Victor Feight

4/25/2022

CS 470 Final Reflection

https://www.youtube.com/watch?v=tHpLLAT0L-M

[[GitHub was used to repo the necessary files for containerization of a MEAN stack. You may also place the dockerfile and Docker Compose file in GitHub, although this is not required. Due to the nature of cloud services and the security features that create unique IDs, it would be both redundant and a security risk to post any of the AWS code to GitHub. This is why the only artifact for your GitHub portfolio for CS 470 is the Final Reflection, which contains a link to your presentation video.]]

You have finished developing a full stack web application in the cloud. In the final iteration, you completed the API and tested the project, then you documented the process using a presentation.

Experiences and Strengths: Explain how this course will help you in reaching your professional goals.

What skills have you learned, developed, or mastered in this course to help you become a more marketable candidate in your career field?

Throughout the course, we worked with **Docker containers** and applications, providing consistent and isolated environments whether deployed on a home machine or in the cloud. In addition, being free of environmental limitations, such containers are portable (cross-platform) and *scalable*, and easy to roll back or test in a CI/CD integration environment.

- Using Docker compose I was able to create containers for a full-stack front-end angular container, a NodeJs backend REST API container, and a MongoDB database container instance, and docker allowed these containers to communicate within an isolated network as a single logical unit allowing secure internal communication, separate from the host network.
- In using this approach, I'm capable of migrating any full-stack application to a cloud provider – application code could be stored on Amazon S3, with any object changes resulting in a trigger to a CI/CD pipeline – a docker compose file can live along side application source code and deploy through our CI/CD pipeline to Amazon ECS (Elastic Container Service).

In addition, I've learned to apply the serverless concept to full-stack applications in the AWS ecosystem – utilizing Amazon BaaS solution, **API Gateway**, to define routes and endpoints for a RESTful APIs; Amazon web APIs can route HTTP requests securely to **Lambda functions**, creating a *lambda proxy* integration. The API consisted of resources, and each API resource exposed one or more API methods with unique HTTP verbs such as GET to handle request.

- I'm now able to author an exports.handler function in NodeJS, and Amazon API gateway will invoke the function synchronously with an event that contains a *JSON representation* of the HTTP request. Integrating AWS API Gateway with Lambda involved giving methods permission to execute my lambda functions. To give the functions permissions and prevent unauthorized access, I created **AWS Identity and Access Management (IAM)** policy that I attached to an *IAM role*. An IAM Role manages who has access to the AWS resources.
- Such roles can be used to grant permission to services such as Lambda functions, DynamoDB tables, or Cloudwatch logging, ensuring The Principle of Least Privelege is adhered to. I attached this policy to actions against my Questions and Answers DynamoDB tables, thus granting my lambda limited permissions, and ensuring the secured lambdas could work on both tables for all CRUD functions.

Describe your strengths as a software developer.

My strengths as a developer include fundamental knowledge of software design principles, development, and testing. I have a broad educational background with excellent analytical, technical, and programming skills. In addition, I demonstrate determination, empathy, team-spirit, and resiliance in my endeavors and interpersonal relationships.

Identify the types of roles you are prepared to assume in a new job.

Some roles I am currently interested in pursuing:

-Entry-level Software Developer or Full-Stack Web Developer.

-Cloud Support Associate or Solutions Architect

-AWS Devops or Cloud Engineer

Planning for Growth: Synthesize the knowledge you have gathered about cloud services.

There are several models to consider when migrating a full-stack application to the cloud.

- One would be a **"lift and shift" model** which while able to migrate a fullstack app quickly does not take advantage of AWS Cloud efficiencies like CI/CD, logging, containerization. etc.
- **Re-platforming** involves making appropriate changes to the API and middleware in a *standard lift-and-shift* so as to better take advantage of cloud architecture and avoid post-migration work.
- **Re-factoring** involves completely re-engineering your fullstack solution into a cloud-native set of microservices from scratch.

With **serverless model**, the purpose **AWS Amazon Gateway**, is to map incoming API calls to a service that can handle the API call. REST is one protocol it supports. AWS Gateway is what triggers the lamba functions, telling them to start running. API Gateway can also act as a "security barrier", meaning only the outside world can access the APIs we deploy, and the API Gateway is the only service with permissions to call our Lambdas.

In a **Amazon Virtual Private cloud (VPC)** I can provision *logically isolated virtual cloud network* where I can launch **EC2 instances** – a security group can be defined which enables restricted access to instances and *secure monitoring* of connections by enforcing rules on *inbound and outbound* connections. The VPC allows me to a traditional network but with automation and scale expected of AWS services – I'm able to provision **private subnets** within a VPC, giving particular access rules to a certain set of resources.

