CSCI 4145 Final Report

Python Flask Web Application with a User Account System and Translation Service

By Shawn Matheson

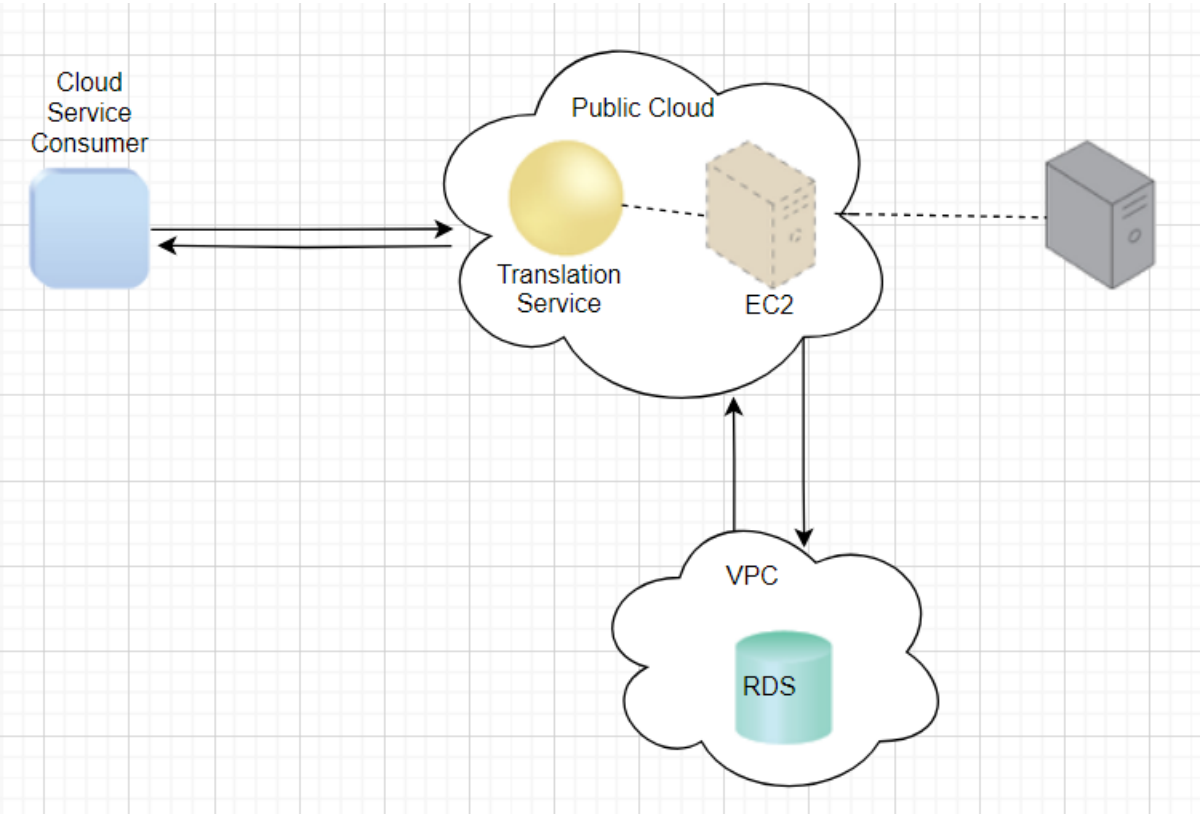
Aug.1st 2023

Gitlab: https://git.cs.dal.ca/courses/2023-summer/csci4145-5409/smatheson/-/tree/main/termassignment/app

Introduction

This application was created as the term project for CSCI 4145 Cloud Computing and is a web application written in Python and using the Python Flask web framework. This application contains a user account system and a translation service that users can use if they create an account and log in. This application uses six AWS cloud services to run and service this web application. When creating this application, I focused heavily on security and usability to try and create a service that would be useful but also a service that users would feel safe using. The goal of this application is to provide the users a service that will allow them to translate text to and from 106 different languages. This service would require users to register an account with a verified email to use the translation feature. This website should be safe to use, and any information used to create an account should be safely stored and unavailable to any bad actors. This application is designed for users who value security and wish to translate text from one language to another. In terms of performance, the goals of this application are that it is fast and responsive, and users are not waiting on backend calls.

