Billogram Feature Design Task

Design Proposal and Implementation

Salinda Wijayakoon : Engineering Manager

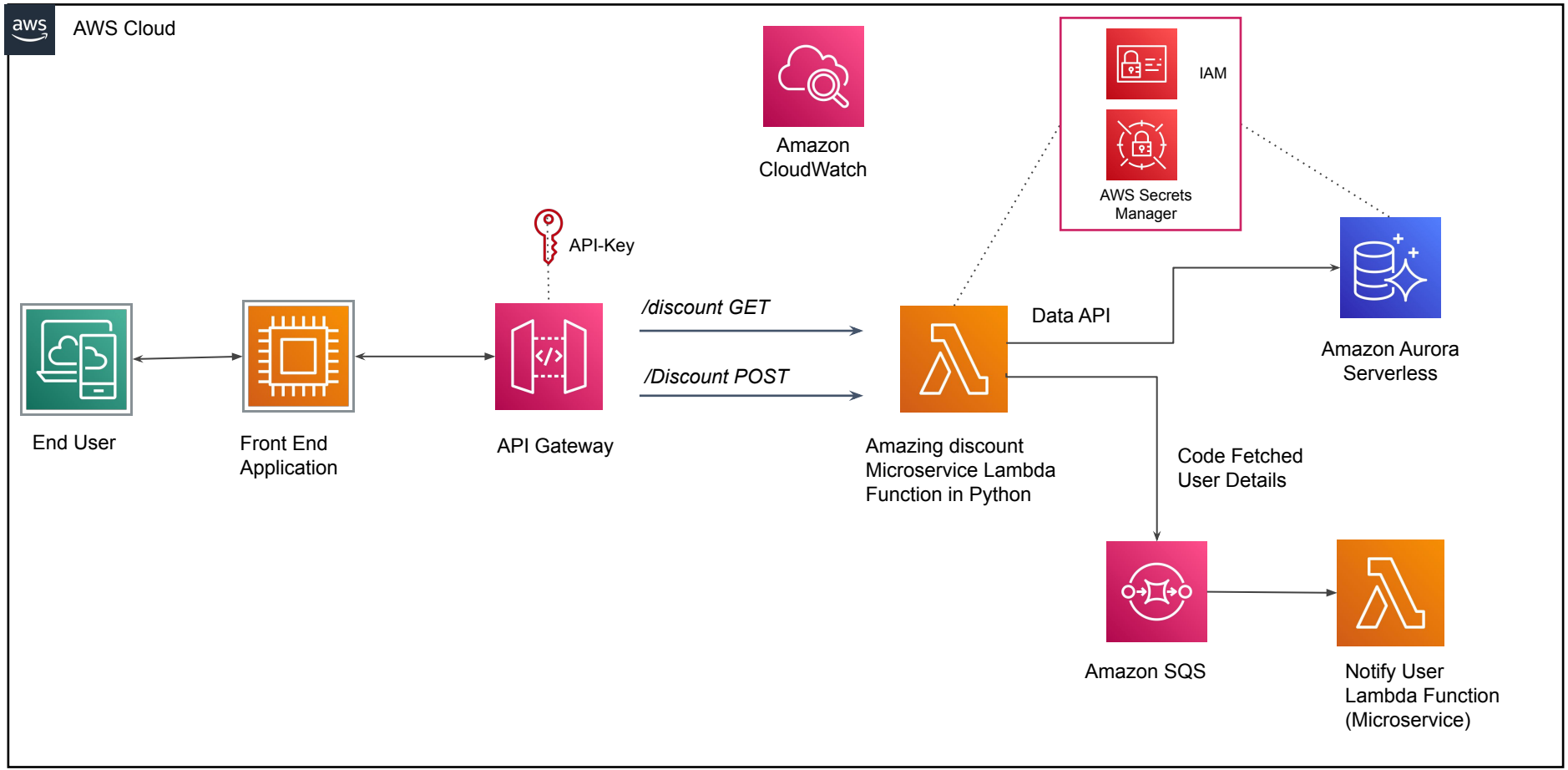
Summary of Content

- Use cases
- Proposed Architecture Overview
- Design aspects
- Implementation and Deployment

Use cases

1. As a brand I want to have discount codes generated for me so that I don’t have to deal with this administration myself.
2. As a logged in user I want to be able to get a discount code so that I can get a discount on a purchase.
3. As a brand I want to be notified about a user getting a discount code so that I can process information about the user for my loyalty programme.

