The linked image cannot be displayed. The file may have been moved, renamed, or deleted. Verify that the link points to the correct file and location.

FINAL YEAR PROJECT REPORT

GEKKO APPLICATION

X14102102 - BSHCYB

INTRODUCTION

The purpose of this document is to outline a detailed overview of the design, development and deployment of a final year project with the National College of Ireland. The following sections of this report will detail the proposed project, the design elements and consideration of its requirements, the in-depth development and technologies used with the application and the implemented solution for deployment of the final application.

We will also reference the research conducted in the course of this development life-cycle and how this research affected the final product in terms of functionality and what has been determined as necessary features based on the intended user base.

BACKGROUND

The emergence of crypto currencies in recent years and more recently the unprecedented rise of Bitcoin pricing has led to a huge amount of interest in this area of technology/finance. This area focus has led to the creation of a large number of new currencies and coins each represented by a trading symbol. Large financial markets and dedicated exchanges have been created in order to buy, sell and trade these currencies against both the Bitcoin standard and recognized international currencies.

As a Cloud Support Engineer, I have become increasingly interested in the technology behind these currencies and how individuals and corporations are building tailored exchanges and wallets in order to trade and profit from registered coins. It is for this reason that my final year project, is to create a web application dedicated to the monitoring of a small subset of these coins.

CONCEPT

From the inception of this project, it had been decided that the finished product would be an elegant solution to a complex field of financial markets. That is to say, the proposed application will display information on financial market symbols and their trends in an easy to understand graphical format.

While elements of this application have evolved, added or removed the Gekko web application, as stated above will be focused on monitoring a small subset of crypto currencies in relation to the current market value of each supported coin. This application will use HTTP API methods and a severless backend in order to retrieve coin data from an established exchange. The purpose of this is to inform users of present coin values and the percentage of change in price over the previous hour, twenty-four hours and 7 days.

In the world of financial trading, prices can fluctuate rapidly across all markets, leading to financial loss during periods of user inactivity. For this reason, the Gekko web application will also include a notification system, by which large changes in market prices will be calculated at regular intervals and once a change in pricing over a defined value has been reached an automated alert system will be triggered and the user will receive a notification.

SCOPE

Below this paragraph, is a list of application functionalities which are intended to be implemented in the Gekko web app. Each entry is a high-level milestone which should be reached during the course of development to ensure the success of the application. The supporting technologies, research and methodologies for the implementation of each functionality listed will be discussed later in this report.

- A secure web application accessible over the public internet.
- The ability to retrieve current crypto currency pricing data.
- The ability to display data in a user-friendly format.
- An option to purchase or trade currencies over a registered exchange.
- The registration and login of Gekko users.
- The automated notification of large pricing changes to registered users.
- A comprehensive support platform through which users can address issues.

DEFINITIONS, ACRONYMS AND ABBREVIATIONS

Throughout this document, abbreviations, acronyms and technologies will be mentioned. In order to remove any confusion around the mention of any of the above, the below list will describe and define each.

AWS - Amazon Web Services is a cloud platform which provides a large number of cloud-based services for development and implementation of applications, solution and software.

Lambda – Is an AWS Service which provides Serverless compute power. This is to say that functions can be executed as they are needed without the need of continuously running servers or back-end machines.

API – An Application Programming interface is a set of communication methods for connecting two endpoints and facilitating the submission or retrieval of data between each endpoint.

HTTP – Hypertext Transfer Protocol is the internet protocol for connection of two machines for retrieving website data. This protocol is based on a client server architecture.

HTTPS – This protocol is the secure version of HTTP and using secure flags over SSL to ensure a secure connection between a server and client.

Bootstrap – This is a free and open source front end framework for website development. Bootstrap provides HTML, CSS and JavaScript templates upon which fully fledged websites can be built.

SNS – Simple Notification Service is an AWS cloud service used for publishing notifications to confirmed endpoints such as HTTP, Email, SMS.

SES – Simple Email Service is a highly scalable email platform which allows sending and receiving of emails through the AWS platform.

DynamoDB – A no SQL database solution provided by AWS. This database allows seamless integration with computing resources, providing an internal endpoint to which resources can write.

POC – Proof of Concept, is a term used in development and means evaluating the validity of a solution.

ELB – Elastic Load Balancers are an AWS resource which can be placed at the edge of a network to receive and order incoming traffic. These ELB's are used to distribute the incoming traffic across the servers which are provisioned behind it.

ALB – Application Load Balancer is a variant of ELB, which is optimized for HTTP and HTTPS applications. This load balancer uses listeners to accept and direct traffic accordingly.

Route 53 – This is the AWS DNS service which uses hosted zones in which DNS records can be created for resolving domain names to a website location.

Hosted Zone – This is a Route 53 resource which contains DNS records and specifies to DNS name servers through which registered domains are resolved.

SSL/TLS – Secure Socket Layer and Transport Layer Security are secure communication methods commonly used in website implementation. These methods allow secure connections between a client and server over the HTTPS protocol.

LAMP – A LAMP server is a server environment in which the server machine is running Linux, Apache, MySQL and PHP.

Apache – This is the most widely used web server solution in the world and allows the distribution of website or web application files over internet ports such as port 80 and port 443.

PHP – Is a programming language commonly used in website development in order to allow front-end interaction to perform back-end tasks.

MySQL – A database solution which uses the SQL querying language.

PHPMyAdmin – A free software tool which has been written in PHP. This tool allows the administration of a MySQL database over the internet.

Exchange – This is a financial term which means a platform through which stock and currency shares can be bought or traded.

Endpoint – This is an abstract term used in development to represent any resource which must be reached by another.

TECHNOLOGIES

A large number of development, automation and cloud technologies were used during the course if implementation for the Gekko application. These technologies have changed from the initial project specification which is listed in the appendix of this report and so this section will outline the technologies used in the final publication of the application and furthermore describe how each technology was used in relation to the application functionality and the rationale behind the adoption of these technologies.

The table below shows a high-level overview of the technologies used and the corresponding areas of application functionality which uses them.

Development Environment	Cloud 9 Linux
Website Application	Bootstrap JavaScript
Coin Statistics/Information	API JavaScript
Support/Contact	PHP API Lambda Python WorkMail Route 53 JavaScript Amazon Connect
Registration/Login	PHP MySQL
Hosting	ALB Route 53 GoDaddy EC2 Linux

CLOUD9

Cloud9 is a cloud-based IDE which allows development within an internet browser. This technology is used as the development environment for the Gekko Application. Our Cloud9 environment was configured with SSH and installed directly onto the production EC2 instance. The choice of this IDE for our development was due to the placement of the environment within our Apache web server which allowed the real time development and update of the application from within the IDE.

LINUX

This is an open source OS which comes in many distributions. The distribution used for our application is the Amazon Linux flavor of Linux which is based on the Red Hat distribution. The choice behind this particular flavor of Linux is the large amount of documentation on the platform in relation to the AWS services which are used through-out the development of this application.

BOOTSTRAP

This is an open source toolkit which is comprised of pre-configured HTML, CSS and JavaScript elements ready for implementation in a website front-end. This is used in the development of the website GUI in order to define containers and styling.

JAVASCRIPT

JavaScript is a high-level programming language commonly used in web applications to create dynamic web pages. While this language is used throughout the application development, very few features depend on JavaScript to achieve functionality. This will most prominently appear as part of the contact form creation in which it was used to create an API connection and used in the Serverless back-end in the form of Node.js.

API

API's are quite prominent throughout the development of the Gekko application. This technology is used to retrieve crypto currency information and also used in the submission of information to a Serverless back-end for the purpose of triggering a Lambda function in order to complete an email delivery.

PHP

This programming language is used more than any throughout this application for the purpose of calling API methods for currency information and also the creation of the registration and login system where a large collection of classes is created to define the methods being used throughout the system. The PHP language was chosen because of its ease of integration with HTML and its capabilities as a back-end development language.

LAMBDA

This Serverless technology is used for a small number of features in the back-end system. These features include the contact email form and the crypto currency monitoring and notification service through which Lambda functions are triggered and provision temporary compute power in order to complete an action. This service was chosen because of a strong familiarity with the service and how it functions.

PYTHON

This programming language is used in the crypto currency monitoring and notification service mentioned previously. The language is used to construct the Lambda functions which perform the data collection and notification actions once triggered. This language was used for our Serverless functions because of its simple syntax and extensive documentation.

WORKMAIL

This is a cloud service provided by AWS which allows the creation of custom mail domains. We use this service during the development of the Gekko application to create a custom email address to be included on the websites contact form. This service was chosen due to the ease of integration with other AWS resources being used in this development.

ROUTE 53

This DNS service is used to resolve the Gekko domain and direct internet traffic to the Application Load Balancer which forward traffic to our host server. This service was chosen due to my own familiarity with the Route 53 DNS configuration. This allowed for a quick and effective set-up.

