Projects and their Information:

**AWS-  
I undertook a course in ASU which had an AWS academy course. The course had a cafeweb server scenario which had challenge labs in each module and the complexity was incremental in nature. The course mostly focused on using AWS resources than the application logic itself. Hence, I got an hands-on experience in using AWS through both the CLI and the User Interface.  
Few projects looked like: hosting the website using S3- On s3, since we can host static websites, I uploaded the website on to an s3 Bucket and configured the bucket policies to accept network traffic from public, enabled versioning on this bucket and anything that is old should be deleted automatically after a certain period.   
in another scenario , I created an EFS and mounted it onto an EC2 instance. First, I created a security group to allow proper inbound access for an NFS communication and associated it with a VPC. Next in an ec2 instance , I created a File system using SSH and mounted it onto the FS.  
In another lab,**

**Aim: Prepare an EC2 instance for hosting a café website with online ordering and staff order viewing capabilities.**

**Accessed AWS Cloud9 IDE on the EC2 instance for an integrated development environment and verified the presence of a LAMP stack (Linux, Apache, MySQL, PHP).**

**Confirmed the web server (Apache) and database (MySQL) services were running and configured them to start automatically on instance restarts.**

**Used commands like sudo chkconfig, sudo service, and service (service) status to manage the services' status and startup behavior.**

**Downloaded and extracted the café application files to the web server's document root directory.**

**Utilized AWS Systems Manager Parameter Store to securely store and manage application configuration data, including sensitive information.**

**Configured the MySQL database by setting the root password, creating necessary tables, and inserting required data for the café application's functionality**

**Conducted thorough testing to ensure website accessibility, functionality, and successful order placement.**

the process scalable and to have a backup in case of any issues **I Created an Amazon Machine Image (AMI) of the configured instance to establish a snapshot of the environment and for using it in a different AWS Region (Oregon) this ensured redundancy and disaster recovery.**

**S3 (Amazon Simple Storage Service):**

**A cloud-based storage service that allows you to store and retrieve data, such as files and images, in a secure and scalable manner.**

**EBS (Amazon Elastic Block Store):**

**Offers persistent block storage volumes for use with Amazon EC2 instances, allowing you to attach and detach storage to your virtual machines.**

**Glacier (Amazon Glacier):**

**A low-cost data archival storage service for long-term data retention, designed for infrequently accessed data.**

**IAM (Identity and Access Management):**

**Provides control over user access and permissions to AWS resources, helping you manage who can do what in your cloud environment.**

**VPC (Amazon Virtual Private Cloud):**

**Allows you to create isolated networks within the AWS cloud, including private subnets, routing, and security groups for better network control.**

**Elastic Load Balancing (ELB):**

**Distributes incoming application traffic across multiple Amazon EC2 instances, ensuring high availability and improved performance.**

**EC2 Auto Scaling (Amazon EC2 Auto Scaling):**

**Automatically adjusts the number of EC2 instances in response to changes in demand, ensuring optimal application performance.**

**RDS (Amazon Relational Database Service):**

**A managed database service that simplifies database setup, operation, and scaling, supporting various database engines.**

**DynamoDB (Amazon DynamoDB):**

**A fully managed NoSQL database service that offers high performance, scalability, and low-latency access to data.**

**Systems Manager (AWS Systems Manager):**

**Provides a unified interface for managing and configuring Amazon EC2 instances and other resources, helping automate operational tasks.**

**CloudWatch (Amazon CloudWatch):**

**A monitoring service that collects and tracks metrics, logs, and events for AWS resources and applications, providing insights into their performance.**

**Lambda (AWS Lambda):**

**A serverless computing service that allows you to run code without provisioning or managing servers, enabling you to automate tasks and respond to events.**

**EFS (Amazon Elastic File System):**

**Offers scalable and shared file storage for Amazon EC2 instances, providing a flexible and efficient way to store and access data.**

**In summary, these Amazon services cover a wide range of cloud computing aspects, from storage to networking, security to monitoring, and database management to serverless computing, empowering you to build robust and scalable applications within the AWS cloud environment.  
  
  
  
  
Good Grammer**

**During my enrollment at ASU, I participated in an AWS Academy course that centered around a cafe web server scenario. The course structure featured incremental challenge labs in each module, gradually increasing in complexity. Emphasis was placed on practical utilization of AWS resources rather than delving into application logic. This hands-on experience provided me with adeptness in navigating both the AWS Command Line Interface (CLI) and User Interface (UI).**

**Project Highlights:**

**As part of the course, I engaged in diverse projects, including hosting a website using Amazon S3. This involved uploading the website onto an S3 bucket, configuring appropriate bucket policies for public access, enabling versioning, and setting up automated data retention.**

**I also successfully established an Amazon Elastic File System (EFS) and linked it to an EC2 instance. This process encompassed creating a security group to allow secure NFS communication, associating it with a Virtual Private Cloud (VPC), and configuring a file system on the EC2 instance through SSH.**

**In a more extensive project, my aim was to prepare an EC2 instance to host a cafe website with online ordering and staff order viewing capabilities. I accessed the AWS Cloud9 IDE on the EC2 instance for an integrated development environment, verified the LAMP stack, and ensured the Apache and MySQL services were operational and configured for automatic restarts. I leveraged commands like "sudo chkconfig," "sudo service," and "service (service) status" to manage service statuses and startup behavior. Subsequently, I downloaded, extracted, and tested the cafe application files, all while securely managing sensitive configuration data using AWS Systems Manager Parameter Store. A meticulous MySQL database configuration and thorough testing ensured the website's functionality and successful order placement.**

**Amazon Services Defined:**

**Here's a succinct overview of the Amazon services covered in the scenario:**

**S3 (Amazon Simple Storage Service): Cloud-based storage for secure and scalable data storage and retrieval.**

**EBS (Amazon Elastic Block Store): Persistent block storage for Amazon EC2 instances.**

**Glacier (Amazon Glacier): Low-cost data archival storage service.**

**IAM (Identity and Access Management): User access control and permission management.**

**VPC (Amazon Virtual Private Cloud): Creation of isolated networks with private subnets and security groups.**

**Elastic Load Balancing (ELB): Distribution of incoming application traffic for high availability.**

**EC2 Auto Scaling (Amazon EC2 Auto Scaling): Automatic adjustment of EC2 instances based on demand.**

**RDS (Amazon Relational Database Service): Managed database setup, operation, and scaling.**

**DynamoDB (Amazon DynamoDB): Fully managed NoSQL database service.**

**Systems Manager (AWS Systems Manager): Unified interface for managing EC2 instances and resources.**

**CloudWatch (Amazon CloudWatch): Monitoring of metrics, logs, and events for resources and applications.**

**Lambda (AWS Lambda): Serverless computing for automated tasks.**

**EFS (Amazon Elastic File System): Scalable and shared file storage for EC2 instances.**

**In summary, these services collectively offer a comprehensive toolkit for building resilient and scalable applications within the AWS cloud environment.**

**AWS projects:** <https://aws.amazon.com/s3/faqs/>

**Project1: Challenge lab3 Creating a static website for a café.**

# **Static website hosting.**

## First created a bucket and enabled the static website hosting on the bucket

## Uploaded the source code to the bucket

## Wrote a bucket policy to allow all public reads to the bucket

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "PublicRead",

"Effect": "Allow",

"Principal": "\*",

"Action": "s3:GetObject",

"Resource": "arn:aws:s3:::bucket914/\*"

}

]

}

* 1. Enabled the versioning feature on the bucket
  2. Created a bucket policy to move old versions to Standarad IA after 30 days and delete old versions after 365 days
  3. Created a second bucket for **cross-region replication**: , a new feature that automatically replicates data across AWS regions. With cross-region replication, every object uploaded to an S3 bucket is automatically replicated to a destination bucket in a different AWS region that you choose. It is free

