UniGuideOnline

William Hadden - 6249537

October 9, 2022

1 Application

1.1 Architecture

This application follows the design principles of a 3-tier architecture. Namely, there are webservers that handle incoming client traffic and display page information, a backend database server for handling application specific data and an API that links them together.

There are two webservers, installed on EC2 instances that both host React applications. One webserver (the public-webserver) takes care of displaying paper information in a dashboard like style which the other (admin-webserver) provides functionality for paper providers; for example, academics to create papers. The backend database is provisioned by DynamoDB, a managed NoSQL database provided by AWS. Finally, there is a CRUD API created using a combination of API Gateway which provides routes and endpoints, coupled with a lambda function that executes custom code in response to requests from the API Gateway.

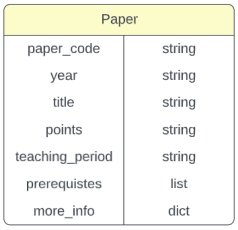
Each webserver is hosted on a customized EC2 instance. A key part of the design is splitting the two different user interactions - that of a paper viewer and paper administrator - onto separate virtual machines. The idea is that the traffic demands on either will almost certainly be different. For example, there are certainly more students than academics at regular universities. As such, we would expect the traffic on the user oriented webservers to be higher. Therefore, this separation of concerns allows more or less instances to be brought online in response to differential traffic/demand. This could be accomplished using a load balancer coupled with auto-scaling in the future. In essence, this separation makes it easy to tweak individual provisioning to match the demand. Furthermore, anticipating this extension, both EC2 instances use pm2 software installed to automatically run their respective webservers on restart.

Each instance is also associated with an elastic IP. The idea here again relates to the load balancing with the operation of multiple instances where the elastic IPs can mask the failures of instances by instant remapping the address to other running instances. Furthermore, upon acquiring a domain name, this can ensure that the domain points to the instance(s).

The API is provisioned using a combination of an API Gateway and a lambda function. The key idea here is running code only on demand. In this instance, the API Gateway service

Figure 1: Diagram of application architecture.

Figure 2: Paper collection data model.



is always visible and able to be accessed. However, the lambda function only runs the code required to accomplish the request when the endpoint is triggered, thereby saving money.

The database backend is provided by the DynamoDB service. This service greatly simplifies the backend logic since AWS takes care of all the particulars. Thus, all that is required is the correct triggering of the provided API.

1.2 Data Model

The database consists of a single collection named papers.

Note that the paper objects can refer to other paper objects. In this case, I opted for a NoSQL database because it more easily links with the graph structure I wanted to accomplish with the paper structures. While some recent RDBMSs do offer tree structures without significant slowdown, I determined that in this case, with the small workload I am anticipating, it is simply easier and less logically intense to use a simple document oriented NoSQL database.

2 Lifecycles

This details the application lifecycle. Note that separate instructions are provided for starting the application from the AWS learner lab.

2.1 Initial setup

These steps detail the inital configurations required for first time setup.

2.1.1 Backend Setup

First we need to initialize the backend DynamoDB database. To do this, first navigate to $https://console.aws.amazon.com/dynamodb/$ to bring up the DynamoDB interface. From here, we then create a table named $paper_table$ with primary partition key id. Note that on once created, successive restarts of the AWS learner lab will restart the table automatically, so no further effort is required.

2.1.2 Webserver setup

To setup the webservers we first navigate to the EC2 console at $https://us-east-1.console.aws.amazon.com/ec2/$ and select *launchinstance*. First, choose the Amazon Linux 2 AMI with a t2.micro instance and a specified key pair. Then create a new security group that allows SSH traffic from $myIP$, HTTP, HTTPS and custom TCP connections on port 3000 for IPv4 and IPv6 from the internet. This ensures that web users can connect to the React webserver that runs on port 3000. Note that the $myIP$ reference will have to be updated periodically.

Then we assign elastic IPs to the instances using the $ElasticIPs$ tab under $NetworkandSecurity$. To do this, simply select $AllocateElasticIPaddress$ and assign each webserver to different Elastic IPs.

Now connect to the webservers by following the instructions under *connecttoinstance* and run the following sequence of commands.

1. $sudoyuminstallgit - y$ to install git
2. $gitclonehttps://github.com/hadwi537/UniGuideOnline.git$ to install the application files (using a private access token if required).
3. Setup Node.js runtime
   (a) $curl-o-https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh|bash$
   (b) $. /.nvm/nvm.sh$
   (c) $nvminstall - -lts$
   (d) $node - e"console.log('RunningNode.js' + process.version)$ to verify installation.

