Deploying a Web App with AWS Lambda

Cloud Infrastructure
BSc in Applied Computing Year 4

Rob Shelly
20068406

April 13, 2018

Contents

1 Introduction

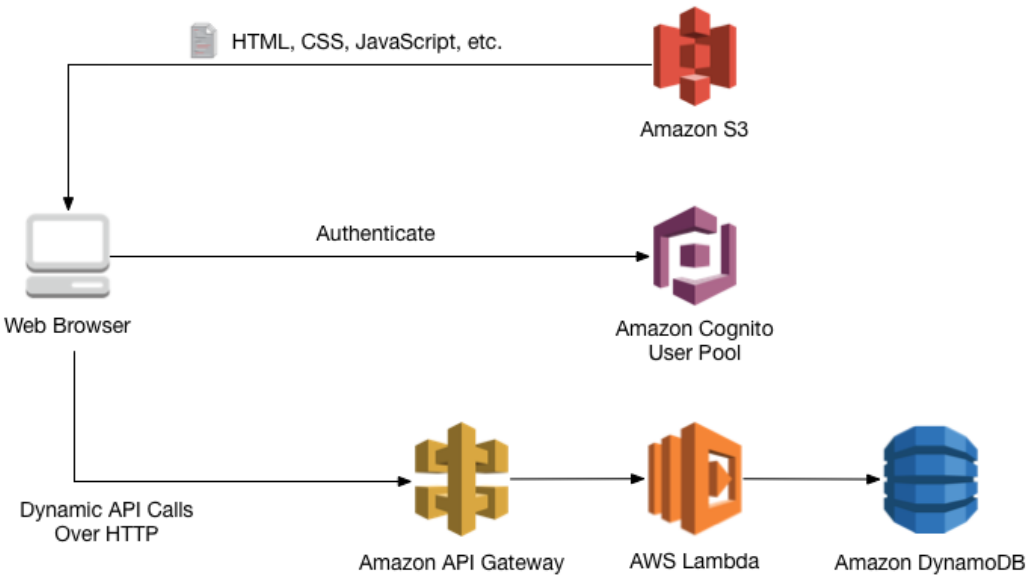
This paper will explore serverless architecture by creating a static website and using a serverless compute service to provide a backend functionality. The practical element of this paper will deploy a sample web app, known as *Wild Rydes*, provided by AWS. This site allows users to register, verify their email address and sign in after verification. Once authenticated the user can request a ride on a Unicorn by entering their location.

The system will be implemented using a number of different AWS services:

- Static web content will be hosted on **S3**;
- The backend functionality will be implemented using **Lambda**;
- Authentication will be provided using **Amazon Cognito**, either through a self hosted **User Pool** of registered users, or sign in using a Google Identity;
- API endpionts will be created for the static website to trigger the backend serverless functions using **Amazon API Gateway**;
- The data base for the system will be implemented using **DynamoDB**.

The use of these together in to architecture shown in Figure 1 allows the serverless implementation of *MEAN stack-like* web app. Services and terms highlight above will be explored in the practical element.

Figure 1: System Architecture



1.1 Objectives

Based on the above work to be completed the following objective have been outlined:

- Learn about serverless architecture;
- Implemented a serverless system using AWS Lambda;

- Explore possibility of creating a website backend using Amazon API gateway with Lambda;
- Examine the use of AWS Cognito for implementing user authentication using a hosted database of registered users and to examine signing in through social identity providers.

1.2 Background

Prior to the introduction of serverless architectures, hosting web apps consisted of deploying the app to a server, either on premise or provided by a cloud service. For example, hosting a MEAN stack application on an AWS EC2 instance. This has the drawback of the financial cost of maintaining the server. Web apps may experience periods of inactivity. However, during these times the servers are still running and therefore continue to incur cost. There is also overhead involved with the provisioning and maintenance of the such servers and auto-scaling of the server. Serverless architectures aim to reduce the impact of these issues.

The above example describes the a web app hosted on a server as this is the service which will be implemented as for the practical element of this paper. However, the same concepts, and alternative described below, apply to any code which a client may wish to run in the in the cloud via a provider.

1.2.1 Serverless Architecture

In a serverless model, a client does not need a server, either locally clouid provided to run their code. Rather, the client provides only their code to a cloud provider which runs the code for them. The provider manages the execution of this code. Thus, it is often described as Function-as-a-Service (FaaS) as the cloud provider completely abstracts the servers away from the clients. Therefore the client does not have to provision or maintain any servers (Stackify, 2017).

It is important to note that serverless computing is not a client uploading some code, running and reading a response or results. The client provides the code and it is run whenever necessary. Hence, serverless computing is described as being an event driven execution model. The code is provided by the client and the execution and life-cycle management of a function is maintained by the provider after being triggered by some event. (Kanso and Youssef, 2017). This will be demonstrated in this paper through the use of Amazon API gateway. The API endpoints will act as triggers for the functions saved in the serverless architecture.

These features of serverless computing provide a number of benefits over traditional server base computing.

- **No server administration:** Clients do not need to provision or maintain servers as the servers are abstracted away from them.
- **No autoscaling:** Clients do not need to configure autoscaling of their servers to adjust to traffic bursts as the provider completely manages the backend servers.
- **Reduced costs:** Clients pay only for execution time instead of paying to run servers constantly, even when not in use.