**Project2:   
Guided lab 4:** Setting Up Amazon Elastic File System (EFS) and Connecting to EC2In the interview, you can explain that you followed a step-by-step process to set up an Amazon Elastic File System (EFS) and connect it to an EC2 instance. The goal was to create a scalable and flexible file storage solution for AWS resources. Here's a summary of what you did:

1. Created a Security Group: You started by creating a new security group named "EFS Mount Target" to allow inbound access for TCP on port 2049 for Network File System (NFS). This security group was associated with the EFS mount targets.
2. Created an EFS File System: Next, you created an EFS file system in the AWS Management Console. The file system was configured without enabling automatic backups and had lifecycle management set to "None."
3. Configured Tags and VPC: You added tags to the EFS file system for better organization. The file system was attached to the specific VPC, "Lab VPC."
4. Configured Mount Targets: You detached the default security group from each Availability Zone mount target and attached the newly created "EFS Mount Target" security group to each Availability Zone mount target. This ensured that the EFS file system could be accessed securely.
5. Connected to the EC2 Instance via SSH: You connected to the EC2 instance using Secure Shell (SSH). For Windows users, you used PuTTY, while for macOS and Linux users, you used the terminal.
6. Created a Directory and Mounted the EFS File System: In the SSH session, you created a new directory and mounted the EFS file system to it. This enabled the EC2 instance to access and use the EFS storage.
7. Examined Performance Characteristics: To assess the performance of the EFS file system, you used the Flexible IO (fio) benchmarking utility. This helped analyze the write performance characteristics of the file system.
8. Monitored Performance with CloudWatch: You monitored the performance of the EFS file system using Amazon CloudWatch metrics. This provided valuable insights into the throughput and data write I/O bytes.

Overall, the process involved creating a scalable and high-performing file storage solution using EFS, ensuring secure access through properly configured security groups, and monitoring the performance to optimize the system's efficiency. The successful completion of these tasks demonstrated your ability to set up and manage AWS resources effectively.

**Challenge lab4:**

**Creating a dynamic website for the cafe**

* Aim: Prepare an EC2 instance to host a website for the café, enabling online ordering for customers and allowing café staff to view submitted orders.
* Connect to the IDE on the EC2 instance: Access AWS Cloud9 IDE running on the EC2 instance to have an integrated development environment and made sure a LAMP stack was already installed :

(linux: cat /proc/version  
Apache:sudo httpd -v   
service httpd status  
MySQL: mysql --version service   
mysqld status   
PhP: php –version  
A LAMP stack is a bundle of four different software technologies that developers use to build websites and web applications. LAMP is an acronym for the operating system, Linux; the web server, Apache; the database server, MySQL; and the programming language, PHP.

* First, I started by ensuring that the web server and database were up and running on the existing EC2 instance. I activated the web server (Apache) and database (MySQL) services and configured them to start automatically after any future instance restarts. This allowed the website to be accessible.  
  sudo chkconfig (service) on ­  
  sudo service (service) start   
  sudo service (service) status

Service:httpd, mysqld

* Next, I downloaded the café application files and extracted them to the web server's document root, making sure they were in the right directory for the website to work correctly.
* To configure the application parameters, I used AWS Systems Manager Parameter Store. This allowed me to securely store and manage the necessary configuration data for the application, like database credentials and other sensitive information.
* For the MySQL database, I set the root password and created the necessary tables and data that the café application required. This ensured the database was ready to handle orders and other application functions.
* After installing the café application and ensuring everything was set up correctly, I tested the website's functionality. I verified that the website was accessible from the internet, and I placed some test orders to ensure the online ordering feature was working as expected.
* To make the process scalable and to have a backup in case of any issues, I created an Amazon Machine Image (AMI) of the existing EC2 instance. This AMI served as a snapshot of the configured environment. From this AMI, I launched a new EC2 instance (ProdCafeServer) in a different AWS Region (Oregon). This region choice provided additional redundancy and disaster recovery capabilities.
* I also configured the necessary AWS Systems Manager parameters in the new region, ensuring that the application could access the required data and resources seamlessly.
* Finally, I verified the new café instance, making sure the website was accessible and all the features, including order placement, were fully functional.  
    
  **Challenge Lab 5:** **Migrating a Database to Amazon RDS  
    
  Aim :**

**The aim of this lab is to migrate the café's existing MariaDB database, currently hosted on an Amazon EC2 instance, to Amazon RDS. By doing so, the café can ensure data consistency, regular upgrades, and patches without the need for manual administration.** **Additionally, the café's web application will be updated to use the new RDS database for storing data related to all future orders**

1. In the first step, I created an RDS instance with MariaDB as the engine type and used the Dev/Test template. I ensured that all the other parameters, such as the DB instance identifier, username, and the precise password , were set as specified.

2. Next, I used AWS Systems Manager to connect to the existing EC2 server, which was hosting the café website. I established a terminal session to proceed with the database migration.

3. I queried the MariaDB on the EC2 server, and to access it, I used the parameter store to retrieve the password with the command "mysql -u root -p". I verified that the database indeed existed on the EC2 server.

4. To ensure data preservation, I captured the existing data by using the mysqldump utility. I executed the command "mysqldump --databases cafe\_db -u root -p > CafeDbDump.sql" to export the data into a file named "CafeDbDump.sql".

5. In the subsequent step, I connected to the new RDS instance, which also utilized the MariaDB engine. To enable this connection, I made sure that the inbound rules were configured to allow SQL traffic on port 3306 for the RDS instance.

Here's why you need to configure inbound rules for an RDS instance:

Network Isolation: Amazon RDS instances are designed to be securely isolated within a Virtual Private Cloud (VPC) by default. This means that incoming traffic is not allowed by default, including SQL traffic (port 3306 for MariaDB/MySQL).

6. I proceeded to import the SQL dump from the EC2 server to the RDS instance using the command "mysql -u admin -p --host <rds-endpoint> < CafeDbDump.sql". This step ensured that all the existing data from the EC2 server's database was migrated to the new RDS instance.

7. The application was initially querying the parameter store to connect to the database installed on the EC2 server. To adapt it to the RDS instance, I made necessary changes in the parameter store. Specifically, I updated the connection information to the RDS endpoint and ensured that the application could retrieve the corresponding password from the store.

The entire migration process was successful, and the café's data is now safely stored in the new RDS instance, providing improved scalability, automated updates, and reduced administrative burden.

Definitions:

* System Manager:  
  Amazon Web Services (AWS) that helps you manage and automate your cloud resources. It provides a unified user interface that allows you to view operational data and manage your AWS resources in a central place.

Key Features:

* + Parameter Store: Securely store configuration data, such as database connection strings, API keys, and other settings. These parameters can be easily accessed by applications and services running on AWS.
  + Session Manager: A fully managed, secure way to control and manage instances (virtual servers) without the need for SSH (Secure Shell) access. It allows you to establish an interactive shell session with your instances directly from the AWS Management Console

**With session Manager do we not need to identify ourselves to the server.** That's correct! With AWS Systems Manager Session Manager, you don't need to identify yourself to the server using traditional methods like SSH keys or passwords. Instead, Session Manager provides a secure and convenient way to establish an interactive shell session with your Amazon EC2 instances (virtual servers) directly from the AWS Management Console or through the AWS Command Line Interface (CLI).

**Module 6 Guided Lab - Creating a Virtual Private Cloud**

In this lab, we create a Virtual Private Cloud (VPC), a private network dedicated to our AWS account. Within the VPC, we create two subnets - one for public-facing resources and another for private resources. The public subnet allows internet access, while the private subnet remains isolated.

To enable internet access for the public subnet, we create an Internet Gateway, which acts as a bridge between the VPC and the internet. We then configure a route table for the public subnet, which directs internet-bound traffic to the Internet Gateway.

For security, we create a Security Group, a virtual firewall that controls inbound and outbound traffic for our AWS instances. We allow HTTP traffic in the Security Group to enable access to our application server.

Finally, we launch an EC2 instance (application server) in the public subnet, ensuring it has access to the internet. The instance is equipped with Apache Web Server, PHP, and the necessary application files. Once the instance is running, we confirm its status and access it via its Public IPv4 DNS address to ensure successful setup and configuration.

