Airbnb Automation with Airtable

Author: Yilin Xie, Jiayi Zhao

September 5, 2023

1 Motivation

Airbnb is a service that lets property owners rent out their spaces to travelers looking for a space to stay (section 2.1). Hosts who post hundreds of listings usually have a huge amount of reservations. It is hard for them to manage their properties. Specifically, when a customer checks out, the cleaning crew need to be notified on time to clean up all dirty rooms, or when the toilet is clogged, the maintenance team need to be notified on time to fix things up. The list goes on and on.

To simplify the analysis process, I implemented an automation system which periodically scrapes the data from Airbnb and helps with the analysis. The entire system can be divided into two parts: **AWS** (Amazon Web Service) and **Airtable** where the former is the backend section and the later is the UI/frontend section. All involved technology frameworks & platforms are introduced in section 2.

2 Short Intro on Platforms

2.1 Airbnb

Airbnb is an online marketplace that connects people who want to rent out their homes with people who are looking for accommodations in specific locations. Originally, Airbnb offered public APIs for developers to play around. However, as the number of users increasingly grew, Airbnb no longer provides public APIs but charges for their uses. It made individual property management harder than before and lowered the efficiency for hosts.

2.2 AWS

AWS is a cloud service offered by Amazon since 2002. It supports technology and programs that help to manage and monitor the environment to ensure performance and cost optimization, to manage security and downtime risks, and to automate remediation of issues to extent possible. In this project, since reservations data update frequently, it is necessary to keep the hosts updated. To do so, I put the major code onto an EC2 instance that runs on AWS, which scraps data from Airbnb and stores a backup to S3 and maintains the most recent data in a DynamoDB database (section 3.2).

2.3 Airtable

Airtable is an easy-to-use online platform for creating and sharing relational databases. The reason why I used it in this project is because most hosts on Airbnb have limited amount of knowledge in computer science, and simply scraping and storing the data in a database without offering a reader-friendly UI is meaningless to them. Airtable is more like an online version of Excel or a more powerful version of Google sheet as it supports data syncing among different tables, running scripts, and setting up automation based on various of triggers.

3 Design

In this section, I will elaborate more on the detailed design of the entire project. Specifically, it has the following three main parts: scripts logic, AWS architecture, and Airtable workflow.

3.1 Scripts

3.1.1 Selenium Behind the Scene

As mentioned before, Airbnb has banned public API access for a few years. Even with the credentials of a host account, it is still difficult to pass the authentication on the website. In the script of this project, I used a package called **Selenium**, which is a suite of tools for automating web browsers. In short, it simulates human behaviors through code, such as clicking a button, filling the input fields, submitting a request through a form provided in the website, etc.

With **Selenium**, the script locates the login form and credentials input fields by either **XPATH** or **CSS Selector** (built-in tools in chrome browser), and then inputs account info to and submit the form to authenticate. Once successfully logged in, the script navigates to the reservation page through URL *https://www.airbnb.com/hosting/reservations* (figure 1).

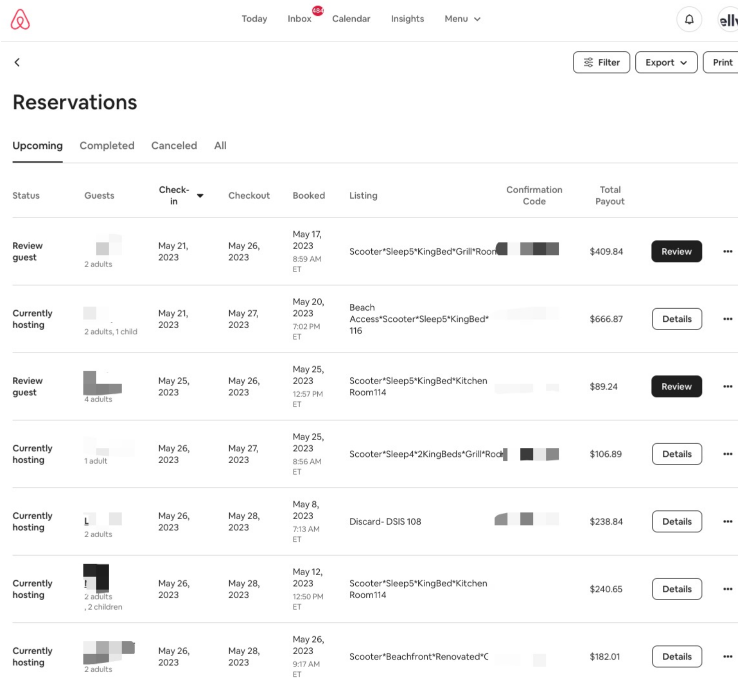

Figure 1: The reservation page in a hosting account on Airbnb. All sensitive info is hidden (the first hidden column are the guests names and the second are confirmation codes).

Note that during the actual execution of the script, the above demo page will not display because the script initialize the web driver as **”headless”**, which is an optional setting offered in **Selenium**. With ”headless” mode on, the user will not be able to see a real browser running

but instead, the executable simulated browser runs behind the scenes. To extract the reservation data needed, the script uses another attribute in the **WebDriver** object: **driver.page_source**, which contains the HTML source code in string format. The extracted data will be stored in a Json/dict object for later usage (section 3.2). An example of the extracted data is as follows:

```
{
   "LBO-xxx": {
      "internal name": "LBO-xxx",
      "reservations": {
         "confirmation": ["code_0", "code_1", "code_2", ...],
         "status": ["status_0", "status_1", "status_2", ...],
         "check in": ["date_0", "date_1", "date_2", ...],
         "check out": ["date_0", "date_1", "date_2", ...],
         "total payout": ["0.0", "1.0", "2.0", ...]
      }
   },
   ...
}
```

Listing 1: Part of the extracted data from the upcoming reservation page.