2 Tools

There are a number of serverless services including AWS' Lambda, Google's Cloud Functions and Microsoft's Azure Functions. This paper will utilise Lambda to implement a serverless architecture.

2.1 AWS Lambda

Lambda is Amazons serverless compute service. Lambda charges based on execution time in the milliseconds. The service allows users to build serverless backends with triggers ranging from many other AWS services and also rd party API requests. It supports a number of different runtime environment including NodeJS, Python and Java (AWS, 2018d). This paper will be executing JavaScript.

2.2 Amazon Cognito

Amazon Cognito is a an authentication service. It not only provides user sign-up and sign-in features but alos allows the client to implemented access control to other backend services through the use of client defined roles AWS (2018c) Cognito allows user sign-in using two different methods:

- Cognito allows users to register and maintains a list such who can then sign-in;
- Users can sign-in to Cognito using an account with a social identity provider such as Facebook and Google.

Cognito is an ideal service to implement a serverless web app as it allows user authentication to be implemented without creating a backend server with a DB of users. For this pape

2.3 AWS Amazon API Gateway

API Gateway is AWS' service for building and publishing APIs. This allows the creation of RESTful endpoints for web app. Just like serverless functions, billing is based only on number of API calls made, making it and ideal low cost feature of this system. It can also be used in conjunction with Cognito to authorise API calls (AWS, 2018a). The API gateway coupled with Lambda is what makes the serverless web app possible. API endpoints for the app can be easily created which act as the triggers for the Lambda functions. This provides the backend service for the web app.

2.4 AWS S3

The final important element of creating a serverless web app is S3. S3 is AWS' storage service. Although S3 provides simple cloud base storage it has the added feature of being able to host static web sites when the necessary files are stored within (AWS, 2018f). Thus, by uploading the static fronted pages of the web app to S3, and configuring the backend with API Gateway and Lambda, the serverless web app is complete.

3 Practical

This paper demonstrated hosting a web application using a sample web site provided by AWS. The website consists of a number of web pages which allow the user to register, verify their email address and sign in. The logic for these functions is provided by the static web pages and AWS Cognito. Once, registered and logged in, the user is directed to a global map at which point the user can enter their location in order to request a unicorn. The request logic is handled by AWS lambda. This function will return the response from the request to the client while also recording the details of the request in a backend database, implemented using DynamoDB.

3.1 Static Web Content on S3

S3 hosts the static portion of the web application. This not only includes static *html* pages but also client side JavaScript, for example the user authentication logic which runs on the client. Therefore,the first step was to create an S3 bucket. This was achieved with the following command:

Code Sample 1: Create Bucket

```
aws s3api create-bucket --bucket cloud-infra-wild-rydes --region eu-west-1 --region eu-west
    -1 --create-bucket-configuration LocationConstraint=eu-west-1
```

Once the bucket was created the static content was added. As mentioned above, the static website is provided by AWS. This means the content can easily copied to the bucket from the provided AWS source bucket:

Code Sample 2: Adding Static Content to S3 Bucket

```
aws s3 sync s3://wildrydes-us-east-1/WebApplication/1_StaticWebHosting/website s3://cloud-
    infra-wild-rydes --region eu-west-1
```

3.1.1 Allow Public Access

In order to make the website visible to a browser the bucket needs to be configured to hosts static web content. First the bucket needs to be given public read permissions. This is done by creating a bucket policy which grants anonymous read access to all assets in the bucket:

Code Sample 3: Add Bucket Read Policy

```
aws s3api put-bucket-policy --bucket cloud-infra-wild-rydes --policy file://bucket-policy.
    json

bucket-policy.json >
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::cloud-infra-wild-rydes/*"
    }
  ]
}
```

Finally, the bucket was configured for static website hosting:

Code Sample 4: Configure Bucket for Static Website Hosting

```
aws s3 website s3://cloud-infra-wild-rydes/ --index-document index.html
```

The bucket configuration as now complete and the web site can be view by visiting the buckets URL.

Figure 2: S3 Bucket Static Web Hosting

However, once Cognito is configured, the IDs of some of their resources needs to be added to the bucket in order client side JavaScript to communicate with them. These IDs must be added to the *config.js* file within the bucket to allow the client-side scripting to authentication and verify against the user pool. Therefore, this file was download so that it could be completed later.

Code Sample 5: Download config.js

```
aws s3 cp s3://cloud-infra-wild-rydes/js/config.js config.js
```

3.2 User Management with Cognito

3.2.1 User Pool

For this exercise user authentication with Cognito will be provided by a user pool. This is a database of users maintained by Cognito. When a user registers they are added they are sent a verification email including a code. They must then visit the verification page and enter the code at which point the are add to the user pool and can now sign into the app.