**S3:**

With S3, The aim was to host a static website.

We did it using the aws console as well as aws cli:

In the console we created a bucket, edited its bucket policies to allow public read access, enabled version on the bucket:

We used to ssh into the aws account instance, using a private key and configure it with the aws account credentials

Created a bucket using the aws s3 api command:

create-bucket

--bucket <value>

Copied all the website files to the bucket

Edited the bucket ownership using commands like: put-bucket-ownership-controls\

--bucket <my-bucket> \

--ownership-controls "Rules=[{ObjectOwnership=BucketOwnerPreferred}]"

aws s3api put-public-access-block \

--bucket <my-bucket> \

--public-access-block-configuration "BlockPublicAcls=false,IgnorePublicAcls=false,BlockPublicPolicy=false,RestrictPublicBuckets=false"

Run the following command to configure the bucket that you created earlier for static website hosting. This command identifies the index.html file as the index document.

aws s3 website s3://<my-bucket>/ --index-document index.html

EBS:

We had to create a snapshot of the volume and retain only 2 of the most recent snashots.

First we have to find out the volume id of the EBS volume: VolumeId

To get a full description of the Processor instance, copy the following command and run it from within your instance:

aws ec2 describe-instances --filter 'Name=tag:Name,Values=Processor'

This is the base AWS CLI command for retrieving information about EC2 instances.

To narrow down the results of the previous command further, copy the following command and run it from within your instance: query option

aws ec2 describe-instances --filter 'Name=tag:Name,Values=Processor' --query 'Reservations[0].Instances[0].BlockDeviceMappings[0].Ebs.{VolumeId:VolumeId}'

Once we find the ebs volume id:   
we have To shut down the Processor instance,

1. aws ec2 stop-instances --instance-ids INSTANCE-ID

 verify that the Processor instance has stopped by running the following command,

1. aws ec2 wait instance-stopped --instance-id INSTANCE-ID

To create your first snapshot of the root volume of your Processor instance, copy the following command, replace VOLUME-ID\_ with your volume id, and run it in your SSH window:

aws ec2 create-snapshot --volume-id VOLUME-ID

to check the status of your snapshot, copy the following command, replace SNAPSHOT-ID your **snapshot-id**, and run it in your SSH window:

aws ec2 wait snapshot-completed --snapshot-id SNAPSHOT

1. To restart the Processor instance, copy the following command, replace the INSTANCE-ID to your instance id and run it in your SSH window:

aws ec2 start-instances --instance-ids INSTANCE-ID

1. aws ec2 wait instance-running --instance-id INSTANCE-ID

**IAM:**

There was a web application hosted on an ec2 instance on aws.

The objective was to create IAM groups and assign IAM users and the required AWS service access

The first group was AppDevelopers group with access to cloud9 and EC2 instance, and the second was DBAdministrators with access to RDS and access to aws systems manager.

So first she will create IAM groups and then attach IAM policies to it. And then assign users to the IAM groups.

if more developers or admins are added on to the team then they can be added to these groups as well.

The new users will inherit these rules permissions correctly by default.

We also used IAM policy simulator to observe the scope of access that is granted by different policies.

Access IAM advisors,to observe what access rights different users are taking advantage of.

Create custom iam policies by using the visual editor

So first we create an APPDeveloper group and attache the ec2readonlyaccess and cloud9environmentmember access.

Then created a user called Nikhil and assigned a password and then added him to the appdeveloper group.

We tested the access by logging in as an app developer I.e Nikhil

Using the credentials generated above

Next we created a user group called dbadmins and attached rds\_readonly\_access and ssmfullaccess

We created another user say olivia and added her to the group. She only had access the rds and ssm but not to the ec2 instances. Then we added ec2readonly and iamreadonly policies to the IAM users group

We tested the access using awsiamsim website.

Where we selected a particular user, and then a particular policy is selected and then ran the simulation to see what are the access a particular user has

We also created custom policies and attached it to the usergroups and played around with it in the sandbox environment.

**Select a service**

**Actions allowed**,

**AutoScaling**

**The aim was to create a**  scalable web server system:

create a new Amazon Machine Image (AMI) from an existing Amazon Elastic Compute Cloud (Amazon EC2) instance. You will use that AMI as the basis for defining a system that will scale automatically under increasing loads.

We used ec2, autoscaling groups and elastic load balancers for this project

scalable web server system:

we ssh into the web server system:

we created a custom ami using the :

aws ec2 create-image --name WebServer --instance-id <NEW-INSTANCE-ID>

The next ai was to create an Auto Scaling EnvironmentA screenshot of a computer

Description automatically generated

First we create a load balancer that pools a group of EC2 instances under a single DNS address.

You will use Auto Scaling to create a dynamically scalable pool of EC2 instances based on the image that you created in the previous section.

Finally, you will create a set of alarms that will scale out or scale in the number of instances in your load balancer group whenever the CPU performance of any machine within the group exceeds or falls below a set of specified thresholds.

Creating an appln load balancer:

Create load balancer in aws console-> give it a name-> configured the vpc->

You will now specify which subnets the load balancer should use. The load balancer will be internet facing, so you will select both public subnets.

A screenshot of a computer

Description automatically generated

For security groups:

We chose httpaccesss securitygroup.  
  
Then we create target groups:

**Analysis**: A target group defines where to send traffic that comes into the load balancer.

The Application Load Balancer can send traffic to multiple target groups based on the URL of the incoming request, such as having requests from mobile apps go to a different set of servers. Your web application will use only one target group.

In the **Listeners and routing** section:

* We **Create target group-> choosed Instances**-> **Advanced health check settings**.  
  The Application Load Balancer automatically performs health checks on all instances to ensure that they are responding to requests
* Entered the threshold values of: 2 for every 10 seconds on a specified path:
* This means that the health check will be performed every 10 seconds. If the instance responds correctly twice in a row, it will be considered healthy.
* Choose **Next**. The Register targets screen appears.:  
  Targets are the individual instances that will respond to requests from the load balancer.

You do not have any web application instances yet, so you can skip this step.

* Review the settings, and choose **Create target group**.

### Scroll to the bottom and choose **Create load balancer**. **Create a Launch Template**

Your Auto Scaling group will use the launch template to know which AMI to use to create new EC2 instances. For this example, you will launch the AMI that you created previously, which automatically configures itself as a web server when it is launched.

1. In the left navigation pane, choose **Launch Templates**.
2. Choose **Create launch template**
3. For **Launch template name**, enter WebServerLaunchTemplate
4. In the **AMI machine image (AMI)** pane, search for WebServer
5. Select **WebServer**. It is under the **My AMIs** group.
6. In the **Instance type** section, select **t2.micro**.
7. In the **Network settings** section, for **Security groups**, select **HTTPAccess**.
8. Expand the **Advanced details** section, and set **Detailed CloudWatch monitoring** to **Enable**.
9. Scroll to the bottom, and choose **Create launch template**

A success message appears.

### **Create an Auto Scaling Group**

Your Auto Scaling group will create a minimum number of EC2 instances that will reside behind your load balancer. In subsequent procedures, you will also add scale-out and scale-in policies that increase or decrease the number of running instances in reaction to alarms from Amazon CloudWatch.

1. In the left navigation pane, scroll to the bottom of the menu, and choose **Auto Scaling Groups**.
2. Choose **Create Auto Scaling group**
3. In Step 1, **Choose launch template or configuration**, configure:
   * **Auto Scaling group name**: Enter WebServersASGroup
   * **Launch template**: Choose **WebServerLaunchTemplate**
4. Choose **Next**

* In the **Network** pane, configure:
  + **VPC**: Choose **Lab VPC**
  + **Subnets**: Choose **Private Subnet 1** and **Private Subnet 2**
* Choose **Next**
* In the **Load balancing** pane, choose **Attach to an existing load balancer**.
* In the **Attach to an existing load balancer** pane, for **Existing load balancer target groups**, choose **webserver-app**.
* In the **Additional settings** pane, select  **Enable group metrics collection within CloudWatch**.

