Housit : Mobile Application for Household Care

Minsoo KIM[2018312439], Seoyoung JO[2020312015], Seoyoung JIN[2018314609],
Jinwook CHOI[2017314098], Sooyoung HWANG[2020314193]

December 12, 2024

Abstract

Housit is an integrated mobile application designed to streamline household management for cohabitants by consolidating essential features like food tracking, expense management, chore allocation, and event scheduling into a single platform. Unlike existing solutions that require multiple apps, *Housit* enhances efficiency and collaboration with its intuitive interface, real-time updates, and category-based organization. The development process involved systematic planning, including API specification, UI/UX design, and backend implementation using technologies such as Android Studio, Spring, and MySQL. Key innovations include barcode scanning for food management and seamless integration of diverse functionalities. Despite time constraints, the team successfully implemented over 95% of the planned features, achieving a functional prototype. Future enhancements, based on user feedback, aim to include advanced features like sorting, recurring tasks, and real bank account integration, positioning *Housit* as a comprehensive solution for shared living environments.

Keywords : Household Management, Co-living, Grocery Management, Shared Expense Tracking, Chore Management, Event Scheduling, Barcode Scanning

1 Introduction

1.1 Project Objective & Problem Definition

Housit is a housecare mobile application designed to help cohabitants conveniently record, share, and access information that is frequently generated and shared in their daily lives.

When living with family, friends, partners, or spouses, various types of information naturally need to be shared. Examples include: managing food and ingredients stored in the refrigerator or pantry, tracking expenses and income in a shared account, dividing and monitoring household chores and adding and sharing schedules.

If decisions and discussions made during face-to-face interactions are not recorded immediately, important tasks like schedules, shopping lists, or chores can be forgotten or left unaddressed. This highlights the critical role of a shared, intuitive platform to record and manage such information.

With the widespread adoption of smartphones, cohabitants often rely on messenger apps for communication, especially when they are not physically together. In 2018, 94.4% of South Koreans reported using KakaoTalk for online communication [KIM18]. While convenient, messenger apps often mix various types of information in a single chat room, making it difficult to locate or retrieve specific details as time passes. This is a common issue observed in many shared households.

1.2 Analysis of Existing Solutions

Several applications exist to address such issues, including: **Food management**: *유통기한 언제지*, *BEEP*, **Shared accounting**: *Banksalad*, *편한 가계부*, *유플래너*, **Chore allocation**: *Tody*, *Sweepy*, **Shared schedules/events**: *Google Calendar*, *Naver Calendar*, *KakaoTalk Calendar*.

Each application offers unique features and benefits. However, using multiple apps often requires repetitive sign-ups, and information becomes scattered across platforms, making it inconvenient to locate or manage. Moreover, the lack of integrated management reduces the overall efficiency of collaboration.

1.3 Proposed Solution

Housit provides an all-in-one platform for recording and managing various types of information generated in shared living environments.

Users can manage tasks such as household chores, finances, food, and shared schedules seamlessly:

- **Food**: Identify the owner of food items, track expiration dates, and monitor quantities to predict future needs.
- **Finances**: Check the current balance of shared accounts, track monthly expenses and income, review past transactions, and plan budgets for the next month.

- **Chores**: Write descriptions for cleaning areas and randomly assign cleaning tasks to members each month.
- **Schedules**: Register and share events, view them in a past-to-present-to-future order, and set reminders for upcoming events.

1.4 Expected Benefits of the App

By consolidating these features into a single application, users no longer need multiple apps to manage their shared household. Unlike messenger apps, which lack structure, *Housit* organizes information into categories such as **Food**, **Finance**, **Chore**, and **Event**, making it easier to search, view, and utilize.

The app also enables real-time sharing of updates among cohabitants, helping them manage shared resources and plan effectively for the future. With its intuitive design and absence of complex sign-up processes, *Housit* significantly lowers the barriers to entry for users.

2 Design for the Proposed Service

2.1 Overall Architecture

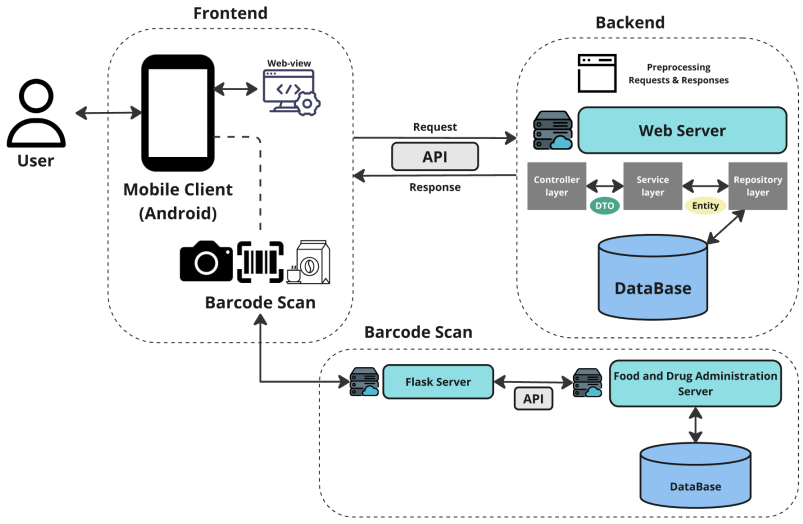

Figure 1: Overall Architecture

The overall architecture of the *Housit* mobile application is illustrated in the **Figure 1**. It can be broadly divided into three parts: frontend, backend, and barcode scan. Through the Android mobile app interface, which is implemented based on web-view, users can input and manage information related to Food, Finance, Chore, and Event. When data needs to be stored on the server, it is sent to the web server via the defined API. The data passes through the controller, service, and repository layers in the app server, where it is processed with appropriate logic and then stored in the server's database. The processed data is then sent back as a response to the mobile client.