Internal names & Listing names: Each item in the demo data dict above corresponds to one or more entries in the reservation page, as one property might have multiple reservations which have different check-in and check-out dates. Note that the keys in the dict are different from the "Listing" in the reservation page. This is because Airbnb lets hosts to have two names for each property: listing name that contains detailed information of a property and only shown to travelers/guests, internal name that is shorter for manage and only visible to hosts themselves. All the internal names are scraped using the same logic from another listing page and pre-processed in this script (figure 2).

```
{
    'Discard- Room-135': 'LBO-135',
    'BeachFront*2 full*WIFI*Security*Laundry*Room-127': 'LBO-127',
    'BeachFront*2 Full*WIFI*Security*Laundry*Room-106': 'LBO-106',
    'Discard-Room-126': 'LBO-126',
    'BeachFront*1 King*WIFI*Security*Laundry*Room-103': 'LBO-103',
    'Discard- Room-132': 'LBO-132',
    'BeachFront*1 King*WIFI*Security*Laundry*Room-116': 'LBO-116',
    'Discard- Room-101': 'LBO-101',
    'StreetFront*2 Full*WIFI*Security*Laundry*Room-138': 'LBO-138',
    '2 Full*WIFI*Security*Laundry*Room-109': 'LBO-109',
    'Discard- Room-133': 'LBO-133',
    'Discard- Room-104': 'LBO-104',
    'BeachFront*2 Full*WIFI*Security*Laundry*Room- 110': 'LBO-110',
    'Discard- Room-131': 'LBO-131',
    ...,
}
```

Figure 2: A part of the listing and internal names map of one property.

3.1.2 Class Diagram

Refer to figure 3 for the script side class diagram. I divided the key functions in this project into 3 parts and wrapped them up into 3 classes (managers). Note there is no dependencies among these classes. In this project, AWS System manager is only used to store the credentials and to achieve the automation of periodical execution of the scripts. Therefore, it is not included in section 3.2.

WebDriverManager handles browser-related operations such as reading in HTML source code, extracting useful information, and storing the data as a field. **RecordManager** handles operations that integrates with DynamoDB which includes all valid CRUD HTTP requests. **Credentials-Manager** only talks to AWS System manager and extract account credentials from a sub-service **Parameter Store**.

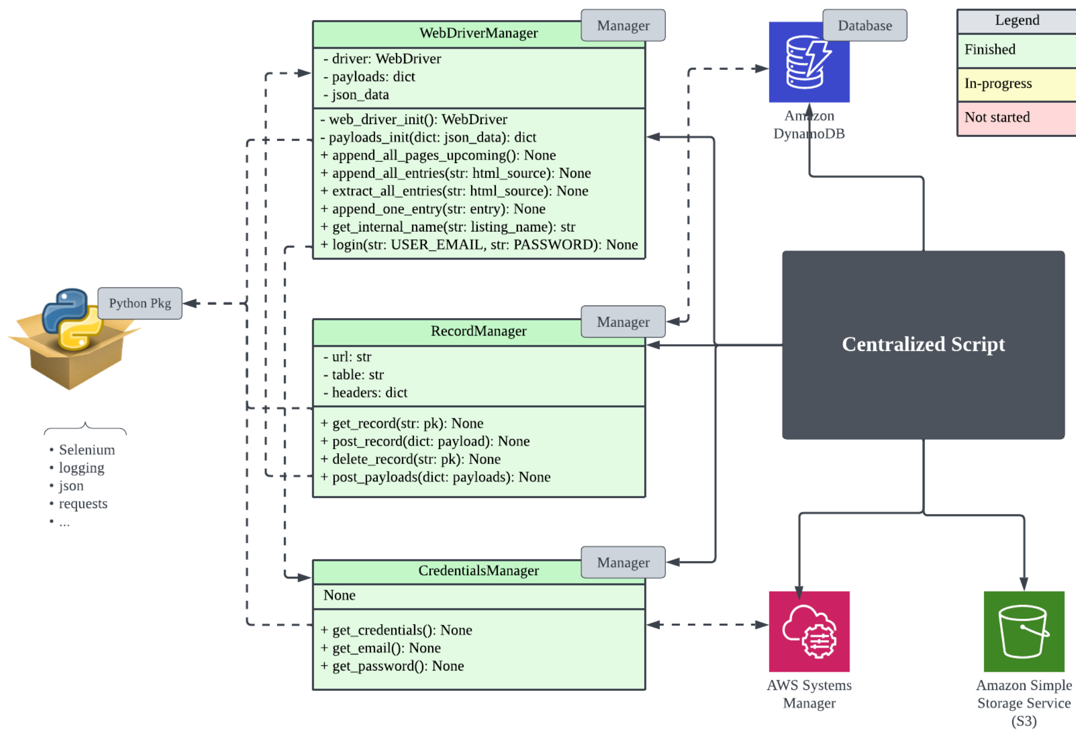

Figure 3: Script side class diagram.

3.2 Cloud - AWS

Since Selenium requires an executable browser installed in the environment, for simplicity, the script will be ran on an EC2 instance (section 3.1). Refer to figure 4, all the scripts in the previous sub-section are on instance t3.micro. Except all other components/services that are necessary for AWS accounts management and web security, the core services used in this projects are **EC2**, **Simple Storage Service (S3)**, **Gateway API**, and **Lambda**.