AMAZON CONNECT

This is the Call Center as a Service offering by AWS. This service allowed us to create and manage a support contact line for the Gekko Application. This center and the corresponding toll-free number will be outlined later in this document. This service was chosen due to its ease of use and deployment.

MYSQL

This querying language was used in conjunction with PHPMyAdmin in order to construct the back-end database to which user account were registered and stored. This will appear later in this document under the development section in relation to the mitigation of injection vulnerabilities. We chose this querying language for our development due to its association with the PHYMyAdmin tool which allows for a graphical interface database.

ALB

We use the Application Load Balancer at the edge of our public network to intercept incoming internet traffic over HTTPS and forward this traffic to the host server. This solution was chosen due to the ease at which an SSL certificate can be applied to the load balancer to support HTTPS.

GODADDY

Tis service was chosen as the domain registrar for the Gekko web domain due to its low pricing on the desired TLD.

REQUIREMENTS

The following section will outline the intended requirements for the finished application. These requirements have been divided into functional and non-functional requirements.

FUNCTIONAL REQUIREMENTS

In this section will define the functional requirements of the Gecko application. These requirements are the foundation for the application and will describe the functions that the application will achieve. Each requirement contributes to the previously specified description of Gecko and each will contribute to the user experience and ultimate validity of the application in the mainstream market.

The functional requirements are discussed below and have been placed in ranking order, beginning with the most important function of the application and descending to the least important.

1. Load Secure Website – As the web application deals with limited API methods, registered accounts as we will discuss later in this report, an intended live exchange, it is important to ensure that any connection to the web application is done so over SSL/TLS using the HTTPS method.

2. Display current coin pricing – The intention of the Gekko web application is to provide current pricing data for registered crypto currencies. For this reason, it is important that the application is able to retrieve currency pricing at each time the coin data is access.

3. Retrieve coin pricing graph – In order to present the gathered coin data in a user-friendly way, we will adopt a common graphical layout for financial information in the form of a ticker graph. These graphs will display the current pricing of the coin, market caps and most importantly the timeline graph of the price value over a twenty-four-hour period. These graphs will be embedded and retrieved through an API from .

4. Display pricing change percentage over defined intervals – On each coin page, it will be important to display the percentage of change in coin price over three defined time values. These time values will be once hour, twenty-four hours and seven days. The importance of this information is to show the user the likelihood of increase or drop of each coin based on its previous trend. This information is used when determining the profitability of any stock or currency in order to commit to a buy or sell transaction.

5. Account Creation – As mentioned in previous sections of this report, a key functionality of the Gekko application is to collect current coin prices and notify users of any large changes in pricing values. In order to push these notifications, we will have to collect a list of registered users who by registering for Gekko have opted into the notification service. An account creation/registration function will be implemented to allow customers to enter personal details which can then be added to our back-end notification system for publishing.

6. Notify users of pricing changes – From above, we understand that notifying users of price changes will occur after a user has registered for an account. Currently to inclusion of each new customer to the notification service is done manually as no automated process could be found to achieve this result. When a user is registered, they do so with an email address and optionally with an Irish mobile phone number. This data can then be extracted from our MySQL database and included in the AWS SNS topic to which price

changes are published. Publishing to the SNS topic will then produce a message which is delivered to each registered endpoint (SMS, Email).

7. Retrieve user submitted contact request – I feel it is important, to include a support section to this site which will help users in navigating and determining the value of information being displayed. It is also important to allow users to reach out to us directly to request the deletion of their account or the exclusion of their registered endpoint from Gekko notifications. Thus, we will include a contact section of the application which will include a HTML5 form which submits an email to a registered address. These emails can then be viewed through the registered Workmail account and responded to using the SES custom MAIL-FROM domain.

8. Change personal account information – Since we gather such information as Name and Phone number during the account registration phase, we should also allow users to change these details after creation. As such we will build an update function which allows users to submit a new name or phone number for the existing email entry of the account in our database.

9. Change password – We should also allow users to change the registered passwords for their accounts, should they wish to do so. For this process we will build a change password page on which the user will confirm the currently registered password and enter a new password to be submitted for the database account entry.

10. Direct users to registered exchange for coin purchase – As of now, the Gekko application has not deployed a stand-alone exchange through which users can purchase or sell their own currency shares. For this reason, each coin page will include a link to an exchange page from which they can buy or sell shares in the current coin. This links in with the Gekko functionality of providing coin information for users which can be then used in determining the appropriate action through the exchange (Buy/Sell).

NON-FUNCTION REQUIREMENTS

The following section will outline the non-functional requirements associated with the proposed application. The titles below represent the four areas of concentration when developing the application and within each section, we will investigate the need and proposed solution for each.

AVAILABILITY

Like most modern web and platform applications, it is important to have a high availability system. This means that the system is accessible and useable by its intended users at any time. This is specifically important for the Gecko application as financial markets are running globally at all times which use a follow the sun method.

High availability means that users will be able to retrieve coin information and analysis at all times to aid in the purchase and sale of stock units across all markets.

The high availability aspect of the application will be naturally implemented by the underlying architecture. The choice to use AWS as a back-end cloud architecture was one made due to its superior availability which guarantee's 99.99+ percent of availability at all times, across most services.

In distributing the application on the AWS platform, we also have the opportunity to use regional data centers across the globe. This will prove advantageous when users across the world access the application which can be leveraged

on hosts closer to the user's location. This too is an important aspect of availability as the reduction in latency for users located a large distance from the development site will be able to access the application with a smaller number of retrieval errors and with far less wait time in the application process loading.

SECURITY

As the Gecko application deals with the registration of users and their personal information. As such, security will be one of the most important non-functional requirements. Of course, in the current environment, with more and more daily aspects of life being migrated to digital and internet-based technologies, security has become one of the largest areas of interest and progression in computer science. The Gecko application is no exception to this trend and will include both inherited and integrated security functions in an effort to protect the application and its users.

As mentioned previously, the AWS platform has many advantages to self-hosted applications. One of these advantages is the established security of the platforms services which will act as the applications back-end. While AWS security is well documented on the internet, the organization also provides auditing reports per account which unfortunately cannot be shared to external parties and require an NDA to retrieve.

The use of AWS as a back-end service, will save time in the development process which would have been otherwise spent securing a server against intrusion. This time can be put back into the development life-cycle and focused on improving areas of the applications and its functionality.

RELIABILITY

It's important for any application to be reliable and avoid system failures or downtime. With the Gecko application, the reliability like the previous non-functional requirements will be inherited from the AWS platform which is a stable provider and has strong SLA's to support the implementation of any application type.

The reliability will also have to be monitored in the development of the application code, building redundancy into the architecture which can handle and resolve failures of all types such as data retrieval which should be able to be refreshed to avoid full system shut down.

EXTENDIBILITY

The prospects of growth for the Gecko application are vast as in the future it may be able to incorporate multiple exchanges and even a brokerage system through which users will be able to buy and sell currencies. For this reason, it is important to build an architecture which can be easily scaled up to incorporate advanced functions and also increased traffic.

Again, the AWS back-end structure will incorporate this extendibility requirement, with easily scaled infrastructures across server hosts, accessible regions and easily integrated services which can be provisioned to the final product seamlessly. The services in AWS also provide cross service dependency and integration which makes the development of new features easier than bottom up creation.

SYSTEM ARCHITECTURE

In order to begin building the Gekko application it was necessary to first build the underlying architecture on which the application environment was to be hosted. This section will outline the technologies previously mentioned as being used as the foundational infrastructure of the final Gekko application.

In order to represent the entire architecture of the application, a diagram has been created and included in figure X.X below. This diagram shows the flow of information and relationships between each resource used as part of the application architecture.

From the diagram above you can see the inclusion of a VPC which acts as our public network. This VPC contains the compute resources, within public subnets. These resources are used to host the Gekko website files. The edge of this VPC uses an ALB to accept and verify internet traffic from our Route 53 DNS. Once an event has occurred on the Gekko website, external resources are triggered/called through the use of an API. This API then forwards site requests to any resources not contained within the VPC.

In order to handle these requests and process necessary actions, Lambda functions are used as a severless compute middleman between the EC2 and endpoint resources. Effectively the Lambda functions used, receive API requests which are then parsed to retrieve the necessary action information. Through the use of SDK's these actions are then forwarded to the resource endpoint and invoked.

NETWORK

This section will outline the underlying network infrastructure which was provisioned to our environment in order to host the Gekko web application and handle internet traffic being passed to host server. The section includes the public networking structure, traffic handling through a provisioned load balancer and security resources to protect the environment from malicious or unauthorized access.

VPC

In order to host our computing resources and allow access from the public internet, we must define a public network on which our servers will exist. This is done through the use of IP ranging and an internet gateway. To begin, we create a Public VPC within AWS and associate a private IP CIDR range for this VPC. Any resource created within this VPC must then exist within this IP range.