Now navigate to the root directory of the one of the webservers (public-webserver or admin-webserver) and run *npminstall*.

The respective app will now be available on localhost:3000 and from the given elastic IP at $http://<elastic-ip>:3000$.

We now configure PM2 to automatically start the webservers and leave them running when the terminal is closed through the following commands.

1. $npminstallpm2@latest - g$
2. $npminstall - gserve$
3. $npmrunbuild$ (from the root directory of a webserver)
4. $pm2servebuild3000 - -spa$
5. copy and paste the output of $pm2startup$ and run it.

2.1.3 API Setup

First setup the lambda function by navigating to $https://console.aws.amazon.com/lambda$ and select $createfunction$. Let the function name be $crud-paper-function$, assign the exectuion role to *LabRole* and ensure that the LabRole has the $Simplemicroservicepermissions$.

Then copy and paste the index.js code in $lambda/index.js$ into the code on the management console and press deploy.

We then create an HTTP API using API gateway. Navigate to the API Gateway console at $https://us-east-1.console.aws.amazon.com/apigateway/$ and choose $createAPI$ using the HTTP API blueprint. Use $crud-paper-api$ as the name and skip the remaming steps to create the API.

Now select this API and choose *Routes* under *Develop*. For each route we defined in the lambda function, we add that route with the specified method. In this case:

- $GET/papers/id$
- $GET/papers/$
- $PUT/items$
- $DELETE/items/id$

Now select $Integrations$ and assign the lambda api created in the previous step to every route.

2.2 Development and Deployment

As stated earlier, the DynamoDB requires no further modification. Unless, a more compute intensive option is desired.

In general, I personally recommend making changes locally. For example, since we are using a React web framework, the webservers can be run from the command line using npm-start where changes to the code can be instantly applied by simply saving the project.

2.2.1 Webservers

Assuming that changes have been made and pushed to remote, connect to the ec2 instance hosting the changed webserver. Then, perform the following steps.

1. $gitpull$
2. $npminstall$ from the root directory of the modified webserver.
3. $npmrunbuild$,
4. $pm2stopall$
5. $pm2servebuild3000 --spa$
6. copy and paste the output of $pm2startup$ and run it in the terminal.

2.2.2 API

For updating the lambda function, navigate to the Lambda function console and copy and paste in the changes.

New API routes can be created by following the steps detailed in the setup section.

3 Estimated costs of cloud services

The estimated running costs are detailed below.

In idle conditions, the monthly cost is about 31.55

With light use, the monthly cost is about 41.38 where the majority of the cost comes from the DynamoDB databsae.

4 User Interaction

The application is designed to be used by a client through a web interface. Users can either view paper data visualizations or create papers through the website interface.

5 Development story

I initally attempted to use terraform in order to automatically provision my resourecs as I figured that this would ultimetly be an easier way to work. However, I was defeated by the IAM role that had to be assigned to lambda function resources would lead t me being either locked out of certain functionality (i.e I couldn't see my functions on the AWS GUI) and to terraform destroy failing because the IAM user couldn't be deleted with it.

Creating the ec2 instances was relatively smooth. As with the last assignment, I used the React framework mostly because I was now familiar with its operation. While I decided to have a static webserver. It is reasbolanly easy to create a dynamic one using React so I felt this option left future development nicely open. However, I did try and create a CI/CD pipeline to simplify the application lifecycle. However, AWS academy does not let you do this.

I then decieded to use DynamoDB primarily because the previous app this is based off was setup to function with a NoSQL database. Furthermore, I found it a bit simpler to interact with from the context of a lambda function than the RDBMS offering.

Clearly I was always going to need a nice way to interact with the different components. I decided to have a serverless function so that I didn't have to run another virtual machine which would be both expensive, slow and difficult to maintain (especially without CI/CD)

6 Future improvements

Not possible with learner lab so would need own account but would make this application x100 better

Creation of user accounts such that the privileges for interacting with resources (such as papers) can be restricted, so only their owners can interact with them

Use of a CDN and load balancing would increase the reliability of the app and allow it to scale out

Would like to have another go at this but again, does not work with learner lab

Href into either through authentication/user dialog to better increase the app