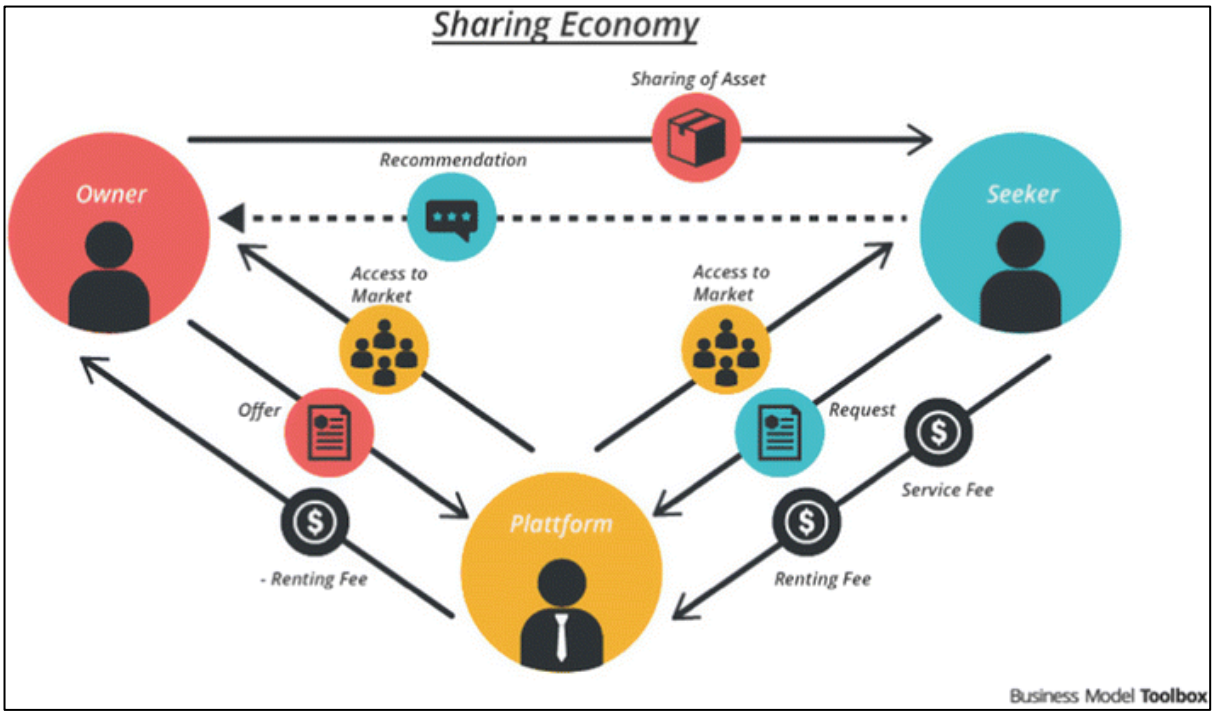

Figure 1 Overview of the sharing economy (Business Model Toolbox, 2018)

1.2 Research Question

How can a peer-to-peer platform effectively address the temporary storage needs in Ireland while fostering collaboration and trust among users?

1.3 Research Objective

The primary objective of this research is to design, implement, and evaluate ***Holdhive***, a platform that connects individuals offering unused storage spaces with those seeking temporary storage solutions.

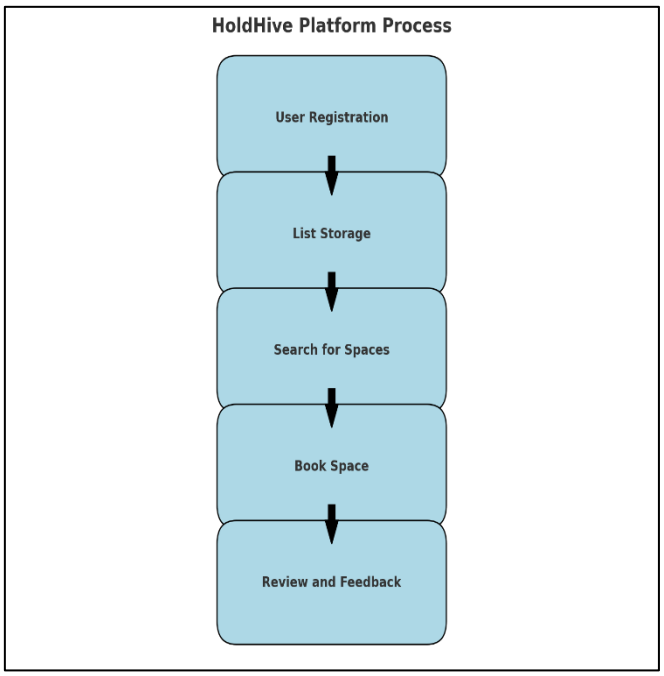

Figure 2 ***Holdhive*** *Booking Flow*

1.4 Significance of Research

This research contributes to the growing body of knowledge on collaborative economies by demonstrating the viability of peer-to-peer storage sharing (Sundararajan, 2016). It provides a practical solution to storage inefficiencies, promoting sustainability and economic accessibility. By addressing temporary storage needs, the platform reduces the reliance on commercial options, lowering costs and environmental impact.

1.5 Scope of the Research

The study focuses on the design and implementation of the ***Holdhive*** platform, including user authentication, secure transactions, and user interface development. It also evaluates the platform's usability and economic impact through user feedback and system performance metrics.

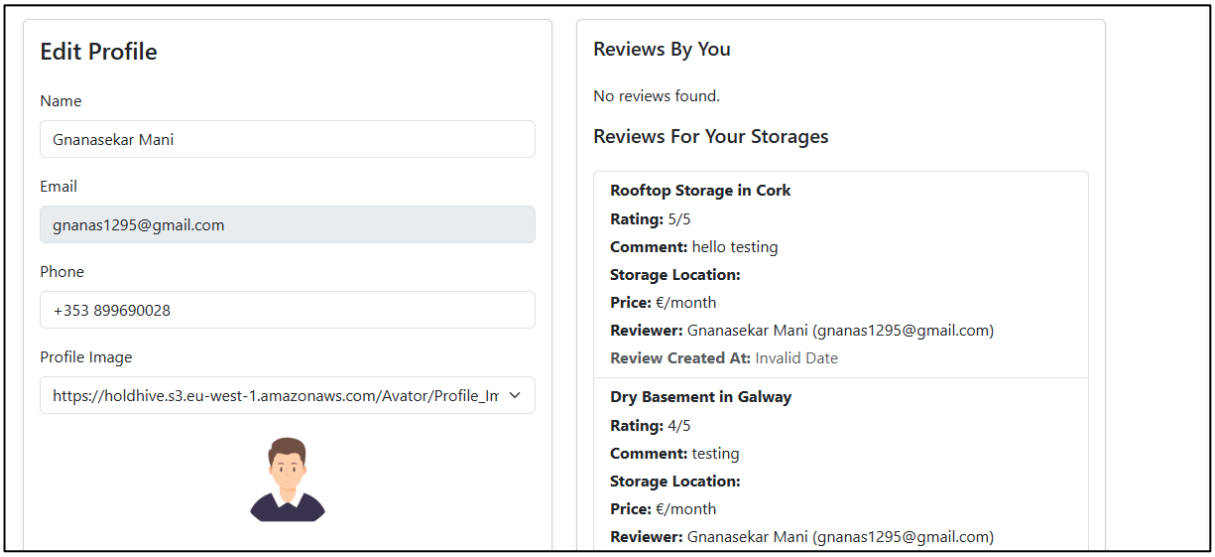

Figure 3 User dashboard

1.6 Research Outcomes

This research aims to deliver a fully functional peer-to-peer storage platform that demonstrates significant user satisfaction, cost-effectiveness, and enhanced space utilization. Insights from this study could inform future innovations in the sharing economy.

1.7 Limitations of the Research

The research is limited to a specific geographical region (Ireland) and does not account for potential legal or cultural barriers in other contexts. Additionally, it relies on self-reported data for usability assessments, which may introduce bias (Trochim, 2021).

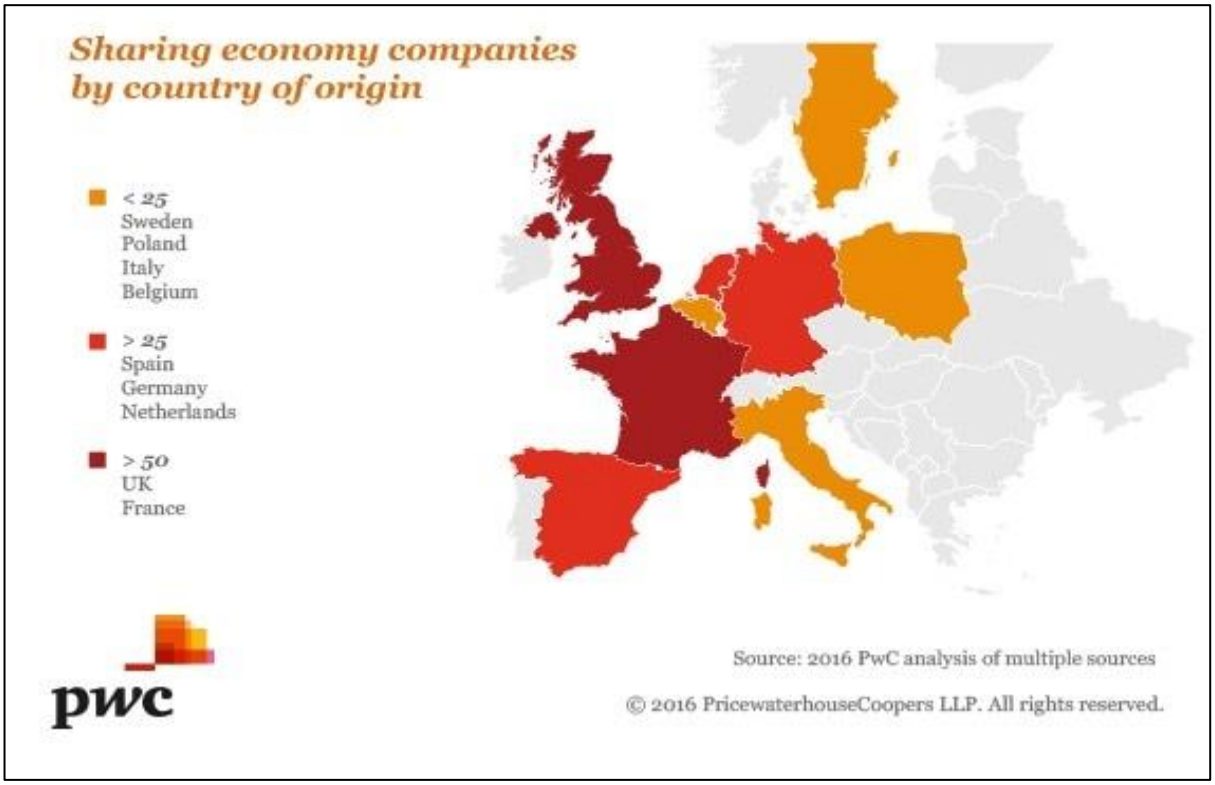

Figure 4 Sharing economy companies PwC Hungary. (2016, July 6).

1.8 Research Roadmap

The research is structured in several phases:

1. Problem identification and literature review.
2. Platform design and development.
3. User testing and feedback collection.
4. Data analysis and refinement.
5. Final evaluation and reporting.

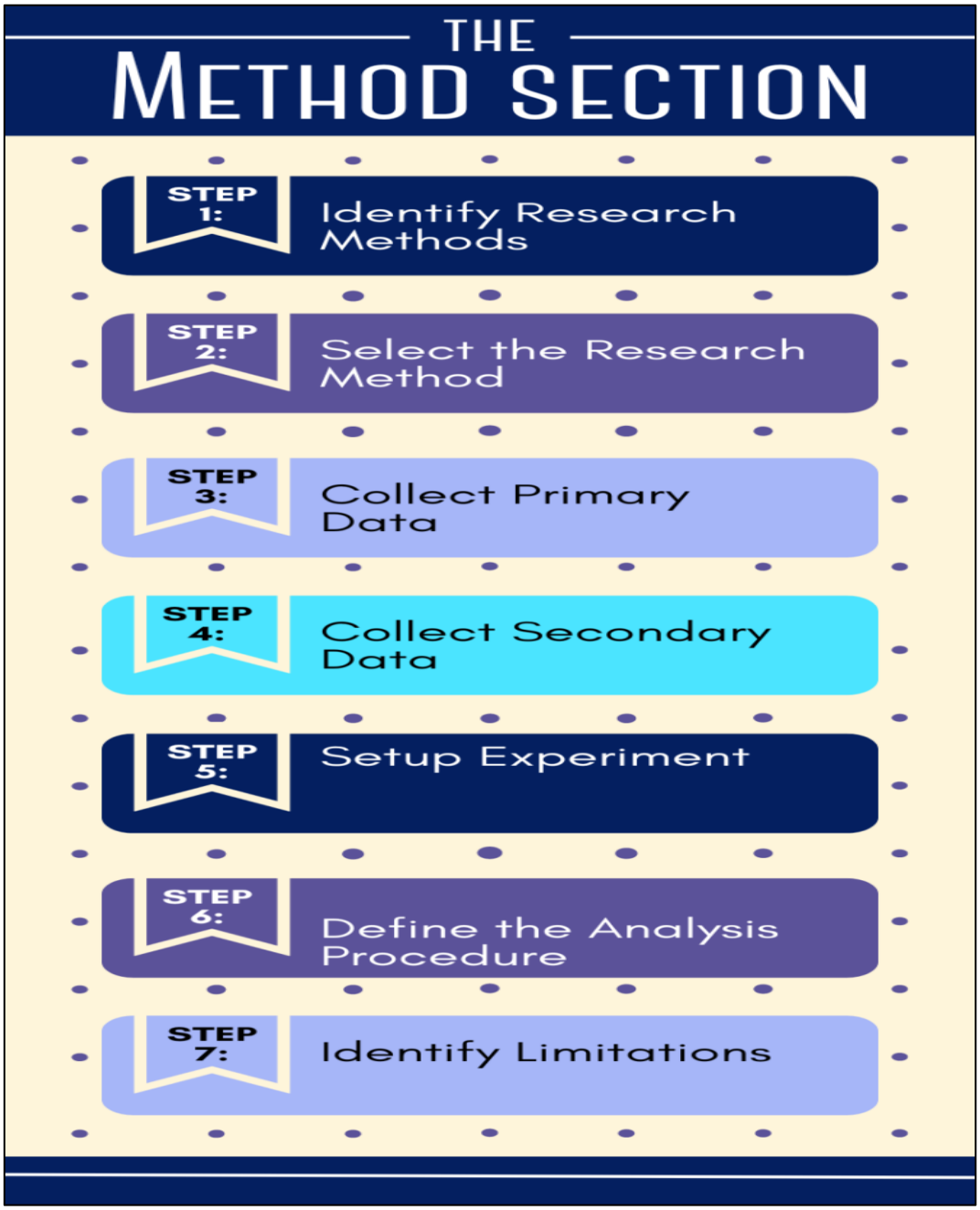

Figure 5 Research Process ResearchVoyage. (n.d.)

This roadmap ensures a systematic approach to achieving the research objectives and addressing the identified problem.

CHATPER 2
LITERATURE REVIEW

2.1 Overview

The rise of the sharing economy in recent years has paved the way for numerous peer-to-peer (P2P) marketplaces (Botsman and Rogers, 2010). Platforms such as Airbnb and Uber have demonstrated how technology can facilitate trust and efficient transactions between strangers (Resnick and Zeckhauser, 2002). In the context of physical storage, ***Holdhive*** is designed to connect individuals who have underutilized storage space ("**Hosts**") with those seeking short-term or long-term storage ("**Renters**").

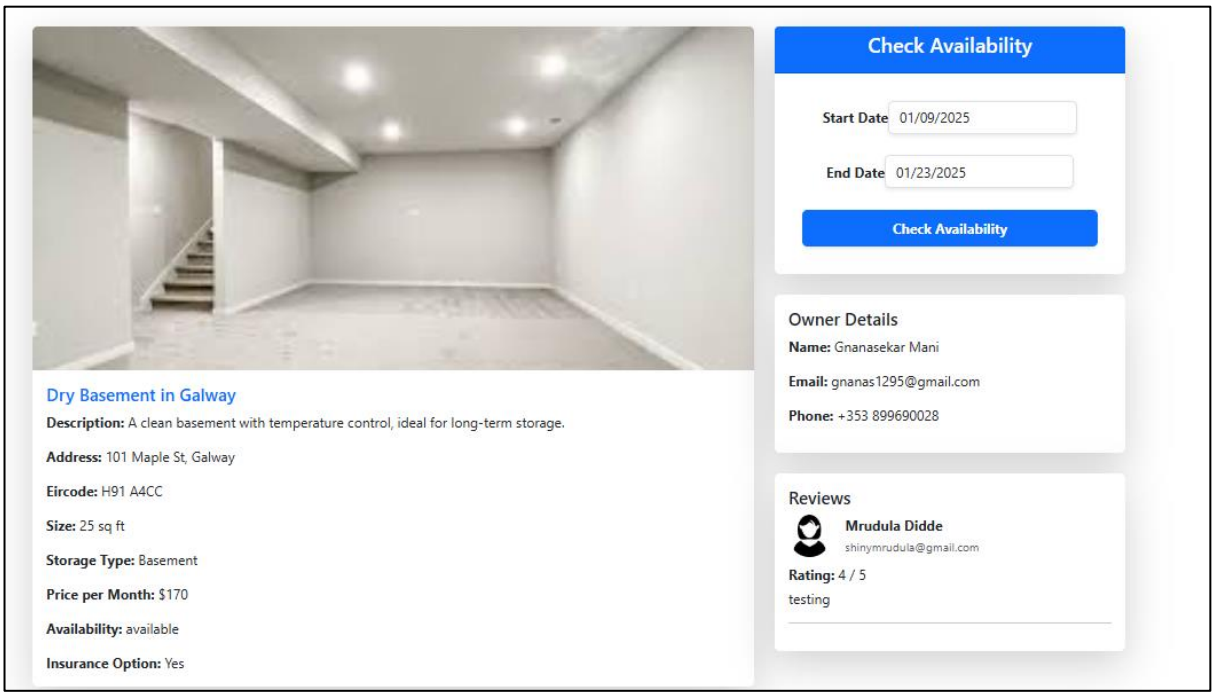

Figure 6 Holdhive- Overview of P2P Storage Interactions.

2.2 History

The concept of **collaborative consumption** gained prominence as a way to optimize idle assets in traditional markets (Botsman and Rogers, 2010). Early P2P models were popularized by eBay's buyer-seller interactions (Dellarocas, 2003), later evolving into specialized platforms for lodging (Airbnb) and ride-sharing (Uber, Lyft).

Self-storage facilities, once the mainstay for personal storage needs, rely on **centralized ownership** and often involve long-term contracts and standardized unit sizes (Owyang, 2016). The ***Holdhive*** model builds on these historical precedents, leveraging P2P principles to offer greater flexibility and community-driven arrangements.

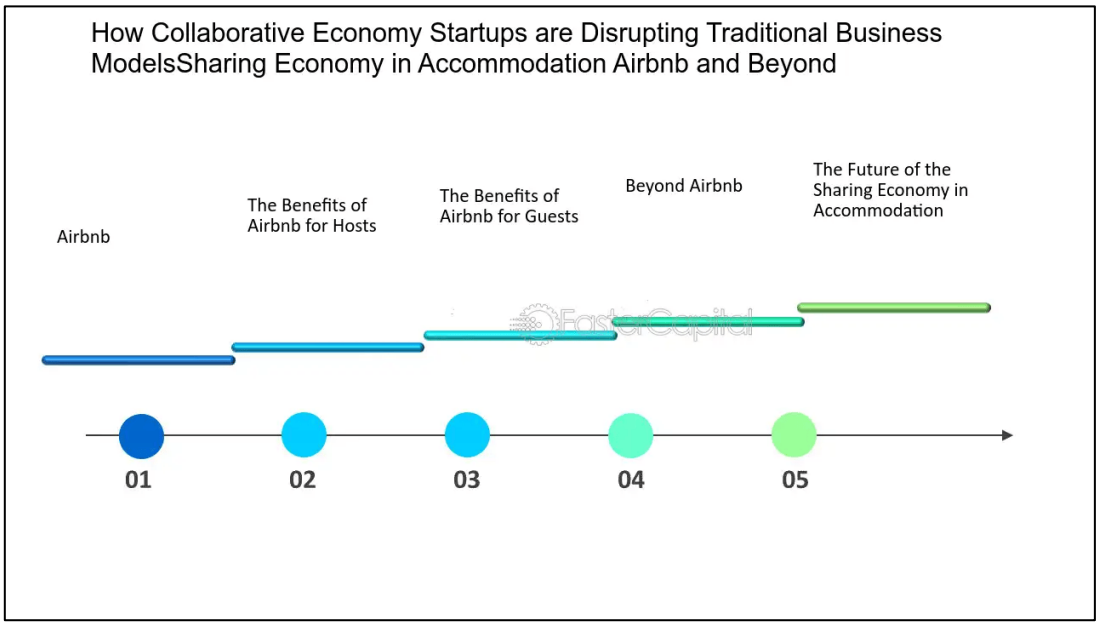

Figure 7 Collaborative Economy Space Timeline FasterCapital. (n.d.)

2.3 Benefits of P2P Storage

P2P storage platforms like ***Holdhive*** offer multiple advantages:

1. **Cost-effectiveness**: Directly connecting Hosts and Renters can lower prices compared to traditional self-storage (Neighbor, 2023).
2. **Community Building**: Localized hosting fosters trust and a sense of shared community (Hawlitschek et al., 2016).
3. **Scalability**: The platform can scale rapidly by enlisting new Hosts, without the heavy infrastructural costs of physical facilities (Botsman, 2013).

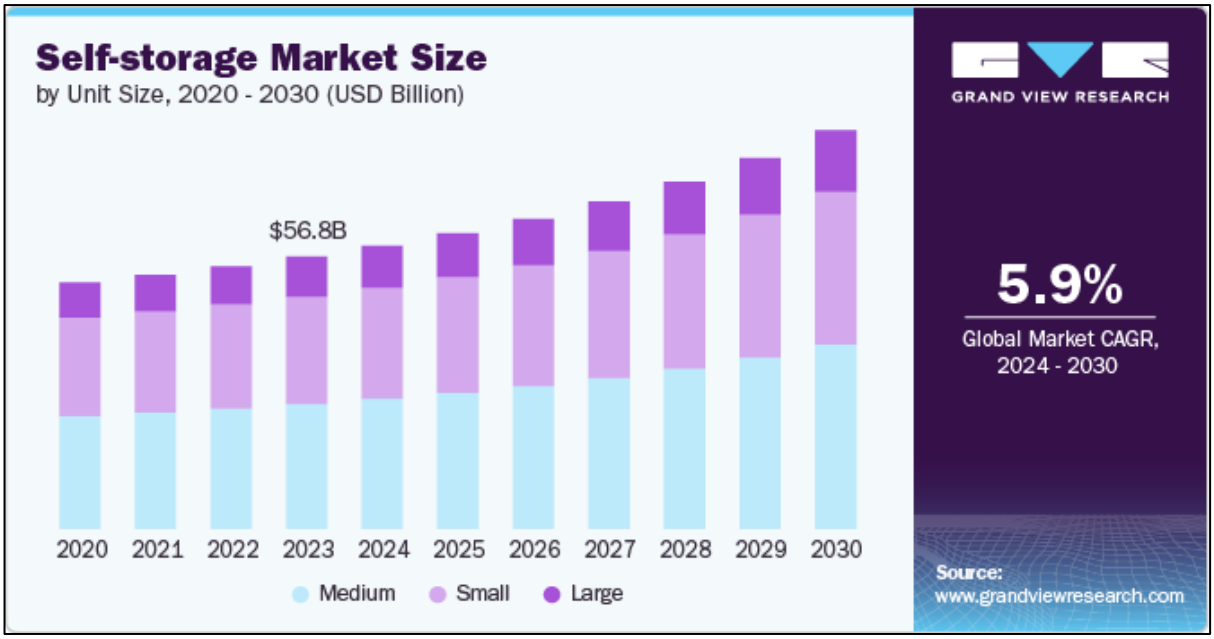

Figure 8 Market Share of Self Storage, Self-storage market size, share and Trends report, 2030 (no date)

2.4 Challenges and Limitations of P2P Storage

Despite its advantages, P2P storage faces notable challenges:

- **Trust and Security**: Users may hesitate to store items with strangers if security measures or user verification are not robust (Jøsang et al., 2007).
- **Legal and Regulatory Issues**: Zoning regulations, insurance requirements, and local ordinances can complicate peer-to-peer arrangements (Codagnone et al., 2016).
- **Inconsistent Quality**: Unlike standardized storage units, private spaces vary in cleanliness, accessibility, and climate control (Zervas et al., 2017).

For ***Holdhive*** to thrive, these barriers must be mitigated by trust mechanisms, transparent policies, and appropriate user support.

2.5 Existing P2P Storage Platforms and Their Limitations

Emerging platforms such as Neighbor and Stow It illustrate the potential for P2P storage models (Neighbor, 2023). However, limitations include:

- **Geographical Focus**: Many existing platforms are confined to major metropolitan areas.
- **Weak User Verification**: Inadequate identity checks can lead to fraudulent or suspicious listings (Resnick et al., 2000).
- **Dispute Resolution Gaps**: Limited audit trails make conflict mediation difficult (Kallahalla et al., 2003).

Holdhive seeks to differentiate itself through a robust audit logging process, integrated authentication (e.g., Firebase), and possible insurance packages to instil confidence in all users.

2.6 Key Considerations

Designing a robust P2P storage system requires attention to:

1. **Privacy and Data Protection**: Ensuring GDPR compliance (European Commission, 2016).
2. **Payment Integrations**: Secure gateways like Stripe or PayPal mitigate risks involved with handling financial transactions (Teoh et al., 2013).
3. **Usability**: An intuitive user interface encourages higher adoption and lowers support overhead (Mashal et al., 2015).
4. **Scalability**: Cloud-based infrastructure supports spikes in user activity without degrading performance (Elmasri and Navathe, 2015).

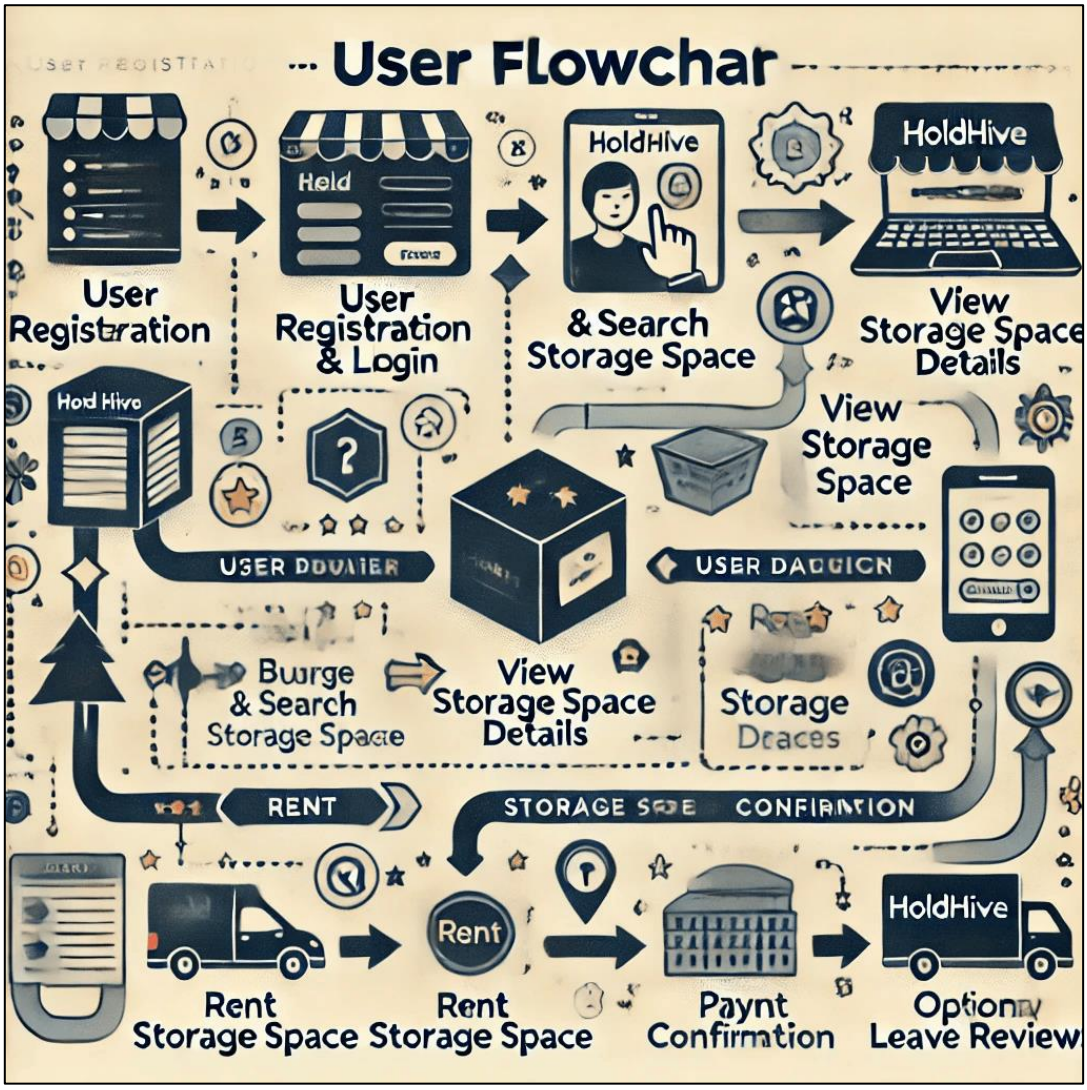

*Figure 9 End to end user flow in **Holdhive**, OpenAI (2023) - AI Generated*

2.7 P2P Storage Implementation Using Python

Holdhive employs a Python-based backend, which offers:

1. **Frameworks**: Lambda with API Gateway using Python for RESTful APIs, enabling efficient route handling and session management (Elmasri and Navathe, 2015).
2. **Database Integration**: Relational databases (e.g., PostgreSQL or MySQL) ensure ACID compliance and robust data integrity (Codd, 1970).

3. **Firebase Authentication**: Simplifies secure user login and provides OAuth-based authentication flows (Firebase, n.d.).

Python's wide ecosystem and community support also facilitate faster iterations and adoption of additional libraries for data analytics or machine learning (Xu et al., 2016).

2.8 Detailed Comparison of Methodologies and Outcomes

2.8.1 Centralized vs. Decentralized Models

While block chain-based solutions like Storj and Filecoin focus on decentralized digital file storage, physical storage requires additional oversight regarding location, property rights, and user security (Benet, 2014). ***Holdhive***'s approach therefore combines the flexibility of P2P with partial centralization for user verification and dispute resolution (Hawlitschek et al., 2016).

2.8.2 Traditional Self-Storage vs. P2P

Conventional self-storage generally involves standardized pricing and multi-month leases. By contrast, P2P storage allows flexible rental durations and localized spaces (**Neighbor, 2023**). However, quality control remains inconsistent without thorough user reviews and platform enforcement.

2.8.3 Platform-Specific Assessment

1. **Trust Mechanisms**: Peer reviews and verified identities can significantly improve user confidence (Dellarocas, 2003).
2. **Scalability**: Cloud-based storage (Amazon S3) and micro services support large volumes of images and data (Amazon Web Services, 2020).
3. **Financial Compliance**: Third-party gateways like Stripe or PayPal align with PCI DSS standards, reducing liability for the platform (Teoh et al., 2013).

2.9 Discussion of Key Findings

Synthesizing the above reveals:

- **Trust and Security**: Central to user adoption, necessitating robust reputation systems and user identity checks (Jøsang et al., 2007).
- **Regulatory and Insurance**: Clear policies and possible insurance frameworks can reduce legal risks and enhance user confidence (Codagnone et al., 2016).
- **Scalable Infrastructure**: Cloud-based solutions (S3, SQL databases) and Python backend can accommodate rapid growth in listings and rentals (Elmasri and Navathe, 2015).
- **Future Enhancements**: Machine learning for matching renters to suitable spaces, and deeper mobile integrations to improve platform accessibility (Xu et al., 2016).

2.10 P2P Storage Solutions and Market Trends

Physical P2P storage solutions are increasingly prevalent, spurred by urban density and the rising demand for flexible, affordable storage options (Neighbor, 2023). Market trends also indicate that many users prefer **short-term rentals and hyper-local storage solutions**, mirroring broader sharing economy behaviours (Owyang, 2016).

Holdhive sits at this intersection of increasing demand and technological opportunity, aiming to harness underutilized residential or commercial spaces in the same way Airbnb leverages underused housing (Botsman, 2013).