SUBNET

Next, we build a public subnet within the previously created VPC and associate a private IP range within that of the VPC CIDR. In order to make this subnet public, we needed to associate a public internet traffic route through which all internet traffic would enter and exit. To do this we created an Internet Gateway and included a rule in our route table specifying all traffic (0.0.0.0/0) should be routed to this location. Route tables rely on explicit ruling, which means any traffic not already included in a defined rule, will be treated as being associated with the rule for all traffic (0.0.0.0/0). This essentially means that if we haven't defined how to handle a certain type of traffic, it is treated as internet traffic and routed to the internet gateway.

LOAD BALANCER

After creating and provisioning the above resources, we created an ALB which would accept, handle and route incoming internet traffic across two or more subnets in our VPC. This ALB was created with only a HTTPS listener on port 443 to ensure secure communication.

Security Group

To begin, we will look at the VPC mentioned earlier. Within the VPC outlined above, we now have a public subnet which routes internet traffic to an internet gateway. In order to secure the instances within this subnet, we construct a security group which defines rules for IP ranges and which ports are accessible by these ranges. As SSH is required by our development environment (Cloud 9), we opened port 22 for any range of IP's. The configuration of our ALB, which listens for internet traffic on port 443 passes traffic requests to our EC2 instances over port 80. This means that the security group required port 80 to be open to all IP's as ALB IP addresses are subject to change unless Static IP ranges are requested through AWS Support.

Serverless

As shown in the architecture diagram at the beginning of this section, the Gekko Web application use the AWS Lambda service in order to trigger functions for site events. Once these events occur on the hosting EC2, API Gateway is used to invoke and the relevant Lambda function and perform the developed action.

The following sub-sections will describe the basic architecture provisioned in order to allow each of the featured Lambda functions to be invoked and perform the desired action on the intended resource endpoints.

Crypto currency data retrieval

Through research of the AWS Lambda service, I discovered the Serverless application repository which hosts a large collection of pre-built Lambda functions and pre-configured resources. This repository pulls directly from the official AWS GitHub repository and the included submissions are those made by AWS Customers. These samples are open source and free to use.

From the Serverless Application Repository in the AWS Lambda service, I found an example of a Crypto Currency monitoring and notification solution. After analyzing this solution, I was able to reverse engineer the CloudFormation template, written in the YAML language, to determine the necessary resources needed to support the solution. Using the provided code and the required resource list, I was able to re-construct this solution using my own resources.

The first Lambda function involved in the solution, is a function which uses a HTTP API to retrieve currency pricing and market data from the Coin Market Cap API. This data and submitted to a DyanmoDB table for storage. For this architecture, it was necessary to create my own DynamoDB table as the storage endpoint for collected currency data.

In order to ensure my Lambda function was able to write to this database table, I used IAM Roles with explicit permission actions. The invocation of this function is then performed by a CloudWatch event which was configured as a scheduling rule using a rate expression of "rate(2 minutes)". This means that CloudWatch will trigger the Lambda function every two minutes.

Crypto currency pricing notification

From the solution outlined above, another function is used to analyze the currency data stored in the fore mentioned DynamoDB table. If this data meets a defined condition, this function will publish a notification to a defined SNS topic.

The formation of this architecture involved creating an SNS topic and registering an SMS and Email endpoint to receive any published notifications. In order to allow the Lambda function to publish to this Topic, we again adopted an IAM Role which explicitly defined permission to perform the required action on the resource.

Like the previous function, we assigned a CloudWatch event to this function which in the form of a schedule expression invokes the Lambda function every two minutes.

SERVERLESS CONTACT FORM

The final Lambda function created for the Gekko web application is one which accepts form data through an API post method. This function then formats an email message and pushing the data to the SES service in order to deliver the email message to a registered Gekko email address.

The formation of this architecture involved creating an API Gateway Resource and Post method in order to interact with the micro-service endpoint. This API collects the form data and posts it to the Lambda function which then parses the information, creates an email body and connects with the SES service in order to forward the mail body.

Again, IAM Roles were used to define the permissions of this Lambda function to perform actions on the SES service. In this case, the trigger which invokes the Lambda function is the API POST method.

For this function and those above, we will discuss the development and code included in each function in the feature development section later in this document.

IMPLEMENTATION

This section will provide a high-level description of how the various features present in the Gekko web application were developed. The following two sub-sections are divided by pure feature development and secure feature development. The difference between these two sub-sections is the focus on existing features and process of the application and how they were secured against common vulnerabilities. Each feature listed in the sections below will contain a unit testing result obtained from valid and invalid entry analysis of the feature after development.

FEATURE DEVELOPMENT

This section outlines the features of the web application and how they were developed. The following are in no particular order and do not carry any ranking in terms of importance.

COIN PRICE RETRIEVAL

One of the main and most basic features of the Gekko application is the retrieval and display of current crypto currency prices. In order to implement this feature, we used a HTTP API provided by Coin Market Cap which through its ticker endpoint can return up to date information on a set of coins or by individual coins specified by ID number.

The Coin Market Cap ticker URL can retrieve coin Symbol, ID, price and percentage changes for up to two hundred crypto currencies. Currently the Gekko application only supports 12 coins, which are all included in the Coin Market Cap, top twenty listed coins. For this reason, we filtered the URL to only return information on the top twenty coins.

By default, the API returns coin prices in US Dollars. Since Gekko is an Irish based application, we further edited the API URL in order to return coin pricing in EUR. The final URL used in the retrieval of currency data is as follows:

https://api.coinmarketcap.com/v2/ticker/?convert=EUR&limit=20

This API returns a JSON output which uses the master object "data", followed by coin ID objects to separate each coin. Within each block of coin data, we then have the "quotes" and "EUR" object parents in which the required information is held. A snippet of this JSON output showing the first coin in the list is shown below for reference:

{

```
"data": {
    "1": {
        "id": 1,
        "name": "Bitcoin",
        "symbol": "BTC",
        "website_slug": "bitcoin",
        "rank": 1,
        "circulating_supply": 17030287.0,
        "total_supply": 17030287.0,
        "max_supply": 21000000.0,
        "quotes": {
            "USD": {
                "price": 8519.81,
                "volume_24h": 6831800000.0,
                "market_cap": 145094809485.0,
                "percent_change_1h": 0.85,
                "percent_change_24h": 0.75,
                "percent_change_7d": -13.62
            },
            "EUR": {
                "price": 7123.28534385,
                "volume_24h": 5711965503.0,
                "market_cap": 121311593789.0,
                "percent_change_1h": 0.85,
                "percent_change_24h": 0.75,
                "percent_change_7d": -13.62
            }
        },
```

"last_updated": 1526166573

In order to retrieve the information relevant to the Gekko application, we will use the PHP coding language at the beginning of our web page. This PHP code calls the API URL outlined above and stores it in the variable "$url". We then use the "file_get_content" PHP method to extract the data from the URL. Once the data is stored in a variable we then parse the JSON data using the PHP "json_decode" method.

After retrieving and decoding the JSON data, we can then query our "$url" variable in order to extract the desired value, in this case the coin price. We do this by creating a variable for each coin. We then set this variable to the value of the coin price from the decoded JSON output by calling the "price" value after the nested parent objects.