1. Choose **Next**

**Analysis**: In the next few steps, you will configure Auto Scaling to target 45 percent CPU Utilization. If average CPU Utilization exceeds this target, additional instances will be launched. If average CPU Utilization falls below this target, instances *might* be terminated.

1. In the **Group size** pane, configure:
   * **Desired capacity**: Enter 2
   * **Minimum capacity**: Enter 2
   * **Maximum capacity**: Enter 4
2. In the **Scaling policies** pane, choose **Target tracking scaling policy**, and configure:
   * **Scaling policy name**: Enter MyScalingPolicy
   * **Metric type**: Choose **Average CPU utilization**
   * **Target value**: Enter 45
3. Choose **Next**
4. On the **Add notifications** page, choose **Next**
5. On the **Add tags** page, choose **Add tag** and configure:
   * **Key:** Enter Name
   * **Value:** Enter WebApp
6. Choose **Next**
7. At the bottom of the **Review** page, choose **Create Auto Scaling group**

### **Verifying the Auto Scaling Configuration**

In this task, you will verify that both the Auto Scaling configuration and the load balancer are working. You will use a pre-installed script on one of your servers that will consume CPU cycles, which will invoke the scale-out alarm.

1. In the left navigation pane, choose **Instances**.
2. Verify that two new instances named **WebApp** are being created as part of your Auto Scaling group.

Wait for the two new instances to complete initialization before you proceed to the next step.

**Tip**: Observe the **Status check** column for the instances until it shows that both status checks have completed successfully.

1. In the left navigation pane, choose **Target Groups**.
2. Select your target group (**webserver-app**).
3. In the **Targets** tab in the lower half of your screen, verify that two instances are being created. Refresh this list until the **Status** of these instances changes to **healthy**.

You can now test the web application by accessing it through the load balancer.

1. In the left navigation pane, choose **Load Balancers**.
2. Select  **webserverloadbalancer**.
3. From the **Description** tab below, copy the **DNS name** value.

The value looks like webserverloadbalancer-xxxxxxxxx.xx-xxxx-x.elb.amazonaws.com.

This value will be referred to as **load-balancer-url** in the next step.

1. Open a new web browser tab, paste the URL into the address bar, and then press Enter.
2. On the web page, choose **Start Stress**.

This calls the application **stress** in the background, which causes the CPU utilization on the instance that serviced this request to spike to 100 percent.

1. Return to the console, and in the left navigation pane, choose **Auto Scaling Groups**.
2. Select  **WebServerASGroup**.
3. Choose the **Activity** tab for your Auto Scaling group, and scroll down to the **Activity History** pane.

After a few minutes, you should see a new entry, which indicates that your Auto Scaling group is launching a new EC2 instance.

This is because CloudWatch detected that the average CPU utilization of your Auto Scaling group exceeded 45 percent, and your scale-up policy has been invoked in response.

**RDS: The main aim was to migrate the databse which ran in an aws ec2 LAMP stack to aws rds.**

**before**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  
The RDS database will be deployed in the same virtual private cloud (VPC) as the instance.  
During the migration process, we built the components illustrated in this diagram, including two private subnets in different Availability Zones, a security group for the database instance, and the RDS database instance itself. After the database has been migrated, you will reconfigure the Mom & Pop Café application to use the RDS instance database instead of the local database.

First step was to ssh into the ec2 instance using a pvt key and configuring the account with access and secret keys.

A CLI Host instance resides in the same subnet to enable the administration of the instance by using the AWS Command Line Interface

We first gathered information about the aws instance where the application was running such as its  **Instance ID**, **Instance Type**, **Public DNS name**, **Public IP address**, and **Availability Zone** using the command

aws ec2 **describe-instances** \

--filters "Name=tag:Name,Values= MomPopCafeInstance" \

--query

aws ec2 describe-instances \

--filters "Name=tag:Name,Values= MomPopCafeInstance" \

--query "Reservations[\*].Instances[\*].[InstanceId,InstanceType,PublicDnsName,PublicIpAddress,Placement.AvailabilityZone,VpcId,SecurityGroups[\*].GroupId]"

2nd  we gathered the IPv4 CIDR Block the subnet id and its cidr block.

aws ec2 **describe-vpcs** --vpc-ids <MomPopCafeInstance VPC ID> \

--filters "Name=tag:Name,Values= MomPopCafe VPC" \

--query "Vpcs[\*].CidrBlock"

Determine the **Subnet ID** and **IPv4 CIDR block** of MomPopCafe Public Subnet 1, which is the only subnet in the VPC. In the SSH window, enter:

aws ec2 **describe-subnets** \

--filters "Name=vpc-id,Values=<MomPopCafeInstance VPC ID>" \

--query "Subnets[\*].[SubnetId,CidrBlock]"

Determine the AZ:

aws ec2 **describe-availability-zones** \

--filters "Name=region-name,Values=<region>" \

--query "AvailabilityZones[\*].ZoneName"

Next we created the prerequisite infrastructure components for the Amazon RDS instance

* **MomPopCafeDatabaseSG** (Security group)
* **MomPopCafeDB Private Subnet 1**
* **MomPopCafeDB Private Subnet 1**
* **MomPopCafeDB Subnet Group** (Database subnet group)

Create the MomPopCafeDatabaseSG security group. This security group is used to protect the Amazon RDS instance:  It should have an inbound rule that only allows MySQL requests (using the default TCP protocol and Port 3306) from instances that are associated with the MomPopCafeSecurityGroup

aws ec2 create-security-group \

--group-name MomPopCafeDatabaseSG \

--description "Security group for Mom Pop Cafe database" \

--vpc-id <MomPopCafeInstance VPC ID>

1. After the command completes, record the returned **GroupID** as:

MomPopCafeDatabaseSG Group ID: sg-zzzzzzzzzz

1. Next, create the inbound rule for the security group. In the SSH window, enter:

aws ec2 authorize-security-group-ingress \

--group-id <MomPopCafeDatabaseSG Group ID> \

--protocol tcp --port 3306 \

--source-group <MomPopCafeSecurityGroup Group ID>

Therefore, we create two private subnets and a database subnet group next.

Create the **MomPopCafeDB Private Subnet 1**. This subnet hosts the Amazon RDS database instance.

For this we had to make sure that the CIDR block for IpV4 was not used up by either the VPC or The public subnet.:

aws ec2 create-subnet \

--vpc-id <MomPopCafeInstance VPC ID> \

--cidr-block 10.200.10.0/23 \

--availability-zone <availability-zone>

Created a second pvt subnet similarly.

And them created a dbsubnet group

aws rds create-db-subnet-group \

--db-subnet-group-name "MomPopCafeDB Subnet Group" \

--db-subnet-group-description "DB subnet group for Mom & Pop Cafe" \

--subnet-ids <MomPopCafe Private Subnet 1 ID> <MomPopCafe Private Subnet 2 ID> \

--tags "Key=Name,Value= MomPopCafeDatabaseSubnetGroup"

### **Create the Amazon RDS MariaDB instance**

You can now create the **MomPopCafeDBInstance** that is shown in the [After Migration topology diagram](https://labs.vocareum.com/web/2353490/2023815.0/ASNLIB/public/docs/lang/en_us/README.md#after-migration-topology-diagram). Using the AWS CLI, create an Amazon RDS MariaDB instance with the following configuration settings:

* DB instance identifier: **MomPopCafeDBInstance**
* Engine option: **MariaDB**
* DB engine version: **10.2.11**
* DB instance class: **db.t2.micro**
* Allocated storage: **20 GB**
* Availability Zone: **MomPopCafeInstance Availability Zone**
* DB Subnet group: **MomPopCafeDB Subnet Group**
* VPC security groups: **MomPopCafeDatabaseSG**
* Public accessibility: **No**
* Username: **root**
* Password: **Re:Start!9**

aws rds create-db-instance \

--db-instance-identifier MomPopCafeDBInstance \

--engine mariadb \

--engine-version 10.6.14 \

--db-instance-class db.t2.micro \

--allocated-storage 20 \

--availability-zone <MomPopCafeInstance Availability Zone> \

--db-subnet-group-name "MomPopCafeDB Subnet Group" \

--vpc-security-group-ids <MomPopCafeDatabaseSG Group ID> \

--no-publicly-accessible \

--master-username root --master-user-password 'Re:Start!9'

* After creating the rds we retrieved its information like: Endpoint address
* Availability Zone

## **Migrate application data to the Amazon RDS instance**

In this task, you migrate the data from the existing local database to the newly created Amazon RDS database. Specifically, you:

a) Open an SSH session to the MomPopCafeInstance.

b) Use the mysqldump utility to create a backup of the local database.

c) Restore the backup to the Amazon RDS database.

d) Test the data migration.