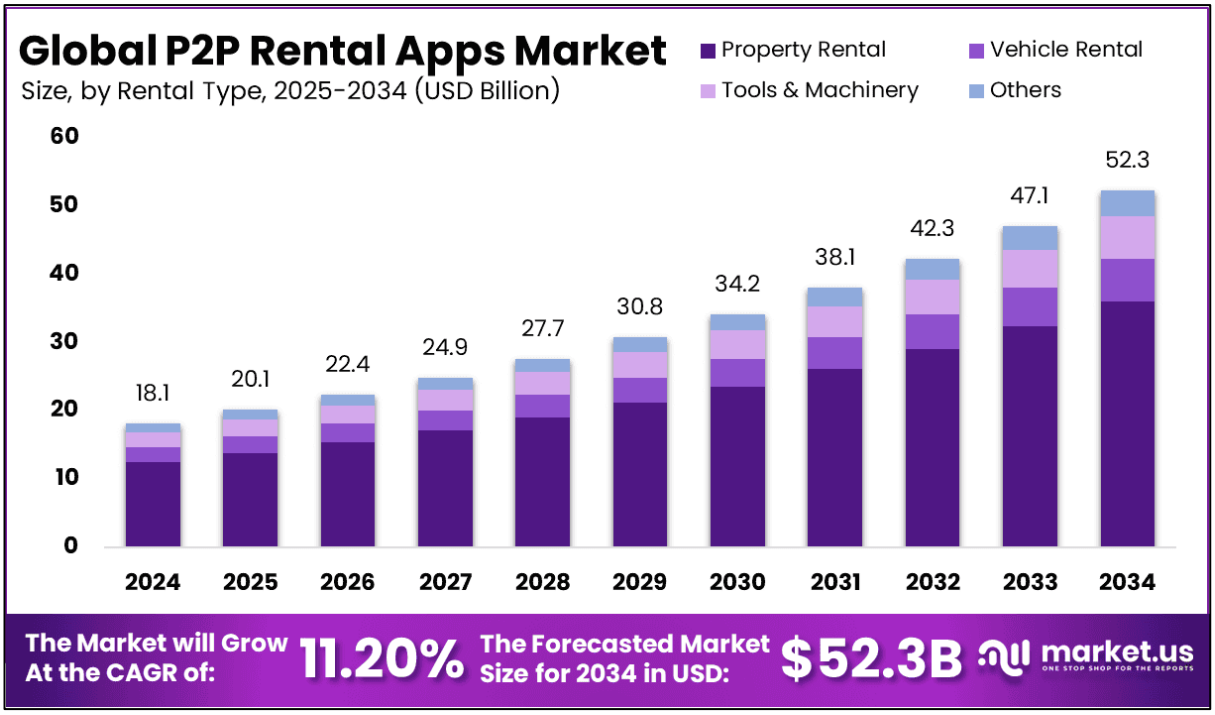

Figure 10 Market Growth for P2P Rental Platforms. P2P rental apps market (2025) Market.us.

2.11 Trust, Reputation, and Security Mechanisms

Building trust in P2P marketplaces is essential (Jøsang et al., 2007). Reputation systems—including star ratings, written reviews, and verified badges—bolster user confidence (Resnick and Zeckhauser, 2002). Security mechanisms must address potential fraud, theft, or property damage.

Holdhive integrates:

1. User Authentication (Firebase): Simplified sign-up and user verification.
2. Reviews and Ratings: Transparent reviews to maintain accountability.
3. Potential Insurance Policies: Partnerships to protect stored items.

2.12 Data Management and Cloud Storage

Effective data management is crucial when handling sensitive personal information, payment records, and property details (Elmasri and Navathe, 2015). **Relational databases** ensure atomic, consistent, isolated, and durable transactions (Codd, 1970), while object storage like **Amazon S3** can house images and documents securely and scalable (Amazon Web Services, 2020).

Key considerations include:

1. **Data Encryption**: Protecting user records in transit and at rest (Kshetri, 2013).
2. **Backup and Recovery Plans**: Minimizing data loss via scheduled backups (Ristenpart et al., 2012).
3. **Compliance**: GDPR obligations for users within EU jurisdictions (European Commission, 2016).

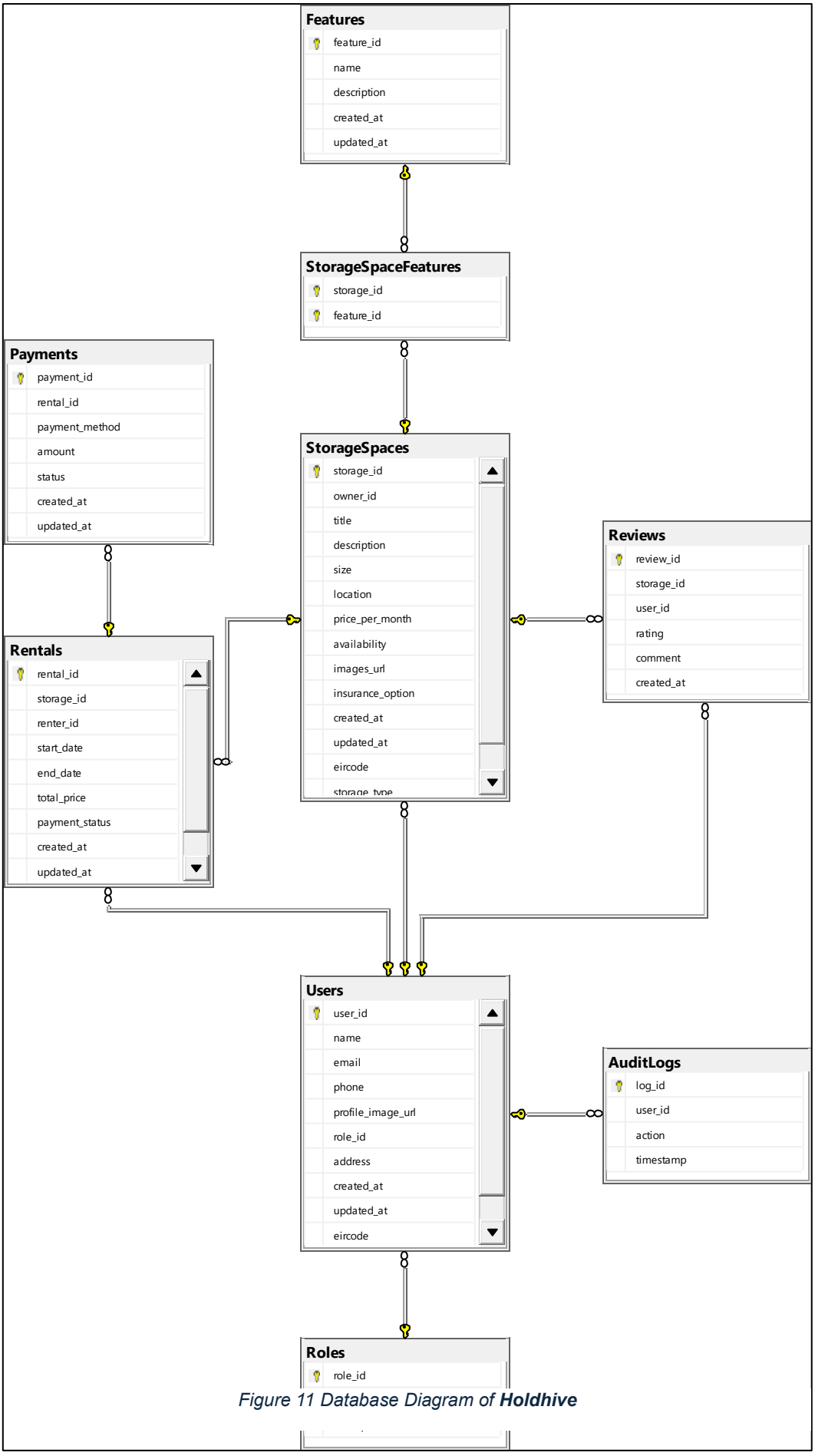

*Figure 11 Database Diagram of **Holdhive***

2.13 Additional Features and Future Considerations

Many sharing economy platforms evolve by adding new features to improve user experience and differentiate themselves (Zervas et al., 2017). For ***Holdhive***, potential enhancements include:

- **Mobile App Integration**: Push notifications, easy photo uploads, geolocation services (Mashal et al., 2015).
- **Insurance Offers**: Partnerships with third-party insurers can protect renters' valuables, increasing trust (Neighbor, 2023).
- **Advanced Matching Algorithms**: Machine learning models to recommend optimal storage spaces to renters based on item size, location, and reviews (Xu et al., 2016).
- **Global Expansion**: Multi-currency support and localized content to tap into emerging markets (Teoh et al., 2013).

CHAPTER 3

SYSTEM DESIGN & METHODOLOGY

3.1 System Design

3.1.1 Actors and Use Cases

Actors

- **Host**: Owns underutilized storage space. Can create, update, and remove listings.
- **Renter**: Searches for and books listed storage spaces. Can leave reviews post-rental.
- **Admin**: Oversees platform operations (e.g., monitoring disputes, enforcing policy). Potentially in a developer/admin role during testing.

Use Cases

1. **Create Listing**: A Host provides details - such as title, description, price, and images - for a new storage space.
2. **Search and Filter**: A Renter applies criteria (location, eircode) to find suitable listings.
3. **Booking**: A Renter selects a storage space, confirms availability.
4. **Review and Rating**: After the rental period, the Renter reviews the space and Host.
5. **Administrative Oversight**: An Admin can view or manage suspicious activities, check logs, or update system settings.

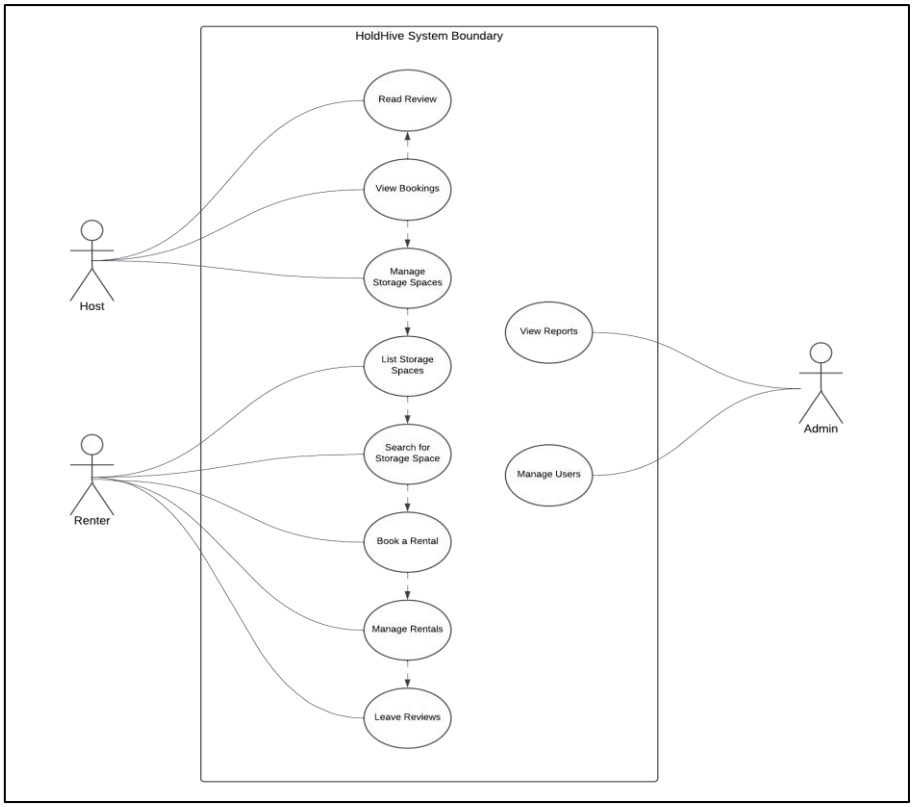

*Figure 12 UCD for **Holdhive***

3.1.2 Data Flows

Holdhive orchestrates multiple data flows between the ReactJS frontend, AWS Lambda micro services, Microsoft SQL Server, and Amazon S3:

1. **Registration and Login**
    - Frontend prompts user credentials.
    - Firebase validates and issues an ID token, which is then used in subsequent requests (Firebase, n.d.).
2. **Listing and Booking Flow**
    - Host uploads listing details (including images to S3).
    - Renter queries available listings (Lambda → SQL Server), checks pricing, finalizes booking (payment triggers a separate Lambda).
3. **Review and Rating**

- Renter submits review → posted to Reviews table in SQL Server.
- Host rating updated automatically to reflect cumulative feedback.

4. **Admin Monitoring**
    - Admin retrieves system logs, checks suspicious bookings, or reviews account info.
    - All logs are stored in CloudWatch for reference (Ristenpart et al., 2012).

3.1.3 Architectural Approach

Holdhive simulates a production-ready environment to ensure scalability and robustness (Amazon Web Services, 2020). The system's architecture includes:

- **ReactJS on AWS EC2**: Serving the single-page interface.
- **AWS Lambda microservices,** orchestrated by API Gateway: Each Lambda encapsulates specific business logic (listings, bookings, payments).
- **Microsoft SQL Server**: A central relational DB for user profiles, roles, listings, transactions, and reviews.
- **Amazon S3**: Storing images securely and offloading large binary data from SQL.
- **Firebase**: Authentication service to manage user signups/logins and issue ID tokens.
- **AWS CloudWatch**: Logging and monitoring for all Lambda functions, capturing performance metrics and errors.

3.1.4 Frontend: ReactJS on EC2

1. **ReactJS**: A JavaScript library enabling a single-page application architecture, delivering a dynamic, component-based UI (Pressman, 2010).
2. **AWS EC2**: The production-like hosting choice, allowing flexible server configurations and potential autoscaling if traffic surges (Amazon Web Services, 2020).
3. **Core Interactions**: Users (Hosts, Renters, and Admins) interact with forms, listings, or dashboards; the React app sends HTTPS requests to AWS API Gateway.

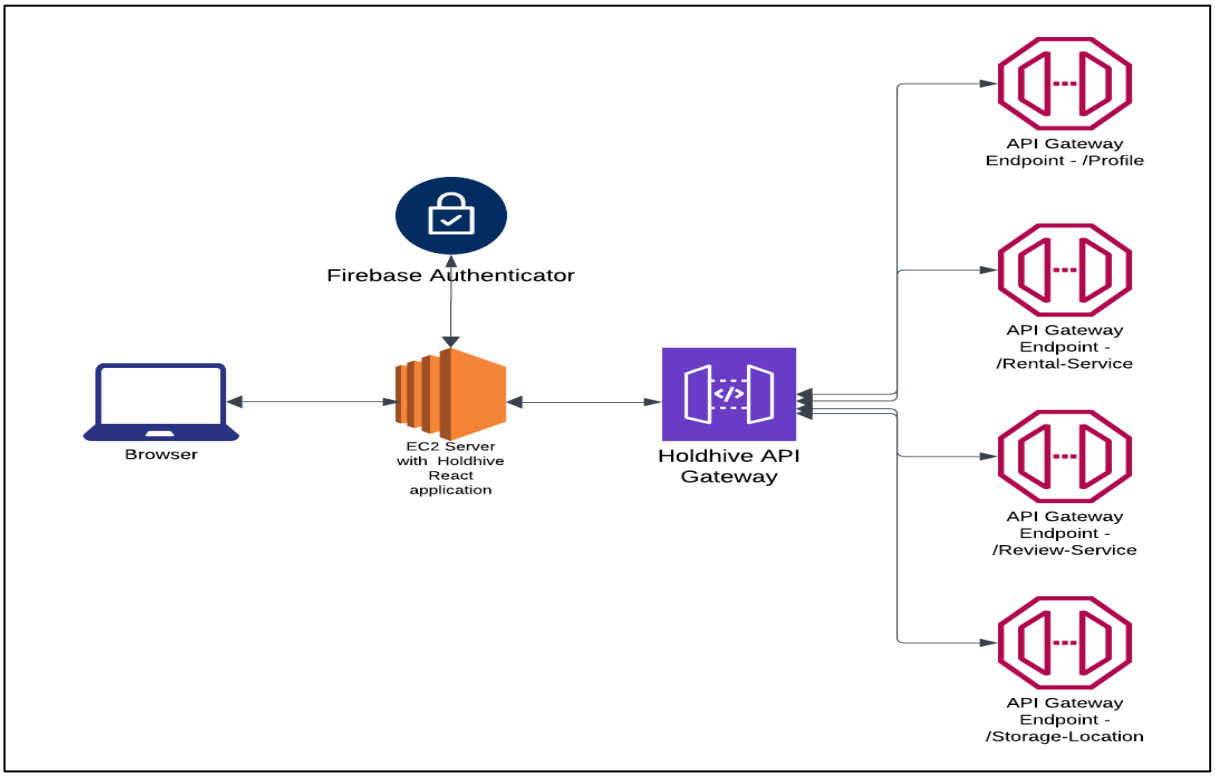

Figure 13 Figure 13 Arch-Diagram-Browser-API-Gateway

3.1.5 Backend: AWS Lambda Microservices

AWS Lambda functions handle discrete backend tasks:

1. **API Gateway Routing**: For each request path, a specific Lambda is invoked—e.g., "/createListing" triggers the listing microservice (Fowler, 2019).

2. **Serverless Scaling**: Lambdas spin up as request volume grows, ensuring minimal response delay (Highsmith, 2004).
3. **Logging**: Each Lambda posts logs to AWS CloudWatch (execution time, memory usage, error stack).

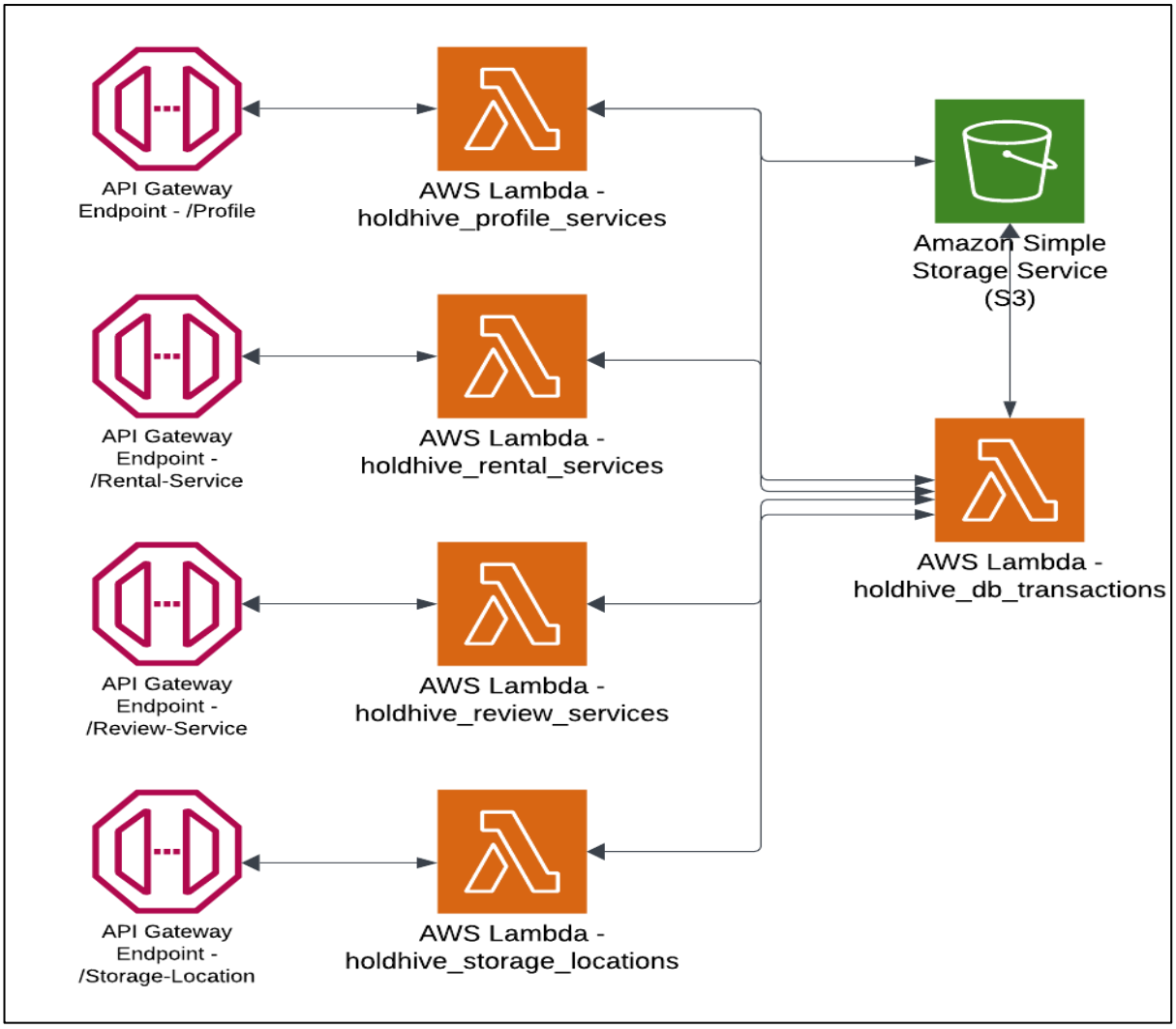

Figure 14 Arch-Diag-API-Lambda

3.1.6 Database Integration: Microsoft SQL Server

To preserve relational integrity and handle structured data:

- **Dedicated Lambda**: Only one microservice (DB-Lambda) holds the DB credentials, centralizing queries and transactions (Codd, 1970; Elmasri and Navathe, 2015).

- **Schema**: Key tables include Users, StorageSpaces, Rentals, Payments, and Reviews. Additional table AuditLogs captures events like listing creation or booking modifications.
- **Production-Like Constraint**: This approach prevents multiple microservices from simultaneously saturating the DB with random connections.

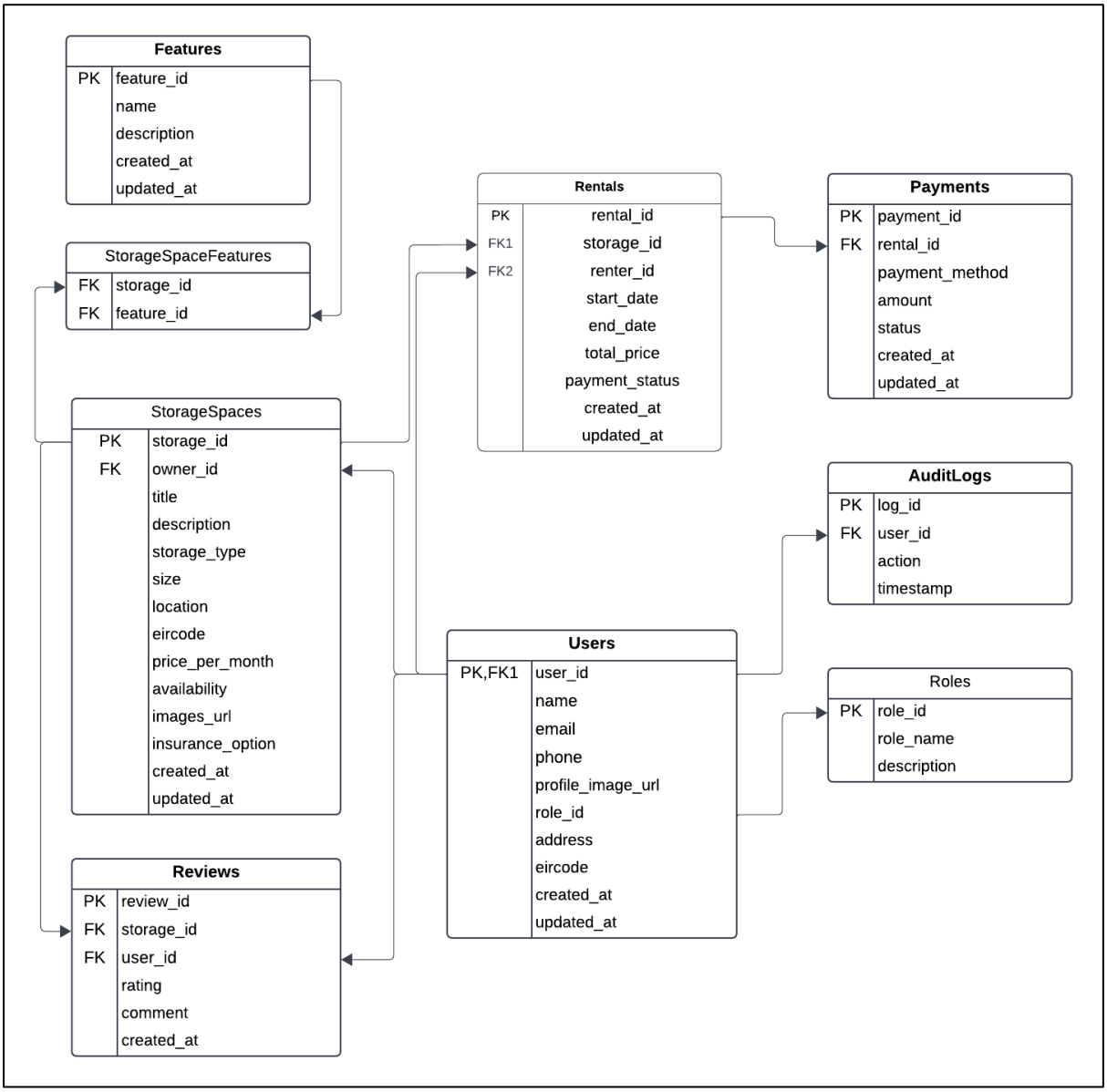

Figure 15 ***Holdhive****-ERD Diagram*

3.1.7 Image Storage: Amazon S3

Amazon S3 stores room/space images and user profile photos:

- **DB References**: The SQL Server stores object keys or URLs for easy retrieval.
- **Scalability**: Offloading images to S3 avoids clogging the DB with large BLOBs (Pressman, 2010).

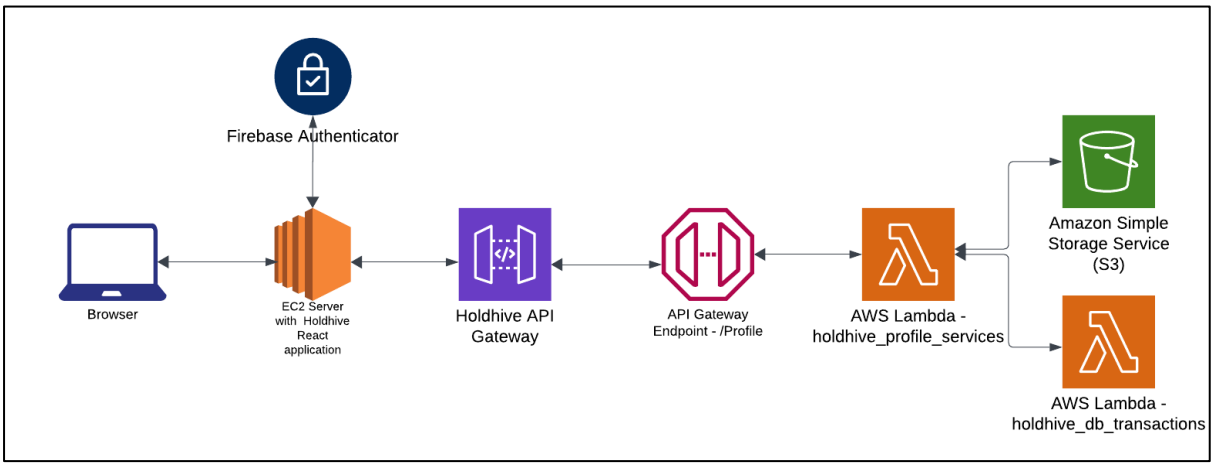

*Figure 16 **Holdhive**-Arch-S3-Upload*

3.1.8 Authentication: Firebase

Firebase Authentication simplifies user management:

- **ID Token Issuance**: Users authenticate (email/password or social) via the React client. Firebase returns a secure token such as userId (Firebase, n.d.).
- **Lambda Verification**: Each Lambda checks the userId before any action (Kshetri, 2013).
- **Role Assignments**: Host vs. Renter can be stored in the DB or embedded in custom claims.

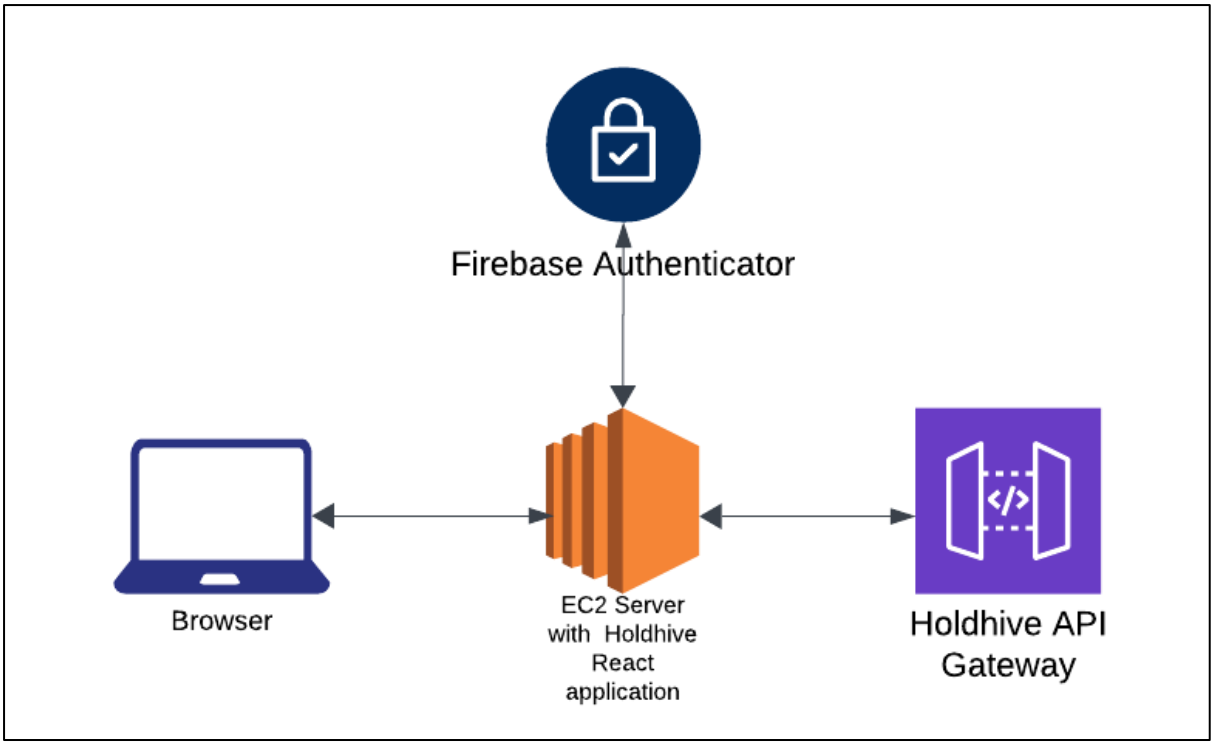

Figure 17 ***Holdhive****-Arch-Firebase-Auth*

3.1.9 Monitoring & Logging

AWS CloudWatch aggregates logs from each Lambda:

- **Real-Time Metrics**: Memory usage, invocation count, execution time.
- **Alerts**: Automatic alarms for error surges, letting you proactively fix issues (Ristenpart et al., 2012).
- **Debugging**: Detailed logs show function input, output, or stack traces for faster troubleshooting.

3.1.10 Rationale for a Production-Like Design

Building ***Holdhive*** on a cloud-native stack ensures:

1. **Scalability & Performance:** Serverless Lambdas react to load, while EC2 handles the React app (Amazon Web Services, 2020).

2. **Security**: A token-based auth system plus a single DB-Lambda fosters controlled data access (Pressman, 2010).
3. **Maintainability**: Microservices can be individually updated or replaced, reducing downtime (Highsmith, 2004).

All source code commits are backed up to GitHub as a secondary repository, reflecting real-world version control practices.