A screenshot of this PHP code can be seen in figure 1 below this paragraph. Once the appropriate values have been assigned to the corresponding variables, we simply use a PHP echo command within each container where we wish the price of each coin to be shown. This can be seen in figure 2.

```php
<?php
$url = "https://api.coinmarketcap.com/v2/ticker/?convert=EUR&limit=20";
$data = file_get_contents($url);
$json = json_decode($data, true);

$btc = $json{data}{'1'}{quotes}{EUR}["price"];
$btc = number_format($btc, 2);
$eth = $json{data}{'1027'}{quotes}{EUR}["price"];
$eth = number_format($eth, 2);
$xrp = $json{data}{'52'}{quotes}{EUR}["price"];
$xrp = number_format($xrp, 2);
$bch = $json{data}{'1831'}{quotes}{EUR}["price"];
$bch = number_format($bch, 2);
$eos = $json{data}{'1765'}{quotes}{EUR}["price"];
$eos = number_format($eos, 2);
$ltc = $json{data}{'2'}{quotes}{EUR}["price"];
$ltc = number_format($ltc, 2);
$ada = $json{data}{'2010'}{quotes}{EUR}["price"];
$ada = number_format($ada, 2);
$xlm = $json{data}{'512'}{quotes}{EUR}["price"];
$xlm = number_format($xlm, 2);
$dash = $json{data}{'131'}{quotes}{EUR}["price"];
$dash = number_format($dash, 2);
$trx = $json{data}{'1958'}{quotes}{EUR}["price"];
$trx = number_format($trx, 2);
$neo = $json{data}{'1376'}{quotes}{EUR}["price"];
$neo = number_format($neo, 2);
$xmr = $json{data}{'328'}{quotes}{EUR}["price"];
$xmr = number_format($xmr, 2);
?>
```

```php
<div class="box alt">
    <div class="row uniform">
        <section class="4u 6u(medium) 12u$(xsmall)">
            <span><img src="images/btc.png"></span>
            <h3><a href="btc.php">Bitcoin (BTC)</a></h3>
            <h3>Price: €<?php echo $btc ?></h3>
        </section>
        <section class="4u 6u$(medium) 12u$(xsmall)">
            <span><img src="images/eth.png"></span>
            <h3><a href="eth.php">Ethereum (ETH)</a></h3>
            <h3>Price: €<?php echo $eth ?></h3>
        </section>
        <section class="4u$ 6u(medium) 12u$(xsmall)">
            <span><img src="images/xrp.png"></span>
            <h3><a href="xrp.php">Ripple (XRP)</a></h3>
            <h3>Price: €<?php echo $xrp ?></h3>
        </section>
        <section class="4u 6u$(medium) 12u$(xsmall)">
            <span><img src="images/bch.png"></span>
            <h3><a href="bch.php">Bitcoin Cash (BCH)</a></h3>
            <h3>Price: €<?php echo $bch ?></h3>
        </section>
        <section class="4u 6u(medium) 12u$(xsmall)">
            <span><img src="images/eos.png"></span>
            <h3><a href="eos.php">EOS (EOS)</a></h3>
            <h3>Price: €<?php echo $eos ?></h3>
        </section>
        <section class="4u$ 6u$(medium) 12u$(xsmall)">
            <span><img src="images/ltc.png"></span>
            <h3><a href="ltc.php">Litecoin (LTC)</a></h3>
            <h3>Price: €<?php echo $ltc ?></h3>
        </section>
```

Unit Testing

Since the implementation of this feature is based on the retrieval of information from a HTTP API, we must ensure that the API is able to connect and return the required JSON data over the intended HTTPS connection on which the website will operate. We already know from the API URL outlined previously that Coin Market Cap use a HTTPS connection when returning API calls, but it is necessary to validate the return of data to another endpoint which operates over the same protocol.

In order to test this, we implement the HTTPS connection of our application through the Application Load Balancer and test repeatedly. Each time the page is refreshed, the data is retrieved successfully without error. This confirms that the HTTPS connection on both ends is not restricted and the information can be shared over secure connection with both endpoints.

Another area of testing is the scalability of our application against the API request limit imposed by Coin Market Cap. The documentation of this website does not explicitly detail a hard limit on API calls, it does however request that no

more than ten calls are made per minute. To test this, we created a Lambda function which would simply call the API URL and return the JSON information.

The purpose of using a Lambda function for this test is to ensure the rapid invocation of API requests without waiting for a website reload delay. After creating and invoking this function a total of 15 times in a one-minute period, each invocation returned the desired output. This tells us that while Coin Market Cap ask that the API is limited to ten calls per minute, this is not a hard limit and some excess calls can still be handled.

COIN STATISTICS

For each coin page, we must retrieve the percentage change in price over the past one hour, twenty-four hours and seven days. In order to do this, we can again use the Coin Market Cap API. Like in the previous feature, we will use PHP code at the beginning of each coin page to call the API URL and assign it to a variable. We will again get the content of the URL and parse the JSON output using the PHP decode facility.

Unlike the previous example however, for this feature we will not be retrieving data for more than one coin per page. As each coin has its own web page on which a graph and percentage changes are displayed we can call only the data relevant to the coin which is being viewed. To do this, we adjust the URL to call only the desired coin by its ID number. As an example, the following URL will call the relevant data for the Bitcoin (BTC) coin page where Bitcoin is identified through the API as ID "1":

$url = "https://api.coinmarketcap.com/v2/ticker/1/?convert=EUR"

Once the JSON has been obtained, we will use PHP variables to store the percentage changes by calling the JSON values using the parent objects. In order to produce a red or green color for the negative and positive changes respectively, we used if statements. The condition of these statements is to compare the retrieved value against 0. If the value is less than 0 we assign the color variable a value of "red". Whereas, if the retrieved value is greater than 0, we assign the color variable a value of "green". This code can be seen in figure 3.

```php
<?php
$url = "https://api.coinmarketcap.com/v2/ticker/1/?convert=EUR";
$fgc = file_get_contents($url);
$json = json_decode($fgc, true);

$p1 = $json{data}{quotes}{USD}["percent_change_1h"];
$p2 = $json{data}{quotes}{USD}["percent_change_24h"];
$p3 = $json{data}{quotes}{USD}["percent_change_7d"];

if($p1 < 0){
  $color1 = "red";
}
if($p1 > 0){
  $color1 = "green";
}
if($p2 < 0){
  $color2 = "red";
}
if($p2 > 0){
  $color2 = "green";
}
if($p3 < 0){
  $color3 = "red";
}
if($p3 > 0){
  $color3 = "green";
}
?>
```

In order to then display these variables and the corresponding values in our HTML containers, we again use the PHP echo commands to return the percentage change values in a HTML table. In order to produce the corresponding color, we insert another PHP echo command as a styling attribute in the form of inline CSS inside the container tag. An example of both the value echo and color echo commands can be seen in figure 4 below.

```
<!-- Sidebar -->
    <section id="sidebar">
        <section>
        <div class="tbl-header">
        <table cellpadding="0" cellspacing="0" border="0">
            <tr>
              <td>Percent change 1hr:</td>
              <td style="color: <?php echo $color1; ?>;"><?php echo $p1 ?>%</td>
            </tr>
            <tr>
              <td>Percent change 24hr:</td>
              <td style="color: <?php echo $color2; ?>;"><?php echo $p2 ?>%</td>
            </tr>
            <tr>
              <td>Percent change 7d:</td>
              <td style="color: <?php echo $color3; ?>;"><?php echo $p3 ?>%</td>
            </tr>
        </table>
      </div>
      </section>
```

FIGURE 4

UNIT TESTING

As this feature relies on the same API used in the previous feature, we can validate by extension that our testing of both the HTTPS connection and the API call limit is confirmed to have passed.

SERVERLESS CONTACT

The next feature to be discussed as part of this section is the inclusion of an email contact form which uses a Serverless back-end in order to form and forward the message body to a registered email address.

The contact form located on the site contact page, uses HTML form and input tags to determine the input values. In order to retrieve the values for these input fields and pass them to the relevant Lambda function, we need to create an API which can be used to reach the Lambda function endpoint. To do this, we created an API Gateway method which passes a JSON request to Lambda. This API is shown in figure 5 below.