When a barcode photo is taken and sent to the Flask server, the barcode ID is read and sent to the MFDS(Ministry of Food and Drug Safety) API. The MFDS server responds with relevant information about the food product in JSON format, which includes the product name and storage method. This data is then parsed and sent back to the mobile client.

2.2 Core Skill

2.2.1 Frontend : Android Studio, Next JS, Vercel

We developed a hybrid web app using Next.js as the main full-stack framework and TypeScript for stability. During the ideation phase, we considered React Native or PWA to support both Android and iOS. However, after reviewing the team's collective skill set, we decided to focus solely on the Android environment and used Android Studio and Kotlin for development. For rapid deployment, we utilized Vercel, which provides automated CI/CD.

2.2.2 Backend : Spring, MySQL, Google Cloud Platform

During the development process, Spring significantly contributed to improving workflow efficiency. Spring Boot was utilized to automate configurations and initial setup, enabling a quick and seamless project launch. Features such as HTTPS and CORS, configured for the first time, were implemented with the support of Spring Security, which simplified the process. Spring JPA helped bridge the gap between object-oriented programming and relational database paradigms, enabling safe and straightforward database design using only Java.

The deployment process involved using Google Cloud App Engine for server and database deployment. Additionally, GCP's services were used to configure DNS and HTTPS settings, and we created simple shell scripts to enable automated deployment on GCP.

2.2.3 Barcode Scan : Android Studio, Flask, AWS Lambda

To provide the barcode recognition function, we built the Flask server using the 'pyzbar' library. The server is responsible for processing barcode images and sending the recognized barcode number to the Ministry of Food and Drug Safety API to return the product name and storage method. The established Flask server was deployed to AWS Lambda using Zappa for increased convenience and stability.

AWS Lambda can be used with the API Gateway to handle HTTP requests, allowing specific Lambda functions to be invoked for each URL path. This allows barcode recognition to be provided in API form, reducing the maintenance burden on the server and providing automatic scaling and high availability.

When a certain button is pressed in an app made with TypeScript on the frontend, the function written in Kotlin using Android Studio is called to take a picture of the barcode and send it to the server. This allows the function to be performed when a button is clicked in the 'webview'.

2.3 Reasoning for Design Choice

2.3.1 Frontend

During the framework selection phase, we had to decide between using React or Next.js within the available tech stack. Considering the short development timeline and the potential for the project to grow in scope, we opted for Next.js. This decision was based on its features like Server-Side Rendering(SSR) for future optimization, built-in API routing within the frontend server, and convenient dynamic routing, which offers advantages over React.

Since React is classified as a library rather than a framework, it is less suited for rapid development. While frameworks often come with a learning curve, our familiarity with Next.js made it the better choice. In hindsight, given the severely limited development time, choosing Next.js - where adhering to its conventions can actually accelerate development - proved to be a smart decision.

2.3.2 Backend

Reason for One-Way Dependency in Layers Our team adopted the waterfall model, but during the initial stages, specific planning and decisions were not fully completed, leading to changes during the development process. To address this, the architecture was designed so that dependencies between layers flow in only one direction. As you can see in **Figure 1** above, The layers are structured to depend on each other in the order of controller $\rightarrow$ service $\rightarrow$ repository, which allowed us to focus on and modify only the affected layer when changes occurred.

For example, if modifications related to the database were required, only the repository layer needed to be adjusted, and there would be no impact on the service or controller. This design approach allowed changes to be reflected consistently and brought two main benefits.

1. Ease of Reflecting Changes

   Since dependencies flow in only one direction, the process of reflecting changes became simple and easy. Additionally, each layer can be modified independently, allowing us to adjust only the required parts and minimize the impact on the remaining layers.

2. Prevention of Side Effects

   Because each layer depends in a one-way direction, there are no circular dependencies, and the ripple effect of changes can be clearly seen. This minimizes unexpected side effects, improving the system's stability and significantly reducing the risks associated with changes.

Reason for Converting DTO in the Service Layer There is no definitive answer to which layer should handle the DTO conversion. The conversion between entities and DTOs can be handled in the controller, service, or repository layers. Moreover, some entities can be converted in the service layer, while others can be converted in the repository layer. However, in this architecture, the entire entity ↔ DTO conversion is handled in the service layer. The reasons are as follows:

1. Re-usability of Entities

   If the repository directly converts to a DTO, reusability might decrease because DTOs are designed to match specific API responses and therefore contain information limited to a specific purpose. On the other hand, in the service layer, entities are often reused multiple times during the process of calling various queries and processing business logic. Entities reflect the database model and contain rich information necessary for business logic, making it more efficient to convert them to DTOs after undergoing several operations. This allows the service layer to handle entities flexibly and convert them to DTOs to match various API responses when needed. This approach increases code reusability and makes maintenance easier.

2. Consistency

   This project required the implementation of many APIs, each of which might require different data formats. If entity-to-DTO conversion were distributed across the repository and service layers, we would have to repeatedly search for where each conversion should be handled for every API. This would decrease the readability of the code and make maintenance more difficult. While modern IDEs can assist with such tasks, maintaining structural consistency is much more intuitive and neat. Therefore, by handling all conversions in the service layer, all conversion logic can be concentrated in one layer, ensuring consistency and making it easier to manage and maintain various APIs more efficiently.

2.3.3 Barcode Scan

Flask was chosen in the process of server development and deployment because it provides fast development and flexibility as a lightweight web framework. In addition, Flask was able to deploy Restful APIs with simple settings, enabling rapid implementation of barcode scanning. By deploying servers to AWS Lambda using zappa, we reduced the complexity of server management and took advantage of 'serverless' architecture.

