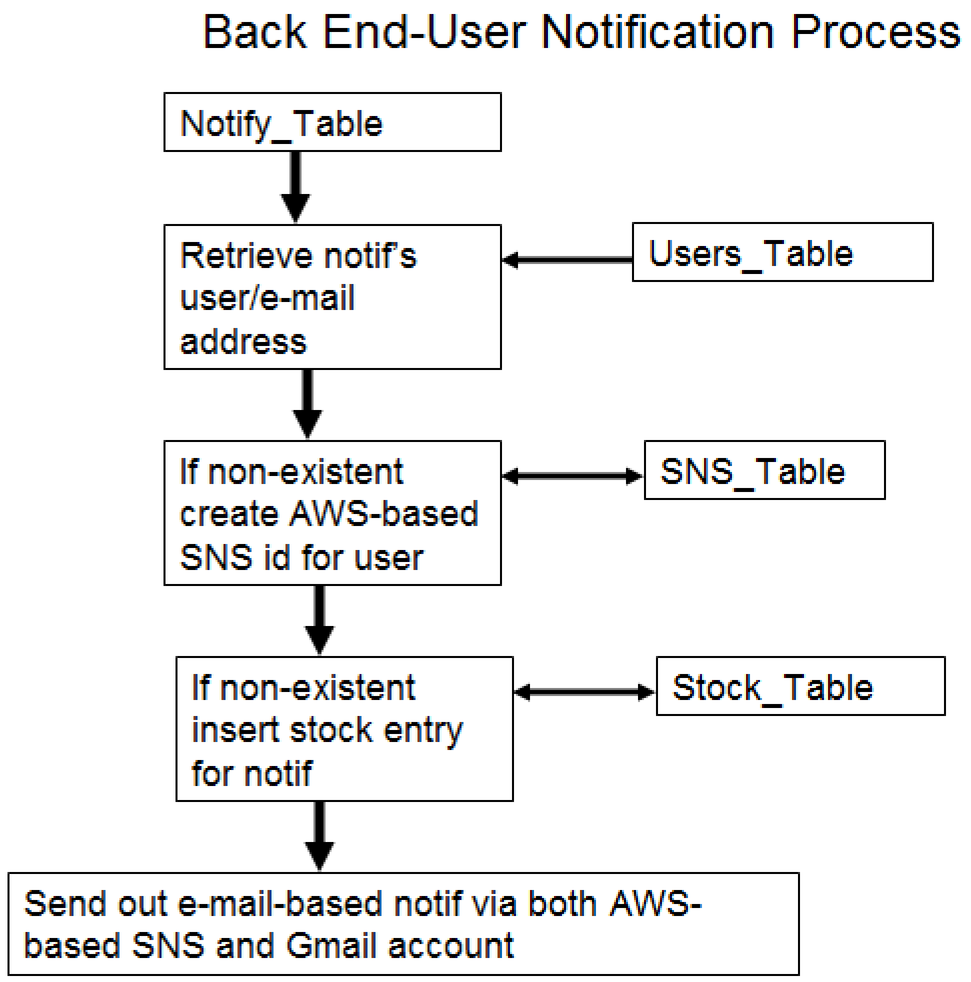
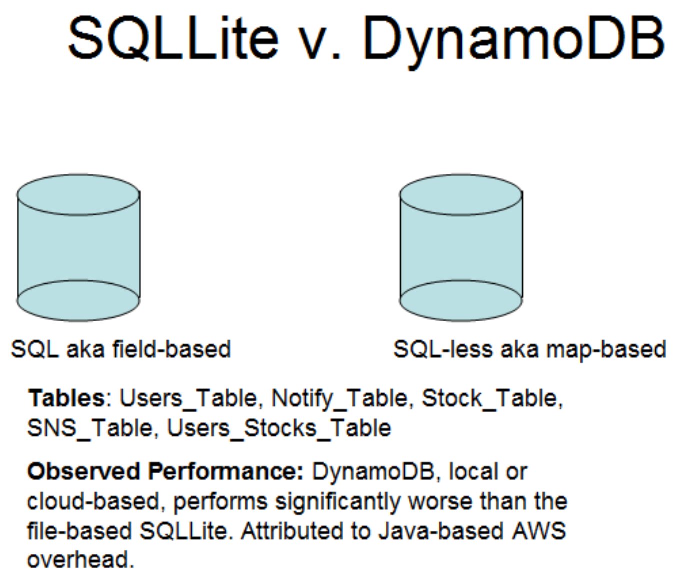
CS9223 INT Cloud Computing
Utkarsh Gautam
Christopher Chan
Final Project Report

Fall 2016
12/22/2016

Stock Notifier: A service to automatically notify users via e-mail when a stock, traded on at least one of the USA exchanges such as NYSE or NASDAQ, is at a certain price as specified by them via a web interface.