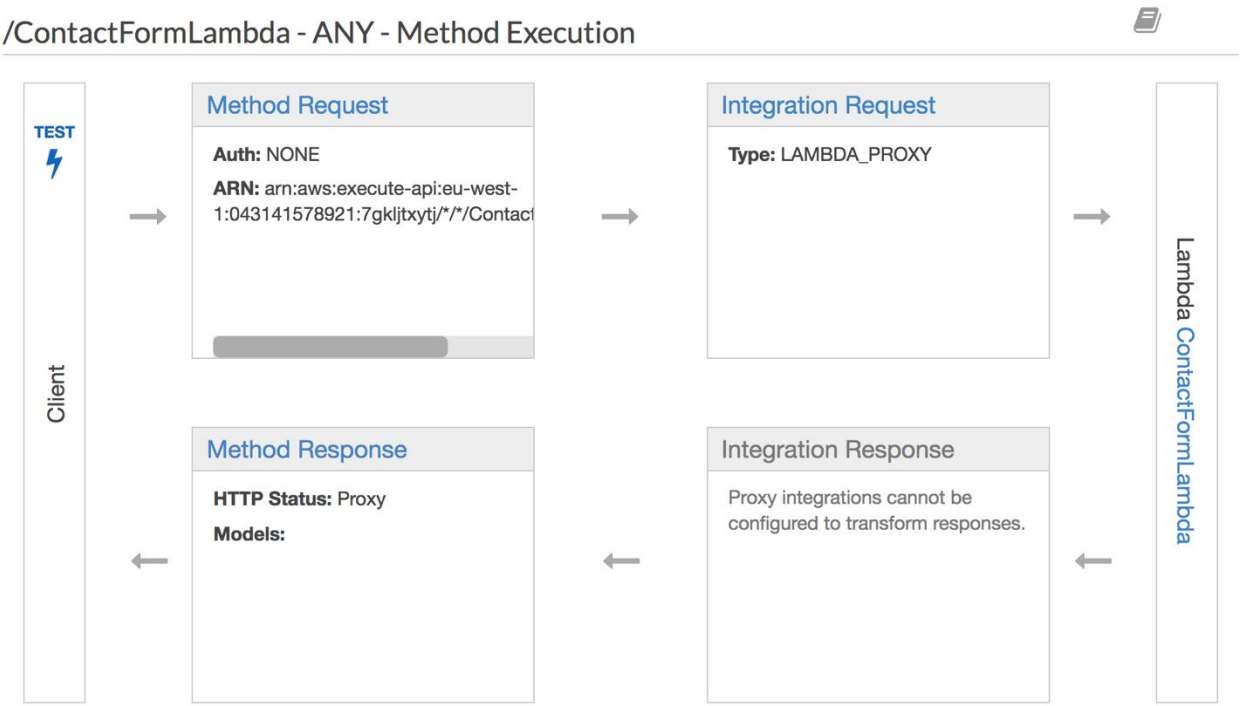


FIGURE 5

After creating the API method, we deploy our API and configure it with the Lambda invocation permissions to allow the API to act as a function trigger. Once the API has been configured, we create a Lambda function which will retrieve and parse the JSON data from the API call. After parsing the data, the "getEmailMessage" method is then called, to format the final email body. Once the message is formatted correctly, the email is pushed to the SES service for final delivery using the "sesClient.sendEmail" operation. The Node.js code used to create the Lambda function which parses and sends the email body can be seen in figure 6 below.

```
exports.handler = (event, context, callback) => {
    console.log('Received event:', JSON.stringify(event, null, 2));
    var emailObj = JSON.parse(event.body);
    var params = getEmailMessage(emailObj);
    var sendEmailPromise = sesClient.sendEmail(params).promise();

    var response = {
        statusCode: 200
    };

    sendEmailPromise.then(function(result) {
        console.log(result);
        callback(null, response);
    }).catch(function(err) {
        console.log(err);
        response.statusCode = 500;
        callback(null, response);
    });
};

function getEmailMessage (emailObj) {
    var emailRequestParams = {
        Destination: {
          ToAddresses: [ sesConfirmedAddress ]
        },
        Message: {
            Body: {
                Text: {
                    Data: emailObj.message
                }
            },
            Subject: {
                Data: emailObj.subject
            }
        },
        Source: sesConfirmedAddress,
        ReplyToAddresses: [ emailObj.email ]
    };

    return emailRequestParams;
}
```

FIGURE 6

As we do not wish to receive any spam messages to our private email address, we create an Amazon Workmail address. Amazon Workmail is essentially an email server for business and provides custom @ domains. The registered email address which is used as the endpoint for the above SES message is "contact@gekko.awsapps.com".

Implementation of this in our front-end web page is done so using an inline JavaScript which reads the input from the form using query selectors identified by the input field classes. This JavaScript defines the endpoint for the Lambda function and forms the email message before submitting using a POST method. This JavaScript can be seen in figure 7 below.

```
</form>
<!-- API -> Lambda handler -->
  <script type="text/javascript">
        // Adds an event listener to our form. When the form is submitted, it will send data to our Lambda function, which in turn, will send us
        document.getElementById('serverless-contact-form').addEventListener('submit', sendDataToLambda);
        // Now for the good stuff. This is the function that will send our data to AWS.
        function sendDataToLambda(e) {
          e.preventDefault();
          // Gets the values of each field in our form. This is the data we'll send to our Lambda function.
          var formEmail = document.querySelector('.form-email').value;
          var formSubject = document.querySelector('.form-subject').value;
          var formMessage = document.querySelector('.form-message').value;
          // This is the endpoint we created in our API Gateway. This is where we make our POST request, which calls our Lambda function.
          var endpoint = 'https://7gkljtxytj.execute-api.eu-west-1.amazonaws.com/prod/ContactFormLambda';
          // Remember those form values we just grabbed? We're going to put them into an object here.
          var body = {
            email: formEmail,
            subject: formSubject,
            message: formMessage
          }
          // Here, we instantiate our Request. This is a special object used by the Fetch API so it knows where to send data, what data to send
          var lambdaRequest = new Request(endpoint, {
            method: 'POST',
            // Quick note: 'no-cors' mode is for development on localhost only!
            mode: 'no-cors',
            body: JSON.stringify(body)
          });
          // Call the Fetch API to make our request
          fetch(lambdaRequest)
            // This is where you can handle errors. This is just an example, so we won't cover that.
            .then(response => console.log(response))
            .catch(err => console.log(err));
        }
  </script>
```

FIGURE 7

UNIT TESTING

We want to make sure that the messaging through the provided contact form is restricted to only allow legitimate messages that contain an email address, subject and message body. In order to test this, we will submit the form a number of times using a combination of all, some and no information being entered in the input fields.

Submitting a form which includes an email address, subject and message body entry we receive the email in our destination inbox. Unfortunately, after re-submitting this form with missing information, we continue to receive email messages missing different portions of information. The final combination of no input, again produces an email in the "contact.gekko.awsapps.com" email inbox. Given more time for development this fault can be easily rectified by including if statements in the JavaScript code which checks for null inputs. If null inputs are then expected the form can produce an error message instead of a successful delivery.

CALL CENTER

A small but interesting feature of the Gekko application which might otherwise be easily overlooked is the inclusion of a toll-free call center phone number. This phone number was obtained through the Amazon Connect service which provisions call centers as a service.

This cloud service allows users to build fully fledge call centers including customer queues and contact flows through a graphical interface. The Gekko support call center was created using an Irish toll-free number (1800-938-588). Once the call center was created a basic contact flow was published to this phone number. The contact flow, uses text to speech input in order to retrieve inbound calls and output an automated voice message alerting customer of the upcoming Gekko exchange.

Once the contact flow was finalized with a termination command, the flow was saved and published. The toll-free number is then configured with the new contact flow. The graphical representation of this call flow and the provisioned text to speech prompt can be seen in figure 8 below.

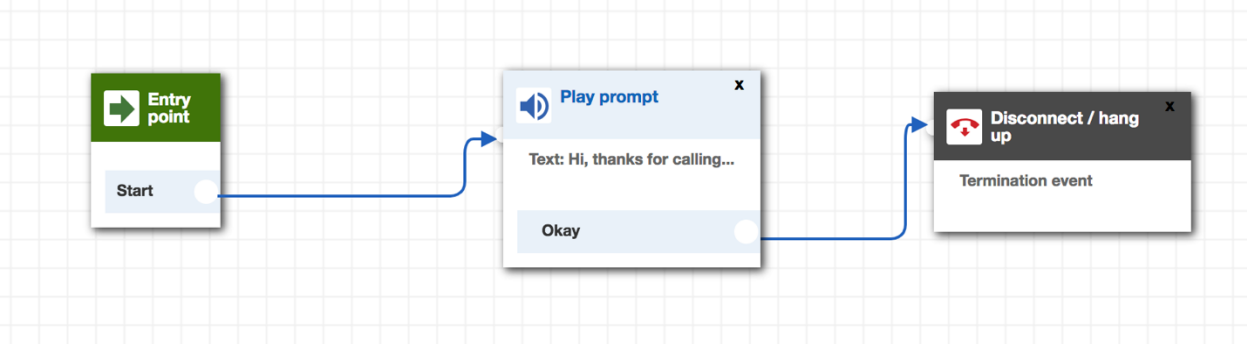


FIGURE 8

UNIT TESTING

There is, very limited testing abilities for this site feature. In order to ensure that the toll-free number can be reached during an inbound contact and that the text to speech prompt plays back correctly our testing consists of a number of test calls made to the number.

Each time the toll-free number was called, the flow performed as expected. For this reason, the testing of this unit is considered to have passed.

SERVERLESS PRICE COLLECTION

As mentioned previously in the architecture section of this document, one of the main features of the Gekko application is a monitoring and alert system based on AWS Serverless resources, which will retrieve coin prices, analyze changes in price values and notify customers of large price changes where the price change minimum threshold is defined in our function code.

The development of this feature involves two Lambda functions. The first function is used to retrieve coin prices and market data from the Coin Market Cap API. Once this information is retrieved by the function code it is inserted into a DynamoDB table. As mentioned before, this solution was obtained through the AWS Serverless Application

Repository as a full stack deployment template. This template was reverse engineered and the resources were created manually for this application.

For the Lambda functions we choose a Python 3.6 runtime, meaning that the Python 3.6 coding language is used in the function handler to perform the necessary actions. The first thing needed when developing this function is to import the boto3 Python SDK. This SDK contains a large number of Python operations used for AWS Resources. This will be needed in order to perform actions on the Amazon DynamoDB table.

When developing the function code, we use logger.info in order to print a debugging statement to the CliudWatch log stream which details each invocation of the Lambda function. Before creating the handler and methods, we also define the dynamo variable as a client of the boto3 SDK. We then define the variable "TABLE_NAME" in order to assign the correct DynamoDB table to the actions being performed. This variable is defined as an os.environ variable, which means the variable value is obtained outside of the function code but rather is defined in the Lambda resources environment variables section.