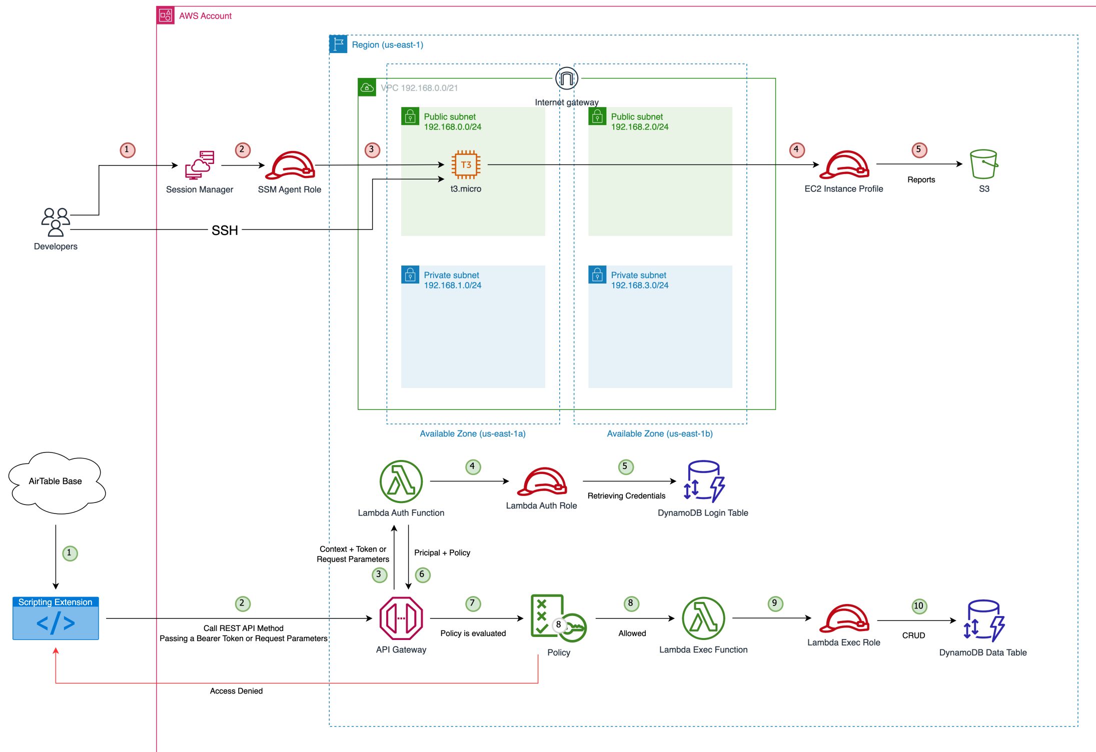

Figure 4: Application Architecture on AWS.

3.2.1 EC2

The selection of proper EC2 instance is based on the requirement of CPU usage of the scripts. In other words, if the host accounts have more properties that need to be managed, an EC2 instance of better performance is required. In this project, the total number of properties is around 200. Therefore, I chose to temporarily test on t3.micro, which is in the free tier of AWS.

Due to security issues, the EC2 instance is not open to public but only developers' IP address. To log into the instance, I can either use the session manager offered by AWS or through SSH with authentication key pair. Once successfully connected with the instance, all the dependencies in the *requirements.txt* can be installed (check github repository).

The EC2 instance in this project has full access to S3 and DynamoDB (section 3.2.2 & 3.2.3) which are two storage services used in this project. Specifically, besides directly posting real-time data into the database, the scripts also send a backup copy of **completed** and **canceled** reservations to S3 (figure 1).

3.2.2 S3

Simple Storage Service, also known as S3, is a cloud object service offered by AWS with scalability, data availability, security and performance. In this project, the main purpose of S3 is to store the backup of all reservation data, including **upcoming, completed,** and **canceled** reservations.

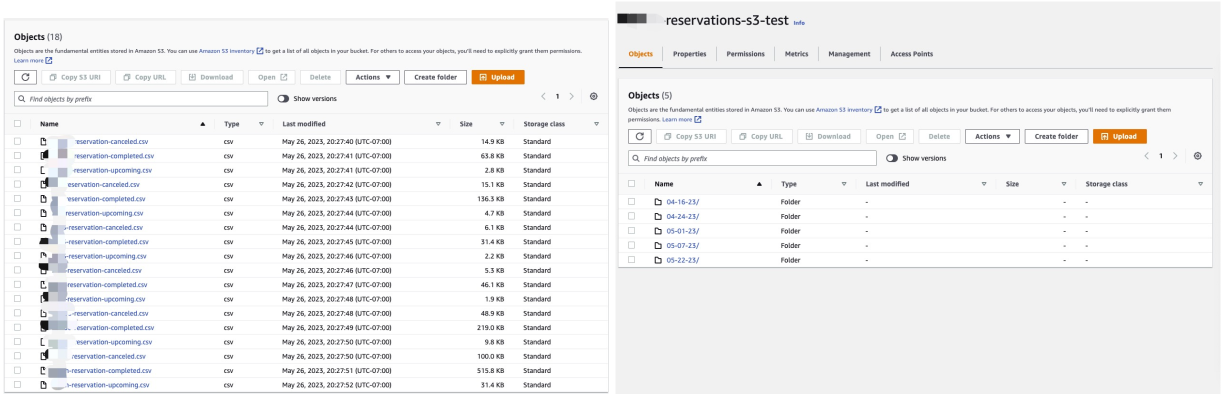

Figure 5: Backup records of reservation data in S3 (sensitive info hidden).

Besides storing backups, S3 in this project also stores the logs monitored by AWS CloudTrail, which is used to debug when the program crashes for unknown reasons.

3.2.3 DynamoDB

DynamoDB is a fully managed, serverless, key-value NoSQL databased designed to run high-performance applications at any scale. Instead of using a relational database, I chose DynamoDB because the scripts do not scrape all information displayed in the reservation page but only what are needed **so far**. In other words, the schema is various depending on the demands of hosts. As mentioned in section 3.2.1, DynamoDB mainly stores the most recent or real-time reservation data of the **upcoming section**. At each time when the scripts run, the upcoming reservations are send directly to DynamoDB through Gateway API and Lambda and overwrites the original data (section 3.2.4 & 3.2.5).

Note that the data in the database has similar structure to the extracted data (Code 1). Namely, the number of rows is fixed as long as the number of properties remains the same. The database is designed this way to reduce the number of items and speed up the searching efficiency. The variable in each item/row is the reservations. At each run, the properties which has no upcoming reservations will have empty field, and those which do will overwrites the original reservation column (Figure 6).

property	reservations
-120	[ ]
-136	[ ]
-138	[ ]
-105	[ ]
-139	[ ]
-108	[ ]
123	[{ "M" : { "total payout" : { "L" : [{ "S" : "386.79" }] }, "confirmation" : { "L" : [{ "S" : "…

Figure 6: Items in the AWS DynamoDB.