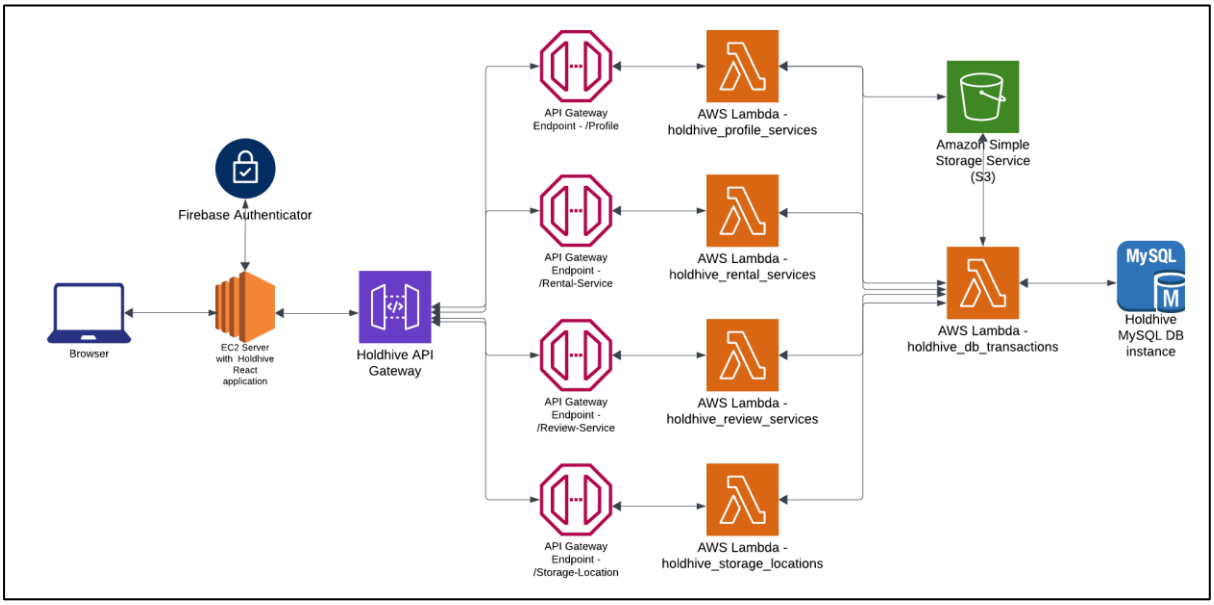

Figure 18 ***Holdhive****-Arch-Diagram*

3.2 Agile Development Methodology

3.2.1 Adaptation of Agile for a Solo Developer

Although Agile traditionally supports multi-person teams (Beck et al., 2001), the solo developer context for ***Holdhive*** involves:

1. **Short Sprints (1–2 weeks)**: Time-boxed periods to deliver incremental features (Cohn, 2005).

2. **Supervisor Collaboration**: Bi-weekly calls allow for demonstration, feedback, and backlog refinement (Highsmith, 2004).
3. **GitHub Commits**: Code changes are continually pushed to a GitHub repository, ensuring a reliable backup and easy version tracking.

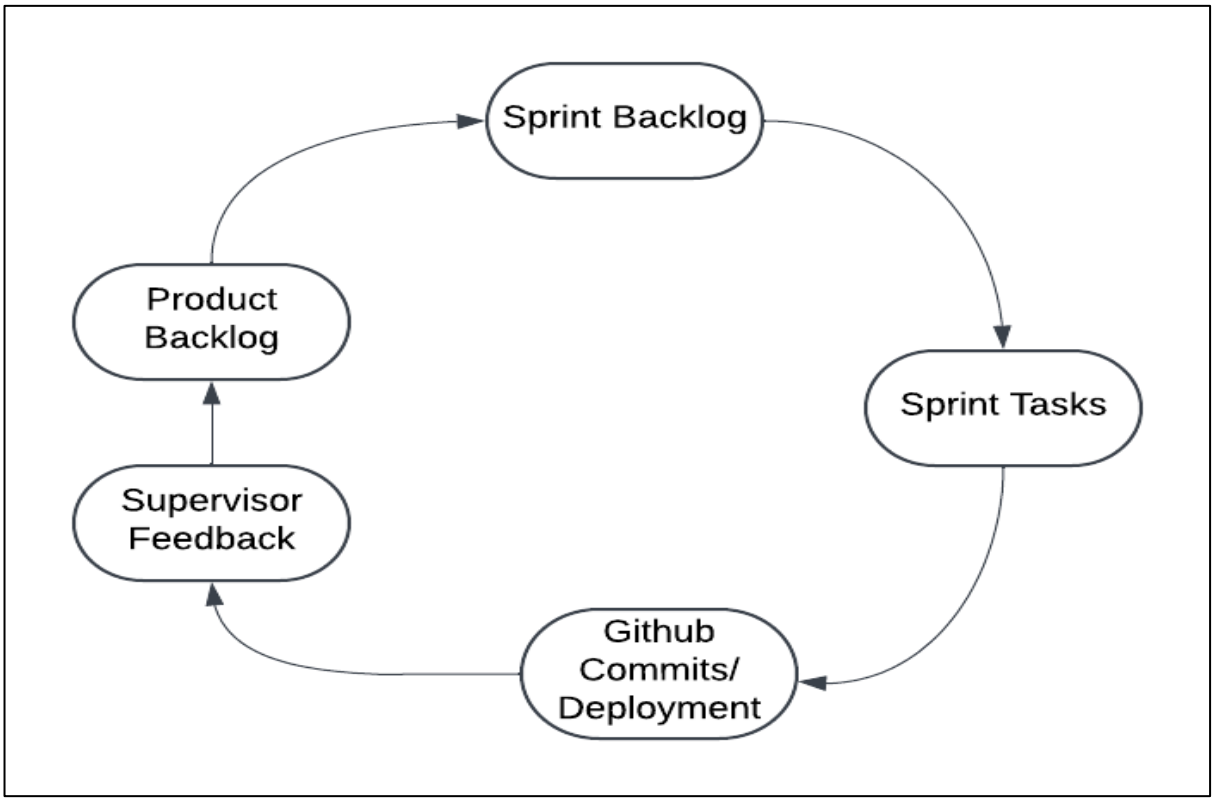

Figure 19 Agile Overview

3.2.2 Sprint Planning & Execution

1. **User Story Creation**: Features like “Implement S3 upload flow,” “Integrate booking microservice with DB” are broken down into tasks (Cohn, 2005).
2. **Time Estimation**: Assign approximate hours or days to each story to pace sprint progress.
3. **Development Cycle**: Code and tests are written, integrated, and deployed. GitHub commits provide a transparent revision history.

3.2.3 Testing & Continuous Integration

Since there is no dedicated QA team, automated testing is critical (Crispin and Gregory, 2009):

- **Unit Tests**: Validate booking logic, listing creation, or DB queries in Lambda functions.
- **Integration Tests**: Confirm that React, API Gateway, Lambdas, SQL Server, and S3 operate cohesively (Fowler, 2019).

3.2.4 Supervisor Calls & Retrospective

Each sprint concludes with a bi-weekly supervisor call:

1. **Demo**: Present newly developed features—e.g., refined listing forms, integrated payments, or enhanced security.
2. **Feedback Integration**: Suggestions on UI, code organization, or architecture feed back into the backlog (Highsmith, 2004).
3. **Retrospective**: You review what went well, any blockages encountered, and adapt the plan accordingly (Schwaber and Sutherland, 2020).

3.2.5 GitHub Backup & Code Management

All code changes are committed to a GitHub repository:

Github Link: https://github.com/gnanas1295/holdhive.com

- **Version Control**: Preserves each change along with commit messages (Pressman, 2010).

- **Branching Model**: You might use a main branch for stable releases and feature branches for individual tasks.
- **Secondary Backup**: This ensures code continuity even if local environments fail.

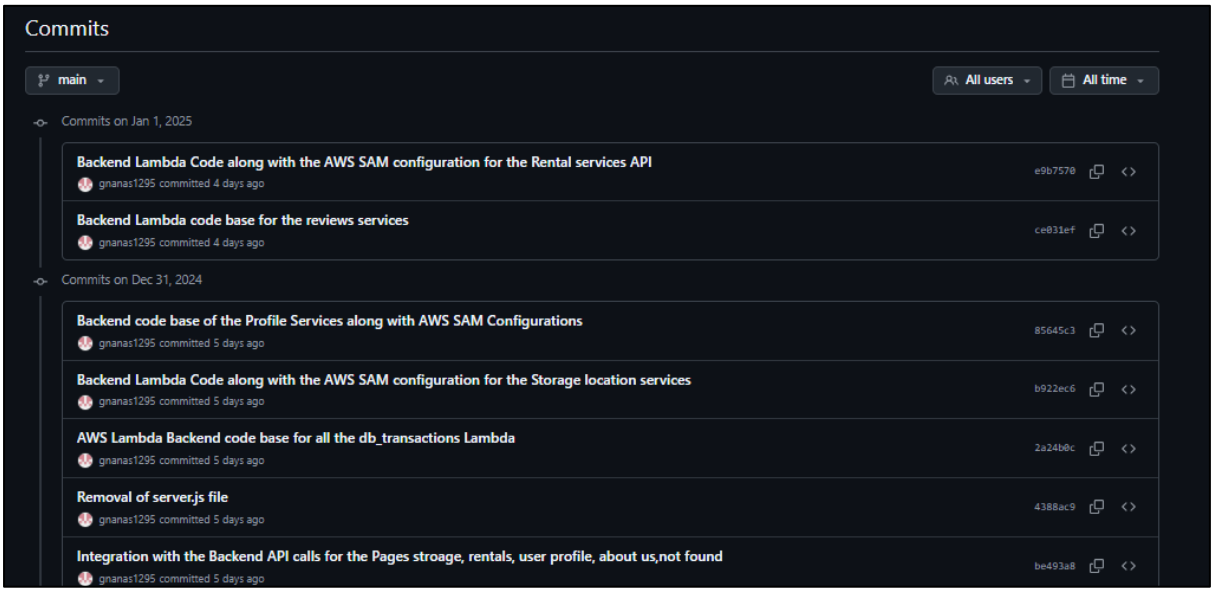

*Figure 20 GitHub Commit History for **Holdhive***

3.3 Survey Design and Feedback

3.3.1 Rationale for the Survey

Before and alongside developing ***Holdhive***, it was essential to understand potential user perspectives on storage needs, trust factors, and feature preferences. The survey aimed to:

1. Validate initial assumptions about peer-to-peer storage adoption.
2. Gather insights on insurance, reviews and ratings, and environmental considerations.
3. Inform design decisions during iterative agile sprints (Cohn, 2005; Highsmith, 2004).

By conducting this survey, we ensured that the **platform's core features** (e.g., insurance options, local storage preferences, pricing flexibility) aligned with real user expectations (Pressman, 2010).

Survey Link: https://forms.gle/nxd93j8K4SoBnbYR7

3.3.2 Survey Design and Questions

The survey was designed to collect both qualitative and quantitative data. It contained Likert-scale questions, multiple-choice items, and open-ended sections, ensuring comprehensive feedback (Crispin and Gregory, 2009). Some of **important survey questions** is as follows:

1. How often do you use or need storage solutions (e.g., self-storage units, renting storage)?
2. Would you consider using a platform like ***Holdhive*** to rent storage space?
3. If you own unused storage space, would you consider listing it for rent on ***Holdhive***?
4. What type of storage space are you most interested in (or have available)?
5. How important are the following features in a storage platform? [Security and surveillance]
6. How important are the following features in a storage platform? [Insurance options]
7. How important are the following features in a storage platform? [User reviews and ratings]
8. Would user reviews and ratings improve your trust in the platform?
9. How important is it for the platform to provide insurance options for stored items?
10. Would you feel more secure if the platform verified the identity of all users?

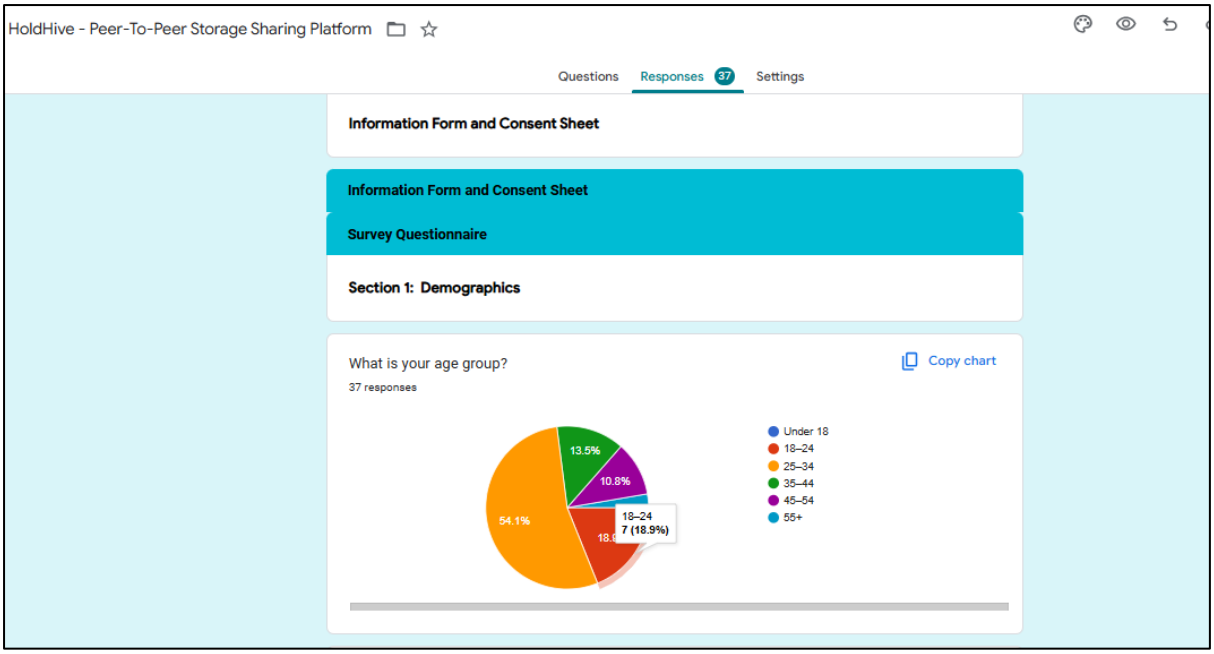

Figure 21 Survey Snippet Responses

3.3.3 Survey Administration and Participants

Participants were reached through online channels (social media, local community groups) to capture a broad range of demographics:

1. **Timing**: The survey was open both before coding began (to shape initial features) and periodically during development (to refine or confirm design choices).
2. **Respondent Demographics**: Age groups, employment status, living area type. This diversity helped validate assumptions about urban vs. rural storage needs (Kshetri, 2013).
3. **Anonymous Option**: Respondents could skip personally identifying questions; only those who volunteered an email were re-contacted for deeper feedback.

3.3.4 Key Findings and Impact on Design

Survey responses indicated:

1. **High Value on Insurance**
    - Over 70% rated insurance as “very important,” prompting an insurance option within the platform.
2. **Importance of Reviews**
    - A majority indicated user reviews and ratings increased trust (Resnick and Zeckhauser, 2002), so a robust review system was prioritized.
3. **Local vs. Price Trade-off**
    - Many respondents prefer local storage, but some would travel further if costs were significantly lower. This informed a location-based filter with cost comparisons.
4. **Environmental Considerations**
    - A notable subset liked the platform’s resource efficiency (reducing unused space), suggesting we highlight environmental benefits in future marketing.

These insights directly shaped product features, including the insurance add-on, advanced filtering for local vs. distant spaces, and a heavier emphasis on ratings and trust signals (Highsmith, 2004). **Code implementations** (e.g., listing forms, insurance toggles) are documented in GitHub commits referencing “survey feedback” in commit messages.

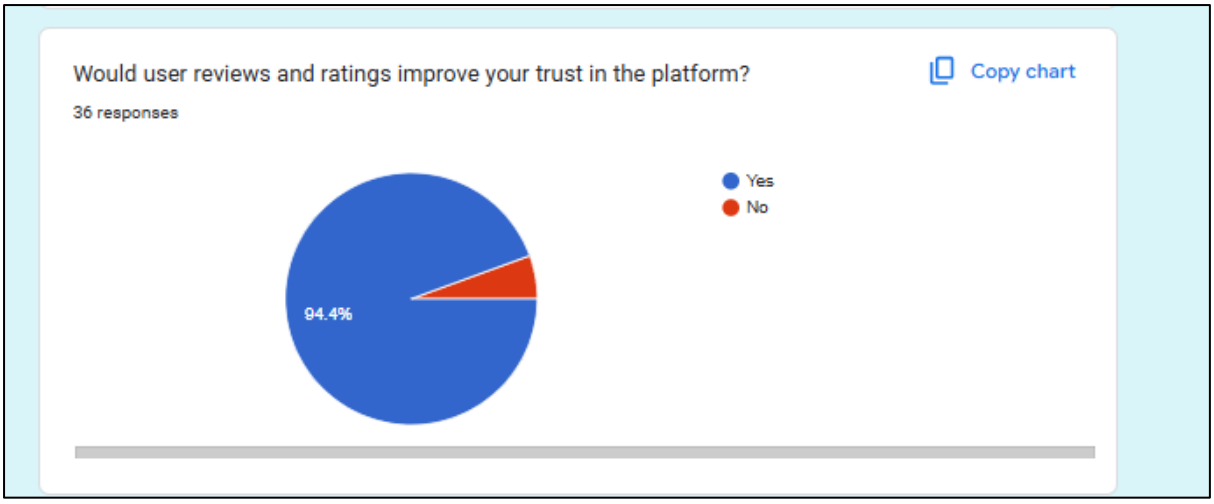

Figure 22 Survey Response Snippet on importance of the platform

3.3.5 Future Survey Extensions

As ***Holdhive*** evolves, follow-up surveys could refine:

- **Dynamic Pricing Acceptance**: Evaluate comfort with variable rates based on demand.
- **Eco-Friendly Features**: Gauge whether highlighting sustainability encourages user sign-ups (Pressman, 2010).
- **Add-on Services**: Potential interest in optional moving help or 24/7 access.

By continually integrating user feedback via iterative sprints, the platform remains user-focused and market-relevant (Crispin and Gregory, 2009).

CHAPTER 4

IMPLEMENTATION

4.1 Hardware and Software Specifications

4.1.1 Hardware

1. **AWS EC2**
    - **Instance Type**: t2.micro (1 vCPU, 1 GiB Memory).
    - **Role**: Hosts the ReactJS front end, accessible at http://***Holdhive***.com:3000/.
    - **Justification**: Adequate for minimal production or proof-of-concept environments (Amazon Web Services, 2020).

2. **AWS Lambda**
    - **Memory Allocation**: 128 MB per function.
    - **Purpose**: Executes backend “microservices” (listing creation, booking/rentals , reviewing, authentication checks).
    - **Auto-Scaling**: AWS automatically spawns additional instances to handle load (Highsmith, 2004).

3. **Microsoft SQL Server (Amazon RDS)**
    - **Instance Type**: db.t3.micro (2 vCPU, ~1 GiB Memory).
    - **Usage**: Storing user data, listings, bookings, reviews, payment info, etc. (Elmasri and Navathe, 2015).
    - **Connectivity**: A specialized Lambda function (using PyODBC or pymsql) manages transactions (Codd, 1970).

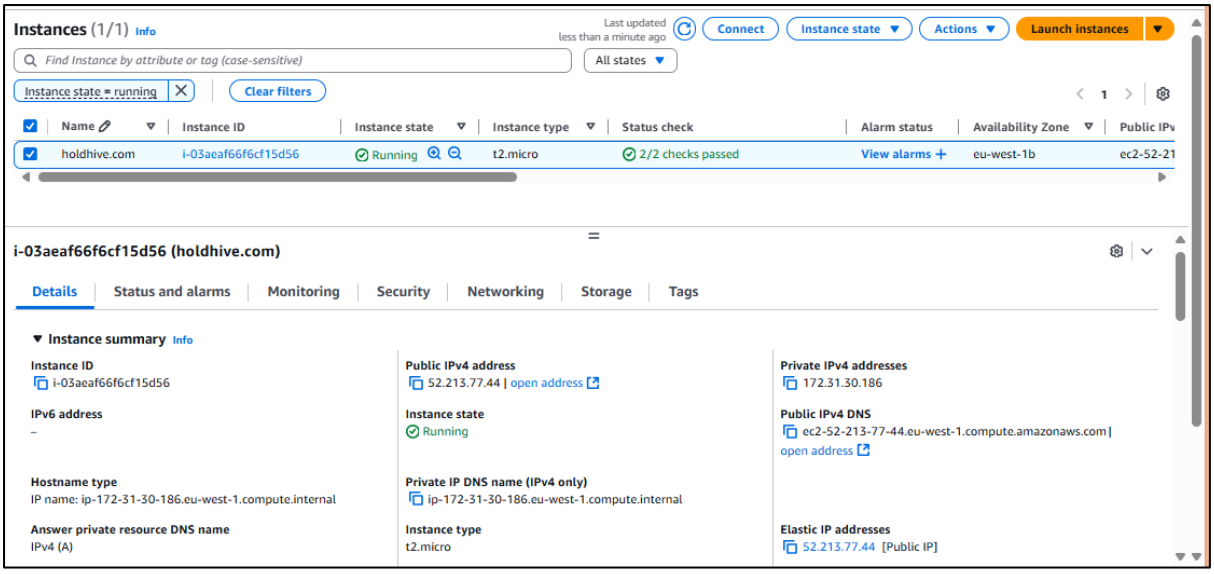

*Figure 23 **Holdhive** EC2 Server*

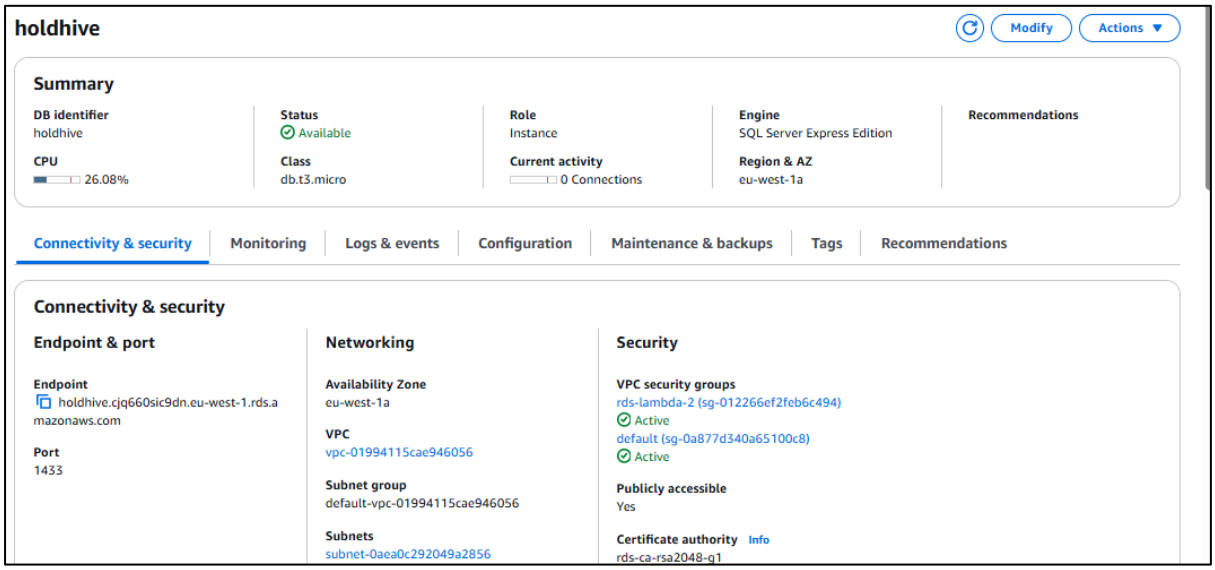

*Figure 24 **Holdhive** RDS Instance*

4.1.2 Software

- **Operating System**: Amazon Linux 2 on EC2.
- **Frontend**: ReactJS (JavaScript) for the single-page application (SPA) (Pressman, 2010).
- **Backend**: Python running on AWS Lambda, accessed via API Gateway (Fowler, 2019).

- **Authentication**: Firebase for user sign-up/login token issuance (Firebase, n.d.).
- **Object Storage**: Amazon S3 for storing images (room photos, user profile pictures) (Amazon Web Services, 2020).
- **Monitoring** AWS CloudWatch for logs and performance metrics.
- **Version Control**: GitHub (https://github.com/gnanas1295/***Holdhive***.com.git) for code commits, branching, and backups (Pressman, 2010).

4.2 Implementation Strategy

1. **Incremental Feature Rollout**
    - Survey insights (Chapter 3) determined priorities: insurance toggle, reviews system (Cohn, 2005).
    - Each feature was developed in sprints, merged into GitHub main branch upon supervisor approval.
2. **Production-Like Setup**
    - **Real AWS resources**: t2.micro (EC2), db.t3.micro (RDS), 128 MB Lambda.
    - Reflects constraints of a typical commercial environment (Amazon Web Services, 2020).
3. **Agile-Inspired Sprints**
    - **Bi-weekly Calls**: Supervisor feedback refined the backlog (Highsmith, 2004).
    - **Minimal Downtime**: Frequent, small updates so the site (http://***Holdhive***.com:3000/) stayed functional.

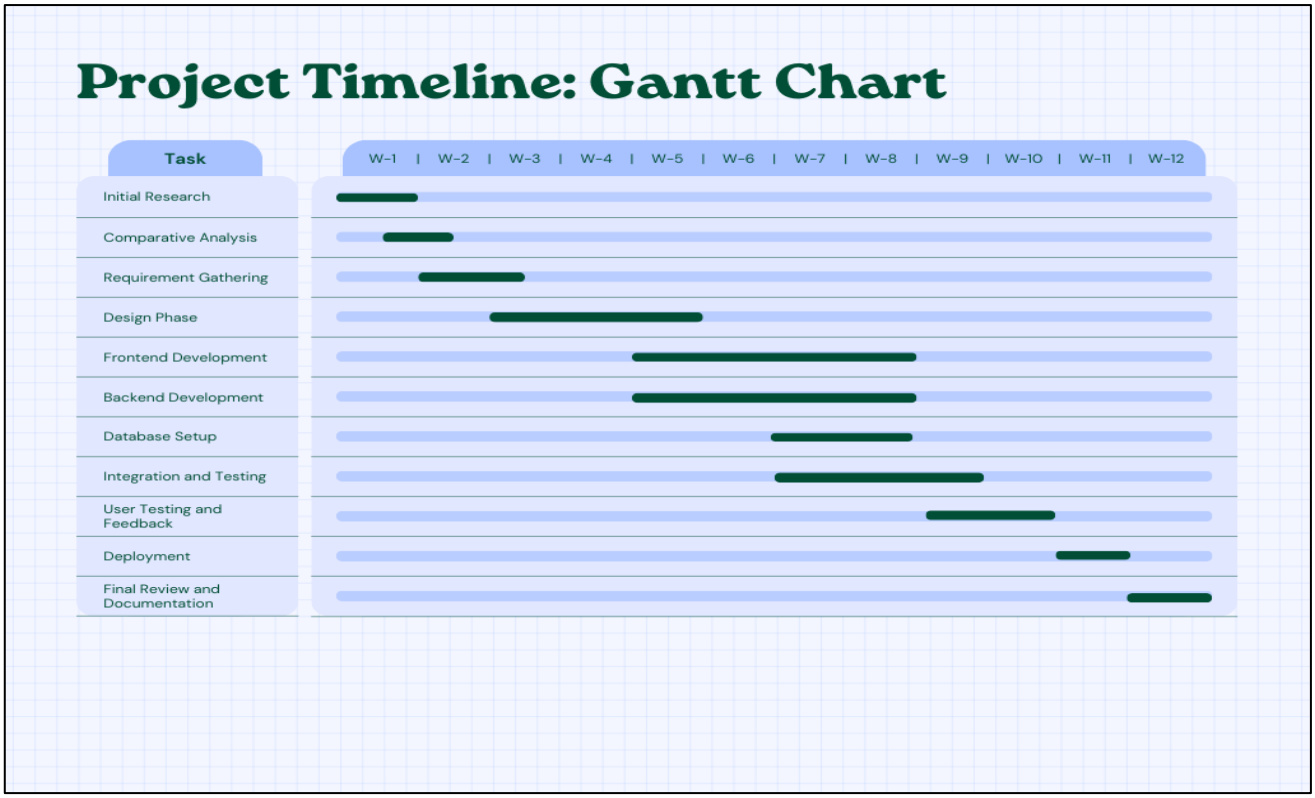

Figure 25 ***Holdhive*** *Gantt Chart*

4.3 Key Libraries and Modules

4.3.1 Frontend (ReactJS)

- **React Router DOM**: Handles SPA routes for pages like /home, /listing, /booking, /rentals (Pressman, 2010).
- **Axios**: Executes HTTP requests to AWS API Gateway endpoints.
- **React Context**: Global state management for user sessions, booking flows.

4.3.2 AWS Lambda (Python)

- **PyODBC**: Connects to MS SQL Server (Codd, 1970).
- **Boto3**: Interacts with AWS services (S3 file ops, CloudWatch logs) (Amazon Web Services, 2020).
- **Requests**: For external REST or microservice chaining.

4.3.3 SQL Server Tools

- **SQL Server Management Studio (SSMS)**: Designing schema, writing stored procedures, triggers.
- **Constraints**: Ensuring data integrity (e.g., no double-booking overlapping dates).
- **Stored Procedures**: E.g., for data analytics

4.3.4 Firebase (Authentication)

- **Firebase Admin** (server-side) or Firebase Web SDK (frontend) for user sign-up/login (Firebase, n.d.).
- **ID Tokens**: Ensures user identity in each API request.
- **Role Management**: Host, Renter, or Admin stored in custom claims or Users DB table (Kshetri, 2013).

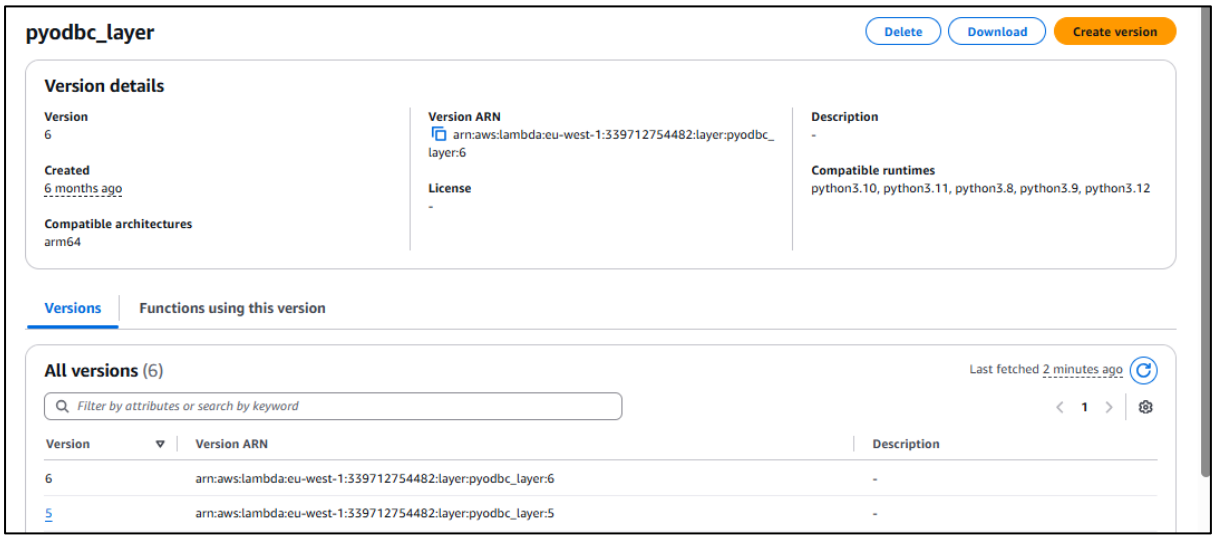

Figure 26 pyodbc_layer creation in Lambda

4.4 API Implementation with AWS Lambda

This section details how each Lambda function is built as a separate block handling a distinct feature (listing, booking, reviewing, etc.). All functions are invoked via **AWS API Gateway** using path-based and resource based routing.

4.4.1 Lambda Architecture

1. **Function Per Feature**
    1. **createListingLambda**: Creates storage listings, gets images to S3, inserts metadata into StorageSpaces table.
    2. **rentalsLambda**: Validates availability, handles insurance toggle, writes to Rentals.
    3. **reviewLambda**: Inserts or updates reviews in the Reviews table, calculates rating averages.
    4. **profileLambda**: Fetches and updates the user details to the users table.
2. **API Gateway Routes**
    - POST /create-listing → createListingLambda
    - POST /rentals → bookingLambda
    - POST /review → reviewLambda
    - GET /listings, GET /rentals, GET /reviews GET /user as read-only endpoints
3. **Authentication Flow**
    - Each request includes a Firebase token/id in body or request.
    - The Lambda function decodes the token, verifying user identity and role (Host, Renter, Admin).