Back End-User Notification Process
Notify_Table
Retrieve notif's user/e-mail address
Users_Table
If non-existent create AWS-based SNS id for user
SNS_Table
If non-existent insert stock entry for notif
Stock_Table
Send out e-mail-based notif via both AWS-based SNS and Gmail account

The original technical expectations was to run a web server hosted on an E2C instance. Users will be able to login to the web interface to specify the stock and price of their e-mail notifications. These specifications would be stored in a S3 bucket. The service of retrieving stock quotes is to be done via the Python-based Google Finance API, https://pypi.python.org/pypi/googlefinance. An instance of a Python-based lambda function with the same S3 bucket where the user notification specifications are stored would run during stock market hours, be responsible for querying the Google server for the latest stock prices, and determining when an e-mail notification needs to be sent out.We were envisioning the back-end to be a python-based lambda function while the front-end would be a web server on an E2C instance. Eventually due to the exorbitant charges based upon number of S3-based access, which had neither direct correlation to the timeframe of the instances nor what constituted access, we finalized towards a Java-based jar file with both e-mail and SQLLite capabilities in addition to AWS access. This allowed us to develop the Java-based notifier process completely outside of AWS resources. The Java-based notifier relied on Google services, specially its REST-based finance website for stock quotes and TLS-based e-mail via SMTP/IMAP protocols. Alongside the Google services with SQLLite, which is a file-based SQL service, the Java-based notifier is capable of running on any Java virtual platform with internet access. This enabled the ability to develop the application in Windows and have it put into operations on a Linux-based E2C instance. The SQL schema utilized was simple by design with a Notify_Table listing out the stocks and price range the users want to be notified on. The Users_Table and Stock_Table are reference tables. The Stock_Table is comprised of stock symbol, stock name, market, url, price, lastupdated. While the price and lastupdated is from the Google Finance API, the stock-specific information can be populated using *.csv files from the websites of markets such as AMEX, NYSE, and NASDAQ


SQLLite v. DynamoDB

Tables: Users_Table, Notify_Table, Stock_Table, SNS_Table, Users_Stocks_Table

Observed Performance: DynamoDB, local or cloud-based, performs significantly worse than the file-based SQLLite. Attributed to Java-based AWS overhead.

As the design progressed towards leveraging AWS resources, we moved towards using the SQL-less DynamoDB and the Push Notification Service (SNS). We ran a localized Java-based version of the SQL-less DynamoDB and developed the interface known as DynamoDBInterface, which is counter to the SQLLiteInterface. The localized Java-based version of DynamoDB was basically DynamoDBLocal.jar listening on port 8000, which creates shared-local-instance.db to store the data. It was observed that DynamoDBLocal.jar while running on our local workstation did so without difficulty, there was connectivity issues on the E2C instance even though port 8000 was verified to be successfully open. DynamoDB's tables operate differently from SQL by only requiring the partition key defined and all other data and fields can be added in. Once development was complete the notify application was developed further to using the online DynamoDB. It is to be noted that DynamoDB is heavily mapped-based and not capable of joint queries, where in SQL tables are merged temporarily upon a common field to fulfill a specific query.

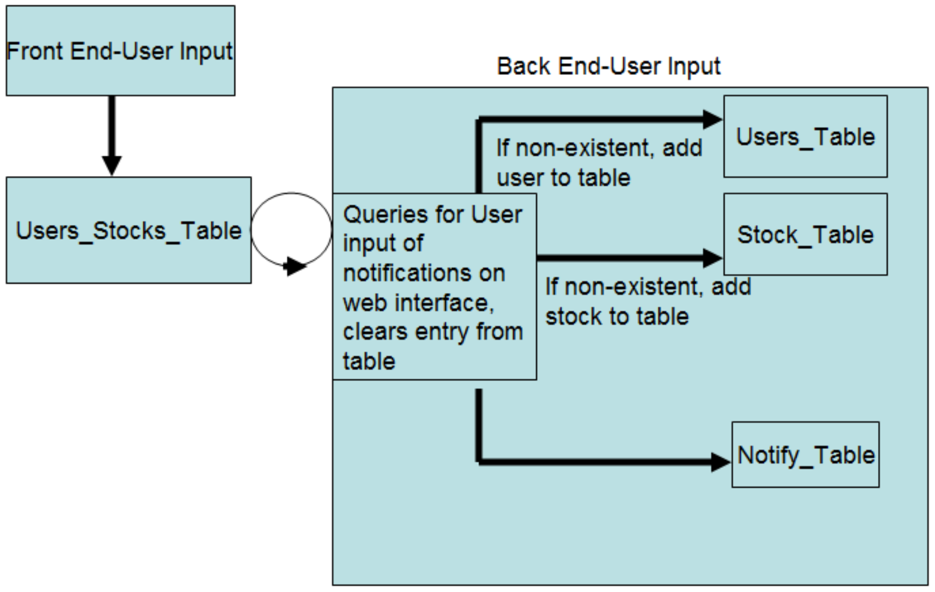
The SNS was designed to have a single topic per a user and utilized to have AWS e-mail users the result of their stock picks. Once the user creates a notification, a SNS-based topic is created unique to the user to serve as a means for communicating the user's stock picks. A first time user once subscribed will be sent an e-mail for the user to confirm his subscription.