Proposed Architecture Overview

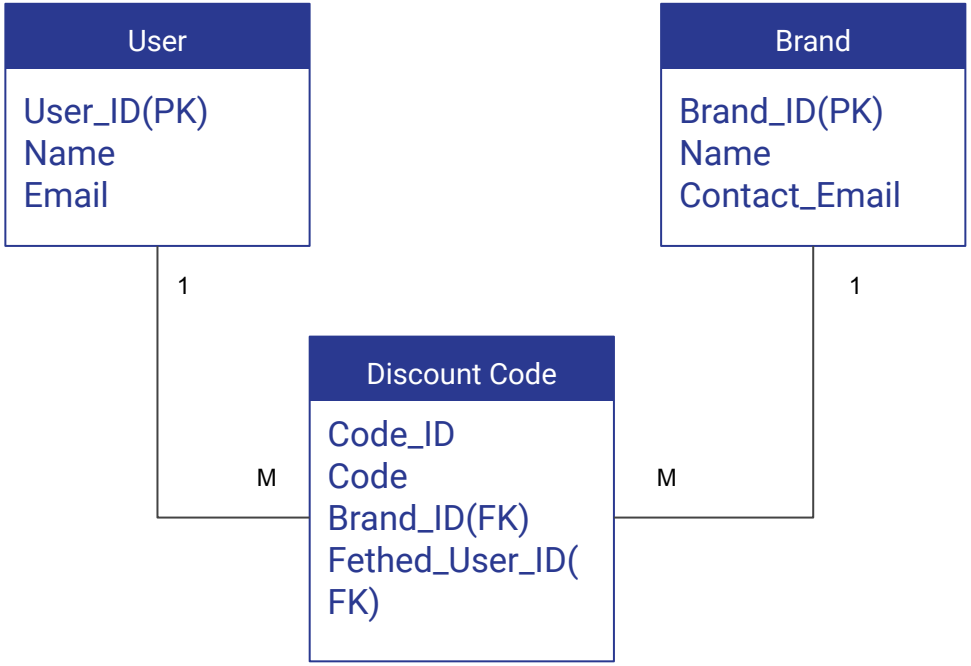


Design Aspects

- Database Design
- Microservice design
- API Gateway
- Synchronous and Asynchronous Messaging
- Scalability
- Authentication and access Management
- Loosely coupled architecture

Database Design

Information Structure



Assumptions:

- one code can only be picked by one user

- it's important to record the user who fetched a code for later marketing purposes

Database Design

Database Selection

I would propose to use RDS due to:

- It's easy to build relationships in RDS database
- The information in hand is relatively fixed and hence its not necessary to have a variable set of columns to be stored in a DB like DynamoDB

Alternatively

- We could store only the discount code data in a DynamoDB and let other microservices deal with user and brand data.

Microservice /discount/GET

1. **Purpose**: Get a discount Code
2. **Http**: GET
3. **Inputs** (as url parameters)
   - User_id
   - brand_id
   - user_email
4. **Functionality**
   - Validate user id and brandid against database
   - Validate email format
   - Send brandid and Get a discount code from database
   - Update the fetched record with Fetched user id(So that it will not be given to another user)
   - Send fetched user details to SQS (Message to be consumed by send notification microservice)
5. **Exceptions**
   - Invalid User ID or Invalid Brand ID
   - Invalid email format
   - No discounts found for the given brand ID
   - Database connection errors(Retry by client in this case)

Microservice /discount POST

1. **Purpose**: Create discount codes
2. **Http**: POST
3. **Inputs** (Json Body)
    - brand_id
    - N(number of discount codes to be created)
4. **Functionality**
    - Validate brand ID against database
    - Validate N is an integer
    - Loop N times and
    - Create and insert discount codes to database
5. **EXceptions**
    - Invalid brand ID
    - Invalid N
    - Database connection errors(Retry by client in this case)

Microservice Design

Technology Selection

AWS Lambda Functions with python

assumption: Discount code usage is seasonal and also not predictable

1. Lambda functions are easy to write and maintain
2. Lambda functions scales automatically based on demand and there is no need of paying for unused resources.
3. AWS maintains the underline infrastructure and hence less maintenance efforts
4. Python is easy to write (In a team setup we will decide which language is most comfortable for us)

API Gateway

Technology Selection

AWS API Gateway

I would propose to use API Gateway REST API interface to expose endpoints towards internal and external applications

- Ability to configure GET and POST methods
- Ability to authenticate the request using a API key
- Ability to configure usage plans and track number of requests for billing purpose etc