3.2.4 Gateway API

Amazon gateway API is a fully managed service that make it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. In this project, so far, there are two main APIs/endpoints: table_name and id (figure 7).

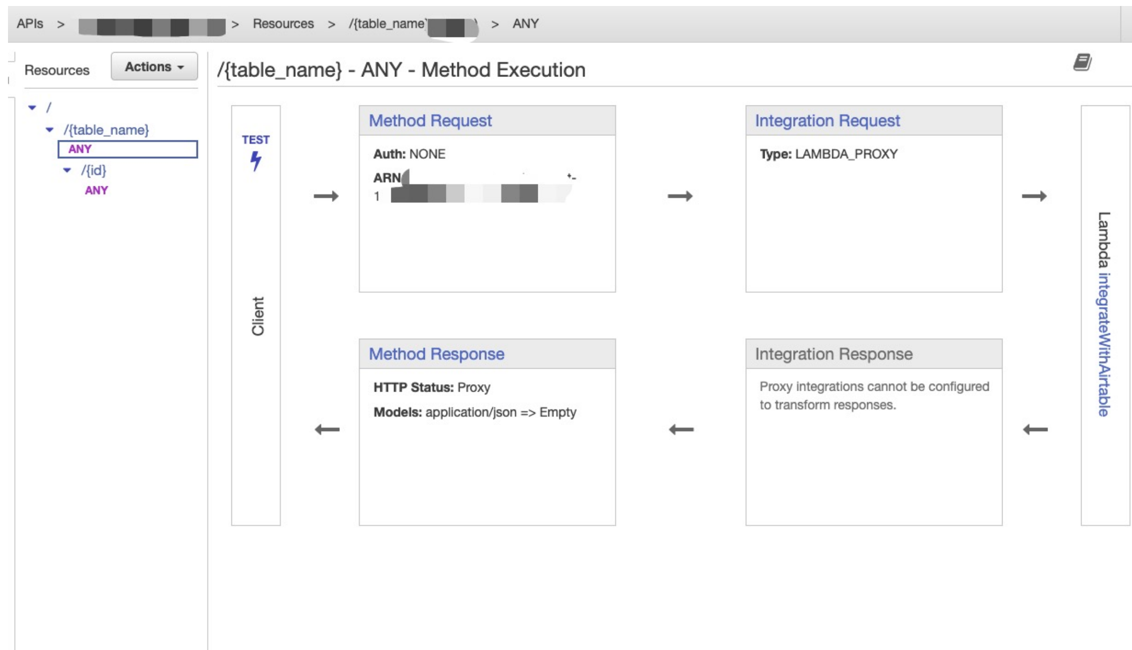

Figure 7: Amazon Gateway API: two major endpoints in this project.

According to the architecture in figure 4, Amazon gateway API in this project plays an important role in transferring data between other services and DynamoDB. Note that although ANY HTTP methods are supported with the table name specified, merely providing a table name will fail the sanity check in Lambda (section 3.2.5). In other words, HTTP requests such as GET all items in the database and POST multiple items into the database is not allowed. In terms of cost optimization, this design reduces the number of requests/search at each API call. On the other hand, from the perspective of security, such design prevents from malicious usage of the API under table name, which can increase the cost of gateway API service.

However, exploring the entire database is still achievable and will be needed under some circumstances. To do so, the developer will need to know the primary key of all the items in the database and use the API for id under the table name to get one item at a call. This is one more layer of restriction as mentioned in previous section (3.1), the primary key/internal name is hidden from public and only hosts can see it.

3.2.5 Lambda

AWS lambda is a serverless, **event-driven** compute service that lets developers run code for virtually any type of application or backend service without provisioning or managing servers. In this project, Amazon gateway API does not directly talk to the database but through triggering

function calls in lambda. Whenever an HTTP request is received, lambda functions will be notified and triggered to handle the corresponding type of requests (figure 8).

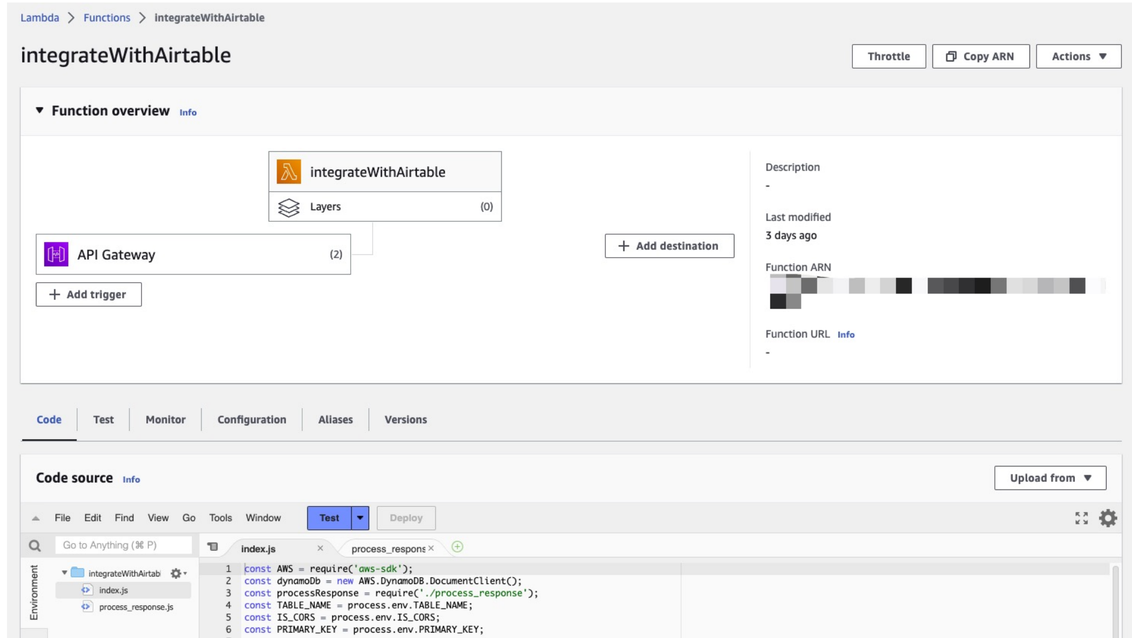

Figure 8: AWS Lambda: connected with Amazon Gateway API.