The pyzbar library supports a variety of barcode formats and provides users with the ability to quickly recognize barcodes in images, enabling users to respond quickly. In addition, by providing reliable information in connection with the API of the Ministry of Food and Drug Safety, users can easily recognize the product name.

2.4 Challenges

2.4.1 Frontend

During frontend development, we faced a significant challenge while creating a bridge for the barcode scanning functionality.

After setting up the WebView through Android Studio, we encountered an issue where the project pushed on GitHub could not be opened on the barcode scanning developer's computer, likely due to version discrepancies in Android Studio. To address this, we created a new project for the barcode scanning functionality. The original code was stored in a new repository, and only the relevant functionality was merged back via PR.

The barcode scanning functionality was initially developed within a separate Activity file that contained all its features. As I hadn't used Android Studio in a while, it took time to figure out how to import the file after merging and use its functions within the main project. Additionally, the barcode scanning feature was tested using a new WebView setup, and we later discovered that the code for loading WebView URLs-unexpectedly-existed in the new Activity file as well. This caused new web pages to open frequently within the app.

After extensive troubleshooting, including verifying the original files and the barcode scanning test web pages, we resolved the issue by extracting only the necessary functions from the new Activity file and integrating them into the MainActivity. By ensuring the WebView URL loading code was executed only once in the project, the issue of unexpected web page openings was resolved.

The goal of the barcode scanning feature was simple: to enable the WebView to call a function that could access the camera. Other functions did not need to be callable from the web. Thus, we used WebAppInterface to make only the camera permission-checking and subsequent processing function accessible from the WebView. This resolved all remaining issues.

Barcode data was successfully transmitted to Android Studio. From there, it was sent to Next.js, requiring additional logic in the web app to handle cases where barcodes were not recognized properly. Fortunately, this was resolved by using the evaluateJavascript function in Android Studio to call a function in the WebView, passing strings back and forth.

As a result, we introduced a curiosity-sparking feature that allowed users to register food items by scanning barcodes, making the app more engaging and user-friendly.

2.4.2 Backend

Most of the APIs and service logic that needed to be implemented in the backend server were just CRUD operations for each resource, so there was little complex business logic. However, some APIs experienced performance issues when fetching related data. The challenge came from the N+1 problem, especially in APIs like the home screen, which required fetching various types of information all at once, tailored to the user's settings. On the home screen, we needed to fetch various data, such as food, events, finance, and chores, based on the user's settings. As a result, separate queries were executed for each resource, leading to unnecessary multiple calls to the database. This issue led to the N+1 query problem, which could impact performance and increase query load.

During the initial development phase, functionality had to be prioritized, so we implemented the features first and deferred addressing the N+1 issue for later. While focusing on implementing the features to ensure the service worked properly, optimization and performance improvements were considered as follow-up tasks.

To address this issue in the future, methods such as query optimization, batch query processing, and appropriate use of lazy or eager loading can be considered. Additionally, implementing a caching strategy to prevent repeatedly fetching the same data could be a good solution. Although performance optimization is a critical factor, starting with complex optimization tasks could interfere with the implementation of functionality, so a step-by-step approach to improvement seemed more appropriate.

2.4.3 Barcode Scan

The original plan was to automatically recognize the product name and expiration date using OCR rather than barcode scanning. However, we tried to recognize the product name and expiration date through 'pytesseract' by going through preprocessing processes such as grayscale, binarization, and noise removal in photos of the product name and expiration date, but the character recognition accuracy was too low, so we changed it to using barcode scanning.

As a result of barcode scanning on the local server using 'pyzbar', it worked normally when we checked it with Postman. However, deploying that server failed because the 'pyzbar' library was dependent on the shared library called 'libzbar'. Therefore, in addition to 'pyzbar' and the PIL library, 'libzbar' had to be added as a layer to AWS Lambda. For this, we were able to solve the problem by compressing 'libzbar' and creating a layer using AWS CLI and adding the created 'layer ARN' to 'Zappa setting'.

After the code implemented by Kotlin in Android Studio requested camera permissions and resource read permissions, we made sure that the photos we took were sent to the server well. However, since it only worked when clicking the button in the existing XML layout, it was necessary to check that the same function was performed in WebView. To do this, we created a website by distributing HTML files including buttons through GitHub, and we used the website as WebView to check that it worked properly when clicking the button to solve the problem.

3 Implementation

3.1 UI/UX Viewpoints

This application is designed to help users navigate intuitively. To ensure an easy UX(User Experience) flow, each feature is clearly distinguished, enabling users to quickly find the information they need. Key features are located on the bottom navigation bar for easy access, guiding users seamlessly to the next steps. Important information is prominently displayed on the home screen, allowing for immediate access. This UX flow helps reduce user stress and enhances the overall enjoyment of using the app.

Color selection is also a crucial factor in shaping the user's first impression of the application. The primary colors used in the application are EC680A, F3F4F6, and 151419. These colors are chosen to create a cheerful and warm feeling, suitable for a home-related application. The orange color (EC680A) brings energy and vibrancy, while the light gray background (F3F4F6) offers a visually comfortable experience compared to pure white. Instead of using pure black (000000), the color 151419 is employed to create a more harmonious blend with the other colors. Such color choices are carefully arranged to resonate the user's emotions.

