
# CNG-495 Capstone Project Stage 3: Final Report of E-Commerce Application with ML Capabilities

**Cem KARASU** **2315398**

**Metehan YILMAZ** **2315638**

**1/9/24**

# Table of Contents

E-commerce Application with ML Capabilities ..... 3
Authentication ..... 3
Managing Items ..... 3
Adding An Item ..... 3
Getting All Items ..... 5
Getting Items by User ID ..... 6
Getting An Item by ID ..... 7
Deleting An Item by ID ..... 8
Getting Item Categories ..... 9
Image Rekognition ..... 10
Inventory ..... 12
AdminFunctions ..... 16
Getting All Users ..... 16
Deleting A User by ID ..... 17
dbFunctions ..... 18
Backuping All Databases ..... 18
Getting All The Created Backups of Databases ..... 18
Deleting A Backup By ID ..... 19
Utilized Cloud Services: ..... 20
Project Statistics ..... 21

## E-commerce Application with ML Capabilities

This project aims to create an e-commerce application where users can conveniently list their products by adding new items. AWS Cognito manages the authentications of each user and customer. Customers can search and add products to their cart. Payment microservice is not included in this project.

This application solves the problem which is faced during listing. It can be listed wrongly categorized due to human error. Our application eliminates this problem by automatically choosing the category for the product based on its image.

To achieve this, the application utilizes image recognition technology from Amazon Rekognition. When users upload an image to add a new item, Amazon Rekognition automatically identifies the object and selects the category for the product. This functionality streamlines the process of filling in item details by automatically populating certain fields based on previously configured specifications of the identified object.

Trendyol is an e-commerce application that is like our application. Main differences are we use image recognition for deciding a category automatically while Trendyol has more capabilities such as payment.

GitHub repository for the back end: https://github.com/ChesterMETU/e_commerce_CNG495.git

GitHub repository for the front-end UI: https://github.com/ChesterMETU/e_commerce_CNG495_ui.git

And you can find the old repository here: https://github.com/metumetehan/e-commerce-CNG-495.git

## Authentication

For authentication, we use AWS Amplify with AWS Cognito. Using Amplify Studio in AWS Console, first we created an Amplify application. Then, to setup authentication, we created a login mechanism and configured it to be logged in using email and password. and we created a sign-up mechanism and configured it to be registered using username, email and password. Then, for the front-end mobile application, we use three Flutter modules: amplify_flutter, amplify_auth_cognito, and amplify_authenticator. Using AWS command line interface, Amplify configurations are pulled using “amplify pull” command. Amplify Authenticator module is used for the UI of login and register.

## Managing Items

### Adding An Item

To add an item, we created a lambda function that is triggered by making a *PUT* request to the

following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items

In the request body, *name*, *price*, *piece*, *creater_id*, *product_description*, and *file* must be included. Here is an example body of a request:

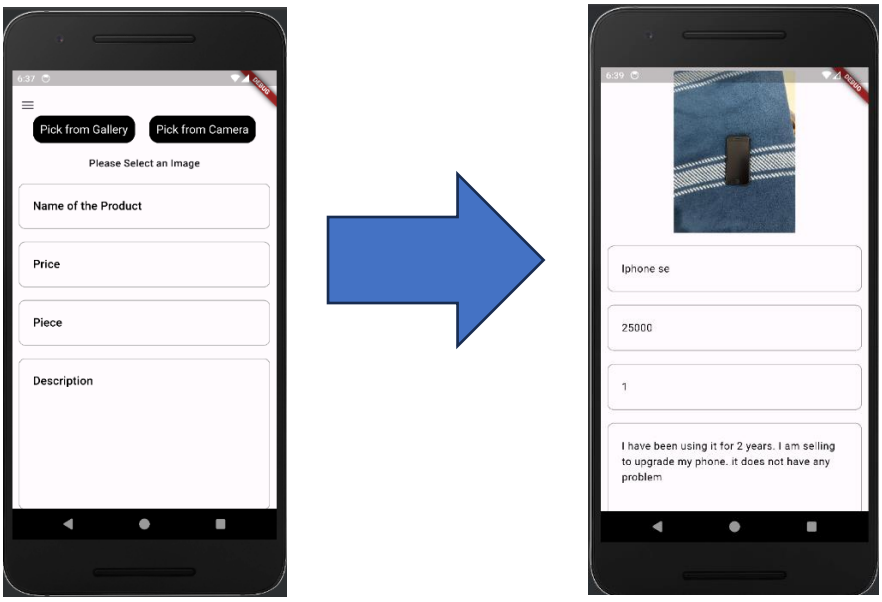
```
    "name": nameController.value.text,
    "price": int.parse(priceController.value.text),
    "piece": int.parse(pieceController.value.text),
    "creator_id": user_id,
    "product_description": descriptionController.text,
    "file": base64img
```