As mentioned in section 3.2.4, without specifying the table name and id will fail the sanity check, which is handled in lambda (Listing 2).

```
if (tableName === undefined) {
  let errorResponse = buildErrorMessage("Error: You're missing the table
      parameter");
  return processResponse(IS_CORS, errorResponse, 400);

}

if (!requestedItemId && event.httpMethod !== 'POST') {
  let errorResponse = buildErrorMessage("Error: You're missing the id
      parameter");
  return processResponse(IS_CORS, errorResponse, 400);
} else if (requestedItemId) {
  const key = {};
  key[PRIMARY_KEY] = requestedItemId;
  params.Key = key;
}
```

Listing 2: Sanity Check in Lambda

3.3 UI - Airtable

Airtable is the frontend section of the entire project. The structure of Airtable resembles that of a database. One Airtable account can have multiple **bases** and one base can have multiple **tables** (Figure 9). In addition, Airtable has built-in extensions which supports JavaScript and TypeScript. The integration with AWS is achieved through such extension and once data is imported, further analysis can be done with different extensions.

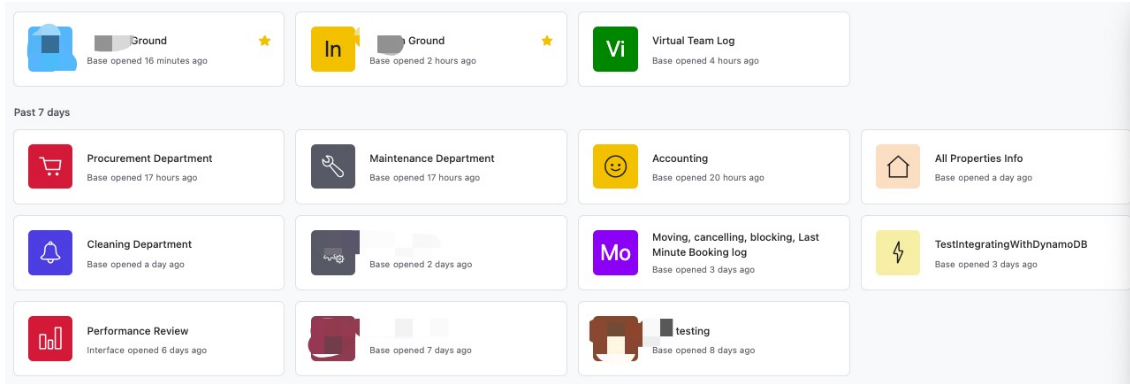

Figure 9: Airtable: some bases in which have tables.

The general workflow of Airtable is in figure 11. The only base/table that integrates with DynamoDB is the base **AllPropertiesInfo**. From **AllPropertiesInfo**, all related data will be passed to other bases using the **sync** functionality offered by Airtable (Figure 10). The most important departments in this project are maintenance, cleaning, and procurement departments, whose data is calculated based on the most current upcoming reservations. In such way, the corresponding crew of those department can see which property/room is dirty, when will the next guest arrive, whether a property/room is available for maintenance, etc.

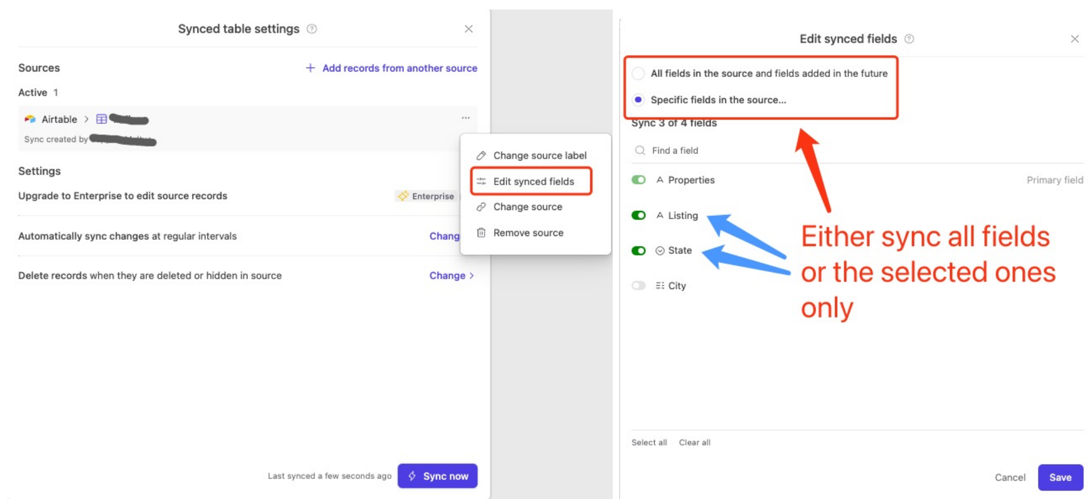

Figure 10: Sync function in Airtable which allows data from one base to be synchronized with that in another base.

Furthermore, the data in **AllPropertiesInfo** is grouped based on different regions and transferred to the corresponding base. This design is for better management as crews in different region should only have access to the data in their own region. Airtable supports account privilege management. The admin account can edit the readability of bases and only assign a group of subusers to view certain bases.