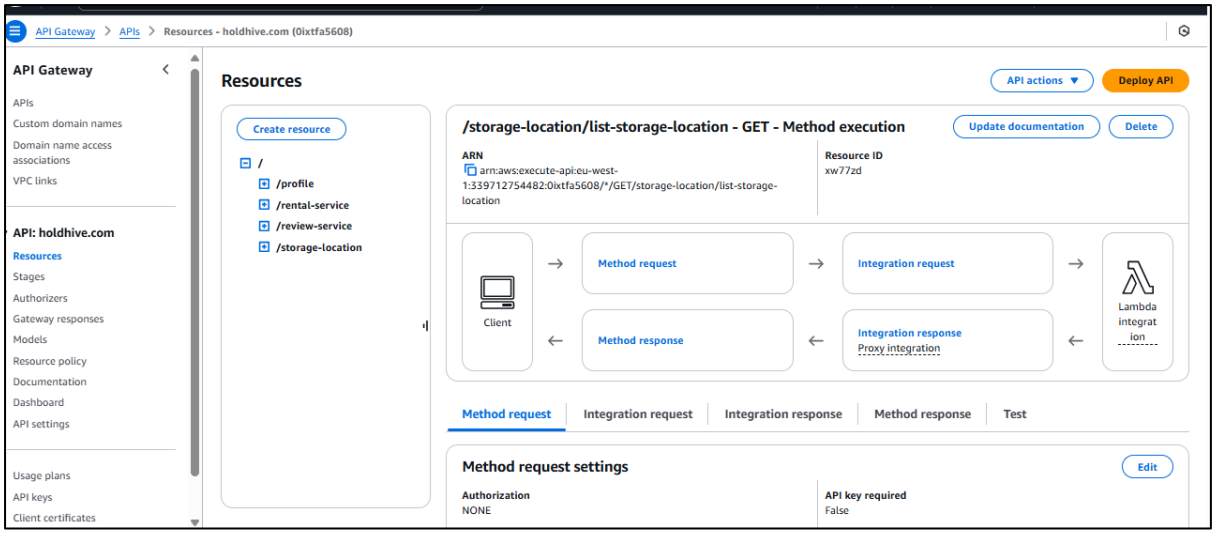

Figure 27 ***Holdhive*** *API Console Snippet*

4.4.2 Data Flow Inside a Lambda

- **Parameter Parsing**: Extract payload (e.g., listing details, booking date, star rating).
- **DB Connectivity**: Use PyODBC to run a stored procedure or direct INSERT/UPDATE.
- **Response Construction**: Return success or error JSON to the React front end.
- **Logging**: Write logs to CloudWatch (Amazon Web Services, 2020).

4.4.3 Example: reviewLambda

- **Input**: { "rental_id": 123, "rating": 5, "comment": "Great space!" }
- **Process**:
  1. Retrive Firebase userID → find user ID.
  2. Check if the user rented that rental_id.
  3. Insert row in Reviews table (rating, comment, user_id, storage_id).
  4. Update average rating for the associated storage_id if needed.
- Output: { "message": "Review added successfully", "status": 200 }

4.5 Web Pages with Authentication and Reviews

This subsection consolidates all main pages, explicitly showing authentication checks and review functionalities.

4.5.1 Home.js

- **Purpose**: Landing page describing P2P storage concept.
- **Auth**: No restriction; open to all.
- **Features**: Quick links ("List My Storage," "Find Storage") that check if the user is logged in before redirecting.

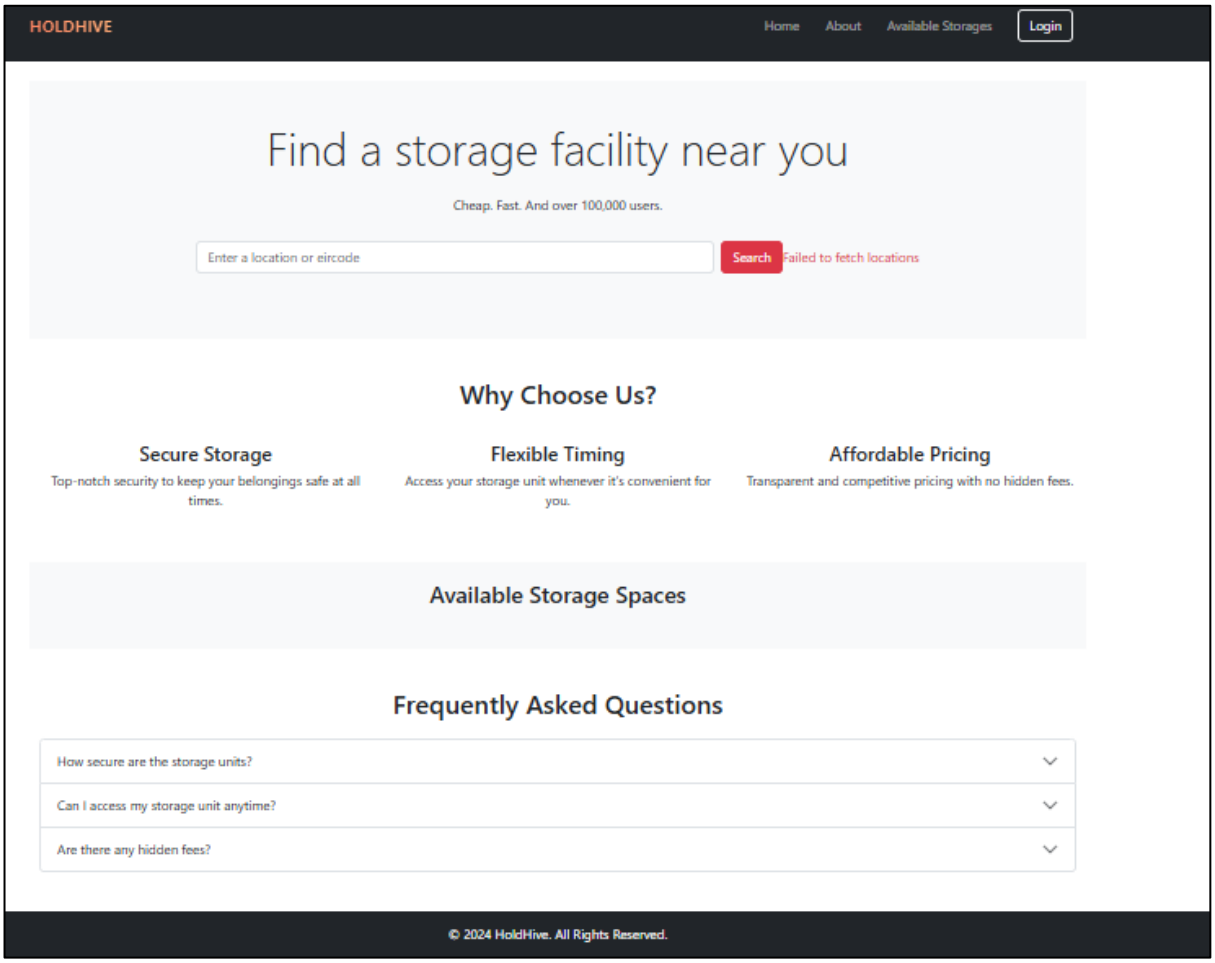

Figure 28 ***Holdhive****-HomePage*

4.5.2 About.js

- **Purpose**: Explains mission, background, potential environmental benefits.
- **Auth**: No restriction; informational page.

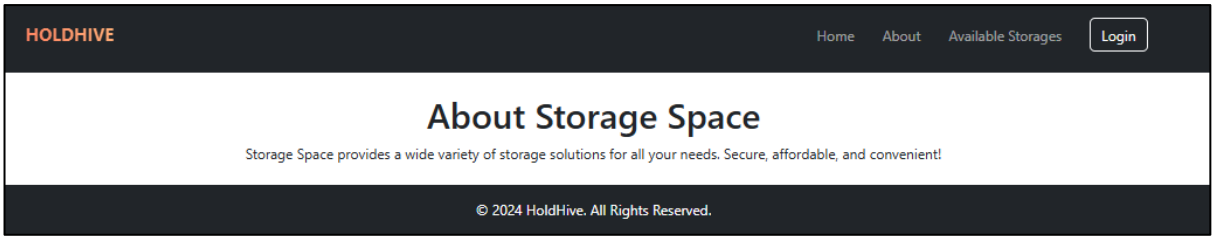

Figure 29 ***Holdhive*** *- AboutUsPage*

4.5.3 Listings.js

- **Purpose**: Host-oriented page to create or update storage spaces.
- **Authentication**: Firebase token must indicate Host role.
- **S3 Image Retrival**: Uses References saved in StorageSpaces table and retrieves the images from the S3

4.5.4 AllStorages.js (Admin)

- **Purpose**: Admin review of all listings in the system.
- **Implementation**: Summarizes data from StorageSpaces; can remove fraudulent or outdated listings.
- **Auth**: Must have admin privileges (Kshetri, 2013).

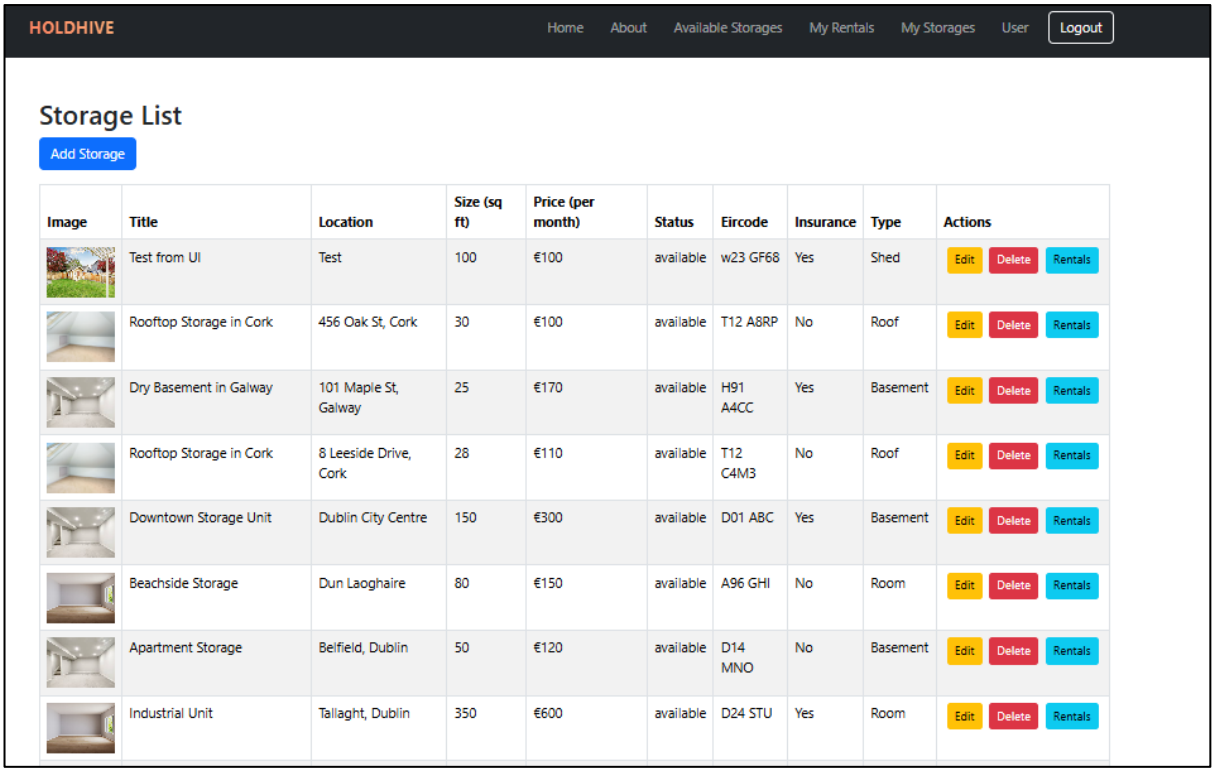

Figure 30 Admin View Storage Location

4.5.5 StorageListings.js (Browse)

- **Purpose**: Public or logged-in user view of available listings.
- **Features**: Filtering by location, eircode, or price.
- **Reviews**: Each listing may display its average rating (queried from Reviews).

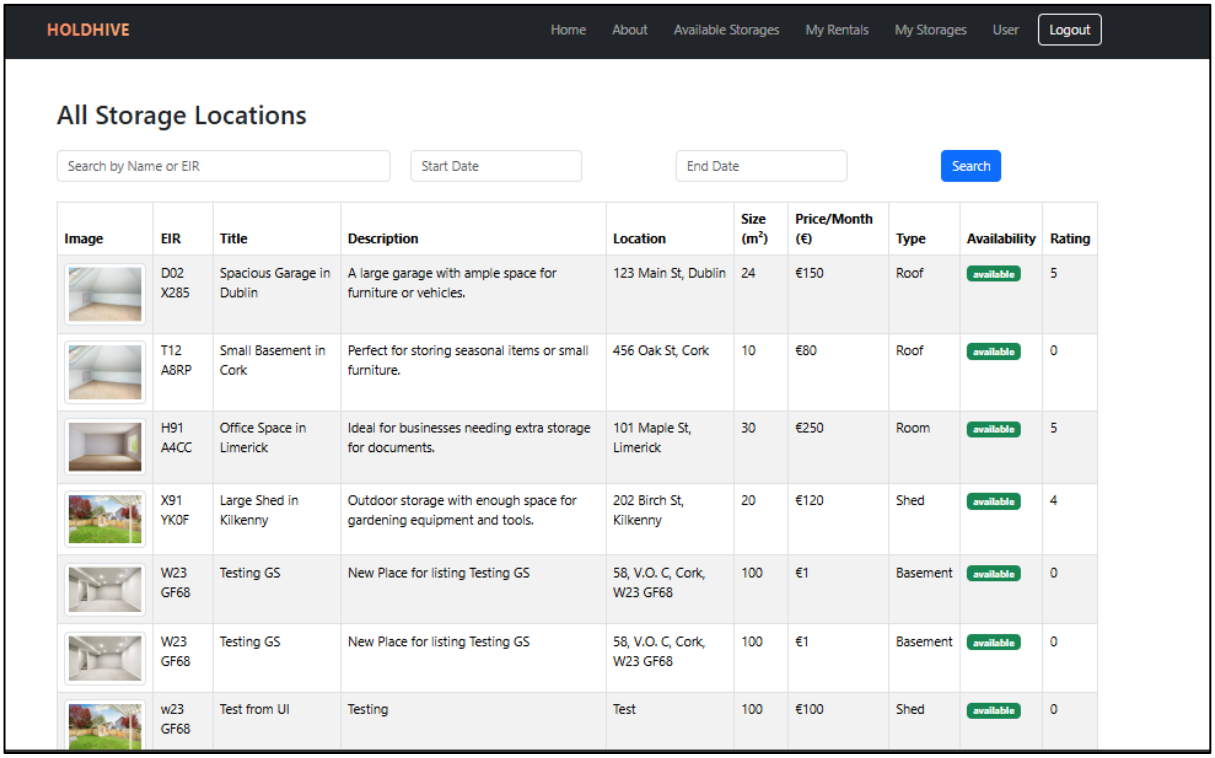

Figure 31 Available Storage Location - Page 1

4.5.6 Booking.js

- **Purpose**: Handles the full rental flow.
- **Auth**: Must be a Renter (Firebase user Id).
- **DB**: On submission, rentalsLambda writes to Rentals, and many things.

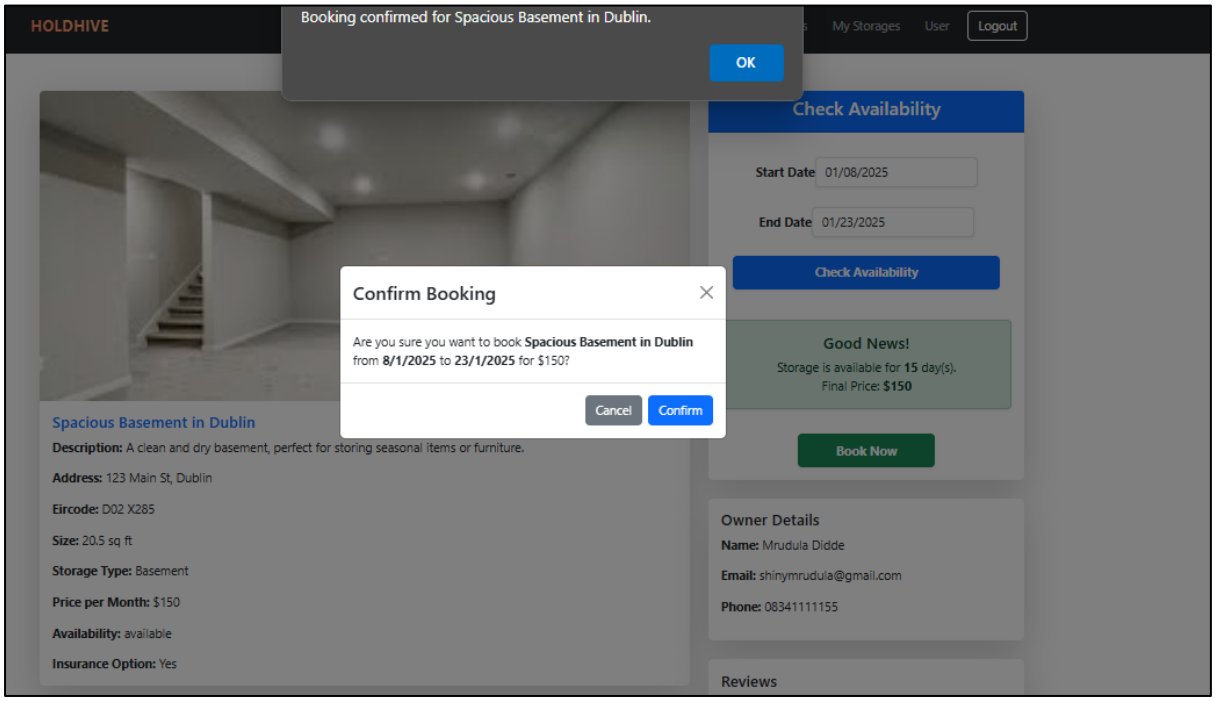

Figure 32 Booking Flow - Page 6

4.5.7 RentalDetails.js

- **Purpose**: Shows specific booking info—dates, final cost, insurance status, etc.
- **Review Prompt**: If rental is completed, user can post a review.
- **Implementation**: rentalLambda and reviewLambda is invoked, referencing rental_id and user ID.

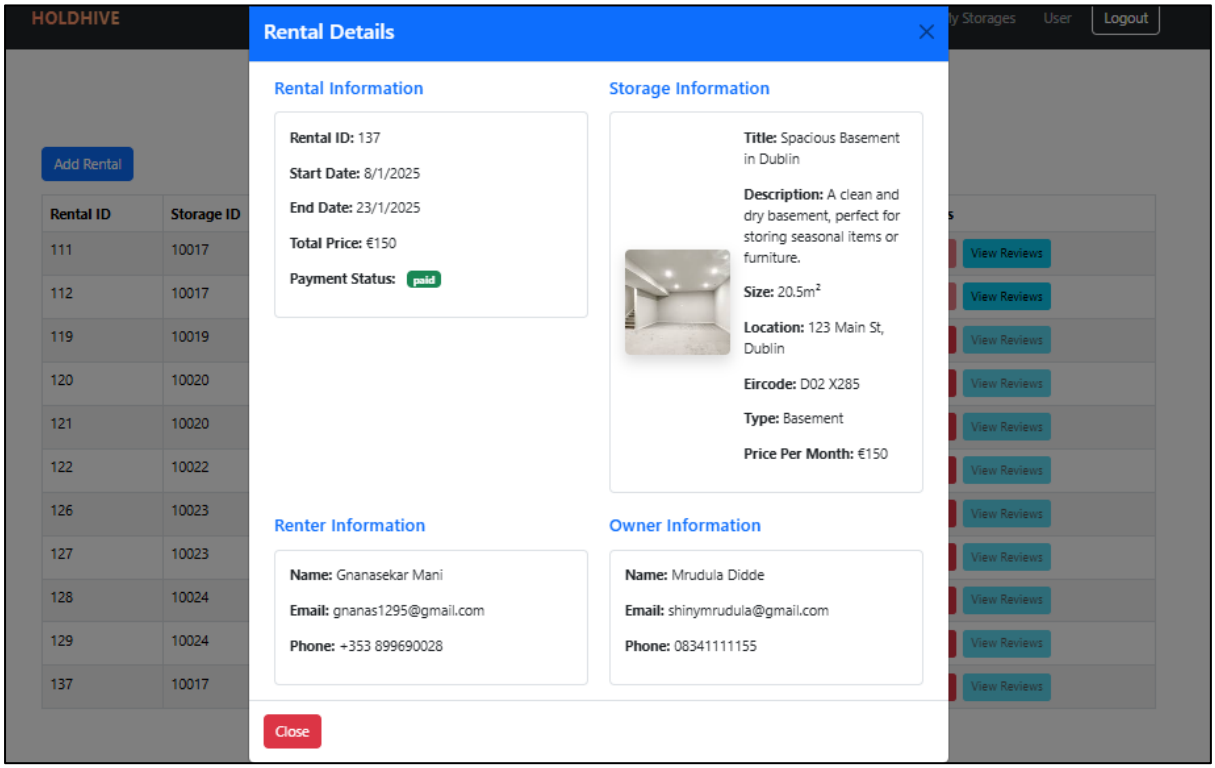

Figure 33 Rental Details - Page 1

4.5.8 AllRentals.js (Admin/Host)

- **Purpose**: Summarizes all rentals, storage spaces, users for administrative or host oversight.
- **Auth**: Host sees only their listings' rentals; Admin sees all (Pressman, 2010).

4.5.9 RentalListings.js (Renter's Bookings)

- **Purpose**: A Renter's personal history of rentals.
- **Review**: If any booking is finished, user can add or edit a review (Resnick and Zeckhauser, 2002).

4.5.10 RentalsByStorage.js (Host)

- **Purpose**: Host's overview of all rentals for a particular storage listing.
- **Auth**: Confirms user is the listing's owner.
- **Reviews**: The host can see the rating or comments left by renters.

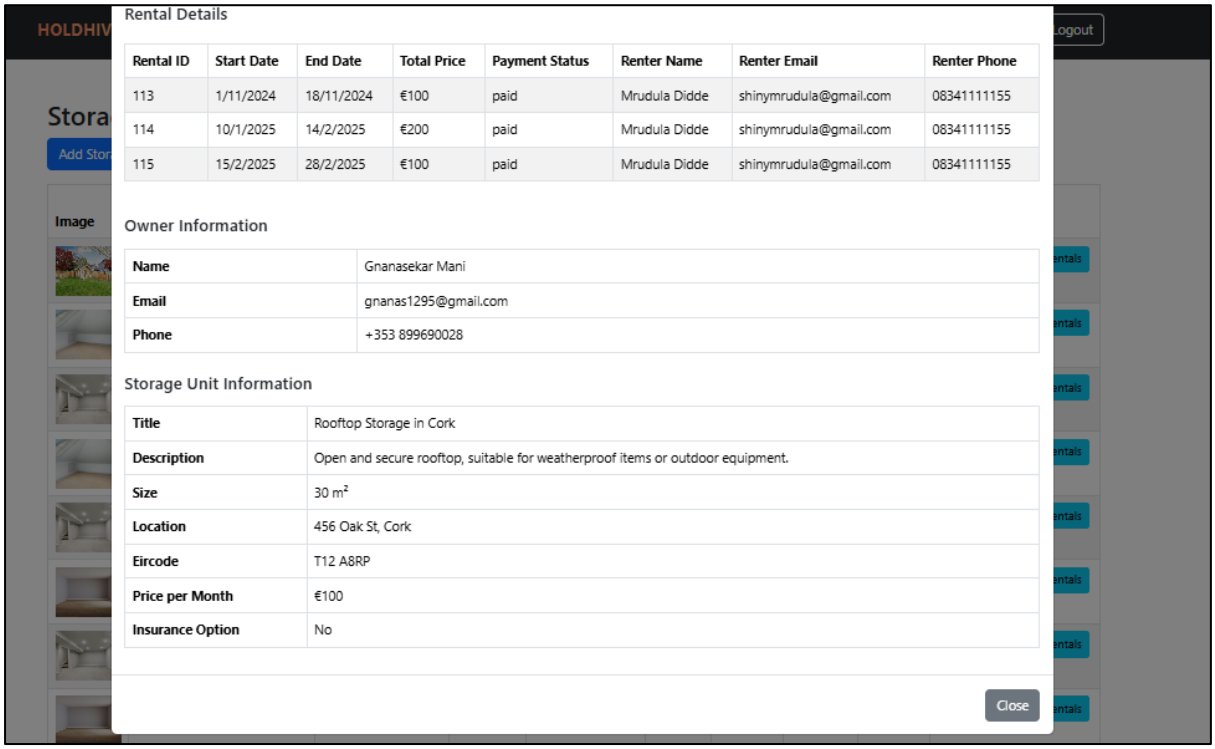

Figure 34 Rentals by Storage - Page 1

4.5.11 Profile.js

- **Purpose**: Displays user data from Firebase or Users table.
- **Host**: Sees a summary of owned listings, potential average ratings from renters.
- **Renter**: Sees booking history, personal info, ability to update certain fields.

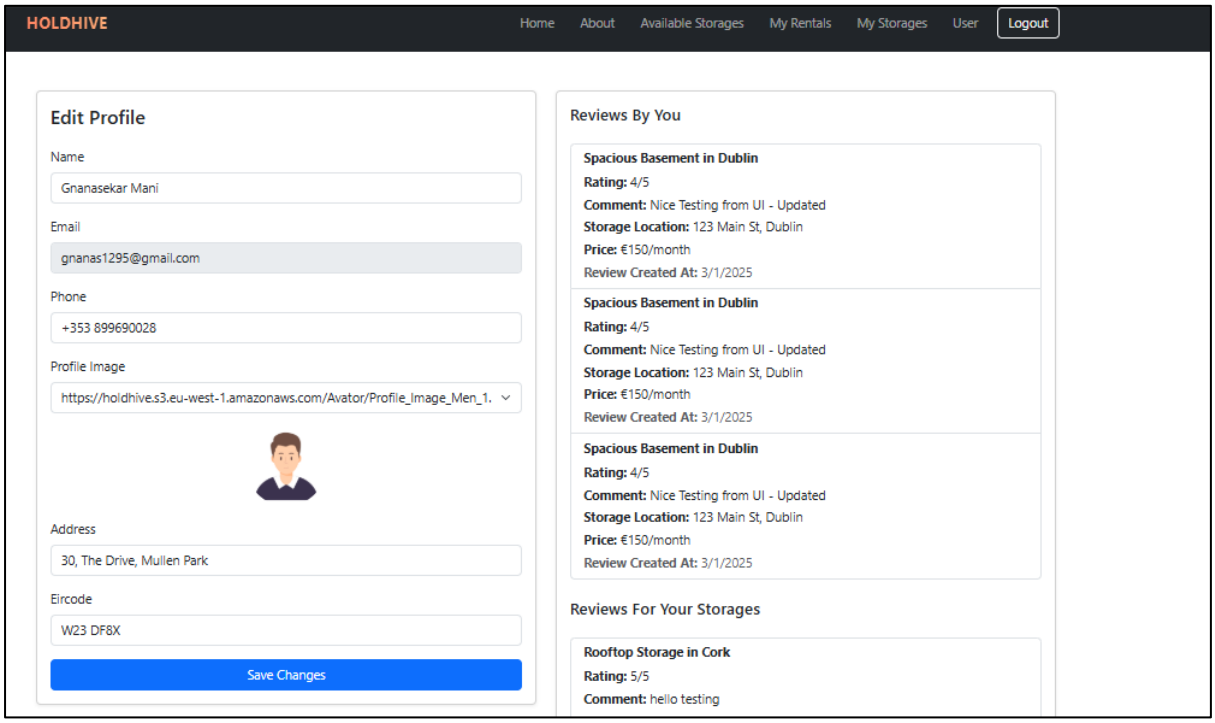

Figure 35 Profile Dashboard

4.5.12 Admin.js

- **Purpose**: Master admin portal for system analytics, suspicious activity logs, or user role management.
- **Auth**: Must be an admin claim in Users table.
- **Review Oversight**: Potentially moderates reported reviews or spam.

4.5.13 NotFound.js (404)

- **Purpose**: Renders a "Page Not Found" message for invalid routes.

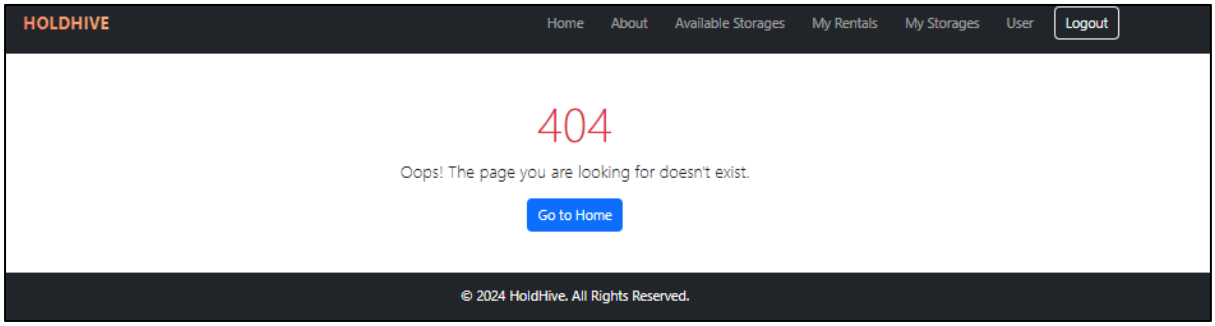

Figure 36 Page not found

4.6 Procedure

4.6.1 User Flow

1. **Access**: http://***Holdhive***.com:3000/ → Explore about P2P storage.
2. **Registration**: Firebase-based sign-up (email/password or social).
3. **User Role**: System checks if user is Host, Renter, or Admin (via DB).
4. **Feature Usage**:
    - Host → "Listings.js" to add storage.
    - Renter → "StorageListings.js" to find a space, "Booking.js" to finalize.
    - After rental, "RentalDetails.js" allows leaving a review.
5. **Review Integration**: The user's rating is aggregated to the listing's overall score.

4.6.2 Development Timeline

- **Phase 1**: Basic pages, minimal DB schema, core Lambdas (createListing, booking).
- **Phase 2**: Add insurance toggle, integrate with booking microservice.
- **Phase 3**: Implement reviewLambda, star ratings, average rating calculations.

- **Phase 4**: Admin functionalities (AllStorages.js, AllRentals.js), code polishing, final testing (Pressman, 2010).

4.7 Ethics

4.7.1 Ethical Considerations

- **Informed Consent**: Survey participants and early testers informed about data usage (Crispin and Gregory, 2009).
- **Data Anonymity**: Minimal personal info stored.
- **Right to Withdraw**: Participants can request data removal or account deletion.

4.7.2 Consent Forms and Appendices

- Contains sample consent forms, disclaimers, user information sheets in surveys.
- Debrief: Final explanation on how user feedback influenced features (Cohn, 2005).

4.8 Obstacles and Resolutions

Below are the key challenges encountered during ***Holdhive***'s development and deployment, plus how they were addressed.

1. **pm2 Process for React/Node**
    - **Issue**: By default, running npm start is ephemeral; closing the SSH session stops the process.
    - **Resolution**: Installed pm2 on EC2 to keep the Node server running continuously.
    - **Benefits**: Auto-restart on crashes, easy logs and scaling.
2. **Lambda Layers for DB Drivers**
    - **Issue**: AWS Lambda runtime lacks preinstalled MSSQL drivers.

- **Solution**: Created a custom layer with PyODBC, tested connectivity to db.t3.micro (Codd, 1970).

3. **API Gateway JSON Parsing**
   - **Problem**: Transforming request/response bodies for different endpoints.
   - **Fix**: Proxy integration passing raw JSON; the Python code in each Lambda handles parsing (Fowler, 2019).
4. **Reviews Authenticity**
   - **Concern**: Ensuring only real renters can leave reviews.
   - **Approach**: reviewLambda cross-checks rental_id with user's Firebase UID.
5. **Production-Level Environment Setup**
   - **Learning Curve**: IAM roles, VPC security, environment variables for DB credentials.
   - **Mitigation**: Created dev → staging → production environment flows.
   - **Responsiveness**: Ensured React UI supports multiple device sizes (media queries, flexible layouts).
6. **Performance & Responsiveness**
   - **Observation**: t2.micro with pm2 handles moderate traffic, but concurrency is limited.
   - **Plan**: Potentially upgrade instance types (t2.small or t3.medium) for higher loads (Amazon Web Services, 2020).

4.9 Additional Learnings and Production Environment Setup

1. **AWS Best Practices**
    - Understood the difference between dev/staging vs. production (IAM roles, cost optimization).
    - Gained experience with setting up VPC and subnets for secure DB connections.
2. **Responsiveness & UI**
    - Ensured React front-end uses responsive design (CSS media queries, flexible layouts).
    - Verified on mobile, tablet, and desktop using Chrome DevTools (Pressman, 2010).
3. **Scalability**
    - Learned how to leverage API Gateway scaling and AWS Lambda concurrency.
    - Realized the db.t3.micro may become a bottleneck if user traffic spikes.
4. **Logging & Monitoring**
    - Integrated CloudWatch for function-level logs.
    - pm2 logs for the React server on EC2, enabling quick debugging.
5. **Real-World Deployment**
    - By using pm2, your React/Node server automatically restarts on system reboot or crashes.

4.10 Summary

This Chapter 4 thoroughly covers:

- **Hardware/Software Setup**: t2.micro EC2 for React, db.t3.micro RDS for SQL, Lambda 128 MB memory.

- **Implementation Strategy**: Agile sprints, incremental feature rollouts, GitHub-based version control.
- **API with AWS Lambda**: Separate microservices for listings, bookings, and reviews, invoked via API Gateway.
- **Web Pages (Authentication & Reviews):** A unified list of ReactJS pages (Listings, Booking, Admin) that check Firebase tokens and incorporate the reviews system.
- **User Procedure**: The typical journey from sign-up to final rating submission.
- **Ethics**: Minimizing personal data, ensuring informed consent, alignment with privacy guidelines.
- **Obstacles**: Overcoming Lambda environment limitations (DB drivers), ensuring real-time connectivity, verifying reviews authenticity.
- **Production-level insights:** On environment creation, responsiveness, and logging best practices.