A user pool was created using the following command:

Code Sample 6: Create User Pool

```
aws cognito-idp create-user-pool --pool-name wild-rydes-user-pool --auto-verified-
    attributes email --policies file://password-policy.json --schema '{"Name":"email","
    Required":true,"AttributeDataType":"String"}'

password-policy.json >
  {
  "PasswordPolicy": {
    "MinimumLength": 8,
    "RequireUppercase": true,
    "RequireLowercase": true,
    "RequireNumbers": true,
    "RequireSymbols": false
  }
}
```

This command creates a user pool with a number of configurations:

- Verification is implemented via email (SMS is also an option)
- A password policy is enforced
- The email attribute is required

3.2.2 App Client

Next an app client (or user pool client) is created. The app client allows the the web application to call unauthenticated API's or functions in Cognito such as *Register* and *Sign in* (AWS, 2018b). The app client was configured as follows, including the Pool Id of the pool created above:

Code Sample 7: Create App Client

```
aws cognito-idp create-user-pool-client --user-pool-id eu-west-1_yXmbzV1SM --client-name
    wild-rydes-web-app --no-generate-secret
```

Finally, the Pool ID and the App Client ID are added to the *config.js* file for the web applications static files.

3.3 DynamoDB Setup

The details of all rides requested are saved in the backend database. This consisted of a DynamoDB table. The table was set up using the follwing command:

Code Sample 8: Create DynamoDB Table

```
aws dynamodb create-table --table-name Rides --attribute-definitions AttributeName=RideId,
    AttributeType=S --key-schema AttributeName=RideId,KeyType=HASH --provisioned-throughput
     ReadCapacityUnits=5,WriteCapacityUnits=5
```

This command creates the table *Rides* with the Primary Key *RideId* which is of type string.

3.4 IAM Role

In order for an AWS Lambda function to run it needs to correct permissions to execute and interface with any necessary AWS services. In this the case the function needs to be able to make entries into a DynamoDB table. The function does this by assuming a Role (AWS, 2018e). Accordingly, an IAM role was created with the following policies attached:

- AWS Lambda Basic Execution Role
- DynamoSB Write Access

This was created on the AWS console.

3.5 Lambda Function

The next step was to create the serverless backend for the web app. This simple app consists of only a single backend function, i.e. request a unicorn. This is provided by AWS Lambda. The function was configured with the following attributes:

- **Runtime:** The runtime environment which should be used to run the function. This function should be run in NodeJS. However, other options such as Python and Java are available allowing functions to be written in a variety of languages.
- **Role:** The role which the function will assume in order to execute with the correct permissions. It is configured with the ARN of the role created above.
- **Handler:** The entrypoint for the function. i.e. the exact function within the provided file(s) which whould run then the Lambda function is called.
- **Code:** The code of the Lambda function. When the function is created using the CLI, the code can provided as a zipped file in S3 which Lambda will retrieve and unzip.

Code Sample 9: Lambda Function

```
aws lambda create-function --function-name RequestAUnicorn --runtime nodejs6.10 --role arn
    :aws:iam::806626264653:role/WildRydesLambda --handler index.handler --code S3Bucket=
    cloud-infra-lambda-function,S3Key=index.js.zip
```

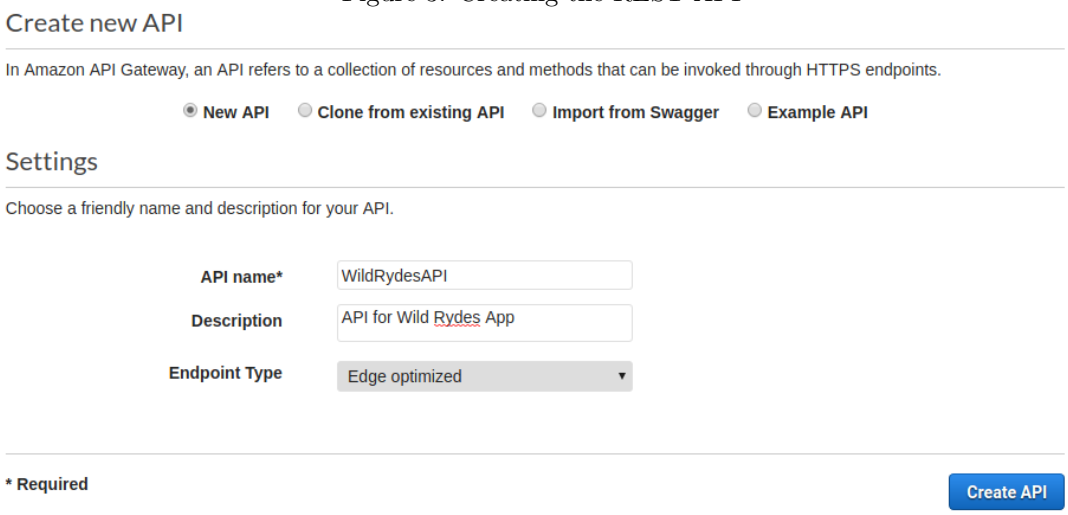
3.6 API Gateway

Due to the complex nature or creating and configuring the API, these steps were carried out via the AWS console. Configuring a the API consisted of a number of steps.

3.6.1 REST API

The first step was to create a rest API. This is shown in Figure 3.