After the ssh seesion is established:  
Use the mysqldump utility to create a backup of the local **mom\_pop\_db** database.

mysqldump --user=root --password='Re:Start!9' \

--databases mom\_pop\_db --add-drop-database > mompopdb-backup.sql

This command generates structured query language (SQL) statements in a file named mompopdb-backup.sql, which can be run to reproduce the schema and data of the original mop\_pop\_db database.

mysql --user=root --password='Re:Start!9' \

--host=<RDS Instance Database Endpoint Address> \

< mompopdb-backup.sql

This command creates a mysql connection to the RDS instance and runs the SQL statements in the mompopdb-backup.sql file.

Finally, verify that the mom\_pop\_db was successfully created and populated in the Amazon RDS instance. Open an interactive mysql session to the instance, and retrieve the data in the **product** table of the mom\_pop\_db database. In the SSH window, enter:

mysql --user=root --password='Re:Start!9' \

--host=<RDS Instance Database Endpoint Address> \

mom\_pop\_db

select \* from product;

## **Task 3: Configure the website to use the Amazon RDS instance**

 This step is simple because the designer of the application followed best practices and externalized the database connection information as parameters in the AWS Systems Manager Parameter Store. In this task, you change the database URL parameter of the Mom & Pop Café application to point to the endpoint address of the RDS instance.

1. In the **AWS Management Console** browser tab, select **Services > Systems Manager**.
2. In the left navigation pane, choose **Parameter Store**.
3. In the **Parameters** list, choose **/mompopcafe/dbUrl**. The current value of the parameter is displayed, along with its description and other metadata information.
4. Choose **Edit**.
5. In the **Parameter details** page, replace the text in the **Value** box with the value of **RDS Instance Database Endpoint Address** that you recorded earlier.
6. Choose **Save changes**.

The dbUrl parameter now references the Amazon RDS database instance instead of the local database.

1. Test the website to confirm that it is able to access the new database correctly. In a browser window, enter the following URL:

http://<MomPopCafeInstance Public DNS Name>/mompopcafe

Substitute <MomPopCafeInstance Public DNS Name> with the value that you recorded earlier.

1. The website's home page should load correctly. Select the **Order History** tab and observe the number of orders in the database. Compare this number with the **Number of orders** that you recorded before the database migration. Both numbers should match.
2. Feel free to place some new orders and verify the successful operation of the website. **Close** the browser tab when you are finished.

* **Restaurant Reccomendation System:  
  My Story  
  Overview.**
* **I built a recommendation system that suggests restaurants to users based on their preferences.**
* **I made use of tools and technologies likeNLP , and libraries like Gensim,LDA**

Dependencies:

Pymongo,HomeBrew,MongoDB,Gensim,NLTK

In an interview, you can explain the above dependencies as follows:

pymongo:

* pymongo is a Python library that allows you to work with MongoDB, a popular NoSQL database. It provides a simple and convenient way to interact with MongoDB from your Python code.

Homebrew:

* Homebrew is a package manager for macOS, which simplifies the installation of various software packages and libraries. It is widely used in the macOS ecosystem to manage software installations and updates.

MongoDB:

* MongoDB is a powerful NoSQL database that stores data in JSON-like documents. It is widely used for handling large volumes of data and scalable applications.

Gensim:

* Gensim is a Python library for topic modeling and document similarity analysis. It is widely used in natural language processing (NLP) tasks, such as text summarization and document clustering.

NLTK (Natural Language Toolkit):

* NLTK is a comprehensive library for NLP tasks in Python. It provides various tools and resources for tasks like tokenization, part-of-speech tagging, sentiment analysis, and more.

Downloading NLTK Data:

* After installing NLTK, you need to download additional data resources (corpora, models, etc.) that are required for certain NLP functionalities. These resources are downloaded using the nltk.download() function in Python.

pymongo: Python library for working with MongoDB.

Homebrew: macOS package manager for easy software installations.

MongoDB: Powerful NoSQL database for handling large data. Stores data in the form of JSOn format

Gensim: Python library for topic modeling and NLP tasks.

NLTK: Comprehensive NLP library in Python. It provides various tools and resources for tasks like tokenization, part-of-speech tagging, sentiment analysis, and more.

* Downloading NLTK Data: Essential NLP resources for NLTK.
* I have used the Yelp academic Dataset , the fields present in the data, are review\_id, user\_id, business\_id, stars, date, text, useful, funny, and cool.
* First read the Yelp dataset file containing reviews in JSON format. I process it and store it in the Mongo\_db
* organizes the reviews in such a way that all the comments a user has given about various restaurants are grouped together under their unique user ID.
* organize the various comments that a restaurant has received from different users and group them together under the specific business ID.
  + [To insert a record, into a collection, we use the insert\_one() method. The method returns a InsertOneResult object, which has a property, inserted\_id, that holds the id of the inserted document.]
* Now preprocess the dataset:

First I load the stopwords list .

Then I use the nltk function -sent\_tokenize on each review, the output is the broken down paragraph to a list of strings   
Then I break each sentence into words using word\_tokenize. The result is a list of words.

The if the word is not in my dictionary of stopwords, I store it.

Then I pass it to the nltk function: pos\_tag . POS tagging is the process of assigning a grammatical label to each word to identify its part of speech, such as noun, verb, adjective, etc.  
Use the WordNetLemmatizer() fnction to lemmatize the word. Basically find the root word. Like playing-play. At the end of this preproccesing, I have preprossed list of words.

Pass the list of words to doc2bow method: The doc2bow(...) is a gensim library method and is used to convert a document (a list of words) into a bag-of-words (BoW) representation.  
Bag\_of\_words representation: It is a way to convert words into numerical vectors, where each vector represents a document's word frequency distribution.

After converting it into a bagofwords, we feed it into the lda model  
LDA assumes that each document is a mixture of various topics, and each topic is represented by a distribution of words. However, we don't know the topics in advance; LDA will find them for us.  
During the training process, LDA iteratively identifies the topics by estimating which words are likely to be associated with each topic and how topics contribute to each document. This is done through statistical inference.

The inference is stored in the database as vectors.   
this is done for each review of Business and user. Which results in a user profile and a business profile.  
All this is stored in the database.  
Now when we want to enter a particular user and find the best recommended and least recommended restaurant we can use cosine similarity.  
so cos 0 is 1 and cos 90, is 0 i.e higly unlikely recommendation.  
the cosine similarity is applied on the user profile against all the business profiles. The output with highest value closest to 1 is higly recommened and with 0 is least recommended.  
**How LDA Works:**

* Imagine you have a collection of recipes, travel stories, and movie reviews in the box. LDA will try to figure out the main themes or topics that might be hidden in these documents.
* LDA starts by guessing what the topics might be. It randomly assigns some words to each topic, just to get started.
* Then, it goes through all the documents, and for each word in a document, it tries to reassign the word to a different topic. It does this based on two things: how frequently that word appears in that topic overall, and how frequently that topic appears in the document.
* It repeats this process several times, shuffling words around and reassigning them to different topics each time.
* As LDA keeps doing this, it starts to get better and better at finding the right topics. Eventually, it reaches a point where the words in each topic are similar and consistently appear together in certain documents.
* At the end of this magic sorting process, LDA gives you the main topics it discovered and which words are most associated with each topic.
* To apply cosine similarity between the two vectors x and y, you need to compute the dot product of these two vectors and then divide it by the product of their magnitudes (Euclidean norms). The formula for cosine similarity is:
* cosine\_similarity = (x dot y) / (||x|| \* ||y||)

Front End:  
The code follows a simple web application architecture using the Flask framework, which is based on the Model-View-Controller (MVC) design pattern. The architecture can be described as follows:

*Model-View-Controller (MVC) Pattern:*

*Model*: In this code, the "Model" is represented by the MongoDB database. It stores data related to users, user preferences, and business profiles. The "NEW\_COLLECTION" and "USER\_PROFILE" correspond to the model components.