By building ***Holdhive*** in a production-like AWS environment and integrating survey-driven features (insurance, ratings), we demonstrate a fully functional, scalable peer-to-peer storage platform.

CHAPTER 5
TESTING AND EVALUATION

5.1 Test Case Descriptions

5.1.1 Testing Approach

All tests were conducted manually, ensuring that ***Holdhive***'s critical functionalities—listing creation, bookings, reviews, authentication, admin oversight, etc.—behaved as expected. Manual testing involved step-by-step user actions, verifying each success or error condition (Crispin and Gregory, 2009; Pressman, 2010).

5.1.2 Test Case Format

1. **Test Case ID**: Unique label (T-001, T-002, etc.).
2. **Feature**: The specific function or user flow under test.
3. **Description**: Brief statement of what is being validated.
4. **Preconditions**: State or setup needed before test execution (e.g., user role, existing data).
5. **Test Steps**: Manual sequence of actions.
6. **Expected Outcome**: Desired system result or response.

5.1.3 Detailed Test Cases

1. **T-001: Create Listing as Host**
    - **Feature**: Listings (Host Flow)
    - **Description**: A Host manually creates a new storage listing.
    - **Preconditions**: User logged in as Host, system has at least one Host account.
    - **Test Steps**:
        1. Go to “Listings” page.

2. Click “Create New Listing.”
3. Fill in details (title, price, location).
4. Select sample image.
5. Submit.

- **Expected Outcome**: New listing visible in the DB’s StorageSpaces table; success message on UI.

2. **T-002: Book Storage as Renter**

- **Feature**: Booking (Renter Flow)
- **Description**: A Renter manually books an available listing.
- **Preconditions**: User logged in as Renter, at least one listing available.
- **Test Steps**:
    1. Go to “StorageListings” page.
    2. Choose a listing, click “Book.”
    3. Enter valid dates if needed.
    4. Submit.
- **Expected Outcome**: Entry inserted into Rentals; confirmation message displayed.

3. **T-003: Leave a Review**

- **Feature**: Reviews

- **Description**: A Renter manually leaves a star rating and comment for a completed rental.
- **Preconditions**: Renter has a finished booking; end date is in the past.
- **Test Steps**:
    1. Open “RentalDetails” page for that booking.
    2. Click “Leave a Review.”
    3. Enter 5-star rating, short comment.
    4. Submit.
- **Expected Outcome**: A new row in Reviews table, updated average rating for the listing.

4. **T-004: Admin Removes a Listing**

- **Feature**: Admin Management
- **Description**: An admin manually removes an unwanted or outdated listing.
- **Preconditions**: User logged in as Admin; listing to remove is valid.
- **Test Steps**:
    1. Go to “AllStorages.”
    2. Select a listing, click “Remove.”
    3. Confirm deletion.

- **Expected Outcome**: Listing no longer appears in user-facing pages; StorageSpaces table entry removed.

5. **T-005: View Rental History**
    - **Feature**: Renter's "RentalListings"
    - **Description**: Renter manually checks all their past/ongoing rentals.
    - **Preconditions**: Renter has at least one booking in Rentals.
    - **Test Steps**:
        1. Go to "RentalListings."
        2. Verify the list displays all rentals.
        3. Click a rental to see details.
    - **Expected Outcome**: Page lists correct rentals for the logged-in Renter; details match DB entries.

6. **T-006: Check Authentication (Firebase)**
    - **Feature**: Firebase Login
    - **Description**: Manually verify login flow with valid credentials.
    - **Preconditions**: User has a registered account (email/password).
    - **Test Steps**:
        1. Go to Home.

2. Click “Login,” enter valid credentials.
3. Check if user is redirected to Profile or relevant role-based page.

- **Expected Outcome**: Firebase user_id issued, user recognized as correct role (Host/Renter/Admin).

7. **T-007: Admin Panel Overview**

- **Feature**: Admin Dashboard
- **Description**: Admin manually checks “AllRentals” and “AllStorages” for system-wide data.
- **Preconditions**: Admin role, existing listings and rentals.
- **Test Steps**:
  1. Log in as Admin.
  2. Navigate “AllRentals,” verify every rental in Rentals table.
  3. Navigate “AllStorages,” verify each entry from StorageSpaces.
- **Expected Outcome**: Admin sees consistent data.

8. **T-008: Error Handling on Invalid Booking Dates**

- **Feature**: Booking Validation
- **Description**: Renter attempts to book overlapping or invalid dates.
- **Preconditions**: Listing already booked for certain dates.

- **Test Steps**:
    1. Try booking the listing for the same date range.
    2. Submit.
- **Expected Outcome**: System shows an error message (“Space not available”), no duplicate entry in Rentals.

5.2 Test Results Summary

The following table summarizes the outcome of each of the **8 manual test cases** described in **Section 5.1**. In this round of testing, **all cases passed** successfully, matching the expected outcomes.

Table 1 Summary of Test Case Results

Test Case ID	Feature	Pass/Fail	Observations
T-001	Listings (Host Flow)	Pass	Listing & S3 image upload worked.
T-002	Booking (Renter Flow)	Pass	Booking recorded, insurance cost.
T-003	Reviews	Pass	5-star rating updated in DB.
T-004	Admin Remove Listing	Pass	Entry removed from StorageSpaces.
T-005	View Rental History	Pass	Renter’s bookings displayed OK.
T-006	Firebase Authentication	Pass	Login token recognized properly.
T-007	Admin Panel Overview	Pass	All data consistent in overview.
T-008	Invalid Booking Dates	Pass	Overlapping date error displayed.

CHAPTER 6

DATA ANALYTICS

6.1 Overview of Analytics Framework

As **HoldHive** matures, data-driven insights become critical for operational efficiency and strategic decisions (Pressman, 2010). By integrating Tableau directly with the Microsoft SQL Server (RDS db.t3.micro), the platform can generate real-time analytics reports on rental trends, storage occupancy, top customers by revenue, and review/ratings patterns (Elmasri and Navathe, 2015). These reports guide Hosts, Renters, and Admins in optimizing listings, managing capacity, and identifying highly valued customers.

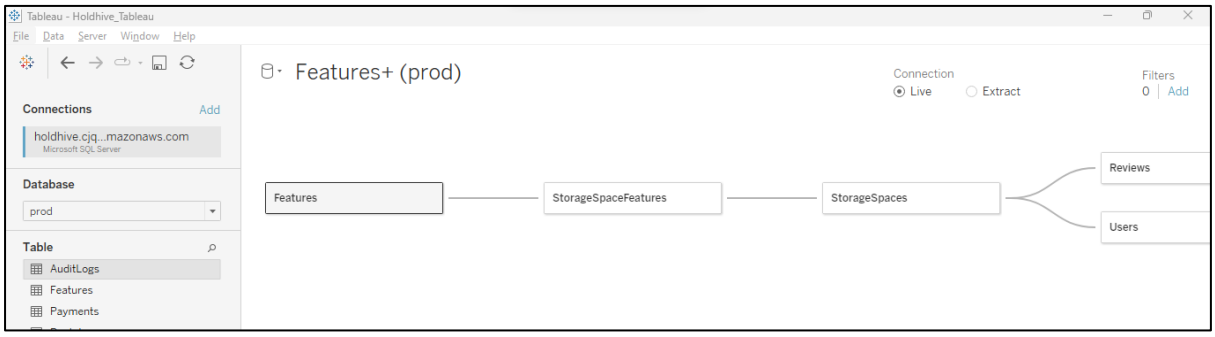

Figure 37 Tableau connection with RDS SQL Server

6.2 Data Pipeline and Integration

6.2.1 Database Schema and Connectivity

- **SQL Server Schema**: Core tables include `StorageSpaces`, `Rentals`, `Reviews`, `Users`, and `Payments` (Codd, 1970).
- **Live Connection**: Tableau connects to the RDS endpoint, retrieving updated data for each report (Fowler, 2019).
- **Data Refresh**: On-demand ensuring that newly created listings, rentals, or reviews appear in dashboards.

6.2.2 Tableau Workbook Organization

- **Workbooks**: Each set of analytics (Monthly Rental Trend, Occupancy, Top Customers, Reviews) resides in dedicated Tableau worksheets.
- **Joins**: For instance, Rentals joined with StorageSpaces to differentiate storage types and occupancy rates.
- **Filters & Parameters**: Date ranges, user roles, or location-based filters can refine the displayed metrics (Crispin and Gregory, 2009).

6.3 Report 1: Monthly Rental Trend (by Storage Type)

6.3.1 Purpose

This report visualizes rental activity over time, categorized by storage type (e.g., “Roof,” “Basement,” “Room,” “Shed”.). It helps Hosts see seasonality and gauge demand patterns (Cohn, 2005).

6.3.2 Data Fields

- **Date (Month-Year)**: Derived from Rentals.start_date
- **Storage Type**: Pulled from StorageSpaces.storage_type.
- **Rental Count**: Aggregates total rentals per month per storage type.

6.3.3 Insights

- Identifies peak months and popular storage categories.

Figure 38 Monthly rental counts segmented by storage type

6.4 Report 2: Storage Space Occupancy (as of Date)

6.4.1 Purpose

Tracks current occupancy rates for each storage listing, assisting Hosts and Admins in capacity planning (Highsmith, 2004).

6.4.2 Data Fields

- **Storage ID / Storage Name**: Unique identifier from StorageSpaces.
- **Occupancy Rate**: Ratio of currently booked count current and future dates.

6.4.3 Insights

- Identifies underutilized vs. fully occupied spaces.
- Helps Hosts optimize pricing or marketing strategies for low-occupancy listings.

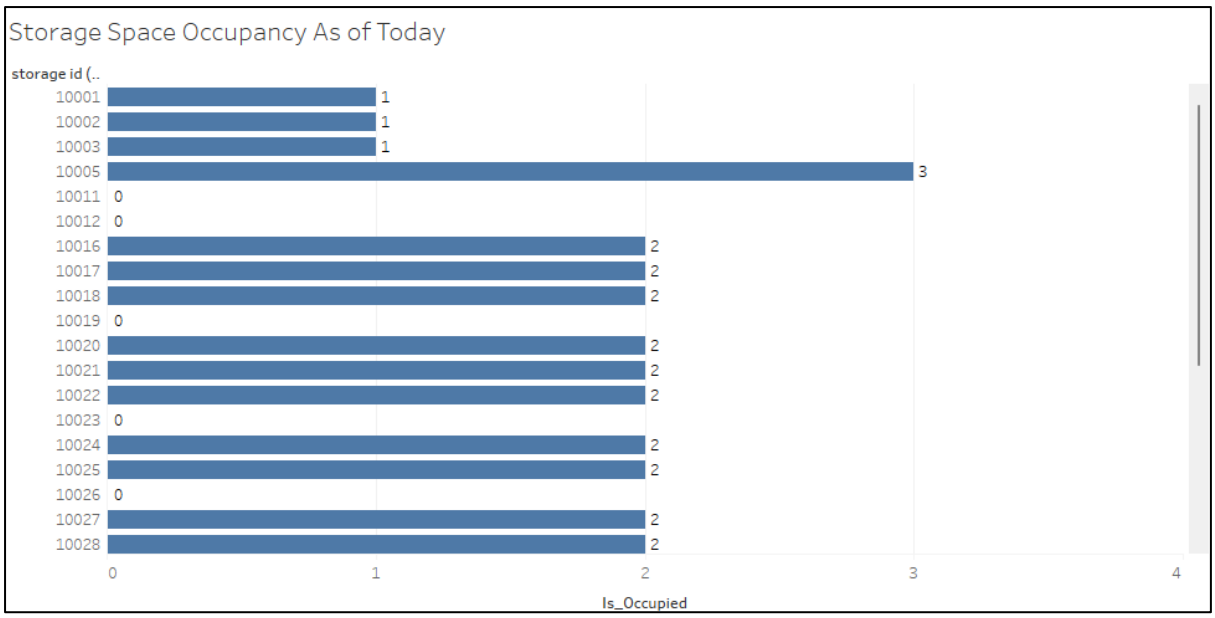

Figure 39 Storage Space Occupancy

6.5 Report 3: Top 3 Customers by Revenue

6.5.1 Purpose

Showcases the **highest-revenue** Renters over a specific period (monthly, quarterly, or all-time). This assists the Admin and finance teams in identifying loyal or profitable customers (Resnick and Zeckhauser, 2002).

6.5.2 Data Fields

- **Customer Name**: From Users table, joined with Rentals or Payments.
- **Total Paid**: Summation of booking fees, insurance surcharges, etc. across all rentals.

6.5.3 Insights

- Identifies repeat business and high-value customers.
- Encourages reward strategies (discount codes, loyalty programs).

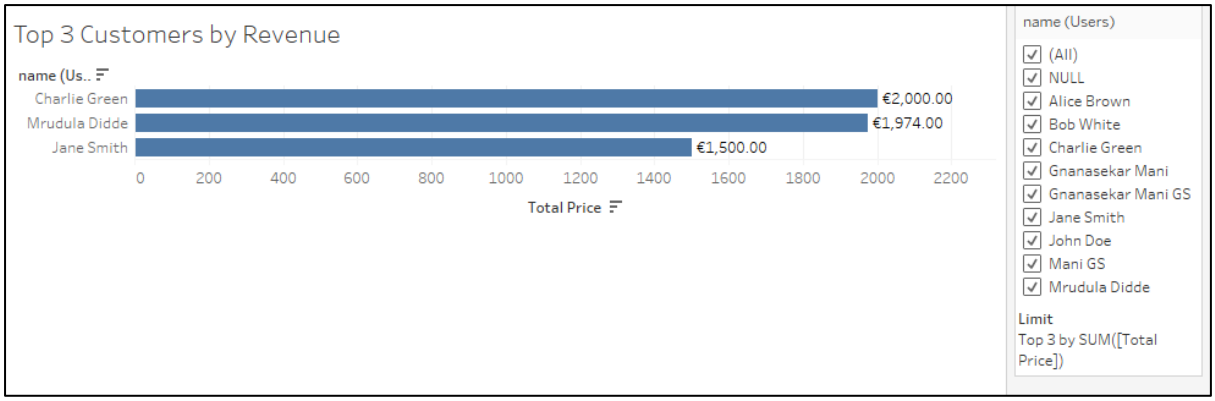

Figure 40 Top 3 Customer by Revenue

6.6 Report 4: Reviews & Ratings per Month

6.6.1 Purpose

Monitors how reviews and star ratings vary over time, reflecting customer satisfaction and platform trust levels (Pressman, 2010).

6.6.2 Data Fields

- **Review Date**: The date a Renter submitted a review.
- **Average Rating**: Mean rating for that month, from Reviews.rating.

6.6.3 Insights

- Trends in **user satisfaction** correlated with changes in features or policies (e.g., new insurance option, price updates).
- Admin can spot **negative rating spikes** needing immediate attention.

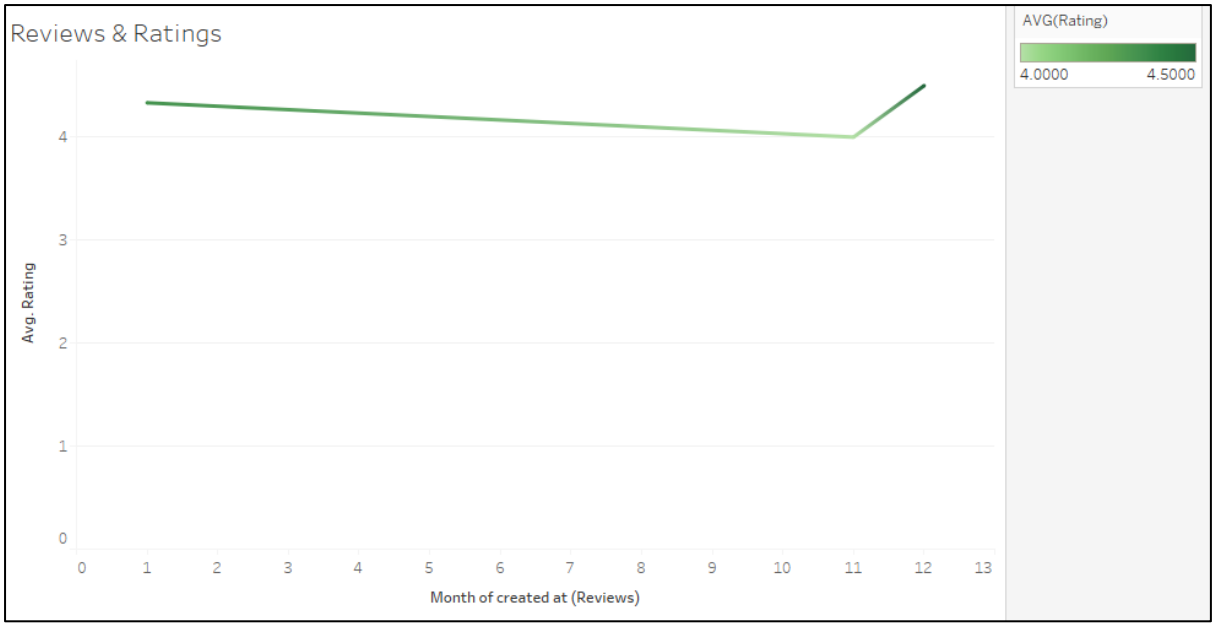

Figure 41 Reviews & Ratings

6.7 Benefits of Tableau Integration

1. **Real-Time Dashboards**: Automatic refresh ensures up-to-date analytics without manual CSV imports (Fowler, 2019).
2. **Interactive Filters**: Users can slice data by location, date range, storage type, or user roles.
3. **Collaboration**: Shared access for Admins, Hosts, or stakeholders through secure Tableau Server or Public (if needed).

6.8 Future Analytics and Enhancements

Although the current reports cover **monthly rentals**, **occupancy**, **top customers**, and **reviews**, additional analytics might include:

- **Geo-spatial Analysis**: Mapping storages by region, spotting coverage gaps (Kshetri, 2013).

- **Pricing Optimization**: Dynamic pricing models for peak vs. off-peak seasons.
- **ML-Based Recommendations**: Suggesting storages to Renters based on past preferences.

6.9 Summary

In Chapter 6, we demonstrated how HoldHive leverages Tableau to deliver real-time analytics directly from the SQL Server database. Four key reports—Monthly Rental Trend (by storage type), Storage Space Occupancy, Top 3 Customers by Revenue, and Reviews & Ratings—provide Hosts, Renters, and Admins with actionable insights. This approach fosters data-driven decisions, improving platform efficiency, user satisfaction, and revenue optimization (Crispin and Gregory, 2009; Pressman, 2010).

Subsequent expansions might integrate more advanced analytics (geospatial, machine learning, or in-depth financial reports) to further refine HoldHive as a robust, intelligence-driven peer-to-peer storage solution.

CHAPTER 7

DISCUSSIONS

7.1 Interpretation of Results

7.1.1 Synthesis of Testing Outcomes

The manual testing (Chapter 5) demonstrated that all core functionalities of HoldHive - including storage listings, bookings, reviews, and admin oversight - operated as intended, with no critical failures.

- **Listings & Booking**: Test results showed that Hosts can seamlessly create new listings, and Renters can successfully book spaces, reflecting a robust integration between the ReactJS front end, Lambda microservices, and the SQL database (Pressman, 2010).
- **Reviews System**: The system effectively allowed Renters to leave ratings and comments upon completing their rentals, updating average ratings in real time (Resnick and Zeckhauser, 2002).

Taken together, these findings suggest a high level of functionality for HoldHive at its current scale, with potential for further enhancements as usage grows.

7.1.2 Insights from Data Analytics

As per Chapter 6, the Tableau dashboards (Monthly Rental Trend by storage type, Storage Space Occupancy, Top 3 Customers by Revenue, Reviews & Ratings over time) demonstrate the viability of real-time data analytics integrated with the platform (Crispin and Gregory, 2009).

- **Monthly Rental Trend**: A stable or growing user interest in certain storage types indicates a seasonal or situational preference.
- **Occupancy**: Current usage patterns highlight potential underutilized spaces, enabling Hosts to make data-driven pricing or marketing decisions (Kshetri, 2013).

These analytics collectively point to a system that can adapt to user behavior and guide strategic improvements.

7.1.3 Alignment with Objectives

The results confirm that **HoldHive** meets its initial objectives:

1. **Seamless P2P Storage Listings**: The architecture (React on EC2, Lambdas, SQL DB) successfully supports listing creation and bookings.
2. **Reviews for Trust Building**: Implementation of a rating system fosters user trust - a key factor in peer-to-peer marketplaces (Resnick and Zeckhauser, 2002).
3. **Data-Driven Insights**: Tableau dashboards provide real-time metrics, aligning with the project's aim of supporting evidence-based decision-making (Highsmith, 2004).

7.2 Practical Implications

7.2.1 Peer-to-Peer Economy Enhancement

With the demonstrated success of a robust listing and review process, **HoldHive** can be viewed as a scalable model for other peer-to-peer platforms:

- **Trust Mechanisms**: The combination of reviews, star ratings, and user roles (Host, Renter, Admin) fosters credibility, potentially applicable to other sharing platforms (Cohn, 2005).
- **Agile Feature Rollouts**: Implementation sprints, continuous integration, and user feedback loops ensure a user-centric approach (Crispin and Gregory, 2009).

7.2.2 Data-Driven Business Decisions

Hosts can refine their rental strategies (pricing, availability) based on real-time occupancy and monthly trends, boosting platform retention and revenue. Admins can identify top customers and reward loyalty, ensuring a positive feedback cycle for user retention.

7.2.3 Feasibility for Wider Markets

Given the production-like environment (AWS resources, Lambda microservices, SQL-based analytics), HoldHive demonstrates a blueprint for scaling peer-to-peer services across geographically dispersed markets. This is particularly relevant in urban areas, where storage demand can fluctuate seasonally and property costs are high (Pressman, 2010).

7.3 Limitations of the Study

7.3.1 Limited User Testing Scale

While the functional and manual tests confirm robust performance at a modest scale, the platform has not yet undergone large-scale user trials. High concurrency demands, regional expansions, or spiking traffic have not been tested extensively on the current t2.micro or db.t3.micro setups (Amazon Web Services, 2020).

7.3.2 Narrow Demographic Sampling

Surveys (Chapter 3) and initial user feedback may skew toward a tech-savvy audience comfortable with **online marketplaces**. Thus, the results might not fully capture the perspective of users unfamiliar with P2P storage or advanced digital platforms (Kshetri, 2013).

7.3.3 Scope of Analytics

Though **four key reports** (Rental Trend, Occupancy, Top Customers, Reviews & Ratings) provide strong insights, many other metrics (e.g., user churn rate, cost optimization, environmental impact) remain unexplored. Future expansions of the analytics module could yield deeper revelations about user behavior (Fowler, 2019).

7.3.4 Security and Compliance Factors

While the system uses **Firebase authentication** and enforces basic data protection measures, a thorough penetration test or formal security audit was out of scope. Additionally, potential GDPR or other regional compliance requirements may impose complex data-handling rules that have not been fully explored (Kshetri, 2013).

7.4 Concluding Remarks

Overall, the results and analytics confirm HoldHive's core viability as a peer-to-peer storage platform, providing a reliable listing/booking system, a trust-enabling reviews mechanism, and meaningful real-time analytics. Despite certain limitations in scale, demographic coverage, and advanced security checks, the project stands as a promising demonstration of how agile development, robust cloud architecture, and data-driven insights can converge to create a practical and scalable solution in the sharing economy domain (Resnick and Zeckhauser, 2002; Pressman, 2010).

CHAPTER 8

CONCLUSION

8.1 Summary of Key Achievements

Over the course, **HoldHive** has evolved from a conceptual peer-to-peer (P2P) storage idea into a **working platform** with:

1. **Robust Architecture**
    - **ReactJS** front end on **AWS EC2**, supported by **pm2** for continuous uptime.
    - **AWS Lambda** microservices handling core business logic (listing creation, booking flows, reviews).
    - **Microsoft SQL Server (RDS)** storing all user, rental, and review data.
    - **Firebase** authentication ensuring secure user logins
2. **Fully Functional Core Features**
    - **Listing Creation**: Hosts can select storage details (size, cost, location) and storage type based upon that images are retrieved from S3 for the listing.
    - **Booking**: Renters can book available spaces and finalize transactions.
    - **Review & Rating System**: Encourages trust by allowing Renters to rate their experiences and Hosts to improve services (Resnick and Zeckhauser, 2002).
3. **Data Analytics & Insights**
    - **Monthly Rental Trend**, **Storage Occupancy**, **Top 3 Customers**, and **Reviews & Ratings** are visualized in **Tableau**, offering real-time, data-driven decisions for Hosts and Admins (Pressman, 2010).
4. **Successful Testing**
    - A series of **manual test cases** (Chapter 5) confirmed that all major workflows operate correctly, matching expected outcomes.

These accomplishments demonstrate the technical feasibility and user-centric focus of HoldHive as a P2P storage marketplace.

8.2 Future Work and Scalability

8.2.1 Large-Scale Performance Testing

While the current infrastructure (t2.micro EC2, db.t3.micro) is sufficient for pilot usage, future endeavours include:

- **Load Testing**: Simulate high concurrency to see if Lambda concurrency or DB capacities need scaling (Amazon Web Services, 2020).
- **Geographic Expansion**: Support multiple regions, adjusting VPC configurations and ensuring minimal latency for cross-regional data access.

8.2.2 Enhanced Security and Compliance

Additional efforts could involve:

- **Formal Security Audits**: A penetration test to identify vulnerabilities in Lambda endpoints, S3 bucket permissions, and user data encryption (Kshetri, 2013).
- **GDPR** and **CCPA** Reviews: Detailed compliance checks for data-handling processes, ensuring personal information is stored lawfully.

8.2.3 Advanced Analytics and Machine Learning

Building on the Tableau dashboards, next steps might include:

- **Predictive Analytics**: Seasonal forecasting of booking demand, dynamic pricing suggestions for Hosts (Cohn, 2005).
- **Recommendation Engine**: Matching Renters to optimal storage listings based on proximity, cost, or climate-control needs (Highsmith, 2004).

8.2.4 Mobile Application Development

A dedicated **HoldHive** mobile app could enhance user convenience:

- **Push Notifications**: Alerts for booking confirmations, listing inquiries, or recommended spaces.
- **Geo-Location**: Real-time distance calculations to identify nearest available storages.

CHAPTER 9

APPENDIX

9.1 Survey Questions

Complete list of Survey Questions is as follows:

1. What is your age group?
2. What is your current employment status?
3. What type of area do you live in?
4. How often do you use or need storage solutions (e.g., self-storage units, renting storage)?
5. Would you consider using a platform like ***Holdhive*** to rent storage space?
6. If you own unused storage space, would you consider listing it for rent on ***Holdhive***?
7. What type of storage space are you most interested in (or have available)?
8. How important are the following features in a storage platform? [Security and surveillance]
9. How important are the following features in a storage platform? [Insurance options]
10. How important are the following features in a storage platform? [Clear pricing]
11. How important are the following features in a storage platform? [Easy-to-use platform]
12. How important are the following features in a storage platform? [User reviews and ratings]
13. How would you describe yourself?
14. What type of storage duration best suits your needs?
15. Which of these additional features would you prefer?
16. Would you recommend ***Holdhive*** to others once it's developed?
17. Would you trust a peer-to-peer storage platform for storing your belongings?
18. Would user reviews and ratings improve your trust in the platform?

19. How important is it for the platform to provide insurance options for stored items?
20. Would you feel more secure if the platform verified the identity of all users?
21. Do you prioritize local storage spaces, or would you prefer options further away but cheaper?
22. Would features like 24/7 access influence your decision to rent a space?
23. Do you expect the platform to assist with moving or transportation services?
24. Do you think a platform like ***Holdhive*** encourages better utilization of unused resources?
25. Would you be more likely to use the platform if it highlighted its environmental benefits?
26. Would knowing you are helping others (e.g., students, temporary movers) influence your decision to list or rent storage?
27. How much would you be willing to pay monthly for storage space?
28. How important is price flexibility (e.g., negotiating with owners)?
29. Would you be more likely to use the platform if discounts or promotions were available?
30. Would you use the platform if the prices were competitive with traditional storage units?
31. Would you be more likely to list your space if the platform offered incentives (e.g., bonuses for first-time listings)?
32. Would dynamic pricing based on location and demand appeal to you?
33. Would you be more willing to rent space on ***Holdhive*** if you received a discount in exchange for allowing non-sensitive data (e.g., storage preferences) to be stored for platform improvement?
34. Do you feel an app providing this service would meet your storage needs?

35. Would you recommend the idea of ***Holdhive*** to friends or colleagues?
36. Do you believe there's a demand for such a platform in your community?
37. If you'd like to be contacted for further feedback or testing, please provide your email address.

9.2 Survey Responses

Survey Responses is attached separately as the PDF along with the report.