Other departments/bases require less from **AllPropertiesInfo** or transfer data back to it. For instance, if a room is broke and the guest want to switch rooms, such operation can only be done manually. All such records are stored in another base and it updates the data in **AllPropertiesInfo**.

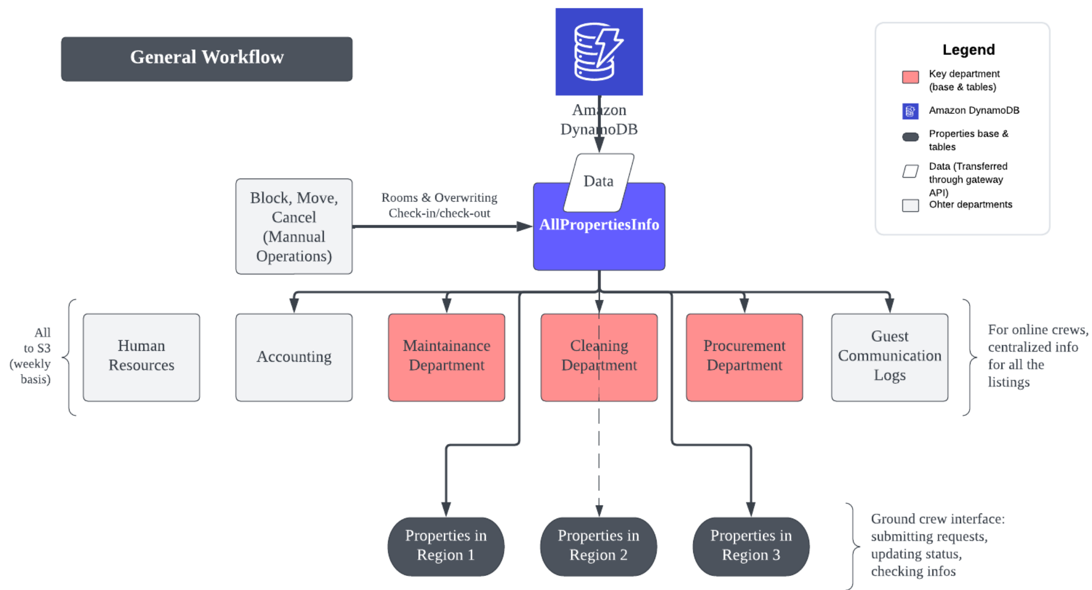

Figure 11: General workflow in Airtable.

4 Security

All crucial operations in this entire project are in AWS cloud. Therefore, maintaining a safe network on AWS is highly required. All used AWS services are granted with the least privilege and each service is assigned with specific roles and policies for them to talk to each other, which are all managed by AWS.

In terms of the key service EC2 in which the scripts run, there are only two ways to log into the instance: 1. through **Session Manager** offered by AWS; 2. through SSH with authentication key pem. Note that although port 22 is allowed for this EC2 instance, it is not open to public. One can only log into the instance under the restricted IP addresses. All other IPv4 and IPv6 portals are closed.

Besides the security of all the AWS services, storing the credentials of the host account of Airbnb has equal level of importance. In this project, instead of storing the user_account and password as variables in plaintext in the scripts, initially I set them as environment variables on

EC2 instance. When the script runs, it can access the credentials from the environment variables. This approach separates the credentials from the code and allows to manage them externally. Nevertheless, in such way, a more complex initialization script is required for each session of connection with EC2 because the environment variable will be gone once a session is terminated. Therefore, a more secure and efficient way to protect the login credentials is through the **Parameter Store** service under AWS System Manager.

5 User Benefits

This application/system is designed for property management companies or individuals. The main purpose/goal of it is to optimize the costs and improve the efficiency by replacing as much manual and humanoid operation with script automation. The automation here not only covers data scraping but also data analysis.

Originally, users have to login to Airbnb and click hundreds of times to view all the reservations and examine them one by one to filter useful information. It was easy to make mistakes while dealing with such a huge amount of data for a person. With the automation system, staff members no longer need to look into the reservation pages and all reservation data has been cleaned up and organized in a table where all useful information is obvious.

Moreover, hosts do not need to share their credentials with other staff or add co-host, which could possibly cause network intrusion, fraud, and identity stealing. The automation is secured under AWS cloud and only the hosts have full access whereas all other staff members' privilege is under control (read-only).

Although there are many mature and powerful property management systems on the market nowadays, they are typically expensive for start-ups or individuals and usually those systems are less scalable for customization. Comparing to them, this project achieves the same functionalities with a minimum cost. The extension in Airtable also enables the project to adapt to different use cases.

6 Feature Description

All functionalities of this project are divided into two subsection (6.1, 6.2).

6.1 Scripts

The core functionality of the scripts is to scrape data from Airbnb and update it to DynamoDB. Except the basic logic from section 3.1, to access the data needed, the very first thing to do is to pass the authentication of Airbnb. For machines that have never logged in an Airbnb account, it requires the user to verify their identiy by confirmation code through either phone-calls or SMS messages. In this project, all the accounts are binded with the same co-host who can receive the SMS confirmation code using Google Voice.

The script has **one and the only one** operation which is not automated and requires mannual

action: **inputting the verification/confirmation code** (Code. 3). For every account's confirmation/verification, Airbnb might randomly generate different pages. Some have the verification code separated as 6 blocks or dashes while others might have just one input field. Therefore, this section needs to be specially handled.