*View*: The "View" is represented by HTML templates (home\_page.html, new\_user.html, new\_user\_display.html, existing\_user.html, and display.html). These templates define the user interfaces and how data is presented to users.

*Controller*: The "Controller" is represented by the Flask app (app). The app handles user requests, processes data, and controls the interaction between the model and view components. The app routes incoming requests to specific functions and decides which templates to render based on the user's actions.

*Simple Web Application Architecture:*

The code defines various routes (@app.route(...)) to handle different URL paths, representing different pages/views of the web application. For example, '/' for the home page, '/new\_user/' for new user registration, and '/existing\_user/' for existing user query.

When a user accesses a specific URL, the corresponding function is executed, which either renders the appropriate HTML template or processes user data and provides restaurant recommendations.

*MongoDB as a Database:*

The code uses a MongoDB database to store user information and restaurant profiles.

The MongoDB database acts as a data repository, and the application can insert, retrieve, and process data from these collections.

Overall, this architecture is a basic web application architecture, where Flask serves as the web framework and MongoDB as the database. The user interacts with the web application through HTML templates, and the Flask app handles the communication between the front-end and the back-end (MongoDB). The application provides personalized restaurant recommendations to users based on their preferences using cosine similarity calculations.  
  
*IS my frontend Stateful?*  
  
Your code is not stateful in the traditional sense. In web development, a stateful application maintains the state of each user session across multiple requests and responses. This means that the application remembers information about each user's interactions and keeps track of their data between different requests. Stateful applications usually use sessions and cookies to store user-specific data.

However, in the provided code, each request is processed independently without maintaining any user-specific state between requests. The Flask application processes incoming requests, executes the corresponding functions, and returns responses accordingly. There is no storage or tracking of user-specific data between different requests, and each request is stateless.

The application uses HTTP methods like GET and POST, and the client sends data in the request body, but this does not make it stateful. It's a standard way of passing data to the server, and the server responds with appropriate data based on the received request.

The code performs specific actions based on the data received in each request, such as adding new users to the database or calculating restaurant recommendations based on user preferences. However, there is no persistent storage or user session management involved to maintain state across requests.

In summary, your code is stateless, as it does not maintain user-specific data between requests, and each request is independent of the others. The server processes each request and responds accordingly without relying on past interactions or user-specific state information.

* **Interview Story:  
  Overview:**
* Situation and Task:
* As part of my capstone project, I undertook the task of developing a restaurant recommendation system, aimed at providing personalized restaurant suggestions to users according to their individual preferences.
* Tools and Technologies:
* To accomplish this, I harnessed a range of tools and technologies, including:
* Utilizing Natural Language Processing (NLP) libraries for text analysis and understanding.
* Employing Gensim, a powerful library, for conducting topic modeling, an essential technique for uncovering patterns in large datasets.
* Leveraging Flask, a versatile web framework, to create a seamless and interactive web interface.
* Implementing MongoDB as the database of choice to manage and store the diverse dataset efficiently.
* Harnessing the programming capabilities of Python to drive the entire application's logic and functionality.
* Action Taken:
* I began by sourcing data from the Yelp academic dataset, which was structured in JSON format. This data was then carefully stored in a MongoDB database, facilitating effective data management and retrieval.
* In preparation for analysis, I conducted preprocessing of the dataset using various libraries including NLTK. This stage refined and organized the data, setting the groundwork for subsequent analysis.
* The outcome of this preprocessing phase culminated in the generation of both user and business profiles represented as vectors. These vectors were indicative of user preferences and the unique attributes of each business.
* To estimate the likelihood of user preference alignment with a specific restaurant, I computed the cosine similarity between the user and business profile vectors. This resulted in a numerical value ranging from 0 to 1, effectively quantifying the compatibility between user tastes and restaurant offerings.
* Final Deliverable:
* Upon concluding the project, I developed a web User Interface that allowed users to input their unique user ID. Through this interface, users could effortlessly retrieve valuable information. Specifically, they could access recommendations for both the best and least recommended restaurants, tailored to their individual preferences.
* In summary, my capstone project encompassed the creation of a restaurant recommendation system, adeptly utilizing a combination of NLP, topic modeling, web development, database management, and programming skills.

**In-depth**

* I worked with the Yelp academic Dataset, which contains fields like review\_id, user\_id, business\_id, stars, date, text, useful, funny, and cool.
* First, I read the Yelp dataset file in JSON format and processed it before storing it in MongoDB. This allowed me to efficiently manage and retrieve records.
* Next, I organized the reviews in a way that all the comments a user has given about various restaurants are grouped together under their unique user ID. Similarly, I organized the comments received by a restaurant from different users and grouped them under the specific business ID.
* To preprocess the dataset, I started by loading a list of stopwords. Then, for each review, I used the nltk function `sent\_tokenize` to break down the paragraph into a list of strings. After that, I tokenized each sentence into words using `word\_tokenize`, resulting in a list of words.
* Before proceeding, I checked if a word is in my dictionary of stopwords and stored only the non-stopwords. Then, I passed the words to the nltk function `pos\_tag` for Part-of-Speech (POS) tagging. This helped identify the part of speech (e.g., noun, verb, adjective) of each word.
* To further simplify word forms, I used the WordNetLemmatizer() function to lemmatize the words, finding their root forms. For instance, "playing" became "play." By the end of this preprocessing, I had a list of preprocessed words.
* I then passed this list of words to the `doc2bow` method from the gensim library. This method converted the list of words into a bag-of-words (BoW) representation. BoW is a numerical vector that represents the word frequency distribution of a document.
* After converting the reviews into a bag-of-words format, I fed them into the LDA (Latent Dirichlet Allocation) model. LDA assumes each document is a mixture of various topics, and each topic is represented by a distribution of words. The model identifies these topics during the training process by estimating word-topic associations and topic contributions to each document using statistical inference.
* The inference from the LDA model is stored in the database as vectors. This process was applied to each review of users and businesses, resulting in user profiles and business profiles based on their inferred topic distributions. All this information was stored in the database.
* Now, when I want to find the best and least recommended restaurant for a particular user, I can use cosine similarity. Cosine similarity is a measure that ranges from -1 to 1. A cosine similarity of 1 means a highly recommended restaurant, while a similarity of 0 means a highly unlikely recommendation (cos 90° is 0).
* To make recommendations, I applied cosine similarity on the user profile against all the business profiles. The output with the highest value, closest to 1, represents a highly recommended restaurant, while the output with the value closest to 0 is the least recommended one.

Challenges face:  
cold start problem,: It occurs when a model lacks sufficient information or historical data about these new users, making it difficult to generate meaningful insights or predictions.  
I solved it using content-based approach and collecting feedback. new users and their recommendation.

**why 50 topics**

Online Banking Website  
**had to build a functional Online Banking System with features like create, login logout a user, user authentication, addinga and displaying the loans, efficient data handling, and designing API’s.   
  