After the Lambda function is triggered, first we create a new item to the DynamoDB table called *item_table* with the attributes in the request body without image *file* and its category is initially set to unknown. After that, item image provided in the request body as base64 encoded image is decoded and put to S3 Bucket called *rekonimage* by setting key to the *item_id*. You can see the associated function below.

```
case "PUT /items":
        let requestJSON = JSON.parse(event.body);

        var IDcounter =  await dynamo.send(
          new GetCommand({
            TableName: counterTable,
            Key: {
              counter: "1",
            },
          })
        );

        IDcounter = IDcounter.Item.counterID;
        await dynamo.send(
          new UpdateCommand({
            TableName: counterTable,
            Key: {
              counter: "1"
            },
            UpdateExpression: 'set #c = :x',
            ExpressionAttributeNames: {'#c' : 'counterID'},
            ExpressionAttributeValues: {
              ':x' : IDcounter + 1,
            }
          })
        );

        await dynamo.send(
          new PutCommand({
```

```
      TableName: tableName,
      Item: {
        item_ID: `${IDcounter}`,
        price: requestJSON.price,
        piece: requestJSON.piece,
        name: requestJSON.name,
        creator_id: requestJSON.creator_id,
        image: requestJSON.file,
        description: requestJSON.product_description,
        category: "unknown"
      },
    })
  );
  const base64Image = requestJSON.file;
  const decodedImage = Buffer.from(base64Image,"base64");
  const command = new PutObjectCommand({
    Bucket: "rekonimage",
    Key: `${IDcounter}`,
    Body: decodedImage,
    ContentType: "image/jpeg"
  });
  const response = await S3.send(command);
  body = requestJSON.file;
  break;
```

*Figure 1 Adding an Item UI representation screenshots*




## Getting All Items

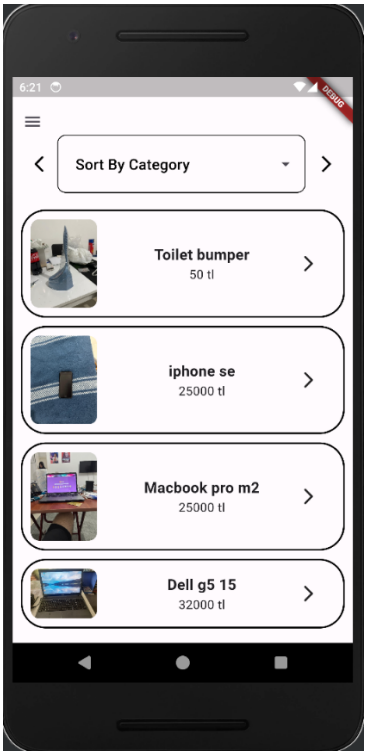
To get all items from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: [https://4151wpvtqb.execute-api.eu-central-](https://4151wpvtqb.execute-api.eu-central-)

1.amazonaws.com/items

After the Lambda function is triggered, first we send a dynamo scan command to the DynamoDB table called *item_table* with only the table name that is *item_table*. And it returns the list of all the items in the database.  Then, it is put in the response body and sent to the client. You can see the associated function below.

```
      case "GET /items":
        body = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        body = body.Items;
        break;
```

*Figure 2 Getting All Items UI implementation screenshot*




## Getting Items by User ID

To get items for a specific user, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/itemsByUser/{id}

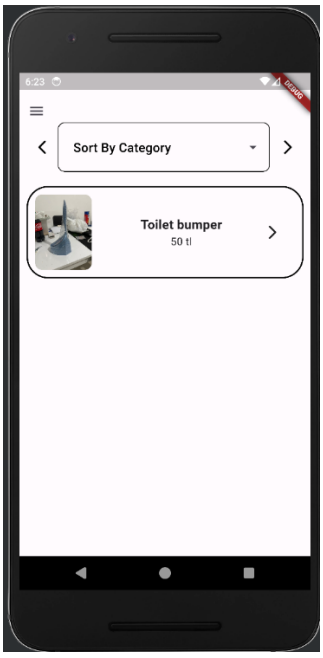
User id must be placed as a path parameter. Here is an example:

https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/itemsByUser/9f7423fe-80a4-45d3-8ac8-27701090ee97