Cloud Structure Graph



AWS Cloud Services

Compute

EC2 - The EC2 service is being used to host the web application, there are several other alternatives that could be used instead of EC2, but I decided that this was the best choice. Some of these alternatives include AWS Elastic Beanstalk, AWS ECS and Docker & AWS Elastic Beanstalk. The reason I chose EC2 over these options comes down to several key reasons. The first reason is that AWS Elastic Beanstalk is a PaaS deliver model, this means that I would have less flexibility, less control, and potentially higher costs. The second reason is that I did not want to develop with docker, since the code in docker containers cannot be changed without making a new docker image. This means every time I edit something or try to debug my application I would need to create a new docker image, docker container and then upload it to ECS.

AWS Lambda – Lambda is the only serverless function service that AWS offers. There are several other serverless function services from other companies but since this application relies completely on AWS it does not make sense to use anything else. However, it is possible that instead of using AWS Lambda I could run the functions in the backend on my application. The problem with doing this is that backend functions take time, and this would affect the speed, responsiveness, and usability of the web app. By using the lambda function, it reduces the amount of backend computation my web app needs to manage.

Storage

Amazon RDS (Aurora) – When considering storage option for my application, there were two options that I was considering. The first option was Amazon RDS, and the second option was DynamoDB, both options could accomplish what I needed from a database. When choosing between these two it came down to what kind of data I was going to be storing, that data being user data. User data is better off in a relational database, this kind of data makes it extremely easy to form relationships between data. This fact alone was enough for me to choose RDS over DynamoDB, if this application were to expand in the future, having this data in a relational database would allow for better database design.

Network

VPC – The goal of using a VPC in this application is to protect the database from unauthorized access. Amazon VPC is the only virtual private cloud service amazon offers, to use an alternative, I would need to use other cloud service providers which would be hard to use with Amazon's EC2. If I had more access to IAM roles in AWS it might be possible to create a IAM role that is the only authorized user of the database which would accomplish my goal like the VPC does and prevent all unauthorized access. However, I do not have access to most of the

IAM role functionality so using a VPC and setting the EC2 instance as the only authorized service allowed to send queries is the best option.

General

AWS Backup – The goal of using AWS Backup is protect the information that is stored in my database and the EC2 instance. The alternative to AWS Backup would be to use RDS's built-in automatic backup option and creating an AMI of an EC2 instance as a backup. From my research I felt that it would be a lot easier to just use a single centralized service to manage the creation and storage of the RDS backup and EC2 backup. With AWS Backup it also allows me to create my own specific backup plan for the RDS and EC2, which gives me more control on how often these backups occur and when they occur.

AWS Secrets Manager - The goal of using AWS Secrets Manager is to add even more security to the applications database. The alternative to this service would be storing the database credentials as environment variables. There are a few reasons why my application is using Secrets Manger over environment variables, the first is that it allows multiple AWS services to have access to these secrets. This means Lambda functions do not need to hardcode their own env variables they can get them from the secrets manager along with my webapp in an EC2 instance. The second reason is it will keep all my secret keys in one location, that way if I need to change them in the future I only need to change them in one place. The final reason is that when these keys are stored in secrets manager, they are encrypted, and this gives greater security.

Deployment Model

The deployment model that my application is using is the Hybrid Cloud Model. With this model I am using a public cloud and a private cloud to support this application. In my app the public cloud contains my EC2 instance that hosts my web app, and the private cloud hosts the database that user information is stored in. Within this private cloud, my RDS database can only communicate and receive queries from my EC2 instance. I could have made this application using only a public cloud which would have functioned the same, however by using a hybrid cloud model I am able to keep my database secure. This addition of a private cloud prevents any source other than my EC2 instance from sending queries to the database, since this database holds user information, keeping this database secure was a high priority when designing my application.

Delivery Model

The delivery model that my application is using is the Software as a Service (SaaS) model. The service that my application is offering is the ability for users to freely translate text between over one hundred languages and this service is hosted on an EC2 instance. The reason I chose this delivery model is that the users who will be using this application only need access to the front-end user-interface. Since the beginning my application was always meant to be public facing and open to the internet. Given that the only reason to visit my application is to use the translation service, no other delivery model would make sense. Everything that a user needs in relation to this application can be done with access to the front-end, everything else is managed by exclusively me.

Final Architecture

The architecture of my application revolves around an EC2 instance that my web application is deployed on and a database that stores user registration data. The EC2 instance hosts the web app which allows anyone on the internet access to be able to create an account and log in to their account and use the provided translation service. To preserve the data gathered when creating an account, an RDS Aurora database is used to store that data. Within this app, users are required to verify the email that they are using to create their account, this is done by entering a code that they receive via their emails. This code is sent in an email using a lambda function rather then in the backend of the app, this will decrease the amount of code running in the back end, therefore increasing performance of the web app. Security is a very important consideration in this application, we use a Virtual Private Cloud (VPC) to make sure that nothing other than the EC2 instance where the web app is deployed has access to the database. On top of that, we use AWS Secrets Manager to store the database username and password, this way we do not have any important credentials hardcoded into the EC2 instance. The last cloud mechanism that is used in this application is AWS backup. With this mechanism we have a centralized backup location for the database and the EC2 instance, it is particularly important that the user data is properly protected because if that data is lost, all the app's users will not be able to access the provided service.