I split the application into client and server side, thus used client server architecture. The client side was mainly the UI where users could create accounts, login, apply for loans and view their loans.**

**The server hosted the application logic, processed the user request, and managed data.  
  
I utilized the Model View Controller architecture. For the application logic.**

**I mainly used mongoose for MongoDB, JavaScript for server and client-side scripting, Pug for template engine, CSS for UI,  
I made use of well-Defined REST API for interacting with the system and performing the CRUD operations and verified this using postman.**

**For authorization a user to the server, I used JWT, JSON Web Tokens.   
// There were about 15 plus API in my routes folder- customer router, loans router, loan**

JWT, which stands for JSON Web Token, is a compact, self-contained means of representing information between two parties in a secure and tamper-evident way. JWTs are often used for authentication and authorization in web applications and APIs. Here's how JWT works:

1. \*\*Creation of the JWT\*\*:

- When a user logs in or an authorization request is made, the server generates a JWT token.

- The JWT typically consists of three parts: a header, a payload, and a signature. These parts are concatenated together with periods (.) to form the complete token.

2. \*\*Header\*\*:

- The header typically consists of two parts: the type of token (JWT) and the signing algorithm being used, such as HMAC SHA256 or RSA.

- This part of the JWT is base64url-encoded JSON.

3. \*\*Payload\*\*:

- The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

- The payload is also base64url-encoded JSON.

4. \*\*Signature\*\*:

- To create the signature part, the server takes the encoded header, the encoded payload, a secret (or private key), and the algorithm specified in the header.

- The server signs this combined data to create the signature.

5. \*\*Token Creation\*\*:

- The header, payload, and signature are concatenated with periods to create the JWT.

- The resulting JWT is returned to the client, usually as part of a login or authentication response.

6. \*\*Token Usage\*\*:

- The client stores the JWT, often in a cookie or local storage, and includes it in subsequent requests to the server. This is typically done in the "Authorization" header of HTTP requests.

7. \*\*Server Verification\*\*:

- When the server receives an API request with a JWT, it decodes the JWT (splitting it into header and payload).

- The server then re-computes the signature using its own copy of the payload, header, and the secret key.

- If the newly computed signature matches the one in the JWT, and the token is not expired or otherwise invalid, the server trusts the information contained in the JWT and proceeds with the request.

8. \*\*Expiration and Claims\*\*:

- JWTs can include an expiration time (exp claim) or other claims like issuer (iss), subject (sub), audience (aud), and more.

- These claims can be used to check the validity and scope of the token.

JWTs are stateless, meaning the server doesn't need to keep track of sessions or user data. However, they should be used with caution and follow best practices to ensure security, such as protecting the secret key, using HTTPS to transmit JWTs, and verifying token signatures on the server.

**Mongoose- a popular JavaScript library used for object modeling and interacting with MongoDB database  
PostMan- Postman is an API platform for building and using API.**

**Work:**

**Bad grammer**I am responsible for developing and maintaining web pages of our departments website using WordPress CMS. I mainly used HTML,CSS and simple JS. . One major project was to migrate the entire website from custom theme to the university theme. So in custom theme, each page was customized and was hard to be in compliance with the university design preferences. Like the buttons used, the colors in them etc. So as a group of three, which was led by me, I migrated the website to the university standard theme and it was a drag and drop interface. I did it over last winter.   
I also take requests from various staff members who are responsible for various pages and update any pages accordingly.   
  
My Department maintained around 50 apps. So I have worked on the entire code base for updating the source code.   
  
In one of the projects I had to remove unused CSS elements. Since the apps were built from a long time and over the years, they have made several changes, I worked on identifying and eliminating such code. How did I acheve this- by testing its usage on a web browser, searching the application code for its usage.   
  
In another project I made an architectural change across the web apps,. So the administrator of each app Be it technical or non-technical was defined in a couple of places in the application and it was hard to maintain and update as the admins changed. So I had to restructure the application where this was initialized in an init method as soon as the application started and this was later used in the rest of the application code. This further helped us to use a structure and access it using indexes when there were multiple administrators.   
  
I was then tasked to develop a standalone application- the contact management system. So in tis application, I iterated over all the web applications init method, retrieved who the current admin of the app was and displayed it on a data table. For this I used jQuery and In the data table function I made an AJAX call to another page. The data that is returned is in JSON format and I populated the data into data table. After this first version it was pushed into production. In the second stage I made further improvements where I wrote logic to the display of one of the columns. It initially showed the entire URL, and I worked on it to just display the app name. I had to recurse through the directory and if it was an index page then list it, since it was the landing page. I wrote the application logic for this.   
  
Right now I am working on converting a jQuery based UI to Fomantic based UI,   
So jqgrid, and its widgets need to use the Fomantic UI. This meant rewriting parts of the code. This is giving me a good experience of data flowing between difrrent parts of the application code and understanding them to make the application work.  
 **Good Grammer**

I am responsible for developing and maintaining web pages on our department's website using WordPress CMS, primarily utilizing HTML, CSS, and simple JavaScript. I have been at the forefront of major projects, including one where I led a team of three in the successful migration of the entire website from a custom theme to the university theme. In the previous custom theme, each page was heavily customized, which made it challenging to align with the university's design preferences – encompassing elements such as buttons and color schemes. Leveraging a drag-and-drop interface, we accomplished this migration during the winter season, adhering to the university's standard theme.

In addition, I am the go-to person for addressing requests from various staff members who oversee different pages on the website. Ensuring prompt updates and modifications is a critical aspect of my role.

Furthermore, I have played a pivotal role in managing our department's portfolio of around 50 applications. This has involved comprehensive work on the entire code base, making updates and enhancements as needed.

In a specific project, I tackled the challenge of eliminating unused CSS elements from legacy applications. Given that these applications had evolved over time, accumulating changes, I employed a combination of techniques such as testing within web browsers and thorough search within the application codebase to pinpoint and subsequently remove redundant code.

Another noteworthy accomplishment was an architectural overhaul across multiple web apps. The task was to streamline administrator management, which had been scattered across different sections of the application code. To address this, I restructured the application architecture by initializing administrator data through an init method at application start. This organized approach facilitated smoother maintenance and provided a dynamic solution for handling multiple administrators, leveraging index-based access.

I also took on the challenge of developing a standalone application – the contact management system. This involved iterating through the init methods of various web applications to retrieve and display administrator information in a data table format. I utilized jQuery for interactive features and incorporated AJAX calls within the datatable function to fetch JSON-formatted data from another page. This project underwent multiple phases, with the second stage involving logic enhancements for column display. I transformed the logic to showcase application names instead of full URLs, necessitating recursive directory traversal and the identification of index pages.

Currently, I am deeply engrossed in the task of transitioning a jQuery-based UI to a Fomantic UI. This undertaking involves rewriting parts of the code to accommodate the Fomantic UI framework. This experience is enriching my understanding of data flow across different sections of the application code and enhancing my overall grasp of application functionality.

DATABASE PROJECT:

* Developed a covid 19 Vaccine management system.
* The main aim of the project was to administer vaccines to people based on their age.
* Keep track of what vaccines were administered and query the database for some statistics
* I used Microsoft SQL Server management studio(SSMS) and used SQL for querying the databse
* To start the project, I did
* I had created an ORM diagram in VS Code, and it had 10 main entities in this project some of the enties.

Some of the table were   
person , medical information, vaccination information, their location  
age table , what age group they belong to  
  
Where is a particular vaccination facility located

I used **Data Definition Language (DDL) Queries: like create table to define my entities and relationships between them**

I used **Data Manipulation Language statements to insert data into my db.**

**There were few main queries I used in this project:**

**1) Stored Procedures:**

• Allot\_v: The procedure is used to allot a vaccine administered to an individual based on age.