Figure 3: Creating the REST API

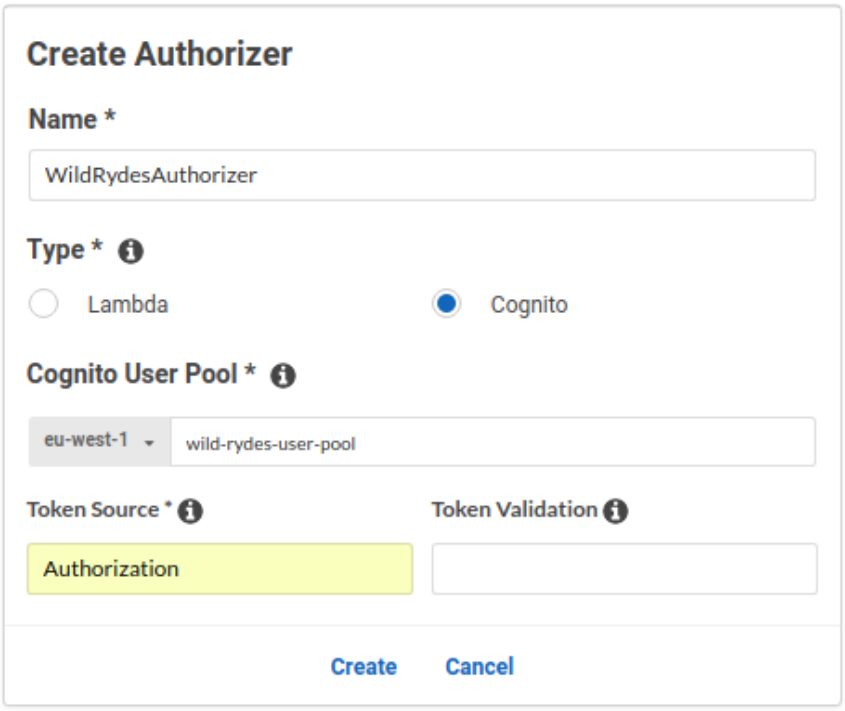


3.6.2 Authorizor for User Pool

The API endpoint to be created is for the *RequestAUnicorn* Lambda function. Users can only access the necessary web page to make this call after authenticating. Therefore, the API Gateway needs a method on ensuring on authenticated request are accepted. This is achieved using an Authorizer connected to the Cognito user pool created above. JSON Web Tokens (JWT) are used to verify authorisation of requests after users have authenticated. The client includes their token with the request to the API and the Authorizer verified this token with the Cognito user pool. Configuring the Authorizer on AWS requires that the header used for authentication be specified.

Figure 4 shows the *Authorization* header is chosen. This is the header which will be used by the API to send user token to the Cognito user pool to verification. Also, shown is the user pool created above

Figure 4: Creating the Authorizer

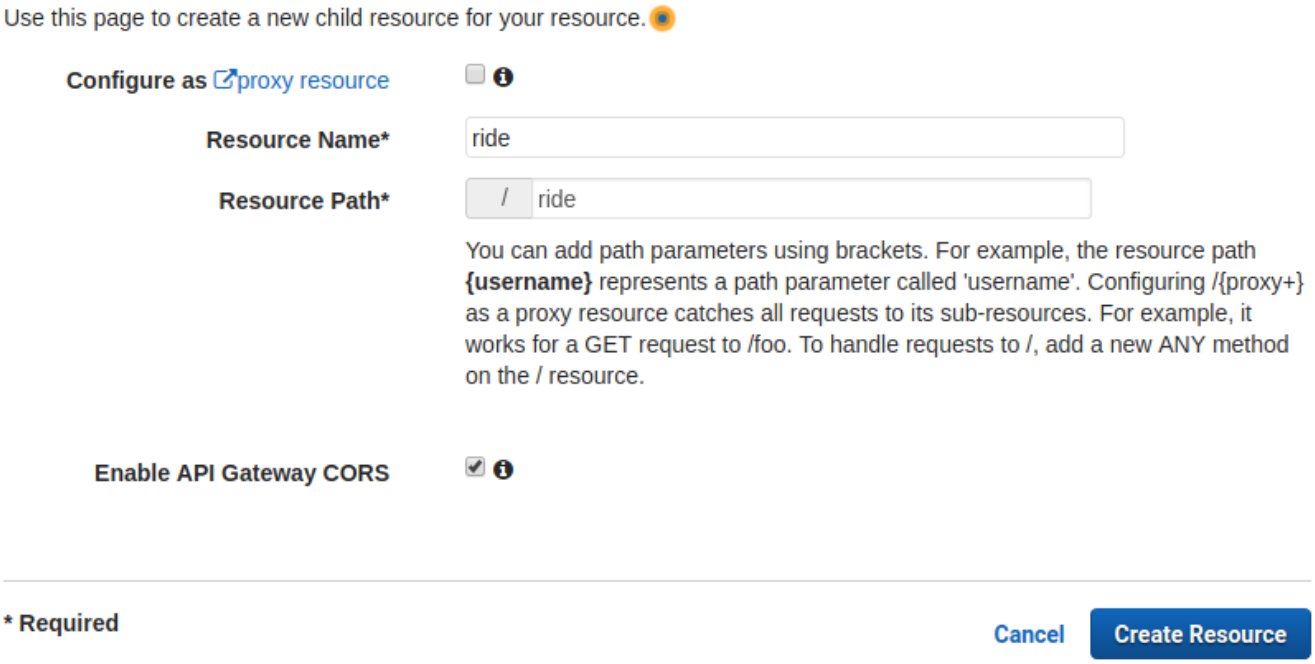


3.6.3 POST Resource Method

The API was then populated with an endpoint. For this app there is only one endpoint i.e. */ride*. The endpoints will accept only HTTP Post requests. Endpoints are defined in API Gateway be as resources. They are created relative to the root domain or parent resources for the purposes of mirroring URL paths. In this case the ride resources' parent is the root domain.