Survey Link: https://forms.gle/nxd93j8K4SoBnbYR7

9.2 Figures / Screenshots

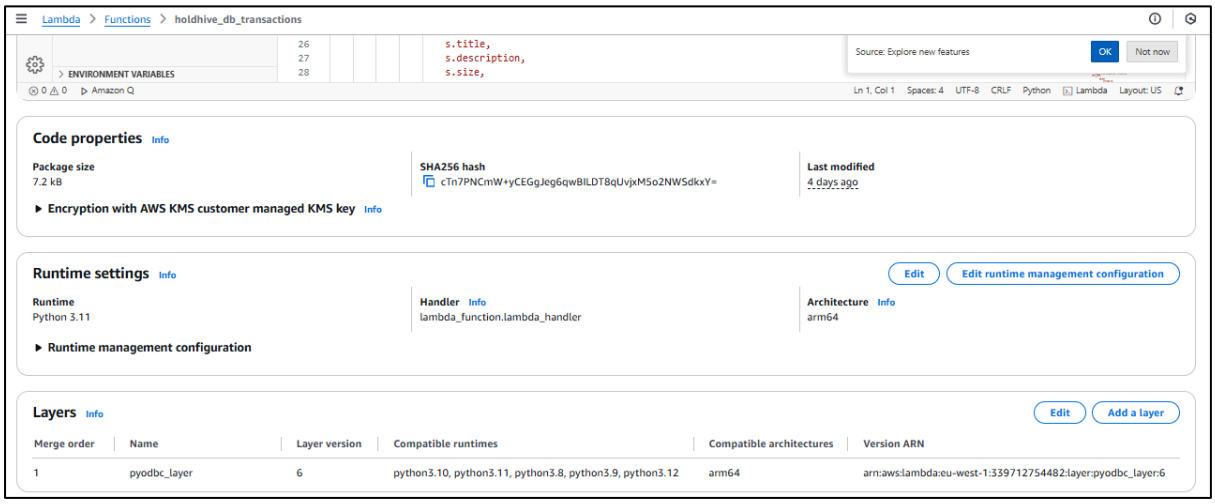

Figure 42 Pyodbc Layer Addition in DB_transactions Lambda

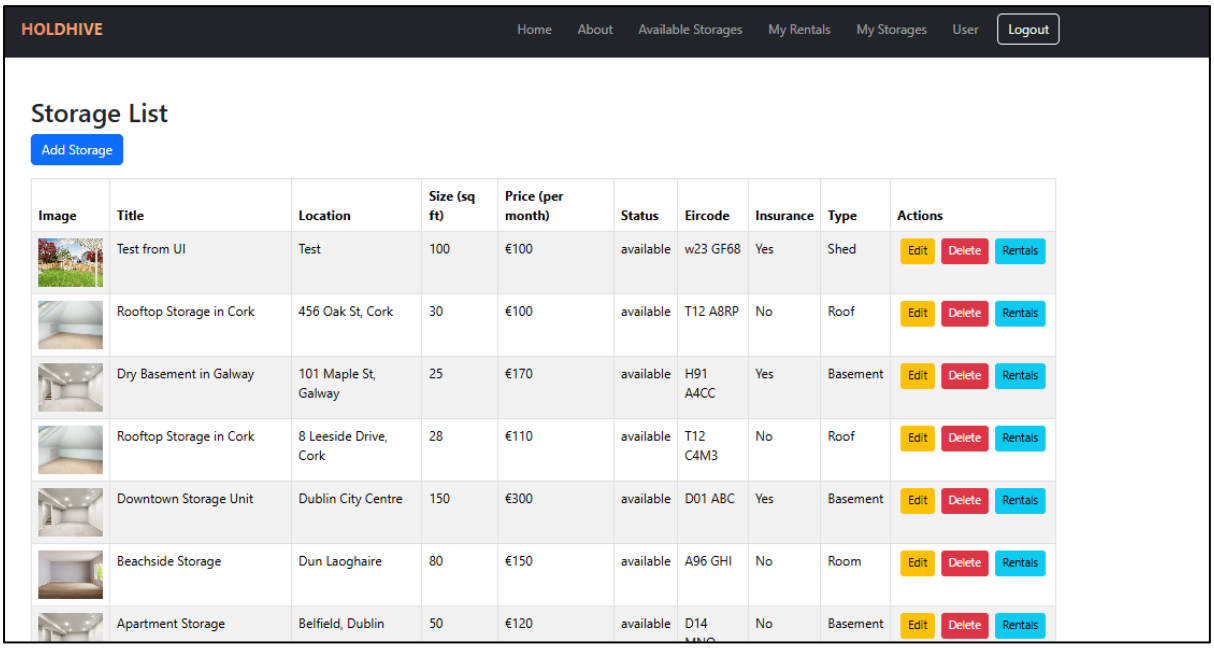

Figure 43 Host-Storage Listings Page-1

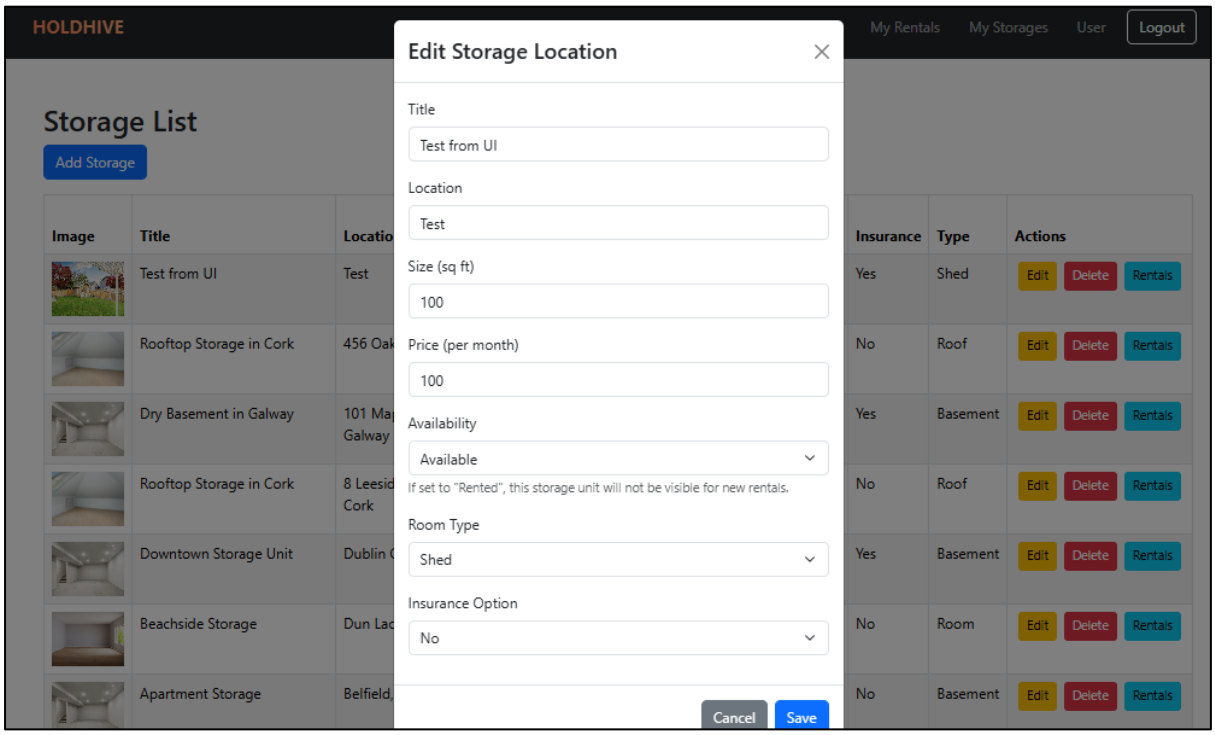

Figure 44 Holdhive Storage Location Update

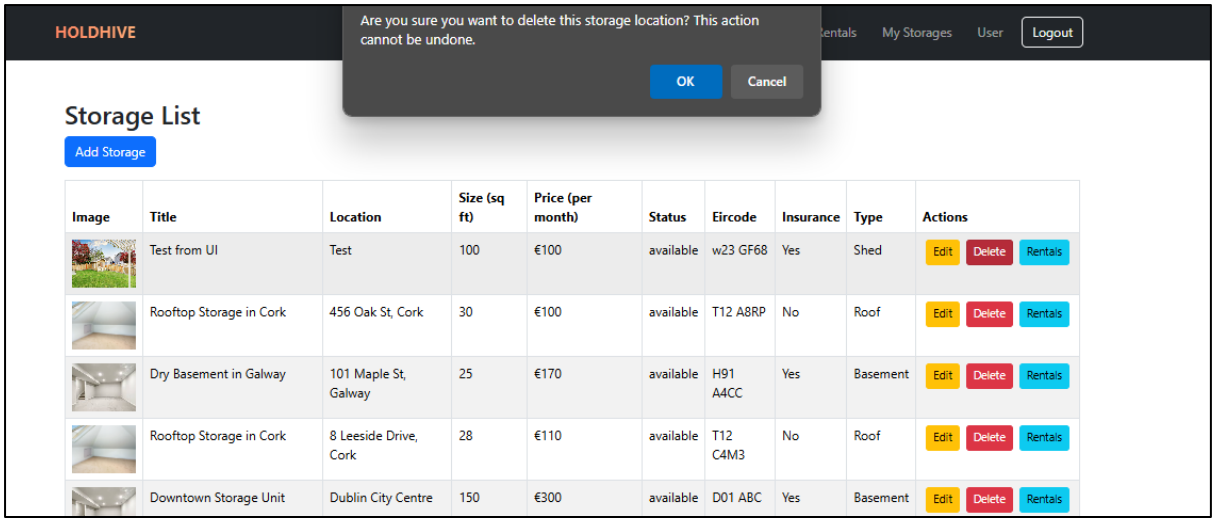

Figure 45 ***Holdhive*** *Storage Location Deletion*

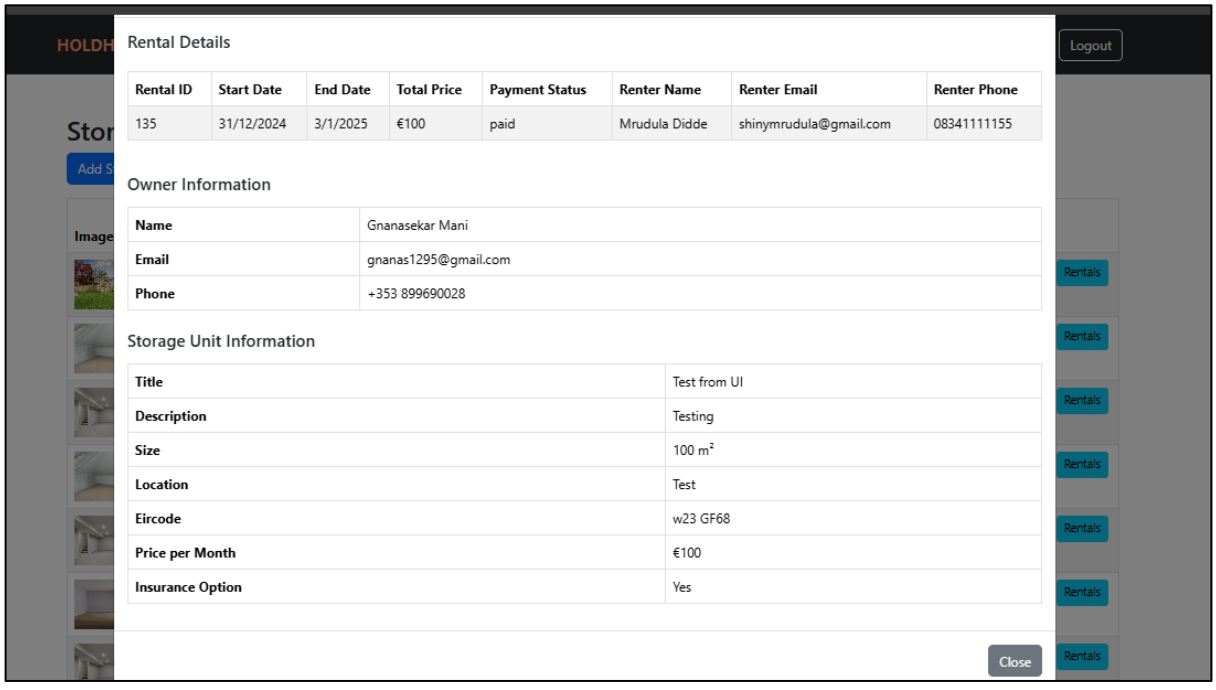

Figure 46 Host- Upcoming Rentals for the Storage

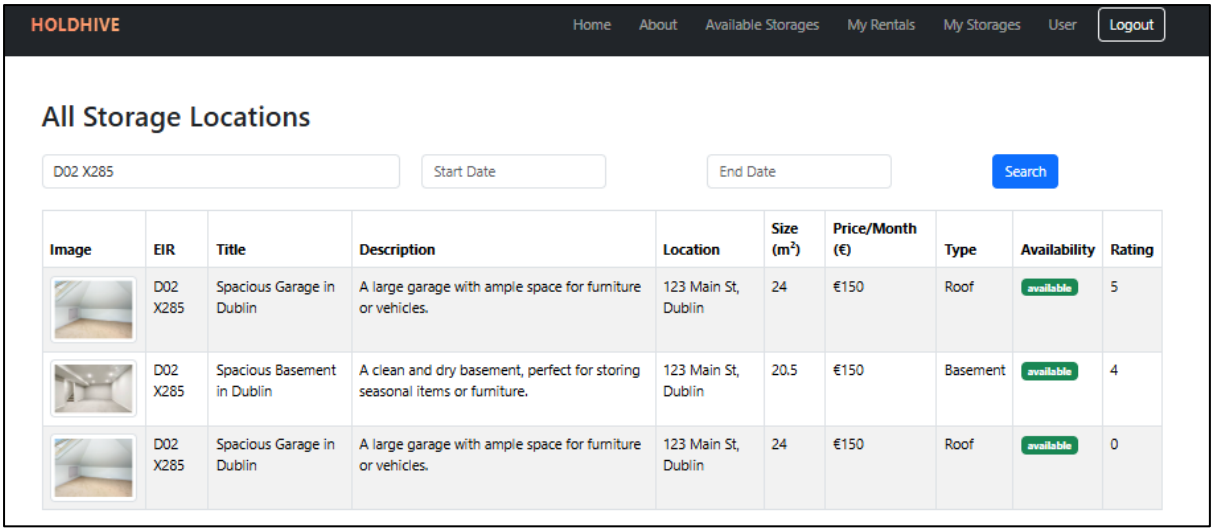

Figure 47 Available Storage Location – Eircode Search Query

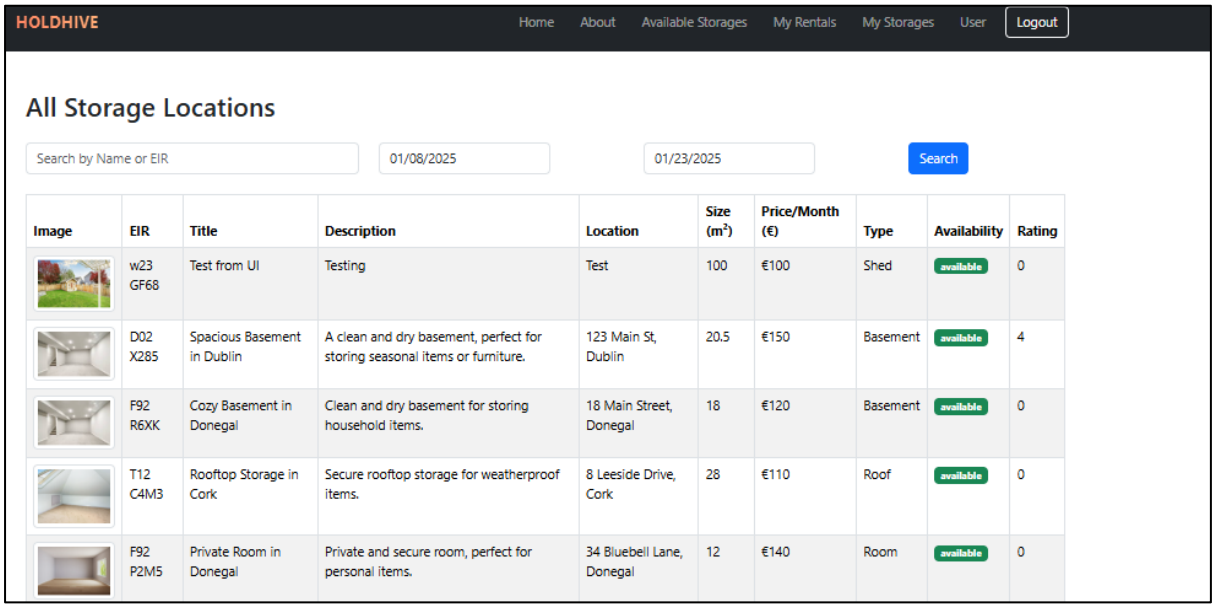

Figure 48 Available Storage Location – Date Filter

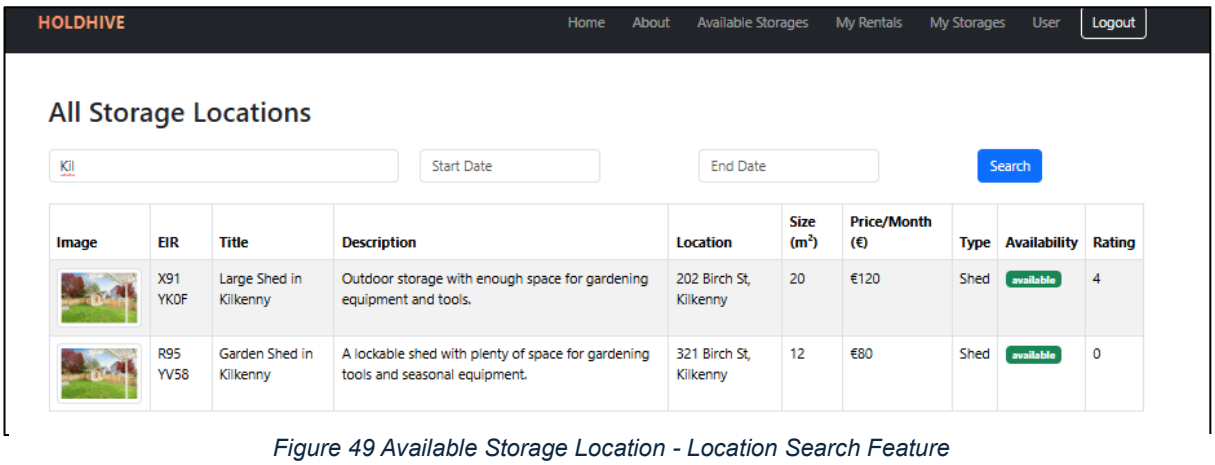

Figure 49 Available Storage Location - Location Search Feature

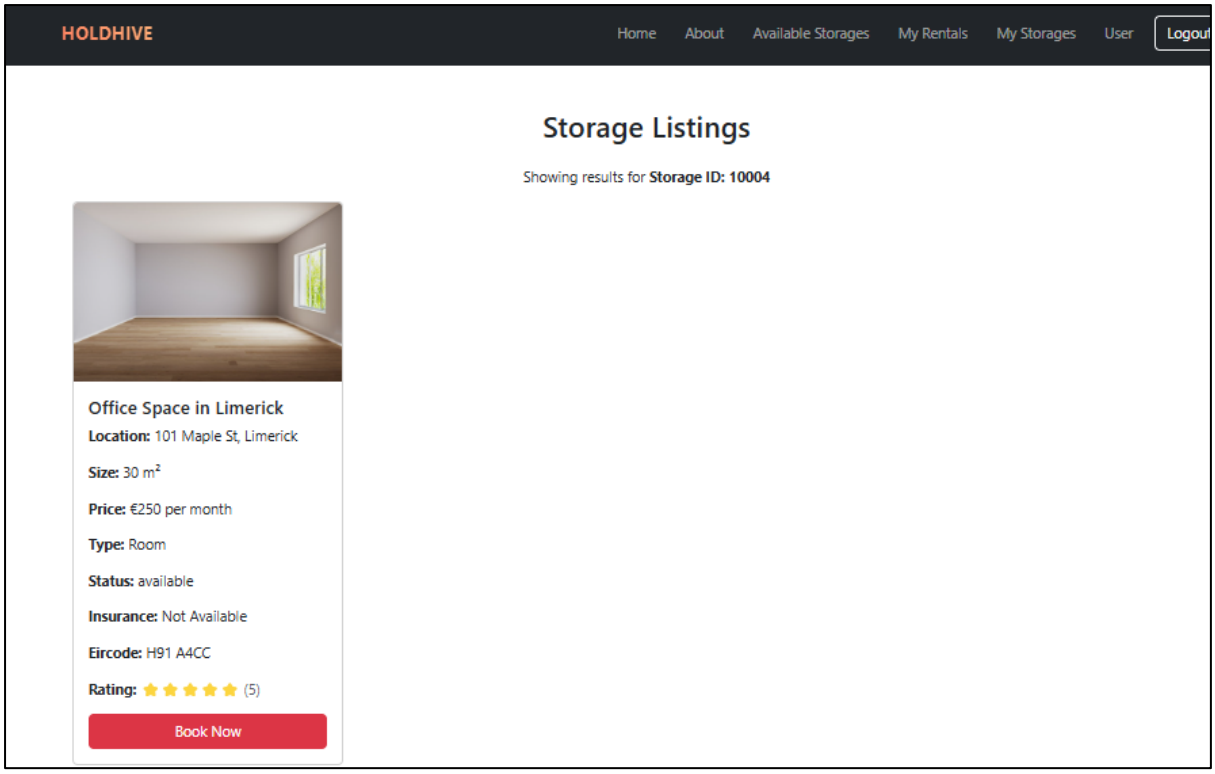

Figure 50 Booking Flow – Page 1

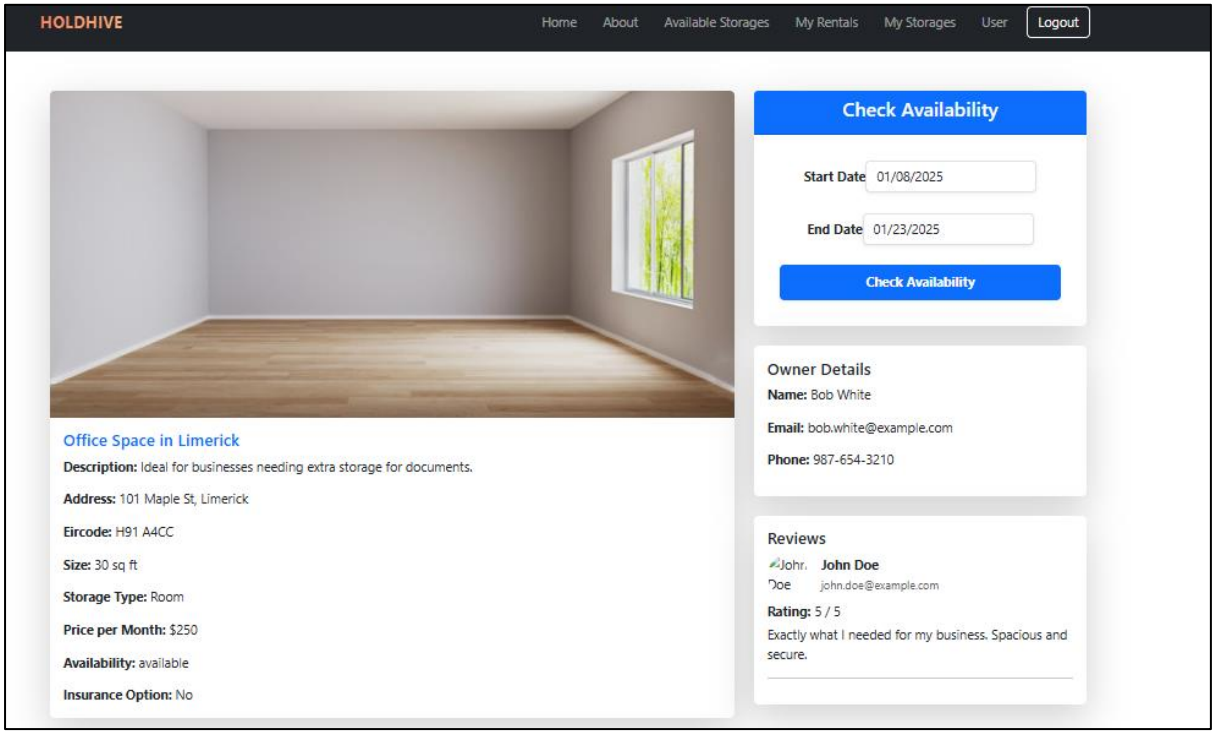

Figure 51 Booking Flow - Page 2

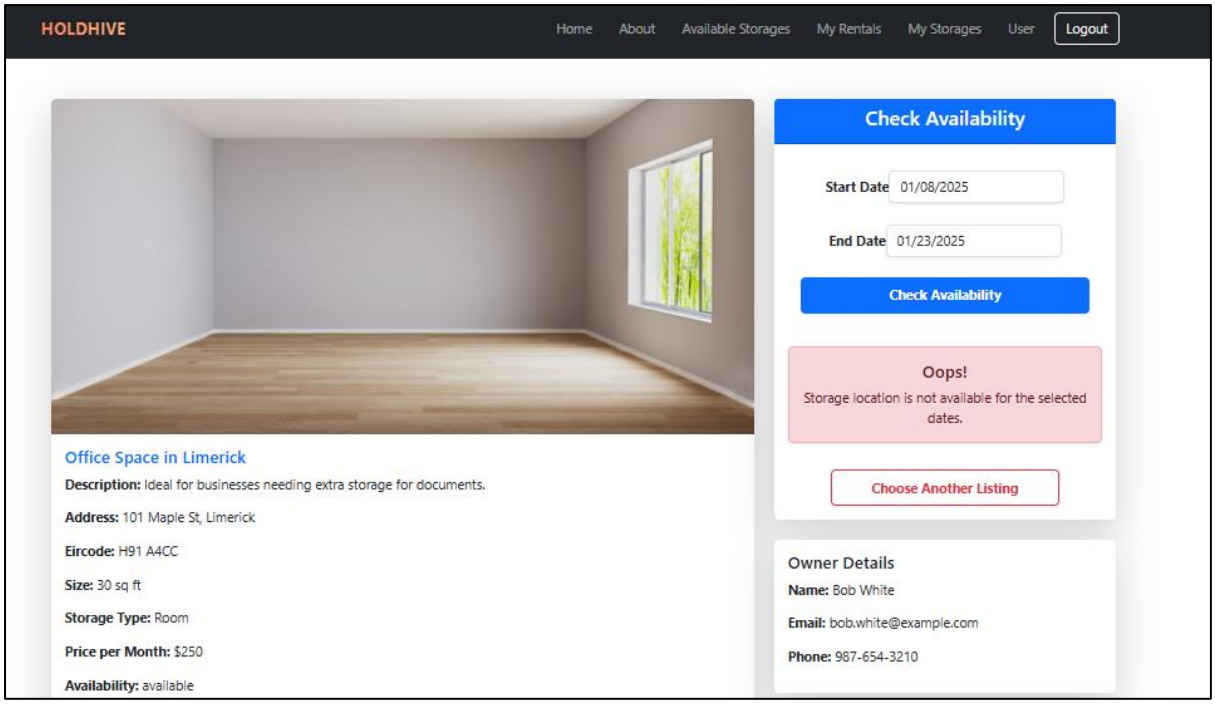

Figure 52 Booking Flow - Page 3(If location is unavailable)

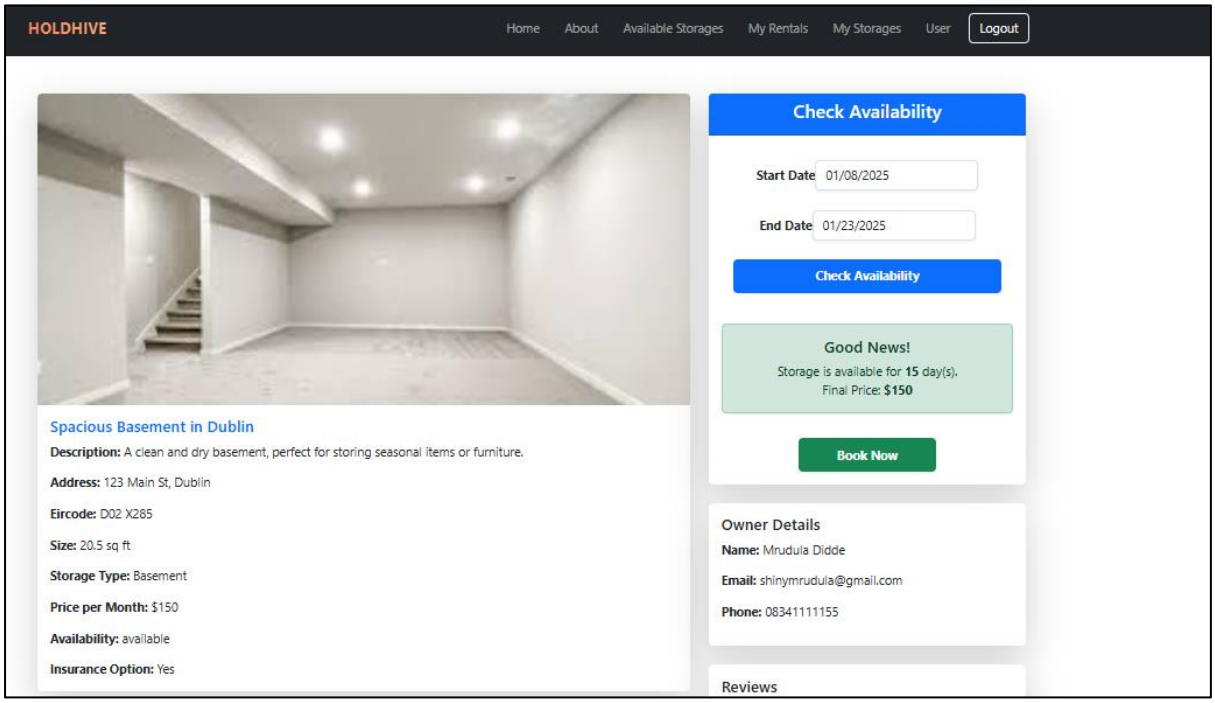

Figure 53 Booking Flow - Page 4 (if location is available)