• Count\_Vaccine\_PostalCode : This Stored Procedure is maily used to find how many people (N) have taken what type of Vaccine (VaccineName) in which Facility at a particular postal code

**2) Triggers:**

• Trigger to update age in Age\_cal Table: This trigger is triggered on inserting a value into MedicalInformation Table

• Dose1Check on Vaccine Table: Making sure Dose 2 can be administered only after Dose1

1. **UDF**

• [stat] is a scalar valued user defined function to find out how many males/females have been infected/not infected

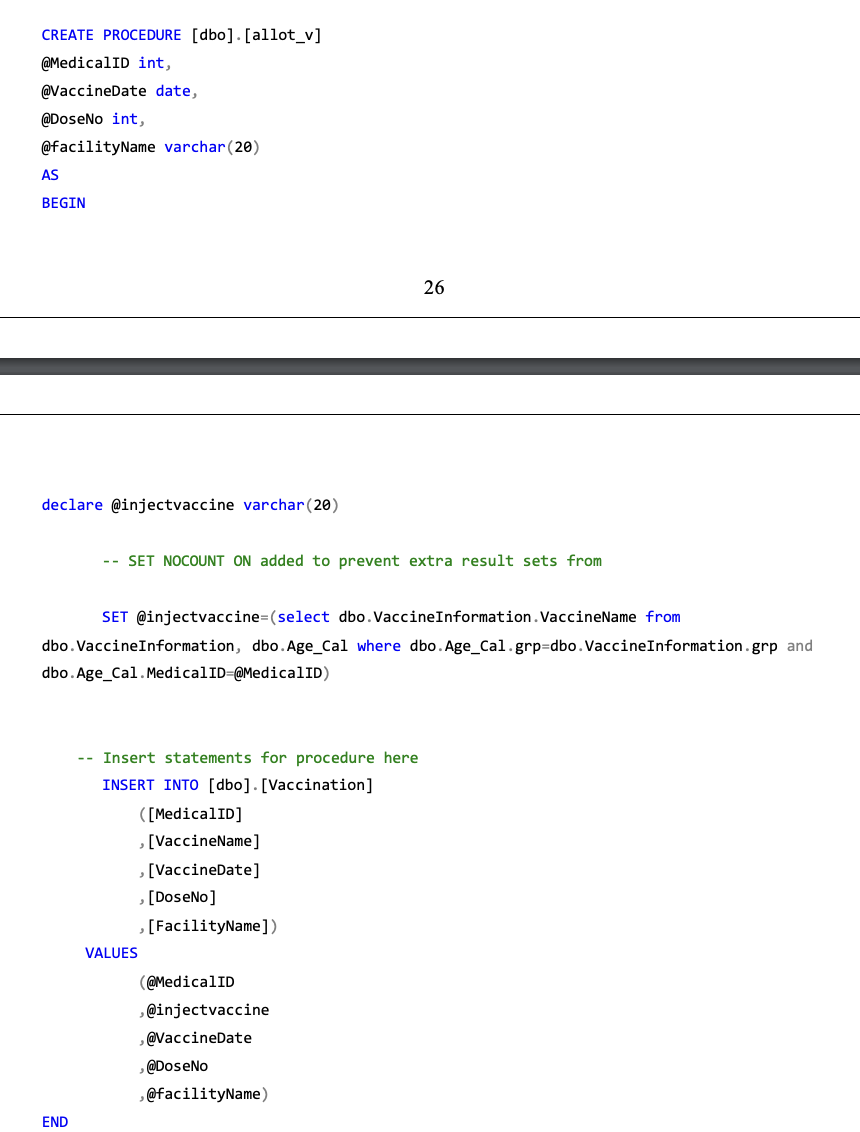
Questions answered in this project include the following :

How many People above a particular age has been vaccinated with 2 doses of Vaccine

• People who have been infected/not infected and have atleaset been given 1 or doses of vaccine • How many People have been administered what type of Vaccine in which Facility.

• The Total statistics of People who have been infected or not grouped by gender.

• People who have been infected with a particular variant in a particular city of a given state

Allot Vaccine:  


A screenshot of a computer

Description automatically generated

2. Count\_Vaccine\_PostalCode

This Stored Procedure is maily used to find how many people (N) have taken what type of Vaccine (VaccineName) in which Facility at a particular postal code :

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated A screenshot of a computer code

Description automatically generated

**Trigger** to update age in Age\_cal Table:

This trigger is triggered on inserting a value into MedicalInformation Table The MedicalInformation Table has DOB as one of its attributes. Using this attribute, the age of the person is calculated. After calculating , it is classified as Child/Teen/Adult and inserted into the Age\_Cal Table.(The values automatically inserted are MedicalID,Calculated age and the classification)

A screenshot of a computer code

Description automatically generated

User Defined Function:

A screenshot of a computer

Description automatically generated

CREATE FUNCTION function\_name ([parameter1 datatype, parameter2 datatype, ...])

RETURNS return\_datatype

AS

BEGIN

-- Function logic goes here

-- You can use SQL statements and calculations

RETURN result; -- Return the result

END;

A screenshot of a computer

Description automatically generated

In SQL, there are primarily four types of queries used to interact with a database:

1. **\*\*Data Query Language (DQL) Queries:\*\*** DQL queries are used to retrieve data from one or more tables in a database. The most common DQL query is the `SELECT` statement, which allows you to specify the columns and conditions to filter the rows you want to retrieve.

Example:

```sql

SELECT FirstName, LastName FROM Employees WHERE Department = 'Sales';

```

2. **\*\*Data Definition Language (DDL) Queries:\*\*** DDL queries are used to define, modify, and manage the structure of database objects such as tables, indexes, and constraints. Common DDL statements include `CREATE`, `ALTER`, and `DROP`.

Example:

```sql

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50)

);

```

3. **\*\*Data Manipulation Language (DML) Queries**:\*\* DML queries are used to manipulate data in the database. Common DML statements include `INSERT`, `UPDATE`, and `DELETE`. These statements allow you to add, modify, or remove records in database tables.

Example:

```sql

INSERT INTO Orders (OrderID, CustomerID, OrderDate)

VALUES (1, 101, '2023-09-27');

UPDATE Employees SET Salary = 60000 WHERE Department = 'HR';

DELETE FROM Products WHERE ProductID = 123;

```

4. **\*\*Transaction Control Language (TCL) Queries:\*\*** TCL queries are used to manage transactions within a database. Transactions are sequences of one or more SQL statements treated as a single unit of work. Common TCL statements include `COMMIT`, `ROLLBACK`, and `SAVEPOINT`.

Example:

```sql

BEGIN TRANSACTION;

-- Execute multiple DML statements here

COMMIT; -- Save changes

BEGIN TRANSACTION;

-- Execute DML statements here

ROLLBACK; -- Undo changes

```

These are the primary types of SQL queries used to interact with a relational database management system (RDBMS). Each type serves a specific purpose in managing and querying data.

A savepoint in SQL is a named point within a transaction to which you can later roll back. It allows you to create checkpoints within a transaction so that if an error occurs or if you want to partially undo the changes made in the transaction, you can roll back to the savepoint without affecting the entire transaction. Savepoints are a part of the

1. Scalar UDF (User-Defined Scalar Function):
   * A Scalar UDF is a function that returns a single value (scalar) based on the input parameters.
   * It can be used in SQL queries wherever an expression or single value is expected.
   * Scalar UDFs are commonly used for calculations, string manipulation, date/time operations, and custom business logic.
   * Example: A function that calculates the total price of items in a shopping cart.
2. Table-Valued UDF (User-Defined Table Function):
   * A Table-Valued UDF is a function that returns a result set in the form of a table.
   * It can be used in SQL queries as if it were a regular table or view.
   * Table-Valued UDFs are useful when you need to encapsulate complex queries or combine data from multiple tables into a single result set.
   * Example: A function that takes a customer ID as a parameter and returns a list of orders made by that customer.

WHY PWC:

Global Reputation: PwC is widely recognized and respected globally.

I read that PWC invests in training and developing its employees: Learning and Development. You'll have access to ongoing training, mentorship, and opportunities to gain new skills and certifications.

Innovation and Technology: PwC has been actively investing in technology and innovation. If you have an interest in cutting-edge technologies and their application in professional services, PwC may offer opportunities in this area.

Work Life balance: I was going through the website and 4th of July