When creating the resource is was important to enable CORS. CORS, or Cross Origin Resource Scripting, is a security concept concerned with APIs called by sources in a different domain. Browsers often restrict API calls to other domains for security reasons. Enabling CORS allows a client running access resources in a different domain via the REST API. This is achieved by adding CORS headers to the request and enabling CORS on the target domain (Mozilla, 2017). In this case, client side files are being server from S3, where API request are directed to API Gateway. Therefore, CORS must be enable on the REST API.

Figure 5: Creating the REST API



With the resource created the HTTP method can be added. The *ride* resource accepts a POST method, as the user sends their location in order to request a unicorn. The lambda function created earlier was selected as the integration for the POST method. Also, the authorizer created above was added to the method to perform the verification against the user pool.

Figure 6: Creating the POST Method

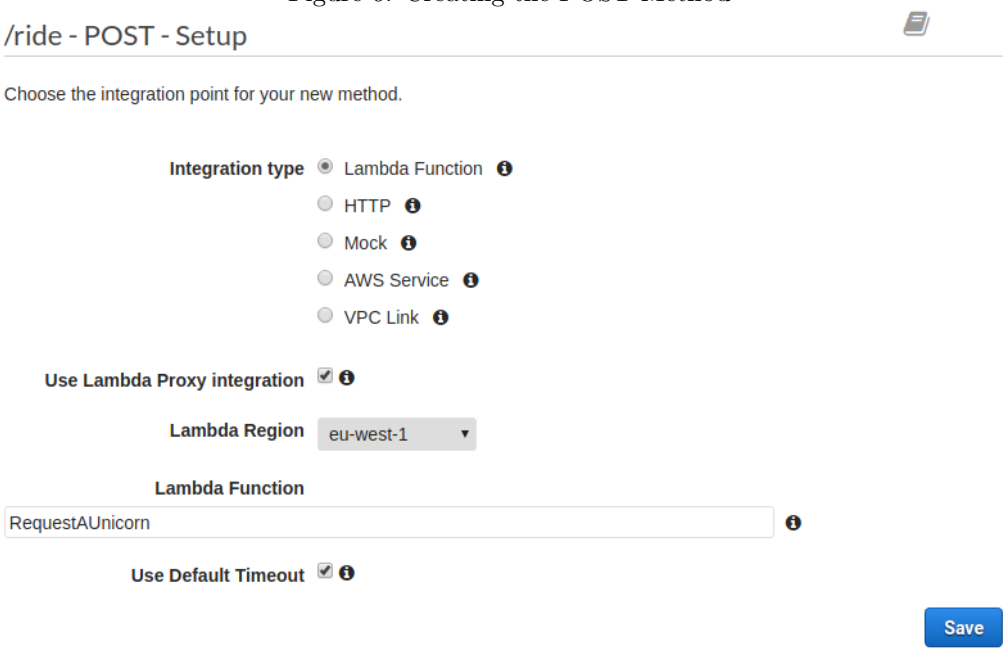
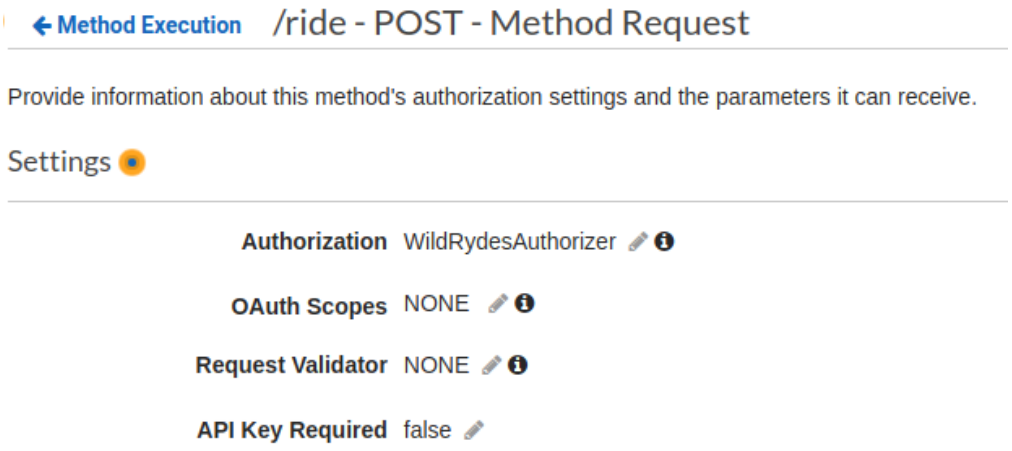
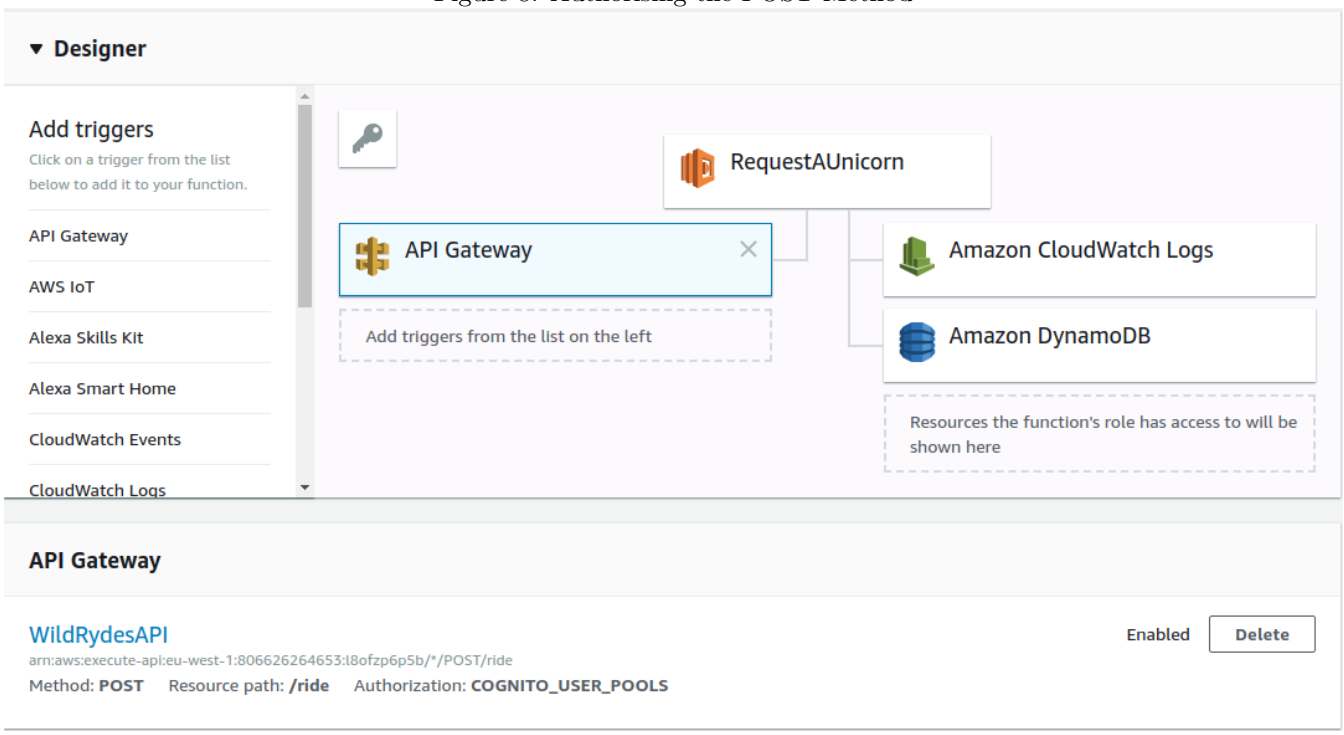