After defining the boto3 client and environment variable, we then create an insert method which calls the DynamoDB "put_item" operation to insert the retrieved coin data into the DB table. After this method we create the event handler which is where the API is called and the raw JSON data is imported. This handler contains a for loop which creates a counter of each new entry. This is to ensure that each new entry under the same coin Id (Table primary key) does not overwrite the previous entry. This is to ensure the analysis of price changes is accurate for the next Lambda function.

The Python code used to create this Lambda function can be seen in figure 9 below and can be found in the submitted code folder.

```
logger = logging.getLogger()
logger.setLevel(logging.INFO)
logger.info('Loading function')

logger.info('Loading function')
dynamo = boto3.client('dynamodb')

TABLE_NAME = os.environ['TABLE_NAME']

def insert(json, timestamp):
    # dynamo db insert logic
    response = dynamo.put_item(TableName=TABLE_NAME,
                               Item={
                                   'CoinId': {"S": str(json['symbol'])},
                                   'Price': {"N": str(json['price_usd'])},
                                   'TimeStamp': {"N": str(timestamp)},
                                   'Name': {"S": str(json['name'])},
                                   'Rank': {"N": str(json['rank'])},
                                   'PriceBTC': {"S": str(json['price_btc'])},
                                   '24hVolumeUSD': {"S": str(json['24h_volume_usd'])},
                                   'MarketCapUSD': {"S": str(json['market_cap_usd'])},
                                   'AvailableSupply': {"S": str(json['available_supply'])},
                                   'TotalSupply': {"S": str(json['total_supply'])},
                                   'MaxSupply': {"S": str(json['max_supply'])},
                                   'PctChange1hr': {"S": str(json['percent_change_1h'])},
                                   'PctChange24hr': {"S": str(json['percent_change_24h'])},
                                   'PctChange7d': {"S": str(json['percent_change_7d'])}
                               }
                              )

def lambda_handler(event, context):
    all_current_prices = requests.get("https://api.coinmarketcap.com/v1/ticker/?limit=20").json()
    timestamp = calendar.timegm(time.gmtime())
    counter = 0

    for price in all_current_prices:
        counter += 1
        logger.info("Starting %s" % price['name'])
        insert(price, timestamp)

    return "Successfully inserted {} coins.".format(counter)
```

FIGURE 9

UNIT TESTING

As we have seen before in this section, the retrieval of data through a HTTP API is relatively secure and entirely functional within our environment. The only testing which needs to be conducted for this unit, is the invocation itself in order to ensure that the data is being imported and written to the DynamoDB table.

Fortunately, the Lambda service provides its own testing facility for each function. To utilize this facility, we create a test input and run it within the Lambda console, after a short processing time, the result of this test is returned as successful and the debugging statement outputs a confirmation message of the data being written to the DB table.

After setting an event trigger for this function to run every minute, we observe the DynamoDB table and confirm the data is being written to the table and is not overwriting any previous entries for same CoinId.

PRICE CHANGE ALERT

As mentioned in the previous section, one of the main features of the Gekko application is the monitoring and notification of price changes in defined crypto currencies. We have already outlined the development of the Lambda function which retrieves pricing and market data and then writes this data to a DynamoDB table. In this section we will outline the development of the second Lambda function which regularly analyzes the DynamoDB data and if any

price change over a defined threshold is detected, forms and publishes a message to an SNS topic containing user endpoints.

As we did for the previous Lambda function, we begin the development of this function by choosing the Python 3.6 runtime and importing the boto3 Python SDK. We then define the variables needed to perform the coded actions. These variables include the list of coins to be evaluated denoted by the "COIN_LIST" variable, the SNS topic which is to be published to denoted by the "SNS_TOPIC" variable, the percentage cutoff value which is the value at which the SNS notification is published denoted by the "PCT_CHANGE_CUTOFF" variable and finally the DynamoDB table name denoted by the "TABLE_NAME" variable. Each of these variables are defined in the Lambda functions environment variables through the Lambda console.

Once all variables have been defined we create a "get_coins" method to retrieve and return the list of coins to be evaluated. The next method forms the response received from the DynamoDB query and appends the object before returning. This response format is then submitted as a query to the table in order to get the list of prices for the corresponding list of coins.

Another method is then created to analyze the most recent price entry against the previous to determine if the percentage of change between both price values is above or below the percentage threshold defined earlier. If this value is above the percentage threshold, the Lambda handler executes the notify user method which uses the boto3 SNS client and publish operation in order to submit the formatted notification to the SNS topic which contains an SMS or Email endpoint.

A screenshot of the Lambda function code from within the Lambda console can be seen in figures 10 and 11 below. These screenshots show the defined Python methods used in the analysis, formation and publication of the notification message.

```
def get_coins(string_list):
    coin_list = string_list.split(",")
    return coin_list

def dynamo_response_to_list(response):
    prices = []
    for item in response['Items']:
        new_obj = {'CoinId': item['CoinId']['S'],
                   'Price': Decimal(item['Price']['N']),
                   'TimeStamp': int(item['TimeStamp']['N'])
                   }
        prices.append(new_obj.copy())
    return prices

def get_prices(coin, now):
    tminus1hr = now - 3600

    response = dynamo.query(TableName=TABLE_NAME,
                            KeyConditionExpression="CoinId = :Coin AND #TS > :SortKey",
                            ExpressionAttributeValues={":Coin": {"S": coin},
                                                       ":SortKey": {"N": str(tminus1hr)}},
                            ExpressionAttributeNames={"#TS": "TimeStamp"}
                            )

    return dynamo_response_to_list(response)

def notify_user(big_changes):
    message = "\n"

    for big_change in big_changes:
        message += "{0} changed {1:.3g} percent, the new price is ${2:.4}".format(
            big_change["symbol"],
            big_change["pct_change"],
            big_change["new_price"])
        message += "    \n"
    logger.info("Sending message: ".format(message))
    sns_topic.publish(Message=message)
```

FIGURE 10

```
def get_price_change(coin, prices, percentage_change_cutoff):

    if len(prices) < 2:
        logger.info("Not enough recent data for %s to determine there was a significant price change." % coin)
        return {}

    sorted_prices = sorted(prices, key=lambda k: k['TimeStamp'], reverse=True)

    current_price_obj = sorted_prices[0]
    old_price_objs = sorted_prices[:10]

    for old_price_obj in old_price_objs:
        current_price = current_price_obj['Price']
        old_price = old_price_obj['Price']

        pct_change = ((current_price - old_price) / old_price) * 100

        logger.info("CP: %d \t OP: %d \t Pct: %d" % (current_price, old_price, pct_change))

        if abs(pct_change) > Decimal(percentage_change_cutoff):
            logger.info("Percent price change was big enough for %s" % coin)
            return {"symbol": coin, "pct_change": pct_change, "new_price": current_price}

        logger.info("Percent change wasn't big enough for %s." % coin)
    return {}

def lambda_handler(event, context):
    timestamp = calendar.timegm(time.gmtime())
    big_changes = []

    coins = get_coins(COIN_LIST)

    for coin in coins:
        logger.info("Starting %s" % coin)
        prices = get_prices(coin, timestamp)
        big_change = get_price_change(coin, prices, PCT_CHANGE_CUTOFF)

        if big_change:
            logger.info("Adding big change for %s " % coin)
            logger.info(big_change)
            big_changes.append(big_change)
    logger.info("Big changes:")
    logger.info(big_changes)
    if big_changes:
        notify_user(big_changes)
```

FIGURE 11

UNIT TESTING

In order to test the expected functionality of this function, we add an email address to the SNS topic. We then enter the DynamoDB table and manually edit entered values in order to breach the defined threshold.

After editing these values, we allow the scheduled event to trigger and invoke the function, after which we receive an email in the registered inbox alerting us of a currency price change for the coins and values we entered into the DynamoDB table.

This test is therefore deemed successful.

ACCOUNT DETAILS UPDATE

While the registration and login system is not in these circumstances deemed as an application feature, we will include the development of a registered user information update system as an included feature. This feature, allows users to update their registered information through the website profile page. The user can choose to edit their name or phone number which will overwrite the previously entered values.

In order to implement this feature, we use the PHP programming language to define an update method in our User.php class contained within the login system. This method defines an if condition under which a user must be logged in to update information. This method then uses the $this variable defined through the login system to push the newly entered values to the currently logged in user.

This method is then implemented in a PHP page and is called using another if condition which determines if a login is detected. If a user is logged in, this condition is met and the update method is called in order to update the user DB entries. The new values are obtained from the HTML form inputs recognized by the input field names.

Figure 12 below shows a screenshot of the development environment which outlines the implementation of the if condition and method call described above.

```
        if($validation->passed()){
            try{
                $user->update(array(
                    'name' => Input::get('name'),
                    'phone' => Input::get('phone')
                    ));
                Session::flash('home', 'Your details have been updated');
                Redirect::to('profile.php');
            }
            catch(Exception $e){
                die($e->getMessage());
            }
        }
        else{
            foreach($validation->errors() as $error){
                echo $error, '<br>';
            }
        }
    }
}
```

FIGURE 12

UNIT TESTING

In order to test this unit, we will attempt to submit an empty form which should fail as through our implementation we have define the name array as being a required value. We will also attempt to access this function via URL without a valid login session.