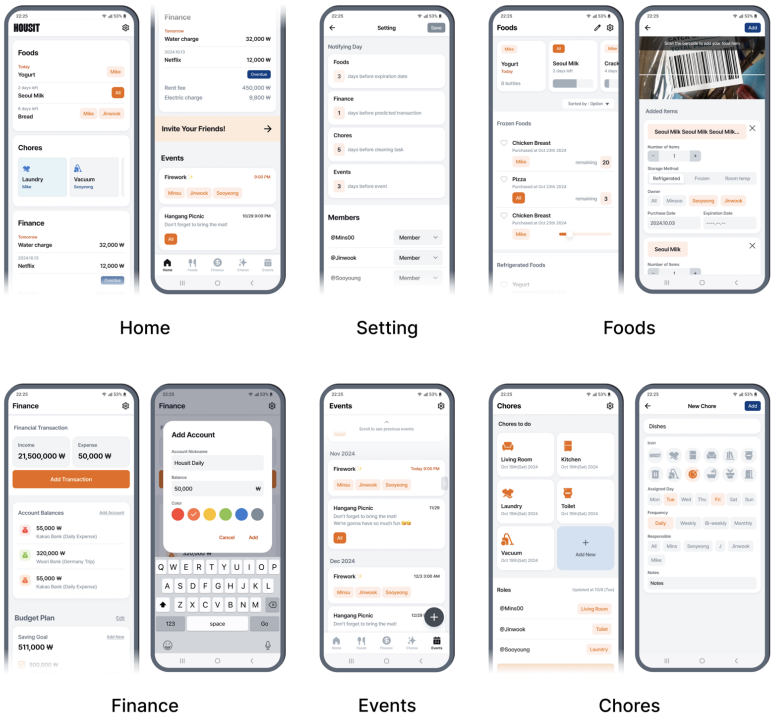

Figure 2: Main Views of *Housit* (Home, Setting, Foods, Finance, Events, Chores)

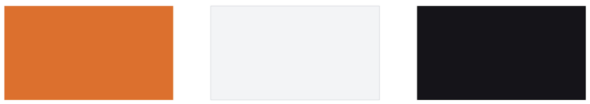

Figure 3: Color Selection for the application

To organize the information effectively, card components are utilized. Given the nature of the application, which involves conveying a significant amount of information, this approach ensures that each piece of text is well-organized and visually appealing. Each card contains essential information, helping users quickly grasp the content at a glance. The card design emphasizes each feature, making it easy for users to find what they need. Adequate spacing and margins between cards reduce visual clutter, which enhance user focus.

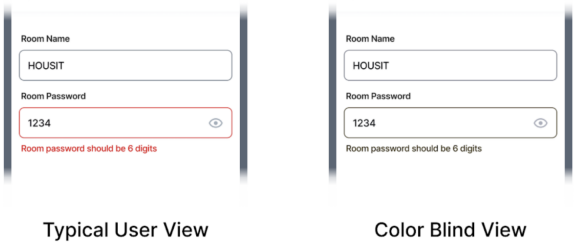

Figure 4: Accessibility Features for Typical and Color Blind Users

Accessibility was a key consideration in the design to accommodate a diverse range of users, including those with visual impairments. When indicating specific situations, not only colors but also icons and accompanying descriptive text are used to enhance the clarity of information. This approach helps users in understanding and perceiving situations more easily. For instance, without descriptive text, color blind users may struggle to interpret information that relies solely on color differentiation, as illustrated in the accompanying images. Furthermore, the contrast ratio between text and background colors is maintained at 4.5:1 or higher, following the WCAG (Web Content Accessibility Guidelines) [Con24], ensuring that all users can easily read and comprehend the content. This enhanced contrast helps even those who experience visual discomfort to easily

recognize the application's content.

The primary focus during the UI/UX design process was to implement a user-centered design. The goal was to create an application that is convenient for a diverse range of users. Ultimately, this UI/UX design will contribute to increasing user satisfaction and encouraging continued use of the application.

3.2 Frontend Detail

3.2.1 Frontend Structure

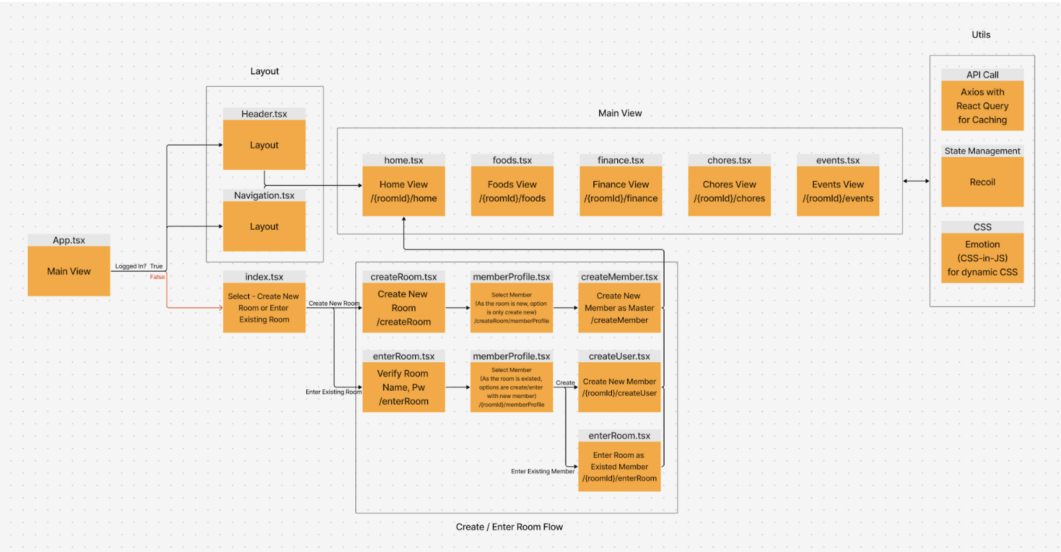

Figure 5: Frontend Page Structure

This front-end architecture, built with React, begins with App.tsx determining user login status. Logged-in users access main views (e.g., home.tsx, foods.tsx, etc.) via a Layout component with shared UI elements like Header.tsx and Navigation.tsx. The app features a room management flow where users can create a new room (createRoom.tsx → memberProfile.tsx → createMember.tsx → createUser.tsx) or enter an existing room (enterRoom.tsx → memberProfile.tsx). API calls are managed with Axios and React Query, state management with Recoil, and dynamic styling with Emotion, ensuring scalability and maintainability.