Figure 7: Authorising the POST Method



Viewing the Lambda function on the console now indicates that the API Gateway has been added as a trigger for the function.

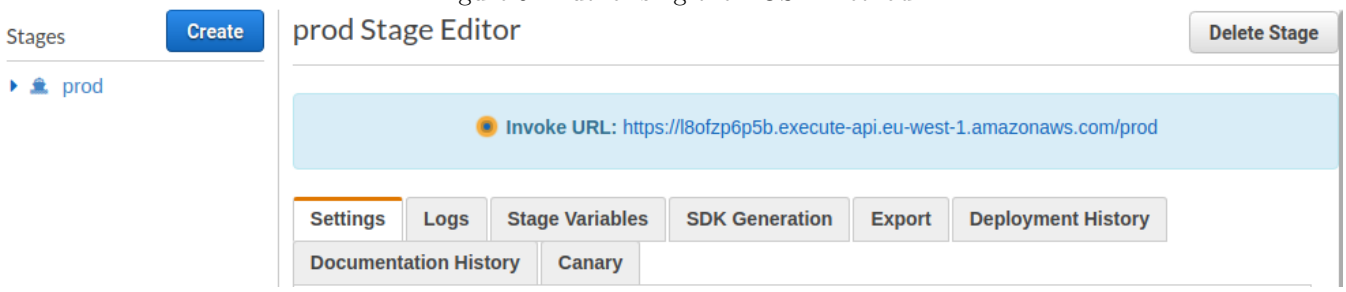
Figure 8: Authorising the POST Method



3.6.4 Deploy API

The final step was to deploy the API. This created a URL at which the API can be targetted. This must be recorded and added to the *config.js* file for the client side scripts in the S3 bucket. This simply consists of clicking *Deploy* and adding a stage name, in this case *prod*.