During developing, among all 7 accounts, so far I only found 3 different confirmation pages which are all handled in the code below. If none of them was found, I can trace the logs and add additional handlers for unseen pages.

```
def confirmation(driver, verification_code):
   """Handles the confirmation/verification of an account.

   Parameters
   ----------
   driver : WebDriver
      The web driver instance for interacting with web pages.
   verification_code : str
      The verfication code used to authenticate a user.
   """
   try: # 6 separated digits (square)
      for i in range(6):
         element_name = "airlock-code-input_codeinput_" + str(i)
         vc_input = driver.find_element(By.NAME, element_name)
         random_sleep(1, 1, 2)
         vc_input.send_keys(verification_code[i])
         print(f"Sending confirmation code {verification_code[i]}...")
      print(f"Done sending {verification_code}")
   except NoSuchElementException:
      try: # 6 separated digits (dash)
         for i in range(6):
            element_name = f"codeinput_{i}"
            vc_input = driver.find_element(By.NAME, element_name)
            random_sleep(1, 1, 2)
            vc_input.send_keys(verification_code[i])
            print(f"Sending confirmation code {verification_code[i]}...")
         print(f"Done sending {verification_code}")
      except NoSuchElementException:
         try: # not separated confirmation code
            vc_input = driver.find_element(By.ID, "airlock-code-input")
            random_sleep(1, 1, 2)
            vc_input.send_keys(verification_code) # paste
            print(f"Pasting confirmation code {verification_code}...")
         except NoSuchElementException:
            logger.error(f"Fail to find correct form.")
            return
```

Listing 3: The confirmation handler in the script

Note that in the code above, there are some lines of **random_sleep** which essentially pause the program for a few seconds. The main reason of doing so is that excessive or suspicious automation behavior may trigger security mechanisms that result in unresponsiveness. Therefore, the script has to mimic human-like interaction patterns, to avoid rapid or excessive requests, and to comply

with any rate limits or CAPTCHA mechanisms of Airbnb.

After authentication, the script will direct the driver to the reservation page and start scraping data page by page. Refer to figure 12, the scraped data was send to the database and was accurate.

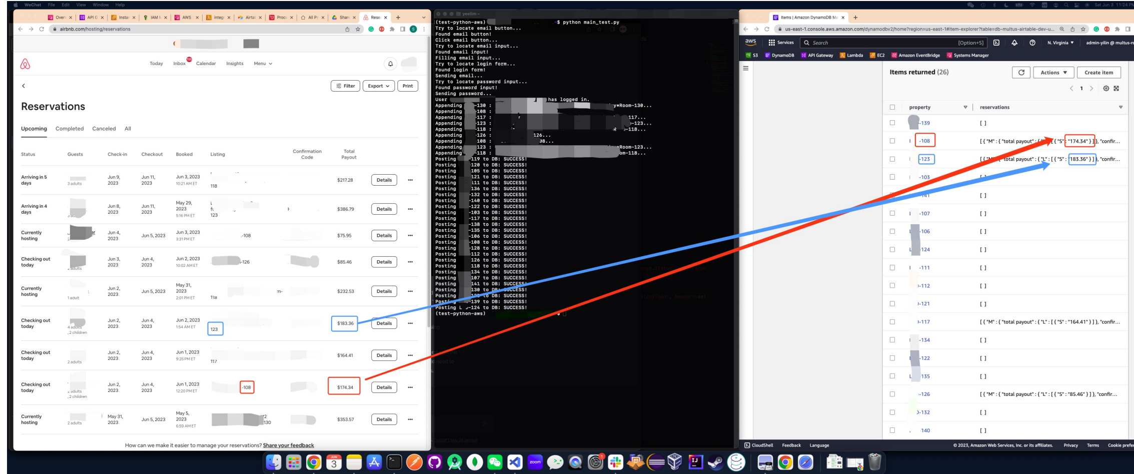

Figure 12: Demo: script scrapes data from Airbnb & send to AWS DynamoDB. On the left is Airbnb reservation page in a normal browser; The middle is the execution of the script on EC2; The left shows the items in AWS DynamoDB, which are just scraped using the script.

6.2 Airtable

In terms of the features in Airtable side, please refer to the use case diagram (Figure 13) and Airtable workflow (Figure 11). The following subsections provide a more thorough description.

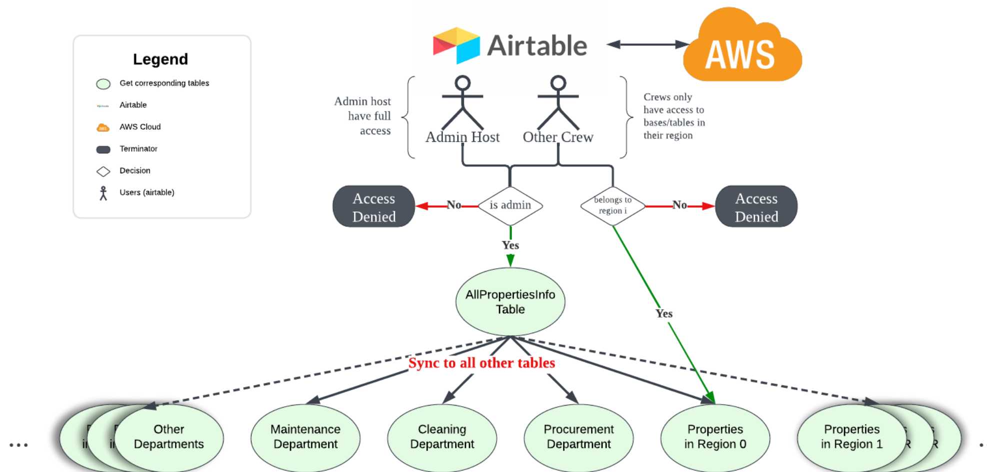

Figure 13: Use Case Diagram

6.2.1 AllPropertiesInfo

This base/table is the only table which integrates with AWS DynamoDB, and all the scraped data will be grab through its extension in JavaScript (Figure 14). Note that the table already has some fixed data such as state, occupancy status, and cleaning status, which will not be gained from database but analyzed using the data from database.