- I understand that VPC provides complete control over my virtual networking environment, with selection of your *own IP address range*, creation of subnets, and configuraiton of **route tables** and **network gateways**—A VPC must be assigned a range of *private IP addresses*, and then spans all of the **availability zones** in a region; one or more subnets can be added in *each availability zone*. A subnet is merely a *partition* of a VPC's IP address range (Out of a VPC's 1024 IP addresses, each of 4 subnet would have 251 IP addresses).
- In addition, a VPC can span *multiple Availability Zones in a region*; you can add subnets into different Azs, and launch instances into the subnets to *improve fault tolerance*. As a VPC resides in an AWS region, a subnet must reside *within a Single Availability zone*. =
- This is convenient for pushing *web-servers* on a public-facing subnet while the backend systems such as databases or application servers can reside in private-facing subnets with no internet access.
    - Network access control lists and security groups can be used for inbound and outbound filtering at the subnet and instance level. Data stored in S3 can have restricted access such that it's only accessible from instances inside your VPC.
    - Note that a **NAT gateway** can be placed on the public internet, to enable instances in a private subnet to connect to the internet, but prevent the internet from making connections with the instances – NAT (network address translation) is what allows translation of addresses between private subnets to public subnets and should be associated with public subnet an **elastic IP**, and the route table should be updated to *point to the NAT gateway*.

I understand that **EBS, Elastic Block service**, behaves like an unformatted external device and can be attached to a single EC2 instance – EBS snapshots can be stored in Amazon S3. By using Amazon Cloudwatch with AWS Lambda, you can automate EBS volume changes.

I understand that **Amazon Machine Images (AMI)** are virtual images with a read-only filesystem that's compressed, encrypted and stored in S3. I can utilize (AMI) to *create, launch, and reuse multiple EC2 instances* conveniently with the same configuration. I understand that AWS provides a file storage, **EFS** that multiple EC2 instances can use as a common data source.

Identify various ways that microservices or serverless may be used to produce efficiencies of management and scale in your web application in the future. Consider the following:

How would you handle scale and error handling?

Deploying your application on EC2 instances *across availability zones* and *managing loads* between them using **Elastic Load balancer** would allow you to distribute traffic to appropriate Availability Zones during outages, *reducing customer downtime*.

- Each instance type—general purpose for web servers, compute optimized , memory optimized (Big Data), storage optimized, accelerated computing (deep learning)—contains *one or more instance sizes*, allowing you to scale your resources to the requirements of your target workload. For scaling, Amazon EC2 instances can be changed to a larger instance type (such as m4.large) to scale your resources to the requirements of the target workload. For handling Backups, there is EBS snapshots with amazon lifecycle manager (DLM).
- Troubleshooting a VPC involves reviewing VPC settings, checking routing tables to ensure the internet gateway is attached, and that your VPC is set to allow outside traffic, if you didn't choose Amazon's default VPC configuration. An internet gateway is horizontally scaled, and highly available by default.

In regards to **Single-Table design**, it's my opinion is that it is what **DynamoDB** was designed for and should be used in most use-cases. You don't technically lose an ability to get multiple, heterogenous items from the database, because by pre-joining your data into item collections, I was able to read multiple items using the same partition key which is a primary benefit of Single-Table design with DynamoDB. In addition, you don't have to manage and configure multiple tables -- which is less cost.

AWS resources can be launched as **Auto-scaling groups** within a VPC. Security groups can be chained to handle *securing a full-stack app in a VPC at scale* – we could have web-tier security group (allowing inbound web traffic over HTTPS on port 443), application security group (allowing only traffic from the web-tier server on HTTP port 80), and database security group would only allow traffic on the TCP port 3306 and nowhere else, allowing strict control of traffic – and *forbidding direct database access from intruders*.

How would you predict the cost?

Different **EC2 instance** types offer differing on-demand linux and windows pricing, for example c5.large instance type is 0.085 USD per hour in N. Virginia Availability Zone.

EBS offers different volumes at various price points – ranging from highly cost effective **$/gB** volumes to *high performance* volumes with high IOPS and *high throughput* design for critical workloads, enabling you to optimize costs and invest in a *precise level of storage*.