Figure 9: Authorising the POST Method



3.7 Updating S3

With the serverless backend complete the final stage is to update the *config.js* file in the bucket. This final configuration is shown in below:

Code Sample 10: Client Side Scripting config.js

```
window._config = {
  cognito: {
    userPoolId: 'eu-west-1_yXmbzV1SM', // e.g. us-east-2_uXboG5pAb
    userPoolClientId: '62tba8dssa0036ig0r30hklbdm', // e.g. 25ddkmj4v6hfsfvruhpfi7n4hv
    region: 'eu-west-1' // e.g. us-east-2
  },
  api: {
    invokeUrl: 'https://l8ofzp6p5b.execute-api.eu-west-1.amazonaws.com/prod' // e.g. https
        ://rc7nyt4tql.execute-api.us-west-2.amazonaws.com/prod',
  }
};
```

The configuration file was copied to the bucket using the following command:

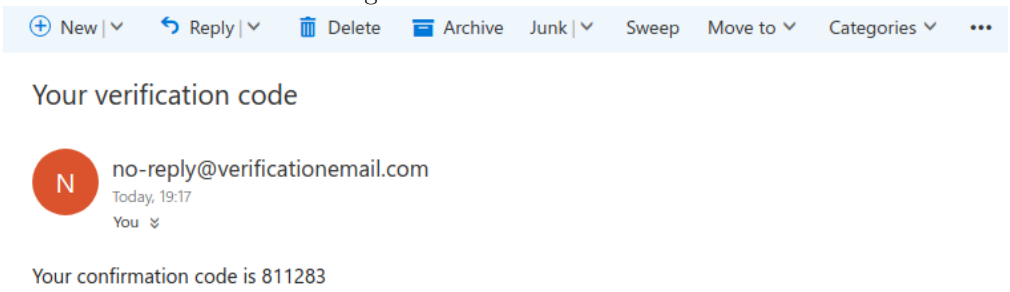
Code Sample 11: Copy config.js to bucket

```
aws s3 cp config.js s3://cloud-infra-term-paper/js/config.js
```

4 Running the App

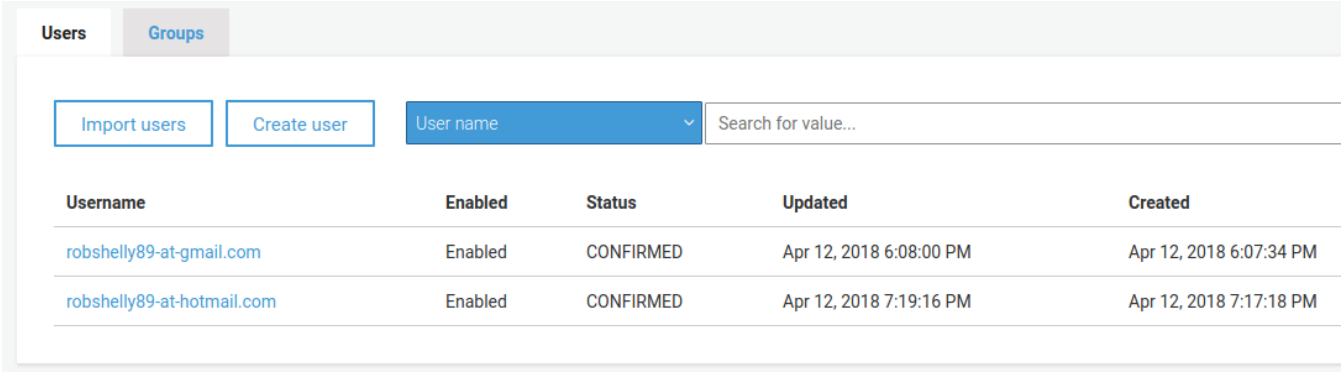
The app was then tested by visiting the *sign in* page of static website. A user was registered and then redirect to the verification page. As expected, an email was received containing a verification code.

Figure 10: Verification Email



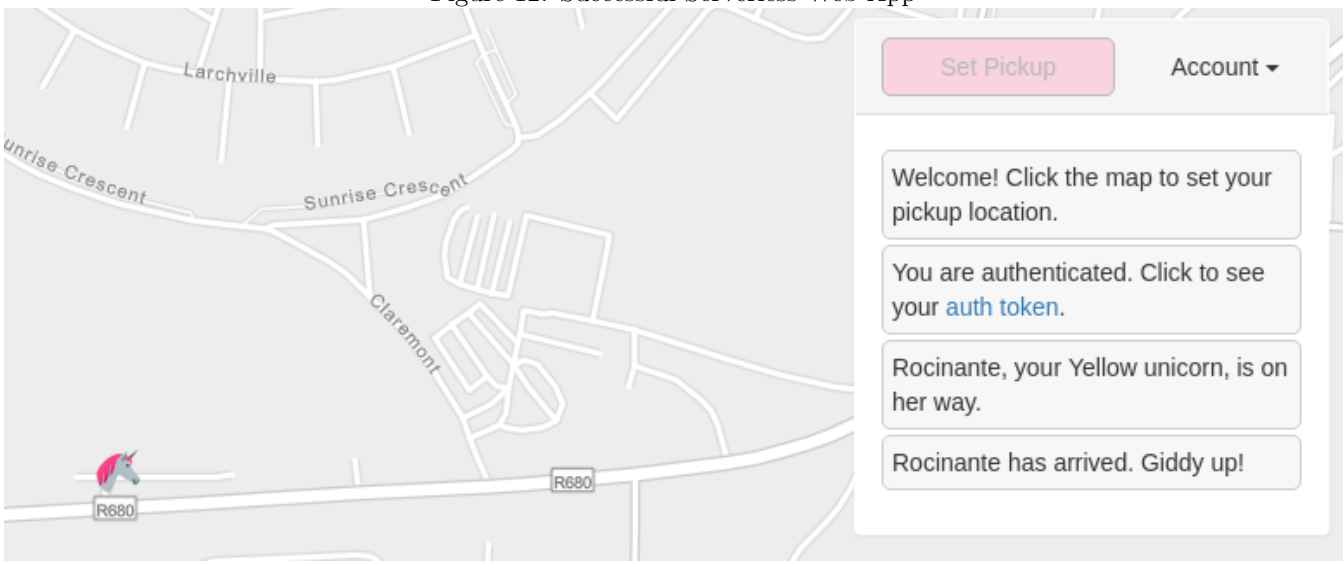
Once the user verified by entering the code, they were redirect to the main page of the app. Viewing the users on the AWS console Cognito page, dmeonstrated that user were successfully registering and verifying.