3.2.2 Connect with Backend

When accessing the backend API, we used axios and React-Query. Axios provides faster API connection and various API call options than fetch. React-Query provides caching and status of API(pending, success, error, etc.). Also, continuously synchronize, update data. With React-Query, users could feel a faster and more stable experience with optimization.

The following codes are part of our API call with axios and React-Query.

```
export const usePostMemberPassword = () => {
  return useMutation({
    mutationKey: [QueryKey.room.passowrd],
    mutationFn: async ({
      roomId,
      memberId,
      data,
    }: PostMemberPassword) => {
      const res = await axios.post(
        `${process.env.NEXT_PUBLIC_API_URL}/room/${roomId}/memberId/${memberId}/join`,
        data,
      );

      return res;
    },
    onSuccess: (data) => console.log(data),
  });
};
```

```
const { mutateAsync: postMemberPassword, isPending } =
  usePostMemberPassword();

const handleClick = async () => {
  if (isPending) return;

  try {
    const { data, status } = await postMemberPassword({
      roomId: String(roomId),
      memberId: String(id),
      data: { roomPassword: password },
    });

    if (status === 401) {
      setIsError(true);
      return;
    }

    push(`/${roomId}/home`);
  } catch (error) {
    console.error(error);
  }
};
```

Figure 6: React Query axios - POST method and Usage of above code

3.2.3 Bridge between Android Studio and Next JS

To use Android Studio - Next.js bridge, we need custom declaration of window object and functions globally. Then, we can confirm that the declared function works properly by defining it when the component that requires the function is mounted.

```
declare global {
  You, 5일 전 | 1 author (You)
  interface Window {
    AndroidBridge?: {
      openCameraWithQuery: (query: string) => void;
    };

    handleProductInfo: (productName: string, expirationDate: string) => void;
    handleProductInfoError: (errorMessage: any) => void;
  }
}
```

```
const openCamera = () => {
  if (
    window.AndroidBridge &&
    typeof window.AndroidBridge.openCameraWithQuery === "function"
  ) {
    window.AndroidBridge.openCameraWithQuery(String(roomId));
  } else {
    console.error("AndroidBridge is not available.");
  }
};

useEffect(() => {
  if (typeof window !== "undefined") {
    if (window.AndroidBridge) {
      window.handleProductInfo = (productName, expirationDate) => {
        console.log(productName);

        handleChangeFoodName(productName);
        setIsError(false);
        setIsLoading(false);
      };
      window.handleProductInfoError = (errorMessage) => {
        console.log("errorMessage", errorMessage);
        setIsError(true);
        setErrMessage(`Barcode error: ${errorMessage}`);
        setIsLoading(false);
      };
    } else {
      console.error("AndroidBridge is not available at useEffect.");
    }
  }
}, []);
```

Figure 7: Declaration of global window Object and Usage of Android Studio - Next.js Bridge

3.3 Backend Detail

3.3.1 Implemented API list

- CRUD(Create, Read, Update and Delete) for Food, Chore, Event, Finance(Account, Transaction, Finance plan), Member, Room
- Fetching Finance & Home info

3.3.2 ERD & Java Class/Interface Diagram

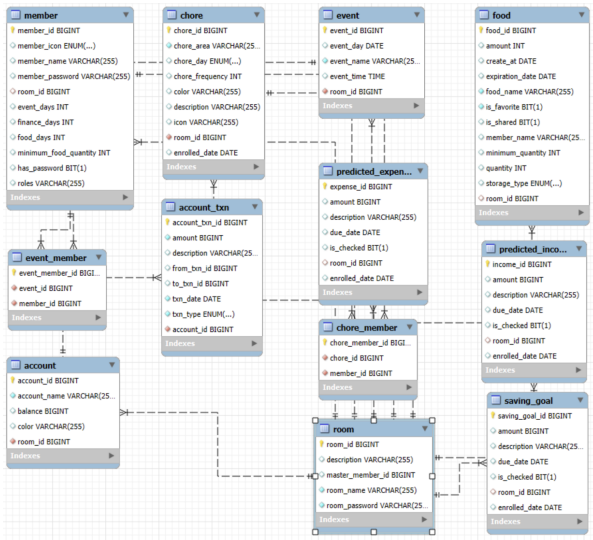

Figure 8: ERD(Entity-Relationship Diagram)