The programming languages used in my application are Python, JavaScript, HTML and CSS. The Backend of my application is written in Python, the reason I chose to use python was because of the web framework Flask which is written in python and used to create web apps. I used python and flask several times during this course in my assignments, my knowledge of it is greater than other options so it was the obvious choice. When designing the look of the web pages in my application I used HTML and CSS. Since my application is a website, using HTML to design the webpages and CSS to style the webpages was a requirement, but to make it a little easier I used Bootstrap as well. To add dynamic behaviour to the webpages in my application I

used JavaScript. Everything that I needed to do for my application to function could be managed by JavaScript. JavaScript acted like an intermediary between the frontend in HTML and the backend in python, sending data to the backend when work needed to be done with that data. Everything within this application required code, every feature, every visual element, every function needed code written to be usable to the users.

This system is deployed to the cloud by setting up a flask web application and deploying it inside of an EC2 instance. This EC2 instance is located within a public cloud allowing users from anywhere on the internet to access the website and use its features. Any data gathered on this website is store within a database that resides within a private cloud that only the EC2 instance has access to. With these two cloud mechanisms, it allows my application to be available and usable by anyone on the internet. This system uses the Workload Distribution Architecture, if the traffic to the website is becoming too much for a single instance of EC2 to manage, an Auto-Scaling group can be created to allow the system to scale horizontally. Currently the system is only using one instance to save money since multiple instances are not needed for this project, but the option to allow this application to automatically scale horizontally is available.

Data Security

As stated in the introduction, security was a primary concern when designing this application. The database is protected by a VPC and the credentials to access the database are protected in AWS Secrets Manager, data given to this application is kept safe in that database. All inputs in this application are protected from injection attacks with regular expressions and all data manipulation is done inside of functions to prevent attacks. Sessions are used to prevent access to unauthorized users and passwords are hashed with a one-way hash that cannot be converted back to the original password.

Security Mechanisms

Virtual Private Cloud – An AWS VPC is used in this application to protect the data stored in the database. By putting the database inside a VPC and attaching the EC2 instance where the web app is stored to it, only the EC2 instance can send queries to the database. This keeps the sensitive data that is stored in that database safe from everyone else on the internet.

AWS Secrets Manager – In this application the AWS Secrets Manager is used to protect important credentials/keys. Currently the username and password that are used to sign into the database are stored inside the secrets manager. To get these credentials a call to the Secrets Manager service needs to be made, this means that only someone with the appropriate AWS

account can make that call and get those credentials. Another benefit is that all secret keys are stored in one place making it easier to use them in different applications, they do not need to be stored in ENV variables every time an application needs them.

Regular Expressions – This application has several web pages that contain user input, this means that there is a risk of injection attacks. To prevent users from entering any data that could do malicious things, inputs are verified using regular expressions. Two of these regular expressions were taken from the internet to ensure their effectiveness, the email and password reg expressions [1, 2]. These regular expressions verify that when a user enters their email address, that it is a real email address and not an SQL statement or JavaScript. By using these regular expressions, we can use and store this data knowing that it does not contain any SQL or JavaScript.

Flask Sessions - This application uses flask sessions to control what web pages users are allowed to visit. When a user logs in, their email and name are added into a local session object, if this object exists they are sent to the main webpage. When a user is logged in they cannot go back to the login screen or the registration/verify screen without first logging out. If a user is logged out, they cannot enter the home page without first logging in. These sessions protect the user and the web application itself by preventing users from going to webpages they should not be able to go to.

Bcrypt Hashing – In order to protect the passwords that users are using when they create their account, this application employs Bcrypt's one way hashing. By hashing the password before storing it in the database, we can ensure that this data remains secure even in the case of a database breach. Since it is a one-way hashing, if someone were to get their hands on the hashed password, they would not be able to convert it back into a usable password.

AWS Backup – This application uses AWS backup by to protect the data stored in the database and the EC2 instance in the case of a failure. A backup plan was created so that everyday the database and EC2 instance are backed up. This increases our services reliability and protects the integrity of the data that is stored in the database and the EC2 instance.