After the Lambda function is triggered, first we send a dynamo *scan* command with a specific attribute (in this case *creator_id* which is taken from the path parameter in the request URL) to the DynamoDB table called *item_table*. And it returns the list of all the items created by the specified user. Then, it is put in the response body and sent to the client. You can see the associated function below.

```
case "GET /itemsByUser/{id}":
    body = await dynamo.send(
        new ScanCommand({
        TableName: tableName ,
        FilterExpression : 'creator_id = :id',
        ExpressionAttributeValues : {':id' : event.pathParameters.id}})
    );
    body = body.Items
    break;
```

*Figure 3 Getting Items by User ID UI implementation screenshot*




## Getting An Item by ID

To get an item from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/{id}

Item id must be placed as a path parameter. Here is an example:

https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/5

After the Lambda function is triggered, first we send a dynamo *scan* command to the DynamoDB table called *item_table* with the table name and key parameter which is the primary key of the *item_table* and it is taken from path parameter in the request URL. And it returns the list of all the items in the database. Then, it is put in the response body and sent to the client. You can see the associated function below.

```
      case "GET /items/{id}":
        body = await dynamo.send(
          new GetCommand({
            TableName: tableName,
            Key: {
              item_ID: event.pathParameters.id,
            },
          })
        );
        body = body.Item;
        break;
```

### Deleting An Item by ID

To delete an item from the database, we created a lambda function that is triggered by making a *DELETE* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/{id}

Item id must be placed as a path parameter. Here is an example:

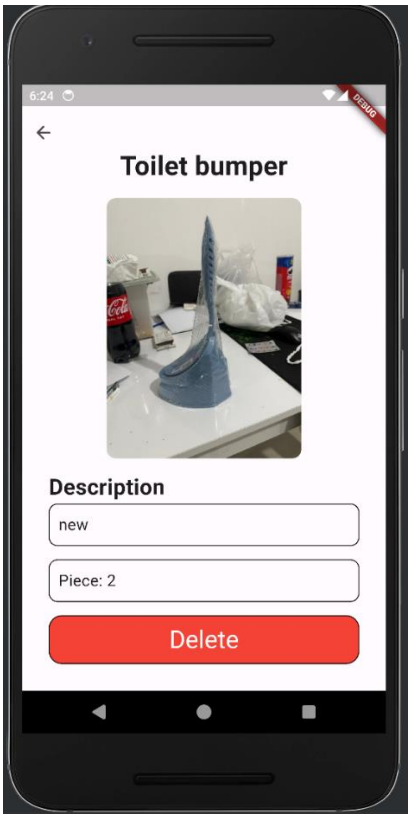
https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/items/5

After the Lambda function is triggered, first we send a dynamo *delete* command to the DynamoDB table called *item_table* with the table name and key parameter which is the primary key of the *item_table* and it is taken from path parameter in the request URL. Then, the id is put in the response body and sent to the client. You can see the associated function below.

```
      case "DELETE /items/{id}":
        await dynamo.send(
          new DeleteCommand({
            TableName: tableName,
            Key: {
              item_ID: event.pathParameters.id,
            },
```

```
        })
      );
      body = `Deleted item ${event.pathParameters.id}`;
      break;
```

*Figure 4 Deleting An Item UI implementation screenshot*




## Getting Item Categories

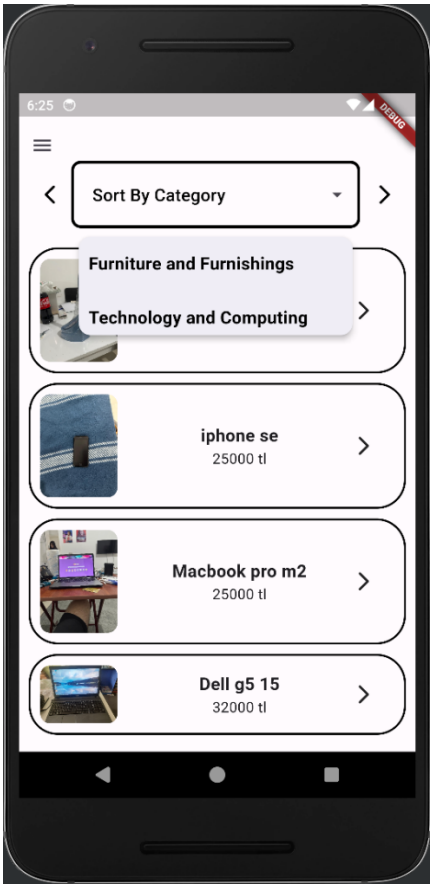
To get all the item categories from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://4151wpvtqb.execute-api.eu-central-1.amazonaws.com/itemcategorys

