



-Hi Everyone, my name is Gino Murin and I'm here today to talk about the benefits of migrating from a traditional full stack environment to a cloud based environment using AWS microservices.

Overview

- Introduction
- **Cloud Development: Advantages and Intricacies**