JavaScript Functions – Since JavaScript runs in the frontend, the code is run on the computer of every user, this opens it up to attacks. By placing all the data manipulation inside functions and not using global variables on important data, we can make sure that this data is not able to be manipulated by any user.

Cost Analysis of moving to on-premises

Moving from the cloud to on-premises will require some expensive upfront costs to pay for the equipment and software. The company would need something to host the web app, a database to store data, software that runs the web app, a generator and power bank, employees to keep things updated/running and more [3,4,5,6].

Here is a list of some of things that would be needed and their price:

1. **Server and Database 2-in-1 - $21,000**
2. **Database backup** – $8,000
3. **Server software** - $2500
4. **Server Rack UPS** - $3000
5. **Back-up Generator** - $1500
6. **Maintenance and Replacement** – $800 yearly
7. **Power and Ventilation** - $1000 yearly
8. **Software and Hardware Support** – $60k yearly
9. **Access Control** - $5,000
10. **Networking equipment (Switch, router, firewall)** – $30,000

Upfront Costs: 21000 + 12000 + 2500 + 3000 + 1500 + 5000 + 30000 = $75,000

Yearly Costs: 800 + 1000 + 60000 = $61,800

My estimation for how much it would cost would be an upfront cost of $75,000 and yearly costs of $61800, these costs would go up if the website receives heavy traffic. These costs are significantly more then the approximately $40 a month being spent on the web application on AWS.

Most Expensive Cloud Mechanism

This application makes use of a lot of different cloud mechanisms in AWS, luckily AWS shows you what your costs and usages are. By looking at these numbers I can see what is using the most amount of money to keep this application running. These numbers show that Amazon RDS is responsible for 94% of the total money spent creating and maintain this application. The next highest is the EC2 instance which is consuming 5% of the total money. With this knowledge, the most important cloud mechanism that should have monitoring is the amazon RDS since it is currently costing the most money by such a large margin.

The Future

There are plenty of way this application could be improved and many features that could be added to make it a much better experience. The first thing I would add would be an account/profile section to the index that would allow users to change their account information. Another thing I would like to change would be the way the app communicates with the database. Currently, the backend of the webapp makes queries to the database, this is because when I wrote this code I had not learned about lambda functions yet. Moving all the database queries into lambda functions would significantly decrease the amount of backend code running which would increase the speed and responsiveness of the application.

Due to limitations with AWS academy account, I was not able to use AWS translate, to keep my plans for this web app the same I had to use the google translate API. I would like to switch it over to AWS translate in the future because the google translate API is quite slow and I believe AWS translate would be faster. Another feature I would like to add is Amazon Transcribe, it also is not available on the AWS academy account so I could not add it. I think it would be useful for people to be able to use their microphone to audibly speak the text that they want translated. I also think you could add Amazon Polly as a feature to this app, this way users could trigger their translated text to be read aloud in that language. All these features would require an AWS account with more permissions and unrestricted access to all the AWS services.

References

[1] Srinivas. (2014, January 30). *Regex for password must contain at least eight characters, at least one number and both lower and uppercase letters and special characters*. Stack Overflow. https://stackoverflow.com/questions/19605150/regex-for-password-must-contain-at-least-eight-characters-at-least-one-number-a

[2] Neumann, O. (2020). *Regex generator*. Regex Generator - Creating regex is easy again! https://regex-generator.olafneumann.org/?sampleText=Apples1234%40email.com&flags=i&selection=0%7CRFC2822+e-mail

[3] *Rack servers : Poweredge Rack Servers*. Dell. (n.d.). https://www.dell.com/en-ca/shop/poweredge-rack-servers/sr/servers/poweredge-rack-servers

[4] *How much does a server cost for a large business?*. Abacus. (2021, December 17). https://goabacus.com/how-much-server-cost-large-business/

[5] *PowerVault 114X tape rack enclosure : Storage: Dell Canada*. Dell. (n.d.-a). https://www.dell.com/en-ca/shop/servers-storage-and-networking/powervault-114x-lto5-lto6-lto7-lto8/spd/powervault-114x/bvcwsk2bca

[6] Maddox, T. (2013, April 8). *Toolkit: Calculate datacenter server power usage*. ZDNET. https://www.zdnet.com/article/toolkit-calculate-datacenter-server-power-usage/

[7] Bootstrap Dev team. (2011, August 19). *Bootstrap Download Links*. Bootstrap. https://getbootstrap.com/docs/4.3/getting-started/download/

[8] Google. (n.d.). *Hosted libraries | google for developers*. Google. https://developers.google.com/speed/libraries#jquery

[9] *Adele's First Love*. Eggradients. (n.d.). https://www.eggradients.com/gradient/adeles-first-love