After the Lambda function is triggered, first we send a dynamo *scan* command to the DynamoDB table called *item_table* with the table name. And it returns the list of all the items in the database. Then, all the item categories in the list are put into a dictionary and distinct categories are send to the client via the response body. Here is the associated function:

```
      case "GET /itemcategorys":
```

```
        var items = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        items = items.Items;
        const distinctValues = {};

        items.forEach(item => {
          const attributeValue = item["category"];
          distinctValues[attributeValue] = true;
        });

        body = Object.keys(distinctValues);
        break;
```

*Figure 5 Getting Item Categories UI implementation screenshot*



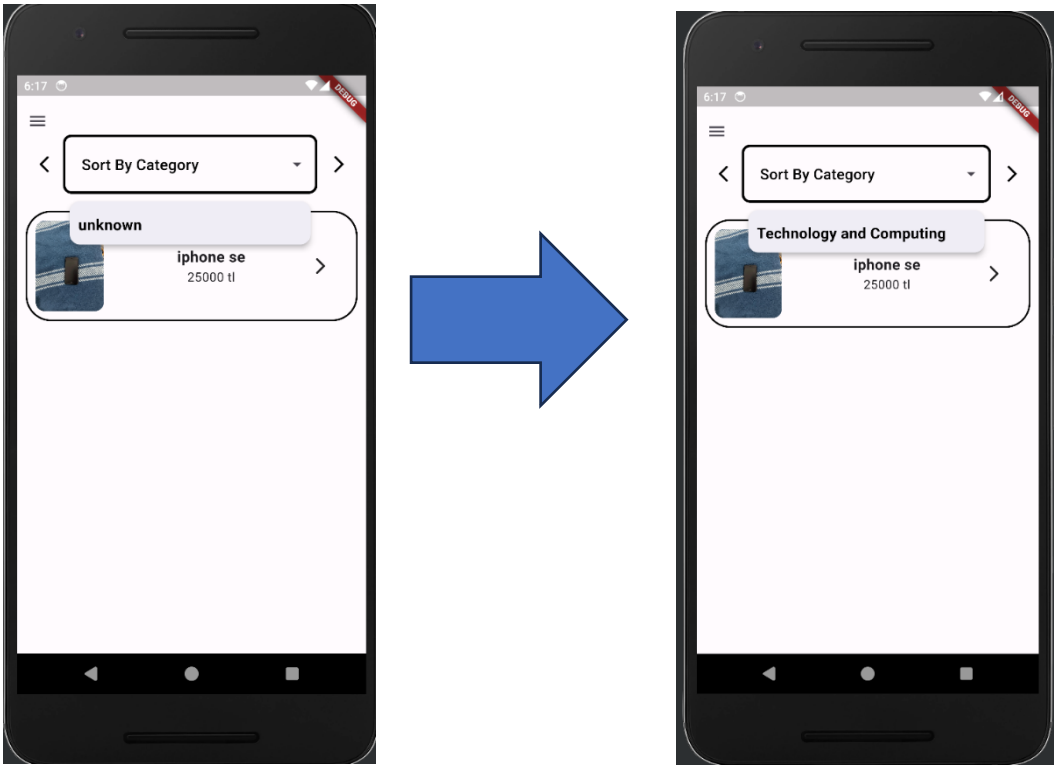

## Image Rekognition

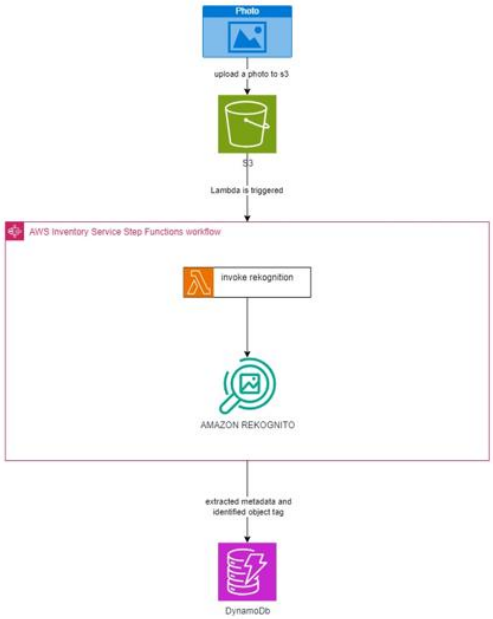
To decide the item category of an item, we created a lambda function that is triggered whenever an item is put into the S3 Bucket called *rekonimage*. First, the function gets bucket name and key of the item from the event header. Using those parameters, the function gets the image from S3 Bucket and then it is sent to AWS Rekognition. AWS Rekognition returns the data that we can get the item category from.

Then, the function updates the item category in the database by sending *update* command with the table name which is *item_table* and key which is S3 item key.

```
bucket = event['Records'][0]['s3']['bucket']['name']
key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
encoding='utf-8')
print(bucket,key);
try:
    result = rk.detect_labels(
    Image={'S3Object':{'Bucket':bucket,'Name': key},},
        MaxLabels=1,
        Features=[
    'GENERAL_LABELS',
    ],
        )
    print(result)