In the migration to AWS resources the SNS_Table and Users_Stocks_Table were created. The SNS_Table is to keep track of each user's unique topic ID. AWS can be used to e-mail the expected end user via the user's unique topic ID. The Users_Stocks_Table is populated by the frontend's HTML/Javascript-based webpage and checked periodically by the process to update the Notify_Table. The *.jar file is capable of running continuously on an E2C instance as an example of Cloud as Infrastructure-as-a-Service (IaaS).

We were looking into AWS Elastic Beanstalk to host the frontend html/javascript. To host it we would require the frontend html/javascript to conform to the Elastic Beanstalk formal format. Originally the html/javascript would sent out a SNS near the end of runtime, but this was removed in addition to a developmental message box.

The frontend is html/javascript and utilizes aws-sdk-2.7.14.min.js to connect to AWS.

Summary: Currently the demonstrated end-to-end index.html can be run on the local machine to update Users-Stocks_Table for communication with the backend. To run the backend using the Backend_Deliverable, first update StockNotify.java such that there are credentials for a gmail account a AWS access.


Front End-User Input
Users_Stocks_Table
Back End-User Input
Queries for User input of notifications on web interface, clears entry from table
If non-existent, add user to table
Users_Table
Stock_Table
If non-existent, add stock to table
Notify_Table

Frontend Setup:

Step 1: Update the authentication information in index1.html

```
AWS.config.accessKeyId = "";
AWS.config.secretAccessKey = "";
```

Step 2: To enable a facebook-based login to fill-in the values upon a login, update the facebook app id and specify a pre-existing ARN role defined in your AWS instance's IAM

```
var appId = ''; //from facebook
var roleArn = '';
```

Step 3: Open index.html with a javascript-enabled web browser

Backend Setup:

Step 1: Update the authentication information on StockNotify.java

```
GmailClient mailClient = new GmailClient("","");
String AWSAccessKeyId =" ";
String AWSSecretKey =" ";
```

Step 2: Run Compile.bat

Compile.bat:

```
javac -classpath ".;./\*" *.java
```

Step 3: Run.bat

Run.bat:

```
java -classpath ".;./\*" StockNotifier
```

Step 4: **It runs best with CORS extension enabled on the browsers.**

Front End Features and Operation:

The Front End uses HTML and JavaScript to provide the user an interface to **search for stocks** and select the stocks they want to follow. It also includes a method to **authenticate the users** by leveraging **Web Federated Identity** and using Facebook login credentials to allow the user to post to DynamoDB. The Front end provides an interface to input communication details of the user and their stock selection, it then posts these details to **DynamoDB** for the backend to process them further.

1. Stock search using AWS Cloud Search

   Users can search for stocks based on the following indexes:

   - Name
   - Symbol
   - Market (NYSE, NASDAQ, AMEX)
   - Industry

   The Web application sends a **GET request** with a **search query** to the **AWS Cloud Search domain end point**, and receives a response in **JSON format** of the stocks that match the search query. The JSON response is parsed and used to **populate** the **select box** from where the user can select a particular stock.

To develop the cloud search domain, we had uploaded a **CSV file of stocks** indexed as mentioned earlier to the AWS Cloud Search. The **IAM policy** for the search endpoint was established as "**Search and Suggester service: Allow all. Document Service: Account owner only**."