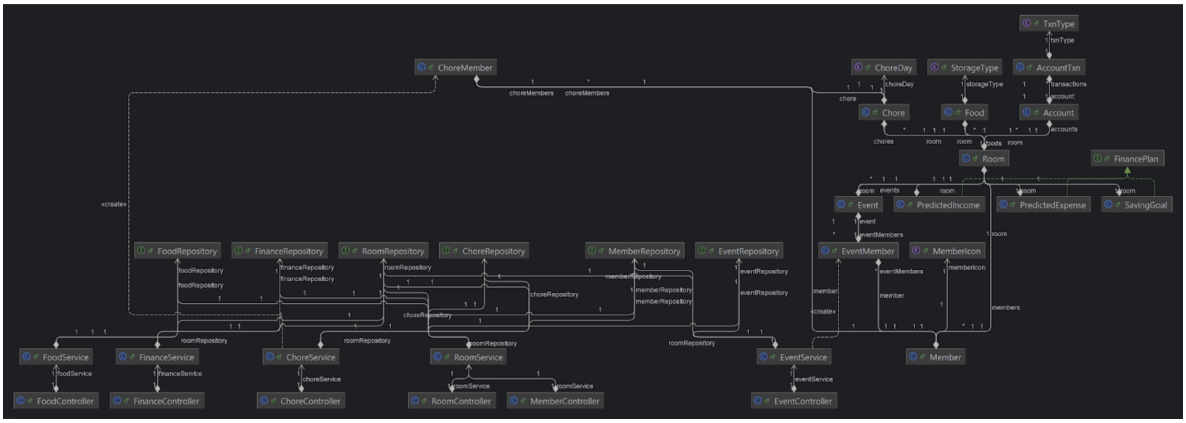

Figure 9: Java Class/Interface Diagram

3.4 Barcode Scan Detail

3.4.1 Connection between Web and App

The WebApp Interface provides a method that can be called in JavaScript.

```
class WebAppInterface(private val activity: MainActivity) {
        @JavascriptInterface
        fun openCameraWithQuery(query: String) {
            activity.checkPermissions() // 권한 확인 후 카메라 열기
        }
    }
```

Figure 10: Code for WebAppInterface

3.4.2 Check and request permission to read images with the camera

Check the camera and image read permission for barcode scanning. If you do not have the permission, request permission through the requestPermissions(). If the request code is REQUEST_CODE_PERISSIONS, open the camera through openCAMERA().

```
private fun checkPermissions(){
        val cameraPermission = ContextCompat.checkSelfPermission(this, android.Manifest.permission.CAMERA)
        val imagePermission = if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.TIRAMISU) {
            ContextCompat.checkSelfPermission(this, android.Manifest.permission.READ_MEDIA_IMAGES)
        } else {
            PackageManager.PERMISSION_GRANTED
        }
        if(cameraPermission!= PackageManager.PERMISSION_GRANTED ||
            imagePermission != PackageManager.PERMISSION_GRANTED) {
            Log.d("permission","request_permission");
            val permissions = mutableListOf(android.Manifest.permission.CAMERA)
            if (imagePermission != PackageManager.PERMISSION_GRANTED) {
                permissions.add(android.Manifest.permission.READ_MEDIA_IMAGES)
            }
            ActivityCompat.requestPermissions(this, arrayOf(android.Manifest.permission.CAMERA,
android.Manifest.permission.READ_MEDIA_IMAGES),REQUEST_CODE_PERMISSIONS)
        }
        else{
            Log.d("permission","already_permitted");
            openCamera()
        }
    }
```

Figure 11: Code for checkPermissions - 1

```
    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
        if (requestCode==REQUEST_CODE_PERMISSIONS){
            if(grantResults.isNotEmpty() && grantResults.all {it==PackageManager.PERMISSION_GRANTED}){
                Log.d("permssion","check_permission")
                openCamera()
            }
            else{
                Toast.makeText(this,"카메라 권한이 필요합니다", Toast.LENGTH_SHORT).show()
            }
        }
    }
```

Figure 12: Code for checkPermissions - 2

3.4.3 Run the camera application

Create the intent to open the camera through the intent (MediaStore.ACTION_IMAGE_CAPTURE). If the camera app is installed, import the external cache directory and create a temporary file, then import the URI of the file generated through getUriForFile(). The camera app will then run after adding the URI to save the photo and the flag that allows the camera to write to the URI.

```
 private fun openCamera() = runOnUiThread {
       val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
       if (intent.resolveActivity(packageManager) != null) {
           externalCacheDir?.let { cacheDir ->
               val photoFile = File.createTempFile("photo_", ".jpg", cacheDir)
               photoUri = FileProvider.getUriForFile(this,
"${packageName}.fileprovider", photoFile)
               intent.putExtra(MediaStore.EXTRA_OUTPUT, photoUri)
               intent.addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
               takePictureLauncher.launch(intent)
           }
       } else {
           Toast.makeText(this, "카메라를 사용할 수 없습니다.",
Toast.LENGTH_SHORT).show()
       }
   }
```

Figure 13: Code for openCamera

3.4.4 Processing results after taking a photo

To process the Intent result of the camera, the ActivityResultLauncher is registered, and when a photo is taken from the camera, uploadImage() is called and the picture is sent to the server. uploadImage() generates an HTTP POST request using the OkHttp client and sends the image in multipart format. If it processes the server's response and is successful, it parses the JSON response, extracts the product name and storage method, and delivers it to the JavaScript function on the web page.

```
takePictureLauncher =

registerForActivityResult(ActivityResultContracts.StartActivityForResult()) {
result ->
               if (result.resultCode == Activity.RESULT_OK) {
                   uploadImage(photoUri)
               }
           }
```

Figure 14: Code for registering ActivityReusltLauncher

```
  private fun uploadImage(uri: Uri){
      val client = OkHttpClient()
      val inputStream = contentResolver.openInputStream(uri)
      val file = File.createTempFile("uploaded_",".jpg",externalCacheDir)
      inputStream?.use { input ->
          FileOutputStream(file).use { output->
              input.copyTo(output)
          }
      }
      val mediaType = "image/jpeg".toMediaType()
      val requestBody = MultipartBody.Builder()
          .setType(MultipartBody.FORM)
          .addFormDataPart("file",file.name, file.asRequestBody(mediaType))
          .build()
      val request = Request.Builder()
          .url("https://1269kdr6v9.execute-api.ap-northeast-2.amazonaws.com/dev/scan")
          .post(requestBody)
          .build()
      client.newCall(request).enqueue(object: Callback {
          override fun onFailure(call: Call, e: IOException) {
              val responseBody = e.message
              println("Response: $responseBody")
              runOnUiThread {
                  webView.evaluateJavascript(
                      "javascript:handleProductInfoError('$responseBody');",
                      null
                  )
              }
              e.printStackTrace()
          }
```