- You're only billed for the *stored incremental changes* for **EBS snapshots**, saving on storage costs. **Elastic volumes** allow you to adapt storage volumes as the needs of your app changes – you can scale up to Petabytes of data in a few clicks.
- AWS IAM enables *access control* to EBS volumes. You can *start or stop an instance* if it has an EBS volume as its root device. Stopped instances *don't incur CPU usage nor data* transfer charges, but **Elastic IP addresses** or EBS volumes attached do. *Re-starting* a stopped instance charges an additional minimum of one minute per usage, but an EBS-backed instance must be stopped before you can change the instance type.

What is more cost predictable, containers or serverless?

EC2 allows instance management, these instances can be spun up or down whenever you want, meaning no need to pay for long-term hardware commitment, with sizes and types of instances depending on use case. For web app purposes, An EC2 would require a load balances, with Auto-scaling group for instances.

ECS model is more in line with Docker ecosystem – treat a pool of EC2 instances as abstract resources deployed in clusters, and these tasks can be long living or one off – operating on Docker containers allows you to scale up or down depending on any key metric. Images are deployed into tasks, operating within the context of a cluster.

To work out the total cost of ownership, in serverless – we can predict prices based on costs for Lambdas, API gateway, or Cognitos.

- For a traditional SaaS application, you'll have to work out how much server resources are necessary for the solution. Assuming a SaaS app with ~100,000 users, and 40,000 sessions/day, equates to ~19 million lambda invocations or about $60 dollars at 3.8 million GB.s / month.
- For an EC2, we might go for an m5.large – with 2 vCPU and 8GB ram, we have ~20million GB.s/month resulting in around $70.

Time to build is a consideration – with serverless, you have redundancy, autoscaling, load-balancing built in which are not available in a traditional system – *autoscaling and failover* must be implemented, as well as setting up a CI/CD process. Added tasks especially for Devops lead to added costs for traditional.

Cost to support going forwards should be considered – with serverless, nothing needs to be changed or updated, whereas with traditional you must consider OS patches for the server, feature deployment for monolithic apps (in EC2 if something goes down it might take down the whole system – costs increase spending time maintaining these traditional systems.

If you've got a SaaS solution for 100,000 users, go with serverless – cost savings might be minimal in exchange for rapid ability to act, reduce development time spent on tasks not growing the business, and even smaller businesses make less sense.

Explain several pros and cons that would be deciding factors in plans for expansion.

As AWS contains **multiple geographic regions**, each with *mutiple availability zones*, allowing *rapid expansion* to virtually any region or country. Storing cloud compute resources in multiple AWS regions, you're provided multiple, **redundant** availability zones which are *fully isolated*, with discrete data centers ensuring *reliability*. When a full-stack app is partitioned *across multiple availability zones*, you're protected from natural disasters and sudden outages, as if one availability zone goes down, *traffic is routed* to other available Azs, leading to less down time and higher customer satisfaction.

- Redundancy and reliability therefore are two huge benefits – from regions, networking links, and PoP edge locations, to load balancers, routers and firmware.
- Lambda provides **automatic auto-scaling** and **pay per use** – for example, peak usages and dips in usage of the day will charge you according to usage, whereas with EC2 you *play a flat rate* – it might work out that you're paying for resources that are underutilized during the day. In addition, one must be aware that if you go over resource limits in EC2 at 100% throttled peak usage, customers will get a *failed service*. Lowering the threshold would result in access over peak usages, but with less utlization of resources per month, resulting in much less used CPU utilization per month.
- Lambda has **redudancy** built in, if one doesn't run, another lambda will fire up, whereas with EC2 you would might need a replicated server (with only ½ utilized). With 3 production servers and 1 backup (and one production goes down), traffic can be split across redundant and 2 other production servers, making added cost 1/3 but still bringing down CPU utilization.

For a company with 1 million sessions per day, it's best to look at the numbers and calculate utilization per month, and how the costs compare between traditional and serverless architectures. You can always build features in serverless and migrate to traditional architecture if necessary, or vice versa.

Serverless gives the flexibility and speed of development, so it's best to stick with it until scaling prices become an issue before thinking about shifting back to traditional.

What roles do elasticity and pay-for-service play in decision making for planned future growth?

Companies can take advantage of the seemingly infinite scalability of the cloud when provisioning resources, scaling up or down depending on business needs. Instance spin-up time is rapid allowing many instances to spin up in moments, and an extensive datacenter footprint allows customers to benefit from cloud economes of scale, reducing total cost of ownership (TCO) over their IT infrastructure. This answer will be updated to reflect my knowledge as time passes.