IAM Policy for Cloud Search:

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
     "Effect": "Allow",
     "Principal": {
       "AWS": "*"
     },
     "Action": [
       "cloudsearch:search",
       "cloudsearch:suggest"
     ]
   }
 ]
}
```

2. User Authentication using Web Federated Identity (WFI)

First the web application was registered with Facebook using Facebook for Developers. JavaScript SDK provided by Facebook and AWS was used to integrate the Authentication provided by Facebook to the AWS Credentials.

Code:

```
FB.login(function (response) {
        if (response.authResponse) {
            AWS.config.credentials = new AWS.WebIdentityCredentials({ //WIF
                RoleArn: roleArn,
                ProviderId: 'graph.facebook.com',
                WebIdentityToken: response.authResponse.accessToken

            });
         }
   }
```

Next, appropriate IAM Policy was attached to the DynamoDB to allow only Facebook Authenticated users to post to DynamoDB

IAM Policy:

```
{
"Version": "2012-10-17",
"Statement": [
 {
  "Effect": "Allow",
  "Action": [
   "dynamodb:BatchGetItem",
   "dynamodb:BatchWriteItem",
   "dynamodb:DeleteItem",
   "dynamodb:GetItem",
   "dynamodb:PutItem",
   "dynamodb:Query",
   "dynamodb:UpdateItem"
  ],
  "Resource": [
   "arn:aws:dynamodb:us-west-2:433603958356:table/Stock_Profiles"
  ],
  "Condition": {
   "ForAllValues:StringEquals": {
    "dynamodb:LeadingKeys": [
     "${graph.facebook.com:id}"
    ]
   }
  }
 }
]
}
```

3. Take user input and post it to **DynamoDB**

The user is asked to input a **unique username** (**Primary Key for DynamoDB**), **email ID** (for **SNS Subscription and Communication**), stock and stock market selection, and threshold price.

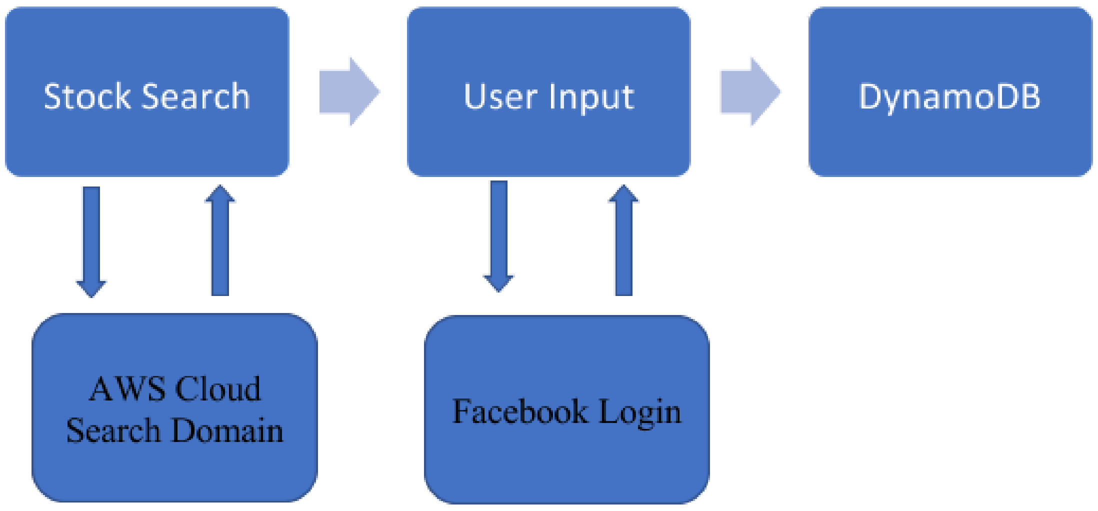
These inputs are posted to the DynamoDB using methods provided by **AWS SDK for JavaScript**.

Code:

```
  dynamodb = new AWS.DynamoDB({ region: 'us-west-2' });
  docClient = new AWS.DynamoDB.DocumentClient({ service: dynamodb });
  username = document.getElementById("username").value;
  emailid = document.getElementById("emailid").value;
  stock1= document.getElementById("stock1").value;
  market1= document.getElementById("market1").value;
  threshold1= document.getElementById("threshold1").value;
```

```
params = {
    TableName: Users_Stocks_Table',
    Item: {
        username: username,
        emailid: emailid,
        stock1 : stock1,
        market1: market1,
        startprice: threshold1,
         stopprice: threshold2
          }
        };
docClient.put(params, function(err, data)
    {
        if (err) console.log(err);
        else
        {
            resultData = data;
            console.log(resultData);
        }
})
```

Operational Flow of Front End:


Stock Search
User Input
DynamoDB
AWS Cloud Search Domain
Facebook Login