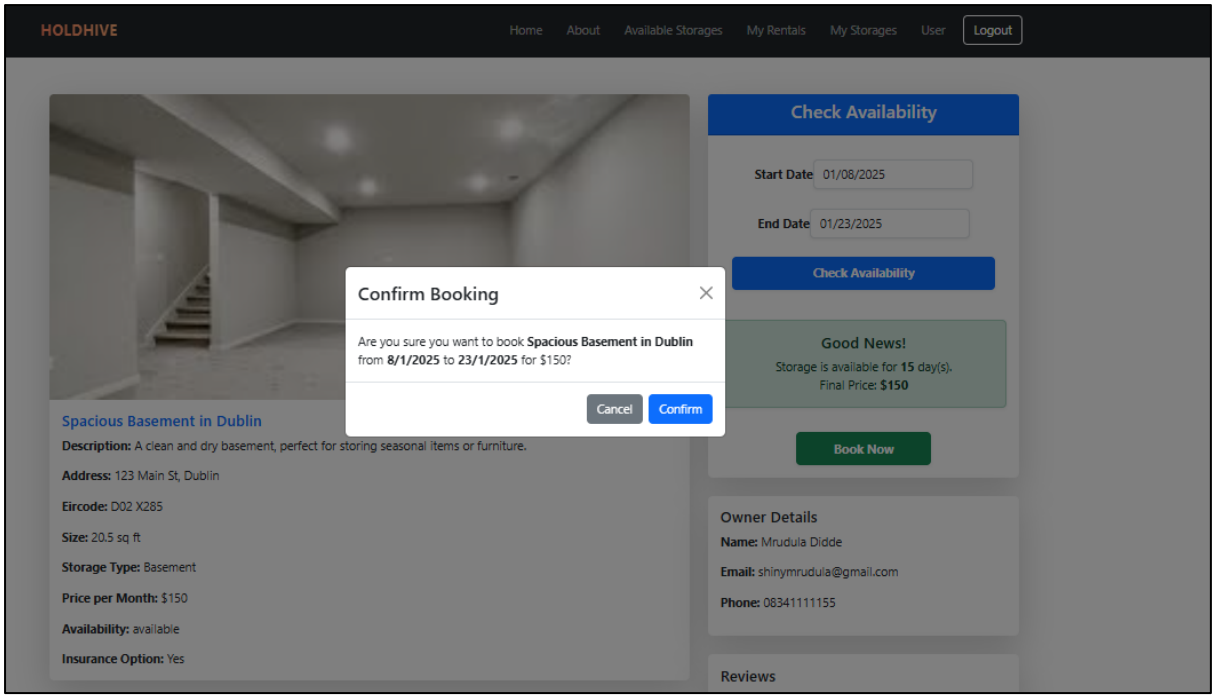

Figure 54 Booking Flow - Page 5

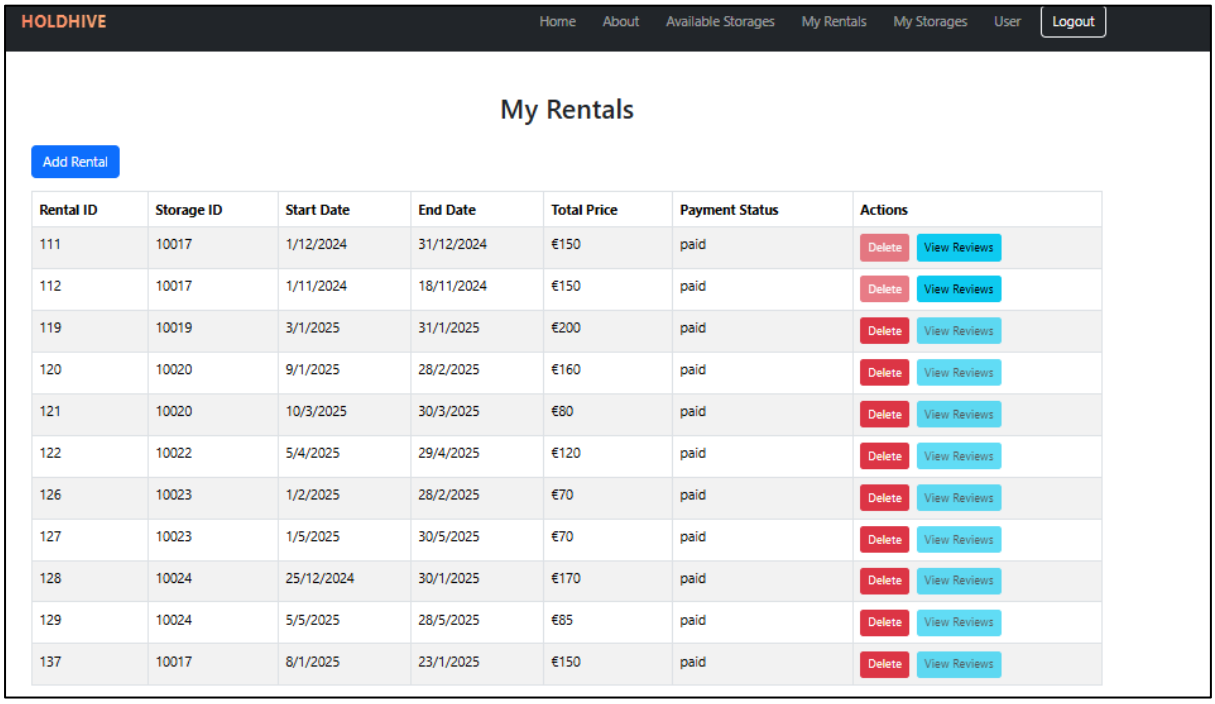

Figure 55 Booking Flow Confirmation in Rentals - Part 7

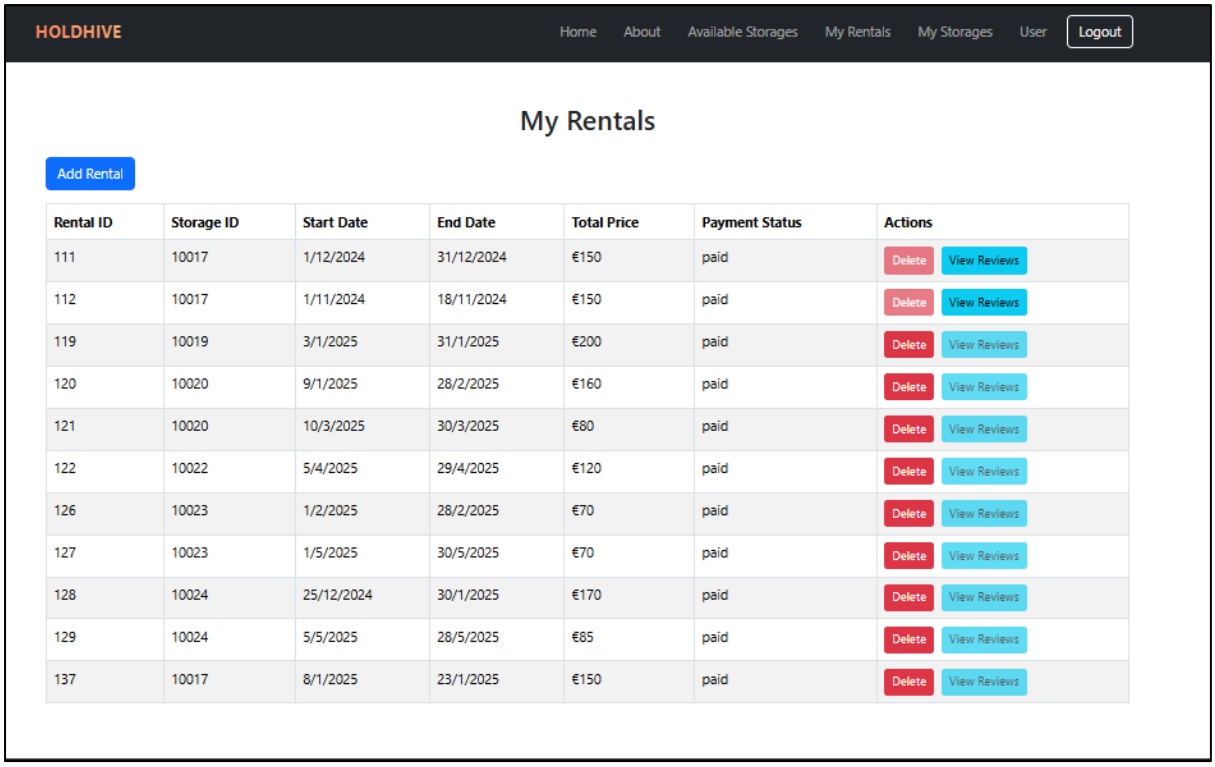

Figure 56 Rentals Details - Page 2

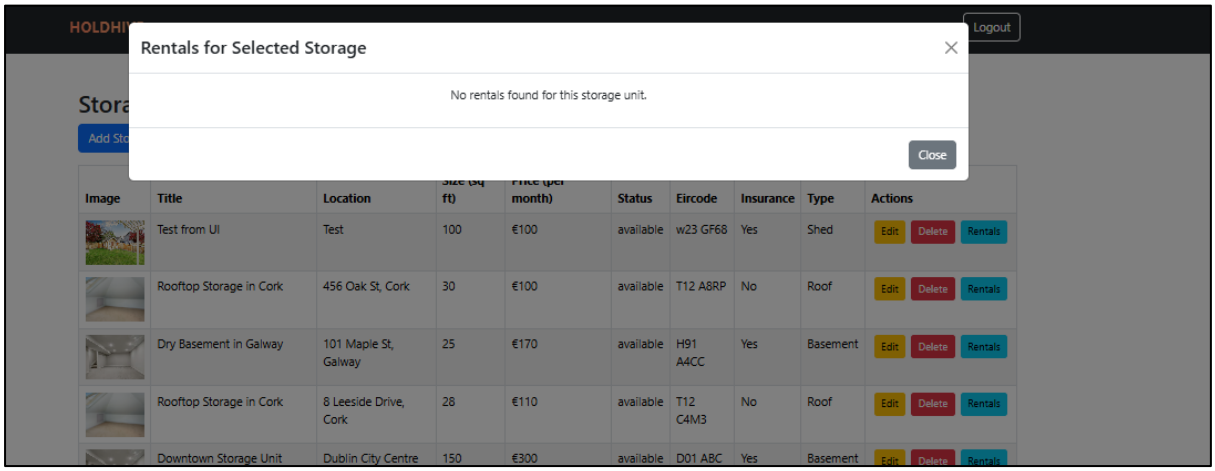

Figure 57 Rentals by Storage - Page 2 if no rentals

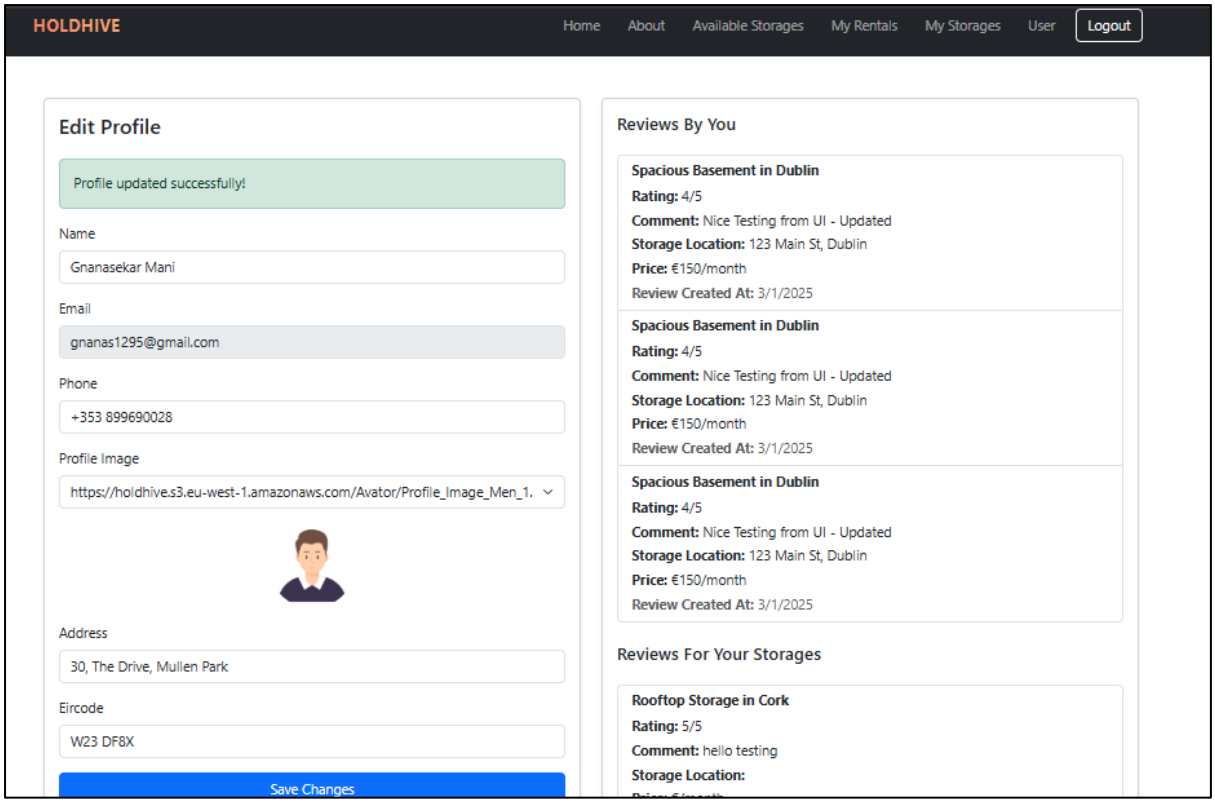

Figure 58 Profile Update Page

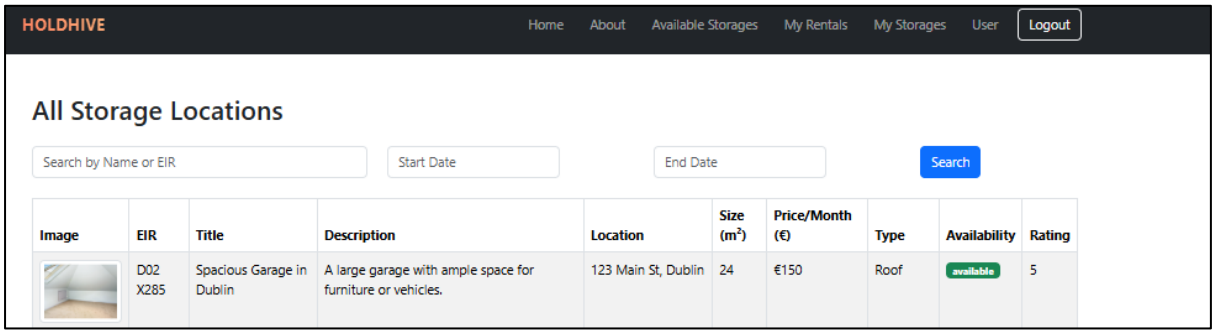

Figure 59 Reviews View Page 1

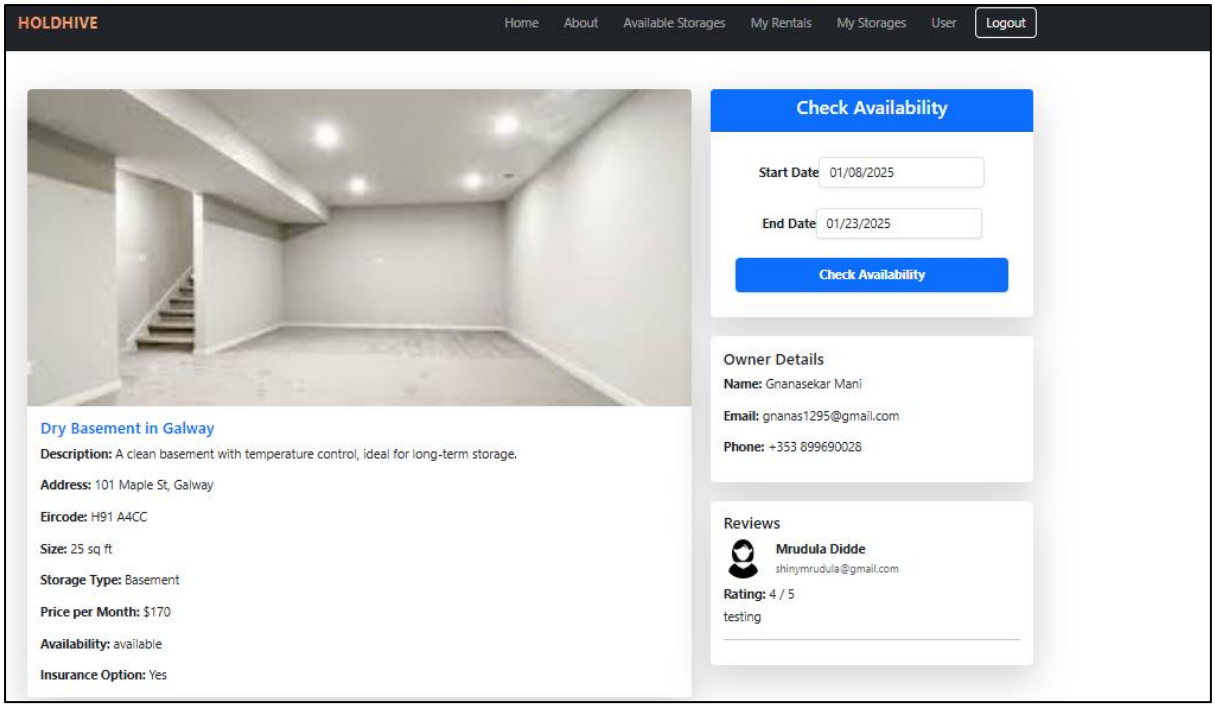

Figure 60 Reviews View Page 2

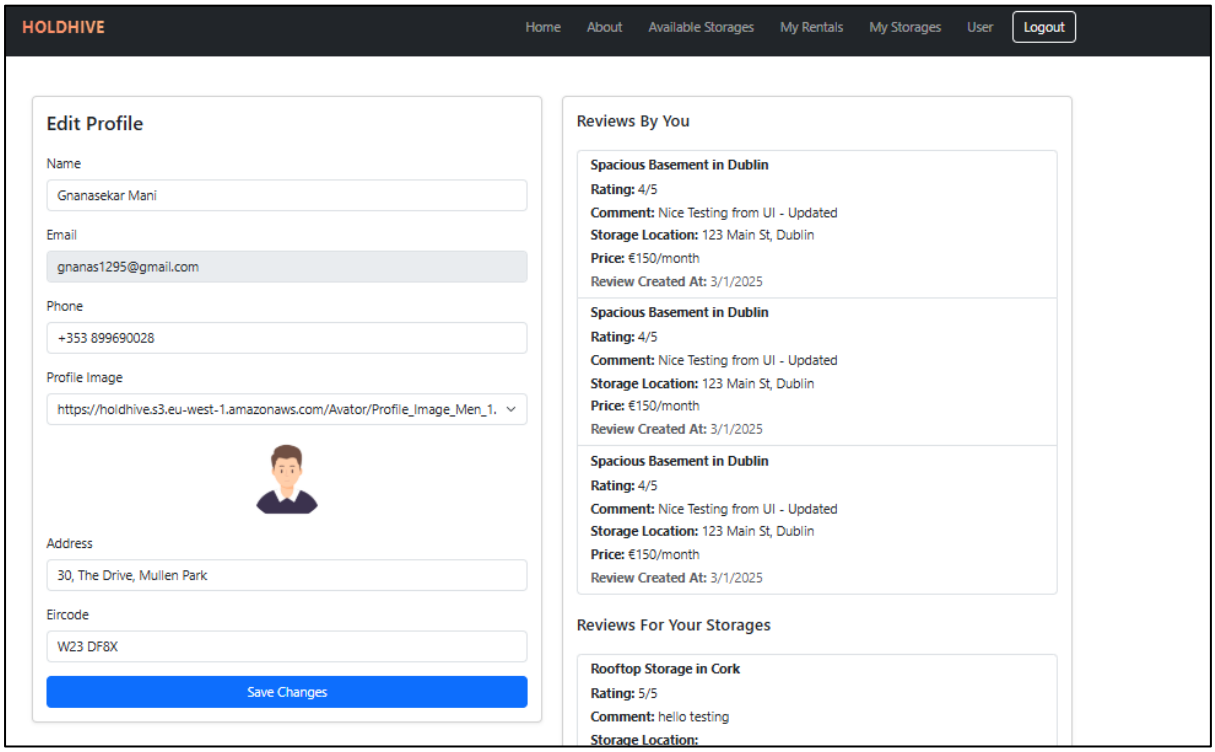

Figure 61 Reviews View Page 3

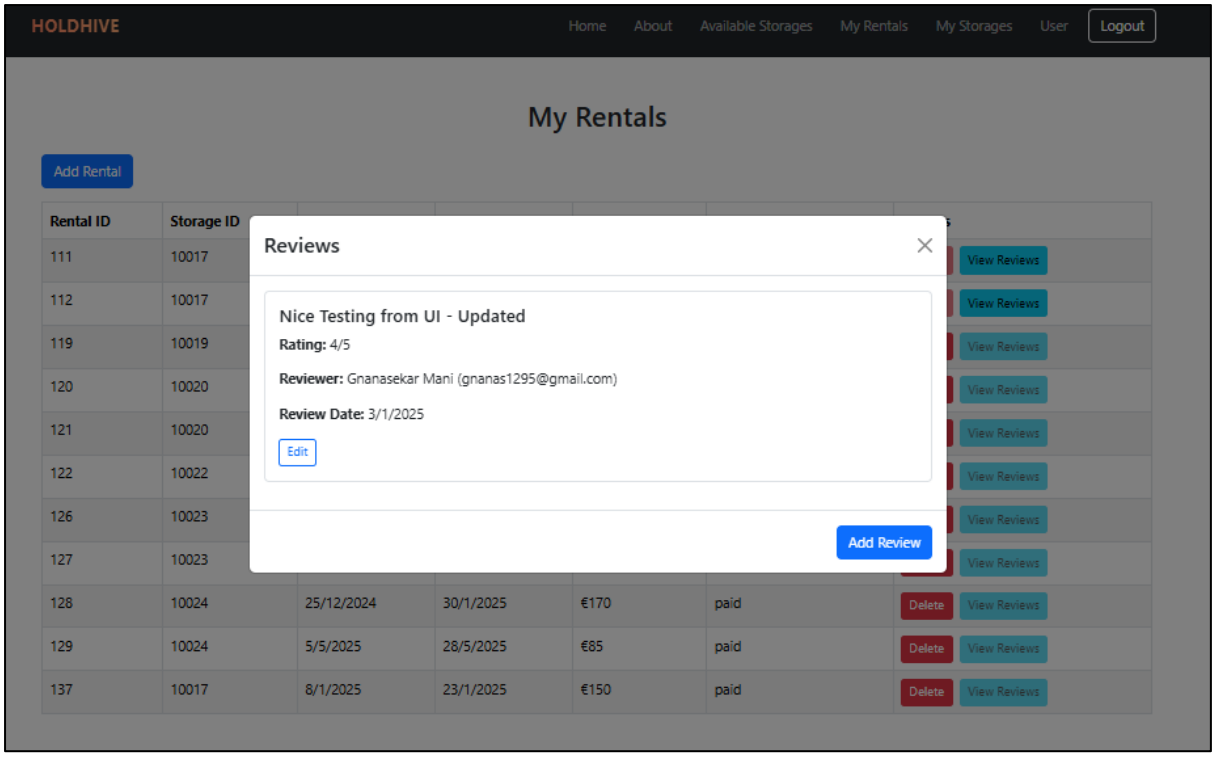

Figure 62 Reviews Addition Page post Rental

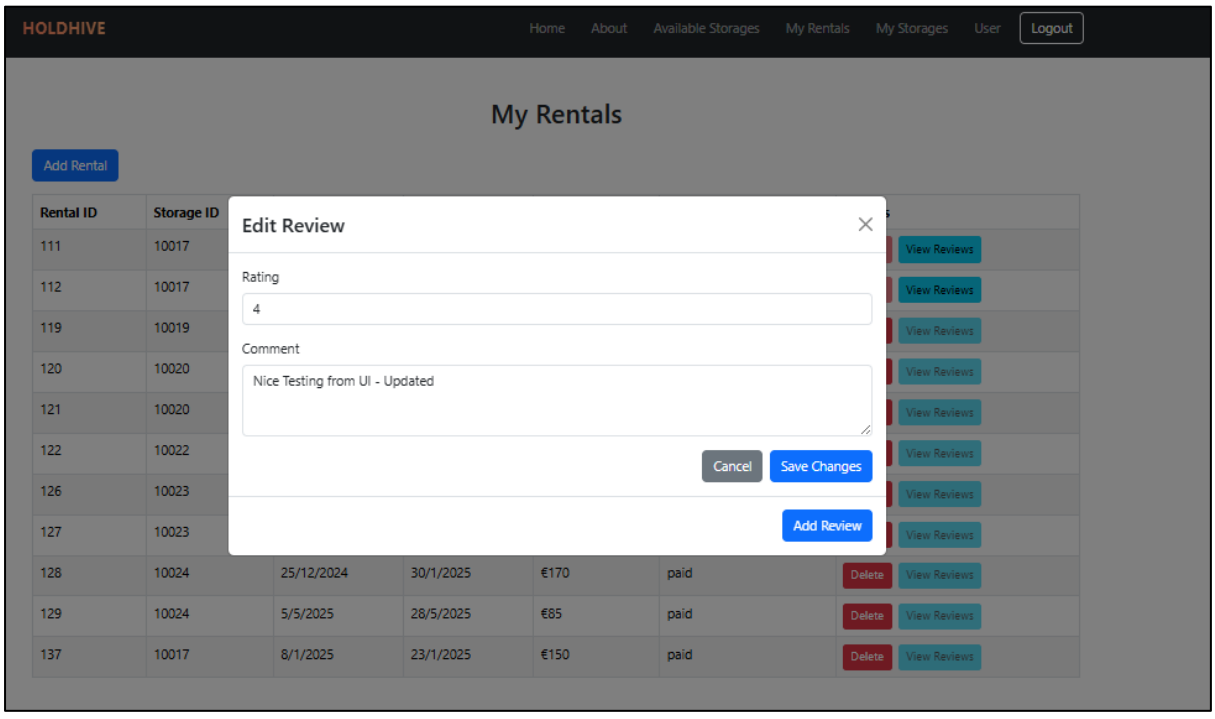

Figure 63 Reviews Updation page post rental

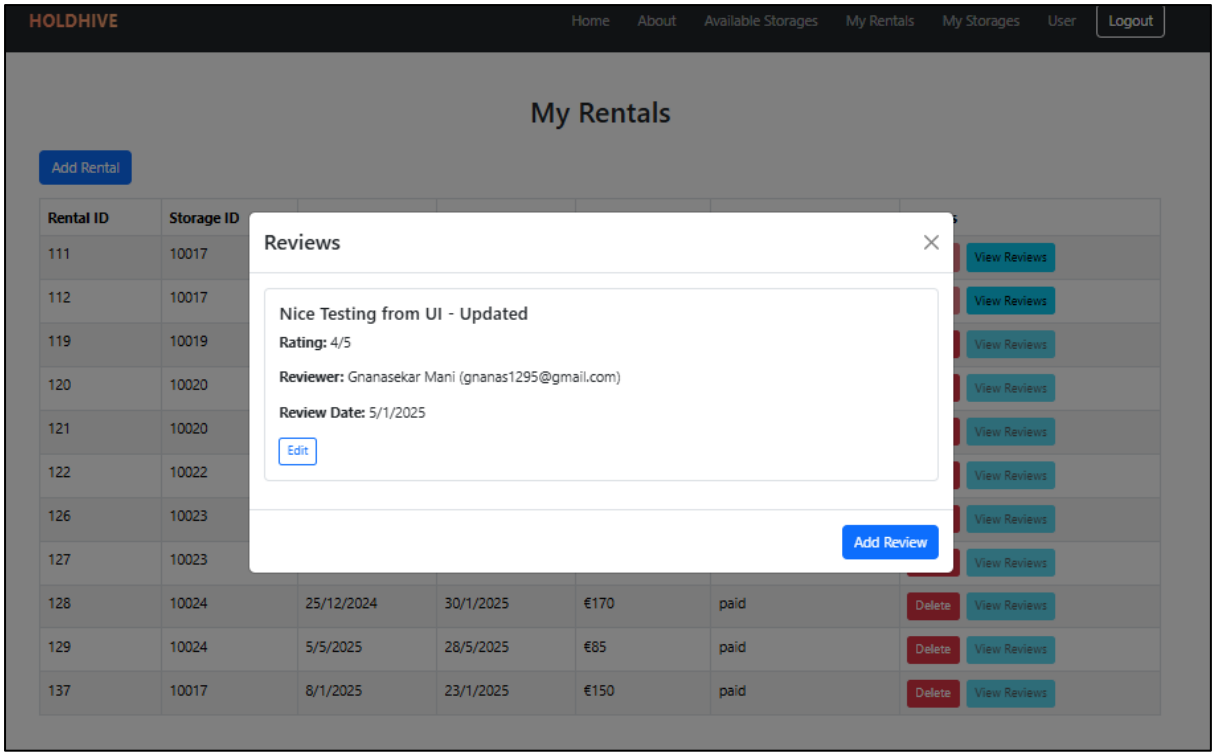

Figure 64 Reviews Page Updation Successful

9.4 Code Base

Note: Only selected portions of the front-end and back-end code appear in this appendix to illustrate key functionalities. The complete code base, including additional features, full API documentation, Tableau reports, and relevant screenshots, is available in the GitHub repository. This approach keeps the main report concise while still offering comprehensive access to the broader code and documentation resources.

9.4.1 Frontend

Home.js:

```
import React, { useState, useEffect } from 'react';

import { Button, Container, Form, Accordion, Row, Col } from 'react-bootstrap';

import "react-datepicker/dist/react-datepicker.css";

import { useNavigate } from 'react-router-dom';

import '../styles/Home.css';
```

```
const Home = () => {
  const [search, setSearch] = useState('');
  const [filteredLocations, setFilteredLocations] = useState([]);
  const [locations, setLocations] = useState([]); // Locations fetched from API
  const [selectedLocation, setSelectedLocation] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState('');
  const navigate = useNavigate();
  const [data, setData] = useState([]);

useEffect(() => {
  const fetchLocations = async () => {
    const apiUrl =
      'https://0ixtfa5608.execute-api.eu-west-1.amazonaws.com/prod/storage-location/list-storage-location';

    try {
      const response = await fetch(apiUrl);
      if (!response.ok) {
```

```
        throw new Error('Failed to fetch locations');
      }
      const data = await response.json();

      setData(data.data);

      const locationTitles = data.data.map((item) => ({
        id: item.storage_id,
        title: item.title,
        location: item.location,
        eircode: item.eircode, // Add Eircode
      }));

      setLocations(locationTitles);
      setLoading(false);
    } catch (err) {
      setError(err.message);
      setLoading(false);
    }
  };
```

```
  fetchLocations();
}, []);


  // Handle search input
  const handleSearch = (e) => {
    const query = e.target.value.toLowerCase();
    setSearch(query);
    if (query.length > 0) {
      const results = locations.filter((loc) =>
        loc.title.toLowerCase().includes(query) ||
        loc.location.toLowerCase().includes(query) ||
        (loc.eircode && loc.eircode.toLowerCase().includes(query)) // Include Eircode in the
search
      );
      setFilteredLocations(results);
    } else {
      setFilteredLocations([]);
    }
  };
```

```
// Handle search selection
const handleLocationSelect = (location) => {
  setSelectedLocation(location); // Store the full location object
  setSearch(location.title);
  setFilteredLocations([]);
};

// Handle form submission
const handleSearchSubmit = () => {
  if (selectedLocation) {
    navigate(`/storage/${selectedLocation.id}`, {
      state: { storageId: selectedLocation.id, location: selectedLocation.location },
    });
  } else {
    alert('Please select a location.');
  }
};

const handleCardClick = (location) => {
```

```
    navigate(`/storage/${location.storage_id}`, {
      state: {
        storageId: location.storage_id,
        location: location.location,
      },
    });
  };

  return (
    <div>
      <div className="hero-section bg-light py-5">
        <Container>
          <h1 className="text-center display-4 mb-4">Find a storage facility near you</h1>
          <p className="text-center mb-4">Cheap. Fast. And over 100,000 users.</p>

          <Form className="position-relative d-flex justify-content-center align-items-center
mb-4">
            <Form.Control
              type="text"
              placeholder="Enter a location or eircode"
```

```
          value={search}
          onChange={handleSearch}
          className="w-50 me-2"
          disabled={loading}
        />
        <Button variant="danger" onClick={handleSearchSubmit} disabled={loading}>
          {loading ? 'Loading...' : 'Search'}
        </Button>

        {error && <p className="text-danger text-center mt-3">{error}</p>}

        {filteredLocations.length > 0 && (
          <ul className="list-group position-absolute w-50" style={{ top: '100%', zIndex:
1000 }}>
            {filteredLocations.map((location) => (
              <li
                key={location.id}
                className="list-group-item"
                onClick={() => handleLocationSelect(location)}
                style={{ cursor: 'pointer' }}
```

```
            >
              {location.title} - {location.location}
            </li>
          ))}
        </ul>
      )}
    </Form>
  </Container>
</div>

{/* Features Section */}
<Container className="my-5">
  <h2 className="text-center mb-4">Why Choose Us?</h2>
  <Row className="text-center">
    <Col md={4}>
      <i className="bi bi-shield-lock-fill display-4 text-primary"></i>
      <h4 className="mt-3">Secure Storage</h4>
      <p>Top-notch security to keep your belongings safe at all times.</p>
    </Col>
    <Col md={4}>
```

```
      <i className="bi bi-clock-fill display-4 text-primary"></i>
      <h4 className="mt-3">Flexible Timing</h4>
      <p>Access your storage unit whenever it’s convenient for you.</p>
    </Col>
    <Col md={4}>
      <i className="bi bi-currency-dollar display-4 text-primary"></i>
      <h4 className="mt-3">Affordable Pricing</h4>
      <p>Transparent and competitive pricing with no hidden fees.</p>
    </Col>
  </Row>
</Container>


{/* Storage Listings Marquee */}
<div className="scrollable-container bg-light py-4">
  <Container>
    <h3 className="text-center mb-4">Available Storage Spaces</h3>
    <div className="scrollable-content">
      {[...data, ...data].map((location, index) => (
        <div
          key={`${location.storage_id}-${index}`}
```

```
            className="scrollable-card"
            onClick={() => handleCardClick(location)}
            style={{ cursor: 'pointer' }} // Add a pointer cursor
          >
            <img
              src={location.images_url || 'https://via.placeholder.com/300'}
              alt={location.title}
              className="scrollable-card-img"
            />
            <div className="scrollable-card-body">
              <h5 className="scrollable-card-title">{location.title}</h5>
              <p><strong>Location:</strong> {location.location}</p>
              <p><strong>Price:</strong> €{location.price_per_month} per month</p>
            </div>
          </div>
        ))}
      </div>
    </Container>
  </div>
```

```
{/* FAQ Section */}
<Container className="my-5">
  <h2 className="text-center mb-4">Frequently Asked Questions</h2>
  <Accordion>
    <Accordion.Item eventKey="0">
      <Accordion.Header>How secure are the storage units?</Accordion.Header>
      <Accordion.Body>
        Our storage units are equipped with state-of-the-art security systems, including 24/7 monitoring and access control.
      </Accordion.Body>
    </Accordion.Item>
    <Accordion.Item eventKey="1">
      <Accordion.Header>Can I access my storage unit anytime?</Accordion.Header>
      <Accordion.Body>
        Yes, you can access your storage unit during our operating hours or opt for 24/7 access for certain units.
      </Accordion.Body>
    </Accordion.Item>
```

```
          <Accordion.Item eventKey="2">
            <Accordion.Header>Are there any hidden fees?</Accordion.Header>
            <Accordion.Body>
              No, we believe in transparent pricing. You only pay for what you use with no hidden
charges.
            </Accordion.Body>
          </Accordion.Item>
        </Accordion>
      </Container>
    </div>
  );
};

export default Home;
```