Figure 15: Code for uploadImage - 1

```
            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    val responseBody = response.body?.string()
                    println("Response: $responseBody")
                    //JSON 파싱하기
                    responseBody?.let {
                        val jsonObject = JSONObject(it)
                        val productName = jsonObject.getString("product_name")
                        val expirationDate = jsonObject.getString("expiration_date")
                        val storageOptions = listOf("상온", "실온", "냉장", "냉동")
                        val storageMethod = storageOptions.find { expirationDate.contains(it)} ?: ""
                        println("Product Name: $productName")
                        runOnUiThread {
                            webView.evaluateJavascript(
                                "javascript:handleProductInfo('$productName', '$storageMethod');",
                                null
                            )
                        }
                    }
                }
                else{
                    // 에러 응답 본문 읽기
                    val responseBody = response.body?.string()
                    println("Response: $responseBody")
                    responseBody?.let {
                        val jsonObject = JSONObject(it)
                        val errorMessage = jsonObject.getString("error")
                        runOnUiThread {
                            webView.evaluateJavascript(
                                "javascript:handleProductInfoError('$errorMessage');",
                                null
                            )
                        }
                    }
                }
            }
        })
    }
```

Figure 16: Code for uploadImage - 2

4 Limitation and Discussion

4.1 Limitation

4.1.1 Incomplete Implementation of Certain Views and Features Due to Time Constraints

The *Housit* project generated numerous innovative and useful ideas during the ideation and feature refinement process. However, designing and implementing all the planned features within the limited timeframe proved challenging.

The final design included the following UI components: 17 screens for room and profile creation, 3 home screens, 10 settings screens, 18 screens for food management, 43 for finance, 6 for chores, and 6 for scheduling, amounting to a total of 103 screens. Within the 10-week period allocated for development, the frontend successfully implemented approximately 95% of the views, while the backend achieved full implementation of all APIs.

To ensure efficiency, the team prioritized essential features and components necessary for managing household information and completed their development for beta testing. Screens for adding and editing items such as food, schedules, cleaning areas, and account transactions were designed to reuse components wherever possible, as their underlying principles were similar. However, the detailed logic for data registration varied significantly, requiring extensive effort in the frontend towards the end of the project.

The following features were left for future development:

- Food: “Like” feature for food items, sorting, and filtering options.
- Finance: Repeating or fixed budget plans on a monthly cycle, and currency selection (e.g., USD, KRW).
- Chore: Logic for monthly repeated chore tasks.
- Setting: Member removal feature and member permission assignment feature.

4.2 Discussion

4.2.1 Integration with Real Bank Accounts

During the project, feedback was occasionally received regarding the finance feature’s shared account functionality. Suggestions were made to integrate the app with real bank accounts, such as those offered by KakaoBank or finance management apps like Convenient Ledger and YouPlanner, to automatically import transaction data (e.g., expenses, income, transfers) into the app.

However, from the early stages of planning, the team agreed that one of the most critical reasons for using a finance management app was to manually record expenses and income to gain a better understanding of financial inflows and outflows, as well as the current financial state. While automation offers convenience, if users fail to recognize where and how much they are spending due to fully automated records, the utility of

a finance management app diminishes. Consequently, we decided not to include the real account integration feature as it did not align with the app's intended purpose. Nevertheless, once the app is developed, distributed, and gains a significant user base, we plan to review market feedback and consumer responses to potentially add this feature as an optional extension in the future.

4.2.2 Balancing Easy Login and Security

There is an inherent trade-off between easy login and security. Simplifying the registration and login process increases the risk of data breaches and hacking, while emphasizing security can make the app and service less user-friendly and create barriers to entry.

The *Housit* housecare mobile app primarily handles information related to food, chores, finances, and shared schedules. It does not collect sensitive personal information such as account numbers, addresses, email addresses, or birth dates. Given this, we determined that reducing barriers to entry by simplifying the registration process was more beneficial. For room creation, users only need a unique room name and a 6-digit password, and for creating sub-profiles, users can set a free-form name with an optional 4-digit PIN.

This approach minimizes user inconvenience while maintaining adequate security. In fact, many financial institutions and even Apple use 6-digit PINs, demonstrating that a 6-digit password—especially when it includes letters and special characters—is sufficient for our service's security requirements before it enters widespread distribution and commercialization.

5 Related Work

5.1 Shared Living Application

5.1.1 Flatastic

Flatastic is a household management application that comprehensively manages tasks necessary for communal living. This app includes features for managing cleaning areas, expenses, shopping lists, and also has a built-in chat feature.

In the **Chores** section, users can create cleaning areas and assign them to each household member. It allows users to set cleaning cycles and sends notifications when the assigned day arrives. For **Expenses**, users can manually record and track spending, making it easy to see who paid for which event and how much. It also aggregates shared expenses and calculates how much a particular member needs to transfer to others. The **Shopping list** feature allows users to list and share items that need to be purchased, which can be checked in real-time. Users can create items and tag them with categories, allowing them to filter or sort the list by specific categories. Additionally, the app includes a **Chatting** feature, enabling communication among members without the need for a separate messenger, all within the integrated application.

With these features combined, *Flatastic* has become the most widely used service in the household management app market. However, it has some drawbacks, such as the lack of a food management feature essential for communal living, the inability to record account balances or budgets aside from expenses, and the limitation of not being able to add detailed descriptions for cleaning areas.