We begin by login in as a registered user and then navigating to the update page. From here we remove the current name value which appears in the name input field. Now that we have a blank form we choose the submit button which refreshes the page and re-enters the original value as the placeholder for the name input field. This in turn is not reflected in the database entry for the user. Therefore, this test is successful.

We finish this unit test by attempting to navigate to the update.php URL without a valid login session. This continuously redirects to the login.php web page as the login condition was not met. This test is also considered a success.

SECURE DEVELOPMENT

This section will outline the steps carried in order to implement a secure environment and to introduce secure features of existing application features and systems. This section is divided into sub-sections which focus on the development of secure enhancements of each relevant section of the application.

HTTPS

As we have outlined in previous sections, we are using an Application Load Balancer to receive and handle incoming internet traffic to the application. This ALB then passes internet traffic to the host server over port 80. However, the initial connection made between the client and the ALB is over HTTPS on port 443.

In order to implement a HTTPS connection, we must first obtain a signed SSL certificate. While unsigned certificates can be used to create HTTPS connection, these certificates are un-validated and cannot guarantee a secure connection. Hence, our decision to obtain a signed certificate.

We chose Amazon as our certificate authorizer since the environment and resources are provisioned by this platform and requested this certificate through the ACM service. Once the certificate was obtained from Amazon, we provisioned the SSL certificate to the HTTPS listener of the ALB. This provides a validated secure connection when connecting to the ALB through the registered domain name.

SSL testing

For testing purposes, we will use the SSLLabs testing tool to evaluate the grade of our server and HTTPS connection. To begin we initiate a test on our site domain which establishes a connection with the IP address of the domain server.

After a few moments of testing the connection, the SSLLabs tool returns a grade for both the ALB IP and EC2 host server. Both results of this test return a grade A result for the ALB and EC2 server. This test is therefore deemed successful. The result of this test can be seen in figure 13 below.

	Server	Test time	Grade
1	52.48.170.206 ec2-52-48-170-206.eu-west-1.compute.amazonaws.com Ready	Sun, 13 May 2018 02:31:59 UTC **Duration:** 133.879 sec	A
2	54.154.250.35 ec2-54-154-250-35.eu-west-1.compute.amazonaws.com Ready	Sun, 13 May 2018 02:34:13 UTC **Duration:** 135.467 sec	A

FIGURE 13

PASSWORD HASHING

In order to secure the passwords of registered users in the user database, we will use hashing to conceal passwords as encrypted strings. This means that password entries are not stored in plain text and therefore cannot be viewed within the database table.

We achieve this password hashing through our PHP login system by using a hash method defined in the Hash.php class. This method creates a $salt variable which is used in the SHA256 encryption of password entries to produce a unique encrypted string to replace the plain text password entry in the DB table. This method within the Hash.php class can be seen in figure 14 below.

```php
<?php
class Hash{
    public static function make($string, $salt = ''){
        return hash('sha256', $string . $salt);
    }

    public static function salt($length){
        return mcrypt_create_iv($length);
    }

    public static function unique(){
        return self::make(uniqid());
    }
}
```

FIGURE 14

This hashing method is called during the registration process on the register.php page. The password input is collected from the registration form and the make method is passed through the Hash class to form the hashed password value during account creation. This can be seen in figure 15 below.

```php
try{
    $user->create(array(
        'username' => Input::get('username'),
        'password' => Hash::make(Input::get('password'), $salt),
        'salt' => $salt,
        'name' => Input::get('name'),
        'joined' => date('Y-m-d H:i:s'),
        'groups' => 1,
        'phone' => Input::get('phone')
        ));

    Session::flash('home', 'You have been registered');
    Redirect::to('login.php');
}
```

FIGURE 15

This method is also called in the changepassword.php page in order to create a hash value for the new password entry that is replacing the previously registered password value.

UNIT TESTING

The only testing method for this unit is to register a new user and observe the hashed value of the entered password in the database directly. In doing so we create a new user with a valid password, entered in plain text to the registration form.

After submitting the registration details, we inspect the database entry for this new user and observe the hashed value under the password column of the user entry. An example of this password hashing entry can be found in image X.X below.

SESSION MANAGEMENT

In order to ensure that login sessions were handled correctly by our login system, we introduced a system of session management where by any new login will be assigned a session which exists while the application is open and the user has not chosen to logout.

This session is produced by the Session.php class which is called during login. If valid conditions are met for a successful login, the session method is called in order to make a session. The Token class is then called in order to create a session and token and assign this token to the current session. This session is then in turn assigned to the name of the user logging in. The session method, method call during login and session token creation can be seen in figures 16, 17and 18 below.

```php
<?php
class Session{
    public static function exists($name){
        return (isset($_SESSION[$name])) ? true : false;
    }
    public static function put($name, $value){
        return $_SESSION[$name] = $value;
    }

    public static function get($name){
        return $_SESSION[$name];
    }

    public static function delete($name){
        if(self::exists($name)){
            unset($_SESSION[$name]);
        }
    }

    public static function flash($name, $string = ''){
        if(self::exists($name)){
            $session = self::get($name);
            self::delete($name);
            return $session;
        }
        else{
            self::put($name, $string);
        }
    }
}
```

```php
public function login($username = null, $password = null){
    $user = $this->find($username);

    if($user){
        if($this->data()->password === Hash::make($password, $this->data()->salt)){
            Session::put($this->_sessionName, $this->data()->id);
            return true;
        }
    }

    return false;
}
```

FIGURE 17

```php
<?php
class Token{
    public static function generate(){
        return Session::put(Config::get('session/token_name'), md5(uniqid()));
    }

    public static function check($token){
        $tokenName = Config::get('session/token_name');

        if(Session::exists($tokenName) && $token === Session::get($tokenName)){
            Session::delete($tokenName);
            return true;
        }
        return false;
    }
}
```

FIGURE 18

Once a user decides to logout of the system, this session is deleted by the logout.php page which calls the logout method. This method within the User class in turn deletes the current session prompting for a new login once a user returns to the application. The logout method called during the log out process can be seen in figure 19 below.

```php
public function logout(){
    Session::delete($this->_sessionName);
}
```

FIGURE 19

PREPARED STATEMENTS

To ensure that users not able to submit SQL queries directly to the back-end database through the use of HTML input fields, we have implemented prepared statements for handling the creation, update, deletion and access of data in the DB.

The implementation of prepared statements involved defining specific query strings within our DB.php class. This involved creating methods for accessing and querying information within the DB. These methods define a specific query string with attributing variables in place of the table name and value lookup. When a user is submitting information, which involves querying the DB, these methods are called and the users input values are validated using the validate class. The values inserted into this defined query string in order to complete the process. A snippet of the DB class and defined prepared statements can be seen in figure 20 below.

```
public function get($table, $where){
    return $this->action('SELECT *', $table, $where);
}

public function delete($table, $where){
    return $this->action('DELETE', $table, $where);
}

public function insert($table, $fields = array()){
        $keys = array_keys($fields);
        $values = '';
        $x = 1;

        foreach($fields as $field){
            $values .= '?';
            if($x < count($fields)){
                $values .= ', ';
            }
            $x++;
        }

        $sql = "INSERT INTO users (`" . implode('`, `', $keys) . "`) VALUES ({$values})";

        if(!$this->query($sql, $fields)->error()){
            return true;
        }

    return false;
}
```

FIGURE 20

We have also included a screenshot of the method being called during the login process which uses the get method to pass a SELECT query to the back-end database. This screenshot can be seen in figure 21 below. Figure 22 shows the validate class calling the DB class's getInstance method in order to perform validation.

```php
$validate = new Validate();
$validation = $validate->check($_POST, array(
    'username' => array('required' => true),
    'password' => array('required' => true)
    ));

if($validation->passed()){
    $user = new User();
    $login = $user->login(Input::get('username'), Input::get('password'));
```

FIGURE 21

```php
public function __construct(){
    $this->_db = DB::getInstance();
}
```

TESTING AND EVALUATION

This section will outline the unit tests performed in the implementation section. Each test will carry a pass or mark fail to signify a successful or unsuccessful implementation against the proposed test. All tests in the table below are unit tests as described above.

API is able to connect and return the required JSON data over the intended HTTPS connection.	**PASS**
Scalability of our application against the API request limit.	PASS
Contact form is restricted to only allow legitimate messages that contain an email address, subject and message body.	FAIL
Ensure that the toll-free number can be reached during an inbound contact.	PASS
Ensure that the data is being imported and written to the DynamoDB table.	PASS

Pricing notification manual entry testing.	PASS
Submit an empty form which should fail as through our implementation we have define the name array as being a required value.	PASS
SSL connection testing.	PASS

OWASP VULNERABILITY MITIGATION

In this section we will outline the vulnerabilities relevant to the Gekko Application as defined in the OWASP Top 10. These vulnerabilities are listed as the ten most prevalent in modern web applications and environments. As such, is important that we list any possible vulnerability of this application and detail the steps taken to mitigate against these vulnerabilities.