    update_response = db.update_item(
        TableName= "item_table",
        Key={"item_ID": {
            "S": key
            }
        },
        ExpressionAttributeNames={
            '#C': 'category',
        },
        ExpressionAttributeValues={
            ':c': {
                "S": result["Labels"][0]["Categories"][0]["Name"],
            },
        },
        UpdateExpression='SET #C = :c',
        ReturnValues="UPDATED_NEW",
    )

    return "Success"
except Exception as e:
    print(e)
    print('Error getting object {} from bucket {}. Make sure they exist and your bucket
is in the same region as this function.'.format(key, bucket))
    raise e
```

*Figure 6 UI representation of Image Rekognition represented by 2 screenshots*




## Inventory

In the application, there is no payment requirements. Therefore, relevant functionalities are not

implemented and to simulate the purchase we created lambda function. It is triggered by making a *POST* request to the following API endpoint: https://6v11idz44b.execute-api.eu-central-1.amazonaws.com/completepurchase

Here is an example of a request body:

```
{
    "items": [
        {
            "id": 2,
            "piece": 1,
            "creator_id": asdg432aggg4a
        },
        {
            "id": 3,
            "piece": 2,
            "creator_id": asdg432aggg4a
        }
    ]
}
```

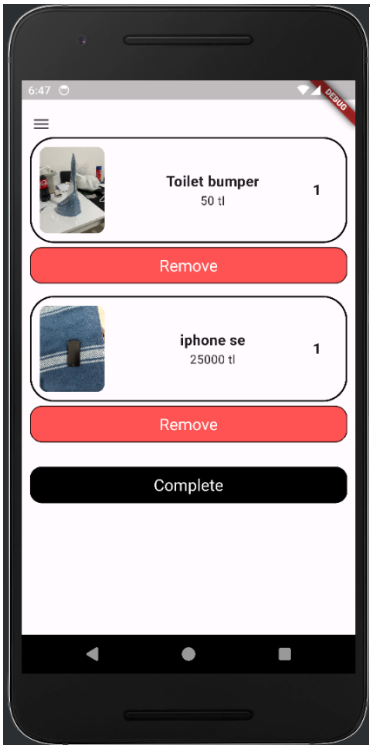
After the Lambda function is triggered, first we check each item whether there is enough amount, if all items have proper amount then database is updated for each item of each piece in the DynamoDB. If any of the items hits to zero amount then it is deleted from the database. Finally, one of the following response messages is sent “purchase successful” or “purchase unsuccessful”. Here is the associated function:

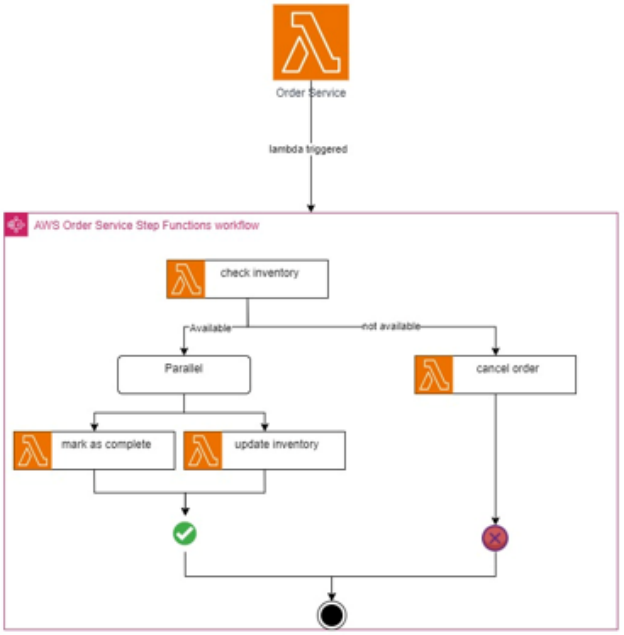
```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  GetCommand,
  DeleteCommand,
  UpdateCommand,
} from "@aws-sdk/lib-dynamodb";
import { S3Client,PutObjectCommand } from "@aws-sdk/client-s3"
const S3 = new S3Client({});

const client = new DynamoDBClient({});

const dynamo = DynamoDBDocumentClient.from(client);

const tableName = "item_table";

export const handler = async (event) => {
  let requestJSON = JSON.parse(event.body);
  let body = "Complete";
```

```
let statusCode = 200;
const headers = {
  "Content-Type": "application/json",
};
let error = 0;

async function asyncForEachGet(items,callback) {
  for(let i = 0;i<items.length;i++) {
    let data;
    data = await dynamo.send(
    new GetCommand({
        TableName: tableName,
        Key: {
          item_ID: items[i].id,
        },
      })
    );

    data = data.Item;
    if(data.piece < items[i].piece) {
      error = 1;
      body = 'The ${data.name} does not meet the quantity you selected';
    }
  }
}

async function asyncForEachGetOp(items) {
  for(let i = 0;i<items.length;i++) {
    let data = await dynamo.send(
        new GetCommand({
          TableName: tableName,
          Key: {
            item_ID: items[i].id,
          },
        })
      );
      data = data.Item;
      if(data.piece === items[i].piece) {

        await dynamo.send(
          new DeleteCommand({
            TableName: tableName,
            Key: {
              item_ID: items[i].id,
            },
          })
        );
      } else {
```

```
          await dynamo.send(
            new UpdateCommand({
              TableName: tableName,
              Key: {"item_ID": items[i].id},
              ExpressionAttributeNames: {
                  '#P': 'piece',
              },
              ExpressionAttributeValues: {
                  ':p': data.piece - items[i].piece
              },
              UpdateExpression: 'SET #P = :p',
              ReturnValues: "UPDATED_NEW",
            })
          );
        }
    }
  }
  try {
    await asyncForEachGet(requestJSON.items);
    if(error != 1) {

      await asyncForEachGetOp(requestJSON.items);
    }

  } catch (err) {
    statusCode = 400;
    body = err.message;
  } finally {
    body = JSON.stringify(body);
  }
  return {
    statusCode,
    body,
    headers,
  };
};
```

*Figure 7 Inventory Screenshot*




## AdminFunctions

We created Admin special functions to be able to manage users and control over database.

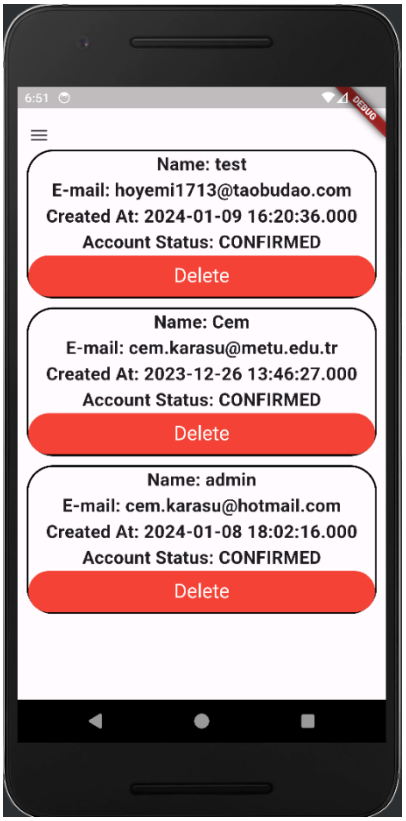
### Getting All Users

To get all users from the database, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://0vx5duwshj.execute-api.eu-north-1.amazonaws.com/users

After the Lambda function is triggered, first we send a dynamo list user command to Cognito user pool. And it returns the list of all the users in the pool. Then, it is put in the response body and sent to the client. You can see the associated function below.

```
case "GET /users":
    input = {
        UserPoolId: "eu-north-1_5mwqN2dXj", // required
    };
    command = new ListUsersCommand(input);
    body = await client.send(command);
    body = body.Users;
    break;
```

*Figure 8 All The Users In Admin Panel UI Screenshot*




### Deleting A User by ID

To delete a user from the user pool, we created a lambda function that is triggered by making a *DELETE* request to the following API endpoint: https://0vx5duwshj.execute-api.eu-north-1.amazonaws.com/users/{id}

User id must be placed as a path parameter. Here is an example:

https://0vx5duwshj.execute-api.eu-north-1.amazonaws.com/users/5

After the Lambda function is triggered, first we send an *admin delete user* command to the Cognito user pool with an id which is taken from the path parameter in the request URL. Then, the deleted user id is put in the response body and sent to the client. You can see the associated function below.

```
case "DELETE /users/{id}":
    input = {
        UserPoolId: "eu-north-1_5mwqN2dXj",
        Username: event.pathParameters.id,
    };
    command = new AdminDeleteUserCommand(input);
    body = await client.send(command);
    body = `Deleted user ${event.pathParameters.id}`;
    break;
```

## dbFunctions

This lambda functions are created for Admin to have control over database.

### Backuping All Databases

To create a backup for all databases, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://5q3rsscxvf.execute-api.eu-central-1.amazonaws.com/backupDb

After the Lambda function is triggered, first we send a *create backup* command to the DynamoDB client. Then, DynamoDB client creates backups for each database. You can see the associated function below.

```
case "GET /backupDb":
    input = {
        TableName: tableName,
        BackupName: "Item_table_backup",
    };
    command = new CreateBackupCommand(input);
    body = await client.send(command);

    input = {
        TableName: counterTable,
        BackupName: "Item_table_backup",
    };
    command = new CreateBackupCommand(input);
    await client.send(command);
    break;
```

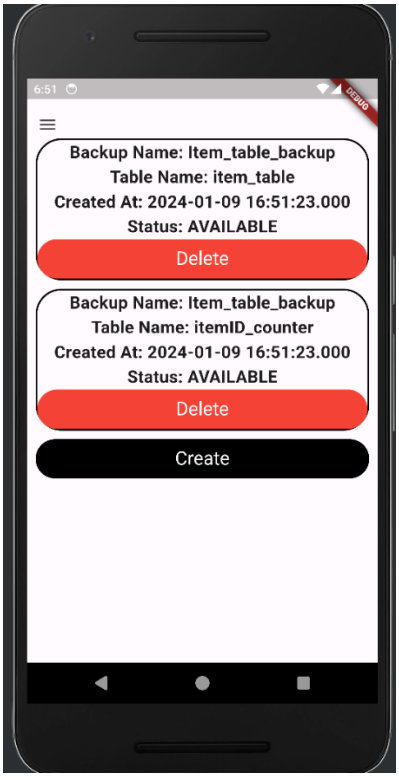
### Getting All The Created Backups of Databases

To get all backups, we created a lambda function that is triggered by making a *GET* request to the following API endpoint: https://5q3rsscxvf.execute-api.eu-central-1.amazonaws.com/listBackupDb

After the Lambda function is triggered, first we send a *list backups* command to the DynamoDB client. Then, DynamoDB client returns a list of created backups. Then they are put to the response body and sent to client. You can see the associated function below:

```
case "GET /listBackupDb":
    input = {};
    command = new ListBackupsCommand(input);
    body = await client.send(command);
    body = body.BackupSummaries;
    break;
```

*Figure 9 Getting All The Created Backups of Databases UI Screenshot*




## Deleting A Backup By ID

To delete a backup, we created a lambda function that is triggered by making a *DELETE* request to the following API endpoint: https://5q3rsscxvf.execute-api.eu-central-1.amazonaws.com/backupDb

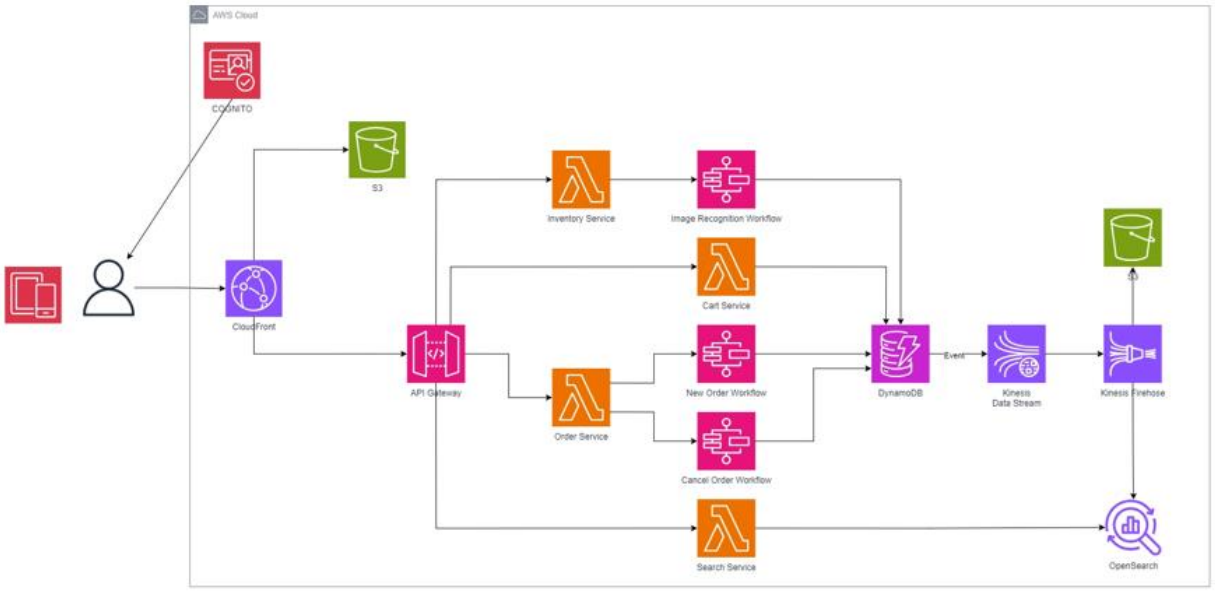
Backup id must be putted in the body. Here is an example of body in a request:

```
{
  "id": "arn:aws:dynamodb:eu-central-
1:837614956853:table/item_table/backup/01704812937247-a232e0c7"
}
```

After the Lambda function is triggered, first we send an *delete backup* command to the DynamoDB client. You can see the associated function below.

```
case "DELETE /backupDb":
    let requestJSON = JSON.parse(event.body);
    input = {
        BackupArn: requestJSON.id,
    }
    command = new DeleteBackupCommand(input);
    await client.send(command);
    body = "Backup deleted";
    break;
```

*Figure 10 General Structure*




## Utilized Cloud Services:

| Part | AWS Service | Programming Language | Lambda Function Name |
|---|---|---|---|
| Authentication | AWS Cognito & Amplify | Node.js 16.x<br>Node.js 18.x | amplify-login-create-auth-challenge-56c36a45<br>amplify-login-custom-message-56c36a45<br>amplify-ecommerceflutter--UpdateRolesWithIDPFuncti-TeqqfJtbBnas<br>amplify-login-verify-auth-challenge-56c36a45<br>amplify-login-define-auth-challenge-56c36a45 |
| Managing Items | DynamoDB & S3 Bucket | Node.js 20.x | items |
| Image Recognition | AWS Rekognito & S3 Bucket | Python 3.12 | rekonObject |
| Admin Functions | AWS Cognito | Node.js 20.x | adminFunctions |
| Database Functions | DynamoDB | Node.js 20.x | dbFunctions |

## Project Statistics

**Authentication:** This part is implemented by Cem in the week W12.

**Managing Items:** This part is implemented by Cem in the weeks W12&W13. The total number of code lines implemented is about 150.

**Image Recognition:** This part is implemented by Metehan in the week W12. The total number of code lines implemented is about 50.

**Admin Functions:** This part is implemented by Metehan in the week W13. The total number of code lines implemented is about 50.

**Database Functions:** This part is implemented by Cem in the week W13. The total number of code lines implemented is about 70.

**Mobile Application:** This part is implemented by Cem in the weeks between W10-W13. The total number of code lines implemented is more than a 1000. Programming language used in this part is Dart. And Flutter framework is used.

**Database:** DynamoDB which is NoSQL Database, is used. Maximum item size is 400kbite. Maximum table size is 10gigabite.