Synchronous & Asynchronous Communication

Method	Type	Comments
/Discount GET	Synchronous	• The client should get a response with a discount code hence this call should be Synchronous
/Discount POST	Synchronous	• The client could get a response after creating the discount codes. • This call can be made asynchronous if the time taken to process is too high so that client doesn't have to hold a session
Sending fetched user details	Asynchronous	• Fetched user details will posted to a SQS for Asynchronous processing

Scalability

Below layers of the architecture is defined in such a way to scale automatically based on the load in the system

1. API gateway scales automatically based on requests
2. Lambda function microservice instances will be initiated automatically based on number of requests
3. Aurora serverless instances will be autoscalled based on demand

Authentication and Authorization

1. API calls are authenticated via API key to prevent unauthorized access
2. Lambda functions uses AWS secret manager, so that it’s not necessary to store DB login details in lambda which reduce the risk of exposure.
3. IAM roles are used to provide necessary authorization to lambda function to execute database commands

Modules are loosely coupled

Architecture is designed in such a way to “loosely coupled” one module from another, so that its possible to replace the modules with alternatives in future if required.

For eg

- API gateway can be replaced by a 3rd party API gateway
- Lambda functions can be replaced with Container based microservices
- Database can be replaced with DynamoDB(the code of connecting to the database has to be changed)
- User notification microservice can be changed without impacting the discount microservice

Implementation & Deployment

Working API Endpoints

What is not implemented

Code in github

Database configurations

Secret Manager Configurations

Lambda function configurations

API Gateway Configurations

Testing

Working API Endpoints

A working API is hosted in AWS environment and below information can be used to test the Endpoints

Discount /GET
Url with parameters:
https://faqxmojp7g.execute-api.us-east-1.amazonaws.com/testenv/discount?userid=111&brandid=SONY&useremail=salindaw@yahoo.com

API KEY in Http headers: x-api-key=PMLNDyyJ434XyL3TSKB4C6YbSQe6eQcS59d13vmL

Discount /POST

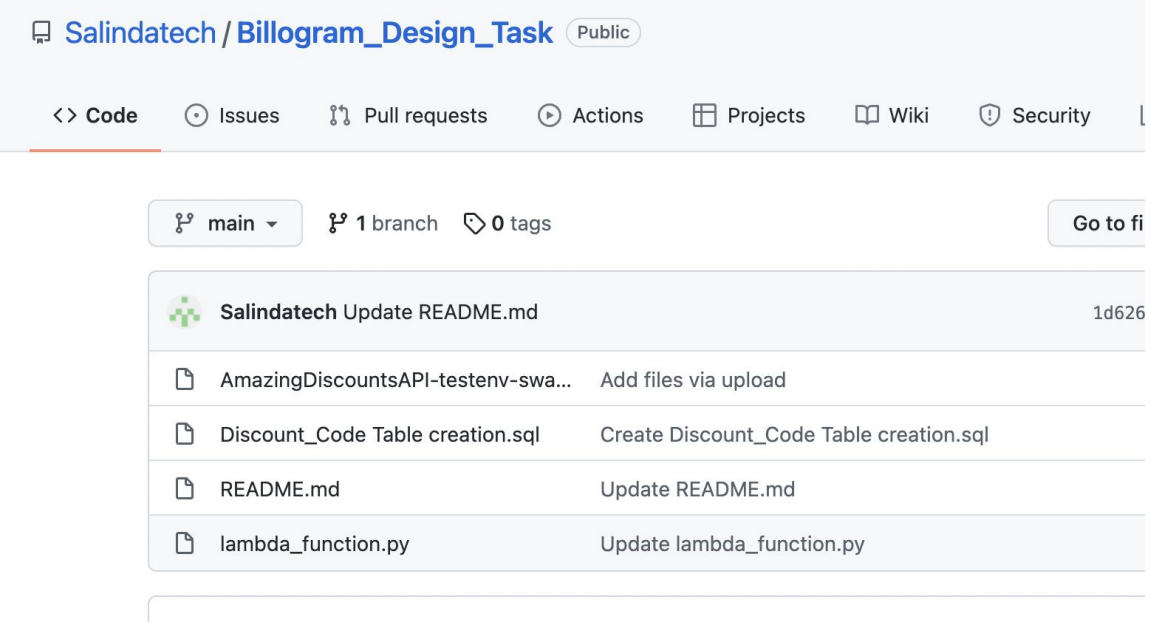
Url: https://faqxmojp7g.execute-api.us-east-1.amazonaws.com/testenv/discount
Body: {"brandid":"SONY", "N":5}
API Key is not Implemented