INJECTION

Injection relates to a number of different injection techniques. The most well known and most commonly present in web applications is SQL injection. This vulnerability is the ability for a malicious entity to enter SQL queries into a site or application input field which directly queries database information.

If this vulnerability is present in an application it may allow for example, a malicious entity during login to pass an SQL DROP query instead of a password which could compromise the integrity of the application data.

MITIGATION

To mitigate against the possibility of SQL injection on the Gekko application, we use prepared statements to define the allowed SQL query for each action. These prepared statements are then populated with attributes based on user input such as a table entry for a login process.

This mitigation technique is the most widely used in development and proves highly effective against this type of attack.

BROKEN AUTHENTICATION

Broken authentication can cover a number of flawed configurations, one of which is session management. In this scenario, a user may be able to access the account of a user that had been previously logged in through a machine or IP address. This occurs when session management is broken and a session is not deleted after a user exits the system

MITIGATION

In order to mitigate against this type of vulnerability we have implemented a system of session management whereby session is assigned during login and deleted during the log out process. We have also chosen not to develop cookie management in this application to defend against "remember me" sessions which may continue after the user has exited the system.

SENSITIVE DATA EXPOSURE

This vulnerability is present where an application does not protect its sensitive data against plain view. For example, this could occur on a site forum when user account details are shown in plain text under the poster's profile information.

MITIGATION

To mitigate against this type of attack, all user information deemed sensitive is stored within the database and is not returned to the application in plain text at any stage other than a login session. During login session this information is also hidden and only queried from the database for the purpose of validation.

USING COMPONENTS WITH KNOWN VULNERABILITIES

Many technologies have known vulnerabilities which are consistently being patched against new and emerging flaws. One such known vulnerability is the Shellshock vulnerability present in early version of Apache web server. This vulnerability allows remote execution of scripting code through CGI webpages stored on the web server.

MITIGATION

To mitigate against this vulnerability all software and packages were updated during development and will continue to undergo regular patching while the application is in production.

INNOVATION

While the concept of this application is not new and many like it exist across the internet, I feel that the inclusion of a Serverless back-end for the notification of price changes in crypto currencies is an innovation into the area of Serverless technologies.

This technology is still new and is continuously being adopted by more and more solutions as a valid option for server retirement. The concept of provisioning compute power only when necessary and only when triggered is one which is expanding at an alarming rate as more developers discover uses for the Lambda service.

COMPLEXITY

While the development of the Gekko Application was not complex, the underlying structure of Serverless technology and a custom API Method to call the required micro-services was a complex task which presented an interesting result. The complexity of this site, while not evident on the surface does show in the back-end coding in the form of its cloud-based resources and secure login system.

SYSTEM EVOLUTION

The Gekko application has experienced a number of evolutions throughout the course of development due to time and knowledge constraints. Embedded in the appendix of this document is a copy of the final project mid-point report which details the sue of Android application programming to build a Machine Learning model used to predict market trends for NYSE stock symbols. Unfortunately, after extensive research and testing, it was discovered that Android application development involved a level of learning outside of the timeline of this applications development. For this reason, the application was moved to a web-based format using HTML, CSS and JavaScript as the foundational programming languages.

Furthermore, the creation of a Machine Learning model was expected to be expedited y the use of the Amazon ML service. After a number of tests with this service, it was discovered that the default configuration of models from this

service are batch predictions. These batch predictions were unable to accurately determine the following trend of pricing related to a stock symbol. In order to achieve the desired prediction result from this model would involve learning the R language entirely to build a Machine Learning Model from scratch. The development of such a model accounting for the period of time involved in learning the R language would have caused an extremely late submission of this project. For these reasons, it was decided that the Machine Learning aspect of this project would be removed. In its place the Serverless technology was adopted. This area of technology has a much shorter learning period.

As the Machine Learning aspect of the project was removed due to the time constraint, it was decided that the Gekko application would become more focused on a specific market within the financial world. This market, chosen through personal interest is that of crypto currencies.

FUTURE DEVELOPMENT

As mentioned in the previous section certain elements of this application were changed due to constraints on time and understanding. For future development I intend to continue my research into the area of Machine Learning and hope to implement this as large-scale feature of the application in the near future.

Throughout this document we reference that the Gekko application does not currently have a platform on which users can buy or sell currency shares. The development of the login system and the profile page were made in the hopes of an exchange being developed in future to support currency share transactions on-site. This functionality paired with the fore mentioned machine learning capabilities would make Gekko a unique resource not currently available through any other application.

Portions of the Application are still administrated manually and involve update and entry on the part of the developer. As an immediate and continuous development of this application after submission, it is my intention to automate all functions and actions included in the application.

REFERENCES

Amazon Web Services, Inc. (2018). *AWS Serverless Application Repository - Amazon Web Services*. [online] Available at: https://aws.amazon.com/serverless/serverlessrepo/ [Accessed 13 May 2018].

Archive.ics.uci.edu. (2018). *UCI Machine Learning Repository: Dow Jones Index Data Set*. [online] Available at: http://archive.ics.uci.edu/ml/datasets/Dow+Jones+Index [Accessed 13 May 2018].

CodePen. (2018). *Random Login Form*. [online] Available at: https://codepen.io/motorlatitude/pen/JFkro [Accessed 13 May 2018].

CryptoCompare. (2018). *Bitcoin, ETH and Zcash price widget, chart widget, news widget and multiple cryptocurrency widgets*. [online] Available at: https://www.cryptocompare.com/dev/widget/wizard/?type=1&theme=0&fsym=BTC&tsym=USD&period=1D [Accessed 13 May 2018].

Docs.aws.amazon.com. (2018). *What Is AWS Lambda? - AWS Lambda*. [online] Available at: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html [Accessed 13 May 2018].

Gist. (2018). *How to install LAMP on a EC2 Amazon AMI*. [online] Available at: https://gist.github.com/aronwoost/1105007 [Accessed 13 May 2018].

HTML5 UP. (2018). *Landed by HTML5 UP*. [online] Available at: https://html5up.net/landed [Accessed 13 May 2018].

Serverlessrepo.aws.amazon.com. (2018). *Serverless*. [online] Available at: https://serverlessrepo.aws.amazon.com/applications/arn:aws:serverlessrepo:us-east-1:713541911133:applications~crypto-monitor [Accessed 13 May 2018].

Witten, I., Frank, E., Hall, M. and Pal, C. (2017). *Data mining*. Amsterdam: Morgan Kaufmann.

YouTube. (2018). *Build a PHP Register & Login System Udemy*. [online] Available at: https://www.youtube.com/watch?v=zvXgsouIzVg [Accessed 13 May 2018].

YouTube. (2018). *How to Make a Bitcoin Price Widget Javascript, PHP, CSS & HTML*. [online] Available at: https://www.youtube.com/watch?v=uVlMueUBGOo [Accessed 13 May 2018].

YouTube. (2018). *Intro to Amazon Machine Learning*. [online] Available at: https://www.youtube.com/watch?v=MWhrLw7YK38 [Accessed 13 May 2018].

Zona, P. (2018). *Using your first microservice with AWS Lambda – Cloud Assessments*. [online] Cloud Assessments. Available at: https://www.cloudassessments.com/blog/using-your-first-microservice-with-aws-lambda/ [Accessed 13 May 2018].

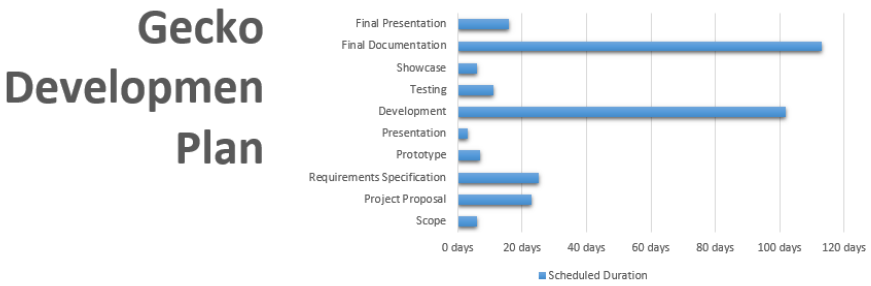
APPENDIX

PROJECT PROPOSAL

Gecko Proposal.doc

PROJECT PLAN

Gantt Chart (1).mpp



MID-POINT REPORT

Gecko Tech (1).docx

USER MANUAL

GekkoUserManul.docx

MONTHLY JOURNALS

GekkoMonthlyJournals.docx