9.4.2 Backend

DB_Transactions_Main.py:

```
from db_client import execute_query
from utils import format_response
from datetime import datetime
```

```
import json


def lambda_handler(event, context):
    """
    Main entry point for handling database transactions.
    Routes requests to the appropriate database operations.
    """
    try:
        print(f"Event: {event}")
        # event = json.loads(event)
        # Extract the action and data from the event
        action = event.get("action", None)
        data = event.get("data", {})


        # Storage API
        if action == "list_all_storage_locations":
            # Fetch all available storage locations -> Working as expected
            # query = "SELECT * FROM StorageSpaces WHERE availability = 'available';"
            query = "SELECT * FROM StorageSpaces WHERE availability = ?;"
```

```
        params = ('available',)
        result = execute_query(query, params)
        print(f"Result: {result}")
        return format_response(200, result)

    elif action == "check_available_storage":
        # Fetch available storage locations based on the given date range
        start_date = data.get("start_date")
        end_date = data.get("end_date")

        if not start_date or not end_date:
            return {"statusCode": 400, "body": {"error": "start_date and end_date are
required"}}

        query = """
            SELECT s.*
            FROM StorageSpaces s
            WHERE s.availability = 'available'
            AND NOT EXISTS (
                SELECT 1
```

```
            FROM Rentals r
            WHERE r.storage_id = s.storage_id
              AND (
                (r.start_date < ? AND r.end_date > ?) -- Rentals that overlap at the start
                OR
                (r.start_date <= ? AND r.end_date >= ?) -- Rentals entirely within the query
range
                OR
                (r.start_date >= ? AND r.start_date <= ?) -- Rentals that overlap at the end
              )
          );
        """
        params = (end_date,start_date,end_date,start_date,start_date,end_date)
        results = execute_query(query, params)
        return {"statusCode": 200, "body": results}


    elif action == "fetch_storage_by_id":
        # Fetch a specific storage location by ID
        query = "SELECT * FROM StorageSpaces WHERE storage_id = ?;"
        params = (data.get("storage_id"),)
```

```
    result = execute_query(query, params)
    return {"statusCode": 200, "body": result}


elif action == "fetch_storage_by_owner_id":
    # Fetch all the storage location of the owner based upon his user_id
    owner_id = data.get("owner_id")
    if not owner_id:
        return {"statusCode": 400, "body": {"error": "owner_id is required"}}


    query = """
        SELECT
            s.storage_id, s.owner_id, s.title, s.description, s.size,
            s.location, s.price_per_month, s.availability, s.images_url,
            s.insurance_option, s.eircode, s.storage_type,
            s.created_at, s.updated_at,
            u.name AS owner_name, u.email AS owner_email, u.phone AS owner_phone
        FROM StorageSpaces s
        JOIN Users u ON s.owner_id = u.user_id
        WHERE s.owner_id = ?;
    """
```

```
        params = (owner_id,)
        result = execute_query(query, params)
        return {"statusCode": 200, "body": result}


    elif action == "add_storage_location":
        query = """
            INSERT INTO StorageSpaces (owner_id, title, description, storage_type, size,
location, eircode, price_per_month, availability, images_url, insurance_option, created_at,
updated_at)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """
        params = (
            data.get("user_id"),
            data.get("title"),
            data.get("description"),
            data.get("storage_type"),
            data.get("size"),
            data.get("location"),
            data.get("eircode"),
            data.get("price_per_month"),
```

```
            "available",
            data.get("images_url"),
            data.get("insurance_option", 0),
            datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
            datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        )
        execute_query(query, params, commit=True)
        return {"statusCode": 200, "body": {"message": "Storage location added"}}

    elif action == "delete_storage_location":
        storage_id = data.get("storage_id")

        # Check for current or future rentals
        rental_check_query = """
            SELECT COUNT(*) AS rental_count
            FROM Rentals
            WHERE storage_id = ? AND end_date >= GETDATE();
        """
        rental_check_params = (storage_id,)
        rental_count_result = execute_query(rental_check_query, rental_check_params)
```

```
        # Extract rental count
        rental_count = rental_count_result[0]["rental_count"] if rental_count_result else 0

        if rental_count > 0:
            # Rentals exist, block deletion
            return {
                "statusCode": 400,
                "body": {"error": "Cannot delete storage space. Active or future rentals are
present."}
            }

        # Proceed to delete storage space if no rentals
        delete_query = "DELETE FROM StorageSpaces WHERE storage_id = ?;"
        delete_params = (storage_id,)
        execute_query(delete_query, delete_params, commit=True)

        return {
            "statusCode": 200,
            "body": {"message": "Storage location deleted successfully."}
```

```
        }


    elif action == "update_storage_location":
        update_fields = ", ".join([f"{key} = ?" for key in data["update_data"].keys()])
        query = f"UPDATE StorageSpaces SET {update_fields}, updated_at = GETDATE()
WHERE storage_id = ?"
        params = list(data["update_data"].values()) + [data.get("storage_id")]
        execute_query(query, params, commit=True)
        return {"statusCode": 200, "body": {"message": "Storage location updated"}}


    elif action == "check_storage_availability":
        storage_id = data.get("storage_id")
        start_date = data.get("start_date")
        end_date = data.get("end_date")


        if not storage_id or not start_date or not end_date:
            return {"statusCode": 400, "body": {"error": "storage_id, start_date, and end_date
are required"}}


        query = """
```

```
        SELECT COUNT(*) AS count
        FROM Rentals
        WHERE storage_id = ?
        AND (
            (start_date < ? AND end_date > ?)
            OR
            (start_date <= ? AND end_date >= ?)
            OR
            (start_date >= ? AND start_date <= ?)
        )
    """
    params = (storage_id, end_date, start_date, end_date, start_date, start_date, end_date)
    result = execute_query(query, params)
    available = result[0]["count"] == 0

    return {"statusCode": 200, "body": {"available": available}}

elif action == "get_storage_price":
    storage_id = data.get("storage_id")
```

```
    if not storage_id:
        return {"statusCode": 400, "body": {"error": "storage_id is required"}}

    query = """
        SELECT price_per_month
        FROM StorageSpaces
        WHERE storage_id = ?
    """
    params = (storage_id,)
    result = execute_query(query, params)
    return {"statusCode": 200, "body": result[0] if result else {}}

#Profile API
elif action == "create_account":
    #Creation of the new Account details addition in the DB
    user_id = data.get("user_id")
    email = data.get("email")
    role_name = data.get("role_name")
    if isinstance(role_name, tuple):
        role_name = role_name[0]
```

```
        role_id = 1001 if role_name == "admin" else 1004
        print(f"RoleName = {role_name}, Role_Id: {role_id}")
        # query = (
        #     f"INSERT INTO Users (user_id, email, role_id, created_at, updated_at) "
        #     f"VALUES ('{user_id}', '{email}', '{role_id}', '{datetime.now().strftime('%Y-
%m-%d %H:%M:%S')}', '{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}');"
        # )
        query = "INSERT INTO Users (user_id, email, role_id, created_at, updated_at)
VALUES (?, ?, ?, ?, ?)"
        params = (user_id, email, role_id, datetime.now().strftime('%Y-%m-%d
%H:%M:%S'), datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
        execute_query(query, params, commit=True)
        return format_response(200, {"message": "User created successfully in the DB"})

    elif action == "update_profile":
        #Updation of the Profile
        user_id = data.get("user_id")
        fields = data.get("profile_data", {})
        email = fields.get("email")
        name = fields.get("name")
```

```
        phone = fields.get("phone")
        image_url = fields.get("profile_image_url")
        role_name = fields.get("role_name")
        if isinstance(role_name, tuple):
            role_name = role_name[0]
        role_id = 1001 if role_name=="admin" else 1004
        address = fields.get("address")
        eircode = fields.get("eircode")
        updated_at = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        # query = (
        #     f"UPDATE Users "
        #     f"SET name = '{name}', phone = '{phone}', profile_image_url = '{image_url}',
role_id = '{role_id}', address = '{address}', eircode = '{eircode}', updated_at =
'{updated_at}'"
        #     f"WHERE user_id = '{user_id}'"
        # )

        # Query with parameterized placeholders
        query = """
            UPDATE Users
```

```
            SET name = ?, phone = ?, profile_image_url = ?, role_id = ?, address = ?, eircode =
?, updated_at = ?
            WHERE user_id = ?
        """
        params = (name, phone, image_url, role_id, address, eircode, updated_at, user_id)
        execute_query(query, params, commit=True)
        # execute_query(query, commit=True)
        return format_response(200, {"message": "User Profile Updated Successfully"})


    elif action == "get_user_profile":
        query = """
            SELECT
                u.user_id, u.name, u.email, u.phone, u.profile_image_url,
                u.address, u.eircode, u.created_at, u.updated_at,
                r.role_name
            FROM Users u
            JOIN Roles r ON u.role_id = r.role_id
            WHERE u.user_id = ?;
        """
        params = (data.get("user_id"),)
```

```
            result = execute_query(query, params)
            if result:
                return {"statusCode": 200, "body": result[0]}  # Return the first result as a
dictionary
            return {"statusCode": 404, "body": {"error": "User not found"}}


        elif action == "list_all_users":
            query = """
                SELECT
                    u.user_id, u.name, u.email, u.phone, u.profile_image_url,
                    u.address, u.eircode, u.created_at, u.updated_at,
                    r.role_name
                FROM Users u
                JOIN Roles r ON u.role_id = r.role_id;
            """
            result = execute_query(query)
            return {"statusCode": 200, "body": result}


        elif action == "update_user_role":
            #Updation of the Profile
```

```
            user_email_id = data.get("email_id")
            query = """
                UPDATE Users
                SET role_id = (SELECT role_id FROM Roles WHERE role_name = 'admin')
                WHERE email = ?
            """
            params = (user_email_id)
            execute_query(query, params, commit=True)
            # execute_query(query, commit=True)
            return format_response(200, {"message": "User Profile Updated to Admin
Successfully"})


        elif action == "remove_user":
            # Deletion of the user and related data
            user_id = data.get("user_id")
            if not user_id:
                return {"statusCode": 400, "body": {"error": "user_id is required"}}


            # Step 1: Check rentals as a renter
            check_renter_query = """
```

```
            SELECT COUNT(*)
            FROM Rentals
            WHERE renter_id = ? AND end_date >= GETDATE();
        """
        renter_count = execute_query(check_renter_query, user_id)
        if renter_count[0][""] > 0:
            return format_response(400, {"error": "Cannot delete user due to active or future
rentals as a renter"})

        # Step 2: Check rentals for owned storage
        check_owner_query = """
            SELECT COUNT(*)
            FROM Rentals r
            JOIN StorageSpaces s ON r.storage_id = s.storage_id
            WHERE s.owner_id = ? AND r.end_date >= GETDATE();
        """
        owner_count = execute_query(check_owner_query, user_id)
        if owner_count[0][""] > 0:
            return format_response(400, {"error": "Cannot delete user due to active or future
rentals for owned storage spaces"})
```

```
# Step 3: Delete reviews made by the user
delete_user_reviews_query = """
    DELETE FROM Reviews WHERE user_id = ?;
"""
execute_query(delete_user_reviews_query, user_id, commit=True)

# Step 4: Delete reviews for user-owned storage spaces
delete_storage_reviews_query = """
    DELETE r
    FROM Reviews r
    JOIN StorageSpaces s ON r.storage_id = s.storage_id
    WHERE s.owner_id = ?;
"""
execute_query(delete_storage_reviews_query, user_id, commit=True)

# Step 5: Delete rentals where the user is a renter
delete_renter_rentals_query = """
    DELETE FROM Rentals WHERE renter_id = ?;
"""
```

```
execute_query(delete_renter_rentals_query, user_id, commit=True)


# Step 6: Delete rentals for owned storage
delete_owner_rentals_query = """
    DELETE r
    FROM Rentals r
    JOIN StorageSpaces s ON r.storage_id = s.storage_id
    WHERE s.owner_id = ?;
"""
execute_query(delete_owner_rentals_query, user_id, commit=True)


# Step 7: Delete storage spaces owned by the user
delete_storage_query = """
    DELETE FROM StorageSpaces WHERE owner_id = ?;
"""
execute_query(delete_storage_query, user_id, commit=True)


# Step 8: Delete the user from the Users table
delete_user_query = """
    DELETE FROM Users WHERE user_id = ?;
```

```
        """
        execute_query(delete_user_query, user_id, commit=True)

        return format_response(200, {"message": "User and related data deleted
successfully"})



    #Rental Service API

    # List all rentals
    if action == "list_all_rentals":
        query = """
                SELECT
                    r.rental_id, r.start_date, r.end_date, r.total_price, r.payment_status,
                    u_renter.user_id AS renter_id, u_renter.name AS renter_name,
u_renter.email AS renter_email, u_renter.phone AS renter_phone,
                    u_owner.user_id AS owner_id, u_owner.name AS owner_name,
u_owner.email AS owner_email, u_owner.phone AS owner_phone,
                    s.storage_id, s.title AS storage_title, s.description AS storage_description,
s.size AS storage_size,
```

```
                s.location AS storage_location, s.price_per_month, s.insurance_option,
s.eircode, s.storage_type
                FROM Rentals r
                JOIN StorageSpaces s ON r.storage_id = s.storage_id
                JOIN Users u_renter ON r.renter_id = u_renter.user_id
                JOIN Users u_owner ON s.owner_id = u_owner.user_id;
            """
        result = execute_query(query)
        return format_response(200, result)

    # List rental by rental_id
    elif action == "list_rental_by_id":
        query = """
                SELECT
                    r.rental_id, r.start_date, r.end_date, r.total_price, r.payment_status,
                    u_renter.user_id AS renter_id, u_renter.name AS renter_name,
u_renter.email AS renter_email, u_renter.phone AS renter_phone,
                    u_owner.user_id AS owner_id, u_owner.name AS owner_name,
u_owner.email AS owner_email, u_owner.phone AS owner_phone,
                    s.storage_id, s.title AS storage_title, s.description AS storage_description,
s.size AS storage_size,
```

```
                    s.location AS storage_location, s.price_per_month, s.insurance_option,
s.eircode, s.storage_type
                FROM Rentals r
                JOIN StorageSpaces s ON r.storage_id = s.storage_id
                JOIN Users u_renter ON r.renter_id = u_renter.user_id
                JOIN Users u_owner ON s.owner_id = u_owner.user_id
                WHERE r.rental_id = ?;
            """
        params = (data.get("rental_id"),)
        result = execute_query(query, params)
        return {"statusCode": 200, "body": result}


    # List rentals by storage_id
    elif action == "list_rentals_by_storage_id":
        query = """
                SELECT
                    r.rental_id, r.start_date, r.end_date, r.total_price, r.payment_status,
                    u_renter.user_id AS renter_id, u_renter.name AS renter_name,
u_renter.email AS renter_email, u_renter.phone AS renter_phone,
```

```
                u_owner.user_id AS owner_id, u_owner.name AS owner_name,
u_owner.email AS owner_email, u_owner.phone AS owner_phone,
                s.storage_id, s.title AS storage_title, s.description AS storage_description,
s.size AS storage_size,
                s.location AS storage_location, s.price_per_month, s.insurance_option,
s.eircode, s.storage_type
                FROM Rentals r
                JOIN StorageSpaces s ON r.storage_id = s.storage_id
                JOIN Users u_renter ON r.renter_id = u_renter.user_id
                JOIN Users u_owner ON s.owner_id = u_owner.user_id
                WHERE r.storage_id = ?;
            """
        params = (data.get("storage_id"),)
        result = execute_query(query, params)
        return {"statusCode": 200, "body": result}

    # List rentals by renter_id
    elif action == "list_rentals_by_renter_id":
        query = """
                SELECT
```

```
                r.rental_id, r.start_date, r.end_date, r.total_price, r.payment_status,
                u_renter.user_id AS renter_id, u_renter.name AS renter_name,
u_renter.email AS renter_email, u_renter.phone AS renter_phone,
                u_owner.user_id AS owner_id, u_owner.name AS owner_name,
u_owner.email AS owner_email, u_owner.phone AS owner_phone,
                s.storage_id, s.title AS storage_title, s.description AS storage_description,
s.size AS storage_size,
                s.location AS storage_location, s.price_per_month, s.insurance_option,
s.eircode, s.storage_type
            FROM Rentals r
            JOIN StorageSpaces s ON r.storage_id = s.storage_id
            JOIN Users u_renter ON r.renter_id = u_renter.user_id
            JOIN Users u_owner ON s.owner_id = u_owner.user_id
            WHERE r.renter_id = ?;
        """
        params = (data.get("renter_id"),)
        result = execute_query(query, params)
        return {"statusCode": 200, "body": result}

    # List rentals by owner_id
    elif action == "list_rentals_by_owner_id":
```

```
        query = """
                SELECT
                    r.rental_id, r.start_date, r.end_date, r.total_price, r.payment_status,
                    u_renter.user_id AS renter_id, u_renter.name AS renter_name,
u_renter.email AS renter_email, u_renter.phone AS renter_phone,
                    u_owner.user_id AS owner_id, u_owner.name AS owner_name,
u_owner.email AS owner_email, u_owner.phone AS owner_phone,
                    s.storage_id, s.title AS storage_title, s.description AS storage_description,
s.size AS storage_size,
                    s.location AS storage_location, s.price_per_month, s.insurance_option,
s.eircode, s.storage_type
                FROM Rentals r
                JOIN StorageSpaces s ON r.storage_id = s.storage_id
                JOIN Users u_renter ON r.renter_id = u_renter.user_id
                JOIN Users u_owner ON s.owner_id = u_owner.user_id
                WHERE s.owner_id = ?;
            """
        params = (data.get("owner_id"),)
        result = execute_query(query, params)
        return {"statusCode": 200, "body": result}
```

```
# Create a rental
elif action == "create_rental":
    # Ensure no overlapping rentals
    query_check = """
        SELECT COUNT(*) AS count
        FROM Rentals
        WHERE storage_id = ? AND (
            (start_date <= ? AND end_date >= ?) OR
            (start_date >= ? AND start_date <= ?)
        );
    """
    params_check = (
        data["storage_id"],
        data["end_date"],
        data["start_date"],
        data["start_date"],
        data["end_date"]
    )
    overlap_result = execute_query(query_check, params_check)
    if overlap_result[0]["count"] > 0:
```

```
            return {"statusCode": 400, "body": {"error": "Storage is already rented for the
given timeline."}}


        # Insert the new rental
        query = """
            INSERT INTO Rentals (storage_id, renter_id, start_date, end_date, total_price,
payment_status)
            VALUES (?, ?, ?, ?, ?, ?);
        """
        params = (
            data["storage_id"],
            data["renter_id"],
            data["start_date"],
            data["end_date"],
            data["total_price"],
            data["payment_status"]
        )
        execute_query(query, params, commit=True)
        return {"statusCode": 200, "body": {"message": "Rental created successfully"}}
```

```
        # # Cancel a rental (update payment status) Cancelled this API
        # elif action == "cancel_rental":
        #     query = "UPDATE Rentals SET payment_status = 'canceled', updated_at =
GETDATE() WHERE rental_id = ?;"
        #     params = (data.get("rental_id"),)
        #     execute_query(query, params, commit=True)
        #     return {"statusCode": 200, "body": {"message": "Rental canceled successfully"}}

        # Delete a rental
        elif action == "delete_rental":
            rental_id = data.get("rental_id")

            # First, delete associated payment records
            delete_payments_query = "DELETE FROM Payments WHERE rental_id = ?;"
            delete_payments_params = (rental_id,)
            execute_query(delete_payments_query, delete_payments_params, commit=True)

            # Then, delete the rental record
            delete_rental_query = "DELETE FROM Rentals WHERE rental_id = ?;"
            delete_rental_params = (rental_id,)
```

```
        execute_query(delete_rental_query, delete_rental_params, commit=True)


        return {"statusCode": 200, "body": {"message": "Rental and associated payments deleted successfully"}}


    #List all Reviews
    elif action == "list_all_reviews":
        query = """
            WITH AvgRatings AS (
                SELECT
                    storage_id,
                    AVG(rating) AS average_rating
                FROM Reviews
                GROUP BY storage_id
            )
            SELECT
                r.review_id, r.rating, r.comment, r.created_at,
                r.user_id AS reviewer_id, u_reviewer.name AS reviewer_name, u_reviewer.email AS reviewer_email, u_reviewer.profile_image_url AS reviewer_profile_image,
```

```
            s.storage_id, s.owner_id, u_owner.name AS owner_name, u_owner.email AS
owner_email,
            s.title AS storage_title, s.description AS storage_description,
            ar.average_rating
        FROM Reviews r
        JOIN StorageSpaces s ON r.storage_id = s.storage_id
        JOIN Users u_reviewer ON r.user_id = u_reviewer.user_id
        JOIN Users u_owner ON s.owner_id = u_owner.user_id
        LEFT JOIN AvgRatings ar ON r.storage_id = ar.storage_id;
    """
    result = execute_query(query)
    return {"statusCode": 200, "body": result}



#List all Reviews based upon Storage ID
elif action == "list_reviews_by_storage_id":
    query = """
        WITH AvgRatings AS (
            SELECT
                storage_id,
```

```
        AVG(rating) AS average_rating
    FROM Reviews
    GROUP BY storage_id
),
RankedReviews AS (
    SELECT
        r.review_id, r.rating, r.comment, r.created_at,
        r.user_id AS reviewer_id, u_reviewer.name AS reviewer_name,
u_reviewer.email AS reviewer_email,
        CAST(u_reviewer.profile_image_url AS NVARCHAR(MAX)) AS
reviewer_profile_image,
        s.storage_id, s.owner_id, u_owner.name AS owner_name, u_owner.email AS
owner_email,
        s.title AS storage_title, CAST(s.description AS NVARCHAR(MAX)) AS
storage_description,
        ar.average_rating,
        ROW_NUMBER() OVER (PARTITION BY r.review_id ORDER BY
r.created_at DESC) AS row_num
    FROM Reviews r
    JOIN StorageSpaces s ON r.storage_id = s.storage_id
    JOIN Users u_reviewer ON r.user_id = u_reviewer.user_id
```

```
            JOIN Users u_owner ON s.owner_id = u_owner.user_id
            LEFT JOIN AvgRatings ar ON r.storage_id = ar.storage_id
            WHERE r.storage_id = ?
        )
        SELECT *
        FROM RankedReviews
        WHERE row_num = 1;

    """
    params = (data.get("storage_id"),)
    result = execute_query(query, params)
    return {"statusCode": 200, "body": result}



#List all Reviews based upon Owner ID
elif action == "list_reviews_by_owner_id":
    query = """
        WITH AvgRatings AS (
            SELECT
                storage_id,
```

```
        AVG(rating) AS average_rating
    FROM Reviews
    GROUP BY storage_id
),
RankedReviews AS (
    SELECT
        r.review_id, r.rating, r.comment, r.created_at,
        r.user_id AS reviewer_id, u_reviewer.name AS reviewer_name,
u_reviewer.email AS reviewer_email,
        CAST(u_reviewer.profile_image_url AS NVARCHAR(MAX)) AS
reviewer_profile_image,
        s.storage_id, s.owner_id, u_owner.name AS owner_name, u_owner.email AS
owner_email,
        s.title AS storage_title, CAST(s.description AS NVARCHAR(MAX)) AS
storage_description,
        ar.average_rating,
        ROW_NUMBER() OVER (PARTITION BY r.review_id ORDER BY
r.created_at DESC) AS row_num
    FROM Reviews r
    JOIN StorageSpaces s ON r.storage_id = s.storage_id
    JOIN Users u_reviewer ON r.user_id = u_reviewer.user_id
```

```
            JOIN Users u_owner ON s.owner_id = u_owner.user_id
            LEFT JOIN AvgRatings ar ON r.storage_id = ar.storage_id
            WHERE s.owner_id = ?
        )
        SELECT *
        FROM RankedReviews
        WHERE row_num = 1;
    """
    params = (data.get("owner_id"),)
    result = execute_query(query, params)
    return {"statusCode": 200, "body": result}



#List review based upon Review ID
elif action == "list_review_by_review_id":
    query = """
        WITH AvgRatings AS (
            SELECT
                storage_id,
                AVG(rating) AS average_rating
```

```
            FROM Reviews
            GROUP BY storage_id
        )
        SELECT
            r.review_id, r.rating, r.comment, r.created_at,
            r.user_id AS reviewer_id, u_reviewer.name AS reviewer_name,
u_reviewer.email AS reviewer_email, u_reviewer.profile_image_url AS
reviewer_profile_image,
            s.storage_id, s.owner_id, u_owner.name AS owner_name, u_owner.email AS
owner_email,
            s.title AS storage_title, s.description AS storage_description,
            ar.average_rating
        FROM Reviews r
        JOIN StorageSpaces s ON r.storage_id = s.storage_id
        JOIN Users u_reviewer ON r.user_id = u_reviewer.user_id
        JOIN Users u_owner ON s.owner_id = u_owner.user_id
        LEFT JOIN AvgRatings ar ON r.storage_id = ar.storage_id
        WHERE r.review_id = ?;

    """
```

```
    params = (data.get("review_id"),)
    result = execute_query(query, params)
    return {"statusCode": 200, "body": result}




#Creation of the new Review
elif action == "create_review":
    # Check if the user has already reviewed the storage
    check_query = """
        SELECT review_id
        FROM Reviews
        WHERE storage_id = ? AND user_id = ?;
    """
    check_params = (data["storage_id"], data["user_id"])
    check_result = execute_query(check_query, check_params)


    if check_result:
        # If review exists, update it
        review_id = check_result[0]["review_id"]
```

```
    update_query = """
        UPDATE Reviews
        SET rating = ?, comment = ?, created_at = GETDATE()
        WHERE review_id = ?;
    """
    update_params = (data["rating"], data["comment"], review_id)
    execute_query(update_query, update_params, commit=True)
    return {"statusCode": 200, "body": {"message": "Review updated successfully"}}

# If no review exists, insert a new one
create_query = """
    INSERT INTO Reviews (storage_id, user_id, rating, comment, created_at)
    VALUES (?, ?, ?, ?, GETDATE());
"""
create_params = (
    data["storage_id"],
    data["user_id"],
    data["rating"],
    data["comment"]
)
```

```
    execute_query(create_query, create_params, commit=True)
    return {"statusCode": 200, "body": {"message": "Review created successfully"}}


#Updation of Review
elif action == "update_review":
    query = """
        UPDATE Reviews
        SET rating = ?, comment = ?, created_at = GETDATE()
        WHERE review_id = ?;
    """
    params = (
        data["rating"],
        data["comment"],
        data["review_id"]
    )
    execute_query(query, params, commit=True)
    return {"statusCode": 200, "body": {"message": "Review updated successfully"}}


#Deletion of Review
elif action == "delete_review":
```

```
            query = "DELETE FROM Reviews WHERE review_id = ?;"
            params = (data["review_id"],)
            execute_query(query, params, commit=True)
            return {"statusCode": 200, "body": {"message": "Review deleted successfully"}}

        else:
            return format_response(400, {"error": f"Unsupported action: {action}"})

    except Exception as e:
        print(f"Error processing request: {str(e)}")
        return format_response(500, {"error": "Internal server error"})
```

CHAPTER 10
REFERENCES

1. Airbnb (2023). About Us. Airbnb. Available at: https://www.airbnb.com/about
2. Botsman, R., & Rogers, R. (2010). *What's Mine is Yours: The Rise of Collaborative Consumption*. HarperBusiness. Available at: https://www.harpercollins.com/products/whats-mine-is-yours-rachel-botsmanroo-rogers
3. Hamari, J., Sjöklint, M., & Ukkonen, A. (2016). The sharing economy: Why people participate in collaborative consumption. *Journal of the Association for Information Science and Technology*, 67(9), 2047–2059. Available at: https://doi.org/10.1002/asi.23552
4. Schor, J. (2014). *Debating the Sharing Economy*. Great Transition Initiative. Available at: https://greattransition.org/publication/debating-the-sharing-economy
5. Sundararajan, A. (2016). *The Sharing Economy: The End of Employment and the Rise of Crowd-Based Capitalism*. MIT Press. Available at: https://mitpress.mit.edu/9780262034579/the-sharing-economy/
6. Trochim, W. M. (2021). *Research Methods: The Essential Knowledge Base*. Cengage Learning. Available at: https://www.cengage.com/c/research-methods-the-essential-knowledge-base-2e-trochim/9781133954774/
7. CBRE (2022). *Irish Commercial Real Estate Market Overview*. CBRE Ireland. Available at: https://www.cbre.ie/en/research-reports
8. Eurostat (2022). *Urbanization Trends in Europe*. European Commission. Available at: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Urban_development
9. BlaBlaCar (2023). *About Us*. BlaBlaCar. Available at: https://blog.blablacar.com/about-us

10. ResearchGate (2023). Overview of the Sharing Economy. Available at: https://www.researchgate.net/figure/Overview-of-the-sharing-economy-Source-Business-Model-Toolbox_fig2_321076799
11. PwC Hungary. (2016, July 6). The sharing economy presents Europe with a €570 billion opportunity. Retrieved from https://www.pwc.com/hu/en/pressroom/2016/sharing_economy_europe.html
12. ResearchVoyage. (n.d.). *How to Write a Method Section for a Research Paper* [Webpage]. Available at: https://researchvoyage.com/how-to-write-method-section-research-paper/
13. FasterCapital. (n.d.). *The Impact of the Sharing Economy on Traditional Industries* [Webpage] Available at: https://fastercapital.com/topics/the-impact-of-the-sharing-economy-on-traditional-industries.html/1
14. Self-storage market size, share and Trends report, 2030 (no date) *Self-storage Market Size, Share And Trends Report, 2030*. Available at: https://www.grandviewresearch.com/industry-analysis/self-storage-market-report
15. OpenAI (2023). *DALL-E 2*. [AI image generation model]. Available at: https://openai.com/index/dall-e-2/
16. P2P rental apps market (2025) Market.us. Available at: https://market.us/report/p2p-rental-apps-market/ (Accessed: 04 January 2025).
17. Amazon Web Services (2020) Amazon Simple Storage Service (Amazon S3) Developer Guide. Available at: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html (Accessed: 3 January 2025).
18. Benet, J. (2014) IPFS - Content Addressed, Versioned, P2P File System. arXiv, 1407.3561. Available at: https://arxiv.org/abs/1407.3561 (Accessed: 3 January 2025).

19. Botsman, R. (2013) The Sharing Economy Lacks A Shared Definition. Collaborative Lab. Available at: https://www.collaborativelab.com/ (Accessed: 3 January 2025).
20. Botsman, R. and Rogers, R. (2010) What's Mine Is Yours: The Rise of Collaborative Consumption. New York: Harper Business.
21. Codagnone, C., Biagi, F. and Abadie, F. (2016) The Passions and the Interests: Unpacking the 'Sharing Economy'. Institute for Prospective Technological Studies, European Commission. Available at: https://ec.europa.eu/jrc/en/publication/eur-scientific-and-technical-research-reports/passions-and-interests-unpacking-sharing-economy (Accessed: 3 January 2025).
22. Codd, E.F. (1970) A relational model of data for large shared data banks. Communications of the ACM, 13(6), pp.377-387.
23. Dellarocas, C. (2003) The digitization of word-of-mouth: Promise and challenges of online reputation systems. Management Science, 49(10), pp.1407-1424.
24. Elmasri, R. and Navathe, S. (2015) Fundamentals of Database Systems. 7th edn. Boston: Pearson.
25. European Commission (2016) General Data Protection Regulation (GDPR).Available at: https://gdpr-info.eu/ (Accessed: 3 January 2025).
26. Firebase (n.d.) Firebase Authentication.Available at: https://firebase.google.com/docs/auth (Accessed: 3 January 2025).
27. Hawlitschek, F., Teubner, T. and Weinhardt, C. (2016) Trust in the sharing economy. Die Unternehmung, 70(1), pp.26-44.
28. Jøsang, A., Ismail, R. and Boyd, C. (2007) A survey of trust and reputation systems for online service provision. Decision Support Systems, 43(2), pp.618-644.
29. Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q. and Fu, K. (2003) Plutus: Scalable secure file sharing on untrusted storage. In Proceedings of the USENIX

Conference on File and Storage Technologies (FAST). USENIX. Available at: https://www.usenix.org/legacy/events/fast03/tech/kallahalla.html (Accessed: 3 January 2025).

30. Kshetri, N. (2013) Privacy and security issues in cloud computing. Journal of Internet Commerce, 12(4), pp.310-320.
31. Mashal, I., Alsaryrah, O., Chung, T. and Yang, Y. (2015) Performance evaluation of mobile crowdsourcing under different spatiotemporal scenarios. Pervasive and Mobile Computing, 24, pp.104-112.
32. Neighbor (2023) Neighbor – The Easiest Way to Store Your Stuff. Available at: https://www.neighbor.com (Accessed: 3 January 2025).
33. Owyang, J. (2016) Collaborative Economy Honeycomb 3.0: Watch It Grow. Crowd Companies. Available at: http://www.web-strategist.com/blog/2016/10/31/collaborative-economy-honeycomb-3-0-watch-it-grow (Accessed: 3 January 2025).
34. Resnick, P. and Zeckhauser, R. (2002) Trust among strangers in internet transactions: Empirical analysis of eBay's reputation system. In Baye, M.R. (ed.) The Economics of the Internet and E-commerce. New York: Emerald Group, pp.127-157.
35. Resnick, P., Zeckhauser, R., Friedman, E. and Kuwabara, K. (2000) Reputation systems. Communications of the ACM, 43(12), pp.45-48.
36. Ristenpart, T., Tromer, E., Shacham, H. and Savage, S. (2012) Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. Journal of Computer Security, 19(1), pp.1-37.
37. Teoh, W.M.Y., Chong, S.C. and Lin, B. (2013) Factors affecting consumers' perception of electronic payment: An empirical analysis. Internet Research, 23(4), pp.465-485.

38. Xu, X., Racanello, S. and Park, E. (2016) The sharing economy's next frontier: Retail. Journal of Marketing Channels, 23(1-2), pp.41-51.

39. Zervas, G., Proserpio, D. and Byers, J.W. (2017) The rise of the sharing economy: Estimating the impact of Airbnb on the hotel industry. Journal of Marketing Research, 54(5), pp.687-705.

40. Amazon Web Services (2020) Amazon Simple Storage Service (Amazon S3) Developer Guide. Available at: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html (Accessed: 3 January 2025).

41. Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001) Manifesto for Agile Software Development. Available at: https://agilemanifesto.org/ (Accessed: 3 January 2025).

42. Codd, E.F. (1970) A relational model of data for large shared data banks. Communications of the ACM, 13(6), pp.377-387.

43. Cohn, M. (2005) Agile Estimating and Planning. Upper Saddle River, NJ: Pearson Education.

44. Crispin, L. and Gregory, J. (2009) Agile Testing: A Practical Guide for Testers and Agile Teams. Boston: Addison-Wesley.

45. Elmasri, R. and Navathe, S. (2015) Fundamentals of Database Systems. 7th edn. Boston: Pearson.

46. Firebase (n.d.) Firebase Authentication.Available at: https://firebase.google.com/docs/auth (Accessed: 3 January 2025).

47. Fowler, M. (2019) Continuous Integration. Available at: https://martinfowler.com/articles/continuousIntegration.html (Accessed: 3 January 2025).
48. Highsmith, J. (2004) Agile Project Management: Creating Innovative Products. Boston: Addison-Wesley.
49. Kshetri, N. (2013) Privacy and security issues in cloud computing. Journal of Internet Commerce, 12(4), pp.310-320.
50. Pressman, R.S. (2010) Software Engineering: A Practitioner's Approach. 7th edn. New York: McGraw-Hill.
51. Ristenpart, T., Tromer, E., Shacham, H. and Savage, S. (2012) Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. Journal of Computer Security, 19(1), pp.1-37.
52. Schwaber, K. and Sutherland, J. (2020) The Scrum Guide. Available at: https://scrumguides.org/scrum-guide.html (Accessed: 3 January 2025).
53. Cohn, M. (2005) Agile Estimating and Planning. Upper Saddle River, NJ: Pearson Education.
54. Crispin, L. and Gregory, J. (2009) Agile Testing: A Practical Guide for Testers and Agile Teams. Boston: Addison-Wesley.
55. Firebase (n.d.) Firebase Authentication. Available at: https://firebase.google.com/docs/auth (Accessed: 3 January 2025).