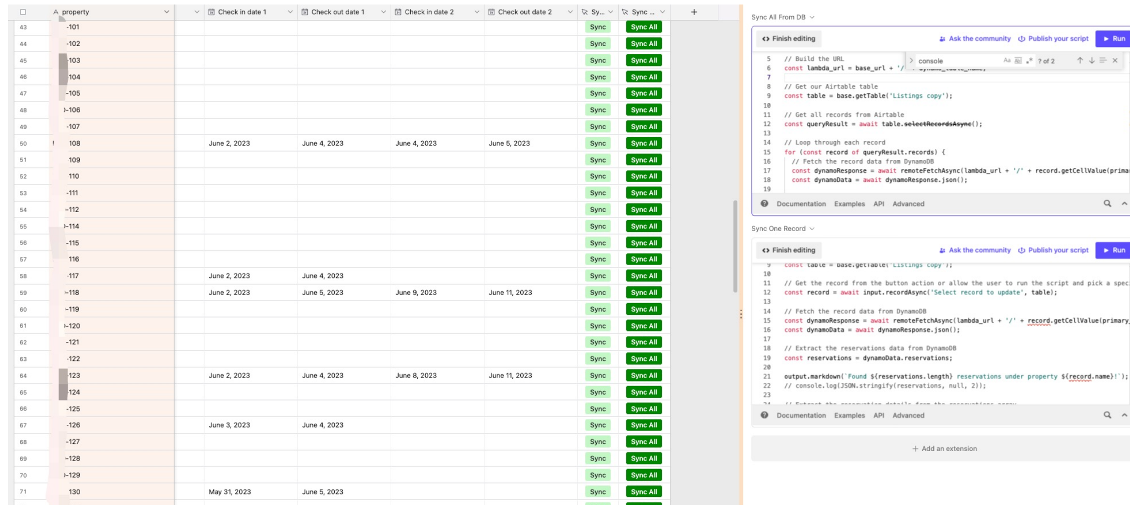

Figure 14: Data extracted using the extension (JavaScript).

The demo of Airtable has two buttons **"Sync"** and **"Sync All"**, which are used to update one property and update all properties on Airtable. These two buttons are linked to the extensions on the right, which is implemented in JavaScript.

6.2.2 Cleaning and Other Departments

Except **AllPropertiesInfo** base, all other bases and tables do not directly integrate with the backend scripts. The very crucial information from AllPropertiesInfo is the current occupancy status of a property and the earliest check-in date for the next guest. Those are used for cleaning crew and maintenance crew to prioritize their work. In terms of the communication among other bases and tables, Airtable provides a sync functionality which allow data to be transferred among different bases/tables.(Figure 15).

Figure 15: Tables in a base for a specific region R (all sensitive info is hidden). From left to right, the first is the occupancy status table in region R whose data is synced from AllPropertiesInfo, the second is the maintenance issue summarization whose "Occupancy" is also synced from AllPropertiesInfo and its "maintenance issue" will be synced to the third table, which is for maintenance crew to schedule inspection and do their work.

7 Descriptions of APIs

In section 3.2.4, although Gateway API has already specified all types of HTTP requests, due to security concerns and performance optimization, this project only supports the following API endpoints (basic CRUD), and all other requests will fail the sanity check in Lambda.

7.1 CREATE one record

Description: This API creates a new property item in the given table. Note that this API is not called by any script ran on EC2 but only in Airtable extension when hosts list a new property to public.
Route : GATEWAY_API_INVOKE_URL/table_name/pk
HTTP method: POST
Input: JSON object (pk & all needed information)
Output: HTTP Status code.
Return: Success/Error message.

7.2 GET one record

Description: This API returns the specific property together with its most recent reservation data. The admin user in Airtable updates **AllPropertiesInfo** table to retrieve the reservation data for further workflow.
Route : GATEWAY_API_INVOKE_URL/table_name/pk
HTTP method: GET
Input: NONE
Output: HTTP Status code.
Return: Returns the user with the property info.

7.3 UPDATE one record

Description: This API updates the reservation data of the given property.
Route : GATEWAY_API_INVOKE_URL/table_name/pk
HTTP method: POST
Input: None
Output: HTTP Status code.
Return: Success/Error message.

7.4 DELETE one record

Description: This API deletes one property item in the given table. Note that this API is not called by any script ran on EC2 but only in Airtable extension when hosts no longer open that particular property to public.
Route : GATEWAY_API_INVOKE_URL/table_name/pk
HTTP method: DELETE
Input: None
Output: HTTP Status code.
Return: Success/Error message.

8 Performance

The performance of the entire application to a great extent, depends on the response time of Airbnb because all ”root” operations need to wait for Airbnb to return the data to the browser. Besides, the newly updated Chrome (since version 113) has special checks for scripting and bots. Specifically, if the browser detects a quick fill of forms or quick scan of a large amount of pages, it will block further HTTP requests and disable all JavaScript in the static site. Therefore, to pass the check, I included some random sleep in the script, which potentially lowers the efficiency of the script but ensures the accuracy of the scraped data.

To speed up the scraping process, two browser options are set in Selenium WebDriver. First, by default, Selenium WebDriver simulates an actual execution of browsers through which developers can see a browser initiated and triggered by code. This ”real” execution requires more CPU and can be turned off, which means no browser will pop up and the execution happens behind the scene. Second, Selenium can be set to prevent from loading static files such as images, icons, etc, which also improves the responsive time of the site. Since the scripts run on the smallest EC2 instance, it is too resource-consuming to turn on the first option. Here is a comparison (CPU-wise) between the performance of one execution of the script with and without the second option being set up (Figure 16). It is obvious that disabling static files is more efficient.

Figure 16: CPU utilization on EC2: 3 executions: the first and second has the setting *'–blink-settings=imagesEnabled=false'* and the third execution enables static files in the browser.

9 In the Future

Based on the current functionalities of this automation, further analysis can be done with the data scraped. For instance, the number of dirty rooms can be calculated and synced to another base on Airtable for the cleaning department, such that the manager can better assign the ground crews.

In terms of the robustness and scalability of this application, as the number of properties enlarges, an instance with better computation power might be needed. On AWS, it is easy to migrate data through **images**. Since we have everything related to Airbnb stored on Parameter Store, whenever Airbnb updates their webpage, the main constants we need to modify is in the parameter store. In other words, the code stays the same.