5.2 Food Management

5.2.1 유통기한 언제지 & BEEP

'유통기한 언제지(When does it expire?)' and ***BEEP*** are apps designed for refrigerator and expiration date management. Their key features include product registration through barcode scanning, receipt capture, and manual entry.

The apps send notifications based on the user-set date and time when an item's expiration date is approaching, which helps prevent food waste caused by expired goods. Additionally, a major advantage is that users can invite other members to manage the same virtual refrigerator in real time.

However, while it is possible to automatically add the name and quantity of products via barcode scanning and receipt capture, users still need to manually input the expiration date. If this feature were enhanced to automatically register expiration dates, the user experience would be significantly improved.

5.3 Finance Management

There are two main types of applications that help communal members manage their finances together: household accounting apps and digital banking apps. The following is an explanation of each type.

5.3.1 Financial Ledger Apps

Financial Ledger Apps apps help users develop financial management skills by manually or automatically recording expenses, budgets, and balances. In South Korea, the three most widely used household accounting apps are ***뱅크샐러드(Bank Salad)***, ***편한 가계부(Simple Household Ledger)***, and ***유플래너(Uplanner)***.

The core functions of Financial Ledger Apps apps include: managing shared accounts/bank books, manually recording and managing balances/budgets/expenses, linking to actual bank accounts, and automatically recording expenses when they occur.

Each of these applications offers different features, as illustrated in the comparison figure below.

Table 1: Feature Comparison of Financial Ledger Apps

Feature/Service	뱅크샐러드	편한 가계부	유플래너
Shared Account/Linked Accounts	△ (up to 2 members)	✗	○
Manual Record of Balance/Budget/Expenses	○	○	○
Bank Account/Card Integration	○	✗	○
Automatic Expense Registration	○	✗	○

One downside of ***뱅크샐러드(Bank Salad)*** is that the number of members who can share a joint account is limited to two. ***편한 가계부(Simple Household Ledger)*** lacks the feature to invite members to manage shared accounts. On the other hand, ***유플래너(Uplanner)*** is the only application that provides all of the above-mentioned functions.

However, we aim to implement this feature of communal finance management within our integrated app, allowing users to manage all aspects of communal living.

5.3.2 Digital Banking Apps

Digital banking apps enable users to manage accounts, make transfers, and apply for loans via mobile devices. In South Korea, an increasing number of banks(such as *KakaoBank*) are starting to support group banking, allowing members to share and manage the transaction history of joint accounts in real-time.

However, the critical drawback of digital banking apps is that they require an additional method to manage budgets separately from income/expenses.

5.4 Cleaning(Chores) Management

5.4.1 Tody

Tody is a to-do list app designed to manage cleaning routines. *Tody* primarily includes a feature to set and manage cleaning cycles. Additionally, it allows users to create specific cleaning areas and, within those areas, set detailed tasks that need to be completed. This makes cleaning a more intuitive and concrete activity.

An interesting feature of *Tody* is the visualization of dust as a character. Users can eliminate the dust by completing cleaning tasks, earning credits in the process, making it feel almost like playing a game. Another advantage of *Tody* is the ability to invite communal members to join and manage the cleaning tasks together.

However, since tasks are assigned in a way that allows users to earn credits by completing the cleaning themselves, it seems necessary to have a feature for assigning and distributing cleaning areas beforehand.

5.5 Event Management

5.5.1 카카오톡(KakaoTalk)

KakaoTalk is currently the most widely used messenger app in South Korea. *KakaoTalk* includes a group event feature that allows users to set dates, times, and notifications, as well as tag participants for the event. Additionally, it has a feature called **Talk Calendar**, where completed events can be viewed in a timeline format.

5.5.2 Google/Naver Calendar, TimeTree

Apps such as *Google Calendar*, *Naver Calendar*, and *TimeTree* offer shared calendar functionalities. These apps allow users to create and share calendars with family, friends, and colleagues, and manage tasks through the to-do function.

If we integrate these features from *KakaoTalk* and calendar apps into a unified household app, it would greatly enhance the ability to create, modify, manage, and track communal schedules.

6 Conclusion

The *Housit* project focused on designing and developing an integrated mobile application capable of managing various types of information generated in shared living environments. By providing essential features such as food management, finance tracking, chore management, and scheduling within a single platform, *Housit* successfully addressed the inconvenience of using multiple separate apps. Its intuitive UI and ease of use optimized the user experience, while enhanced organization, search, and sharing capabilities significantly improved the efficiency of shared living.

The development process followed a systematic approach from feature design to implementation. Starting with idea refinement and API specifications, the team proceeded with design tasks using Figma, frontend development based on Android Studio, and backend development using Spring and MySQL. One of the key features, barcode scanning, enabled users to input barcode information through the camera, retrieve product details via integration with the MFDS API, and seamlessly incorporate this functionality into the Android app. Despite the limited timeframe, the team collaborated effectively to implement over 95% of the overall design and successfully conducted beta testing.

Housit holds great potential to become a solution that maximizes the efficiency of shared living. Moving forward, advanced features such as sorting and filtering, recurring task logic, and real bank account integration will be developed based on user feedback. This project goes beyond merely recording and storing information, positioning *Housit* as a platform that effectively supports collaboration and communication in shared living environments, delivering even greater value to its users.

References

[Con24] World Wide Web Consortium. Web content accessibility guidelines (wcag), 2024. March 7, 2024.

[KIM18] Jongho KIM. 카카오톡, 국내 모바일 메신저 점유율 94%…10대 이용자는 소폭 하락. *아주 경제*, 2018.