Figure 11: Users



Finally the main aspect of the serverless web app could be tested by entering a location on the map and choosing *Request a Unicorn*. This returned a response to the user containing a Unicorn and some details, shown if **??**. The web also demonstrated that the user has successfully received a JSON web token, verifying that the user authentication is working.

Figure 12: Successful Serverless Web App



Viewing the *Rides* table on DynamoDB also verifies that the Lambda backend is functioning correctly as the rides and relevant information have been entered into the table.

Figure 13: Rides Added to Backend Database

Rides Close

Overview | Items | Metrics | Alarms | Capacity | Indexes | Global Tables | Backups | Triggers | Access control | Tags

Create item | Actions

Scan: [Table] Rides: RideId

RideId	RequestTime	UnicornName	User
ESEJSe1P0MKOCSMQo8l1Rw	2018-03-27T23:03:11.416Z	Shadowfax	robshelly89-at...
H_sDBhErMR5wfDM71l5d7Q	2018-02-20T15:52:09.765Z	Shadowfax	the_username
WZRowooHRRoTWvTA9ih88g	2018-02-20T16:28:11.511Z	Rocinante	robshelly89-at...
ED3Yt6kWZiPFvQt5_5Z5gA	2018-04-12T18:59:14.399Z	Rocinante	robshelly89-at...
zdMeaxTkwvQjypmB44Rx0w	2018-04-12T18:41:24.859Z	Rocinante	robshelly89-at...
21ZfwROd6bPOoro1Hs3NJw	2018-03-27T22:41:53.736Z	Bucephalus	robshelly89-at...
JKqKbyNcYTbetdNYkxr08A	2018-04-12T01:41:14.384Z	Rocinante	the_username

5 Conclusion

The practical element of this paper successfully demonstrated serverless architecture using AWS Lambda. In this case, a serverless architecture was implemented to create a MEAN style web application without the use of any servers. The fronted was provided entirely by S3, serving HTML web pages and client side scripts as a static website. The backend was provided by a Lambda function triggered by API endpoints configured in AWS' API Gateway. Finally, user authentication was provided by AWS Cognito. Therefore, the entire web application was hosted in the cloud without deploying, configuring or maintaining servers. This also has the benefit or reduced costs.

A notable drawback of the system created in this paper is vendor lock-in. This system was deployed as a number of AWS services all configured together to create a single architecture. Migrating a system such as this to another vendor such as Google Cloud Platform or Microsoft Azure could prove to be quite a monumental task. Knowledge and experience would be needed in each of the corresponding services on the new platform before the system can be migrated.

It is possible that designing this system as a traditional MEAN stack application which can be run on a server with the necessary software installed (e.g. NodeJS) could make it far easier to port to another provider. In order to port such a system only knowledge of the new platforms compute service would be necessary. Once a server was create. the necessary software could be installed and the application started. This approach however, introduces the overhead of server maintainence.

It must also be noted however that vendor lock-in could also be avoided while still employing a serverless architecture. With the correct configurations and permission, the various services needs could be provided by different vendors. For example, the static website server from S3 could be configured to target and API created using Cloud Endpoints, Google Cloud Platform's API service.

6 Bibliography

AWS. Amazon API Gateway, 2018a. URL https://aws.amazon.com/api-gateway/.

AWS. Configuring a user pool app client, 2018b. URL https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-settings-client-apps.html.

AWS. Amazon cognito, 2018c. URL https://aws.amazon.com/cognito/.

AWS. Aws Lambda, 2018d. URL https://aws.amazon.com/lambda/.

AWS. Aws Lambda Permissions Model, 2018e. URL https://docs.aws.amazon.com/lambda/latest/dg/intro-permission-model.html.

AWS. Static web hosting, 2018f. URL https://aws.amazon.com/websites/.

Ali Kanso and Alaa Youssef. Serverless: beyond the cloud. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 6–10. ACM, 2017.

Mozilla. Cross-origin resource sharing (CORS), 11 2017. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

Stackify. What is Function-as-a-Service? Serverless Architectures Are Here!, May 2017. URL https://stackify.com/function-as-a-service-serverless-architecture/.

Appendix A Resources

The tutorial followed and source code for this paper can be found at the following Github repository:
https://github.com/aws-samples/aws-serverless-workshops/tree/master/WebApplication

Files of particular interest:

RequestUnicorn Lambda Function:
https://github.com/aws-samples/aws-serverless-workshops/
blob/master/WebApplication/3_ServerlessBackend/requestUnicorn.js

Client Side User Authentication Logic:
https://github.com/aws-samples/aws-serverless-workshops/
blob/master/WebApplication/1_StaticWebHosting/website/js/cognito-auth.js