What is not Implemented

1. Concurrent transaction handling: There is a risk that same code is fetched by two users if they request at the same time.
    - A solution would be to create the discount codes “on the fly” and provide them to users and then record in DB, rather pre creating them.
2. Error handling: More granular level error handling is required for better programming and reporting errors
3. Only discount code table is created in the database.
4. Validation methods and calling SQS methods are not implemented
5. Reading values from Jason Body in POST method is not implemented

The Code

Github Url: https://github.com/Salindatech/Billogram_Design_Task.git



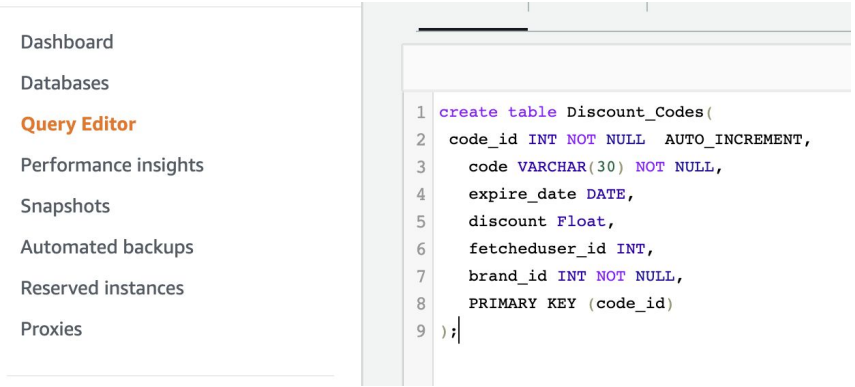
Database configurations

Please use below options when creating the RDS database

1. Creation Method: **Standard create**
2. Engine Type: **Amazon Aurora**
3. Edition: **Amazon Aurora MySQL-Compatible Edition**
4. Capacity Type:**Serverless**
5. DB cluster identifier: Choose a name
6. Master username:choose a name
7. Password: Choose a pass
8. Virtual private cloud (VPC): Default was chosen to make it simple
9. Existing VPC security groups:Default was chosen to make it simple
10. Additional Configurations: Web Service Data API: **Check Data API** (This is mandatory since Lambda function use Data API to connect to the database)
11. Create the database

Table creation

Connect to the database using Query Editor or some other means and execute the provided Discount_Code Table creation.sql



Secret Manager Configurations

Please use below options when creating secret manager

1. Navigate to : AWS Secrets Manager:Secrets:Store a new secret
2. Secret Type: Credentials for Amazon RDS database
3. Enter Username and password
4. Secret Name: Choose a name
5. Create a secret with other default options
6. Copy the Secret ARN (this will be used to configure in Lambda function)

Lambda Function Configuration

Please use below options when creating lambda function

1. Copy the Lambda function and create a new Python lambda function(lambda_function.py)
2. Change below details

    - databasename='bestbrands'
    - dbclusterarn='arn:aws:rds:us-east-1:118383306190:cluster:marketing'
    - db_secret_store_arn='arn:aws:secretsmanager:us-east-1:118383306190:secret:rds-db-credentials/cluster-PQUJSVPFEJTFCY3INBPCKJ4J5A/admin-O1wdTq'

3. Edit and add a Role with below permissions

	Policy name
▸	AmazonRDSDataFullAccess
▸	AWSLambdaBasicExecutionRole

API Configurations

Please use below options when creating the API

1. API→Create REST API
2. Add a resource “discount”
3. Create GET and POST Methods
    - Integration type: Lambda Function
    - Use Lambda Proxy integration: Should be checked
    - Select the lambda function

4. Now the API should be ready for testing

Testing

1. First call the POST method to create discount codes
2. Then call the GET method to get a discount

POST https://fagxmojp7g.execute-api.us-east-1.amazonaws.com/testenv/discount

Params | Authorization | Headers (8) | Body | Pre-request Script | Tests | Settings

none | form-data | x-www-form-urlencoded | raw | binary | GraphQL | Text

```
1 {"brandid":"SONY", "N":5}
```

Body | Cookies | Headers (7) | Test Results | 200 OK

Pretty | Raw | Preview | Visualize | JSON

```
1 Discount codes were created successfully
```

GET https://fagxmojp7g.execute-api.us-east-1.amazonaws.com/testenv/discount?userid=111&brandid=SON...

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings

Key	Value	Description
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
x-api-key	PMLNDyyJ434XyL3TSKB4C6YbSQe6eQc...	

Body | Cookies | Headers (7) | Test Results | 200 OK 434 ms 351 B

Pretty | Raw | Preview | Visualize | JSON

```
1 {'Discount_Code':'SONY2bc248e6-7d54-11ec-b4a4-f27edf240890'
2 }
```

Thank You!

Thanks taking time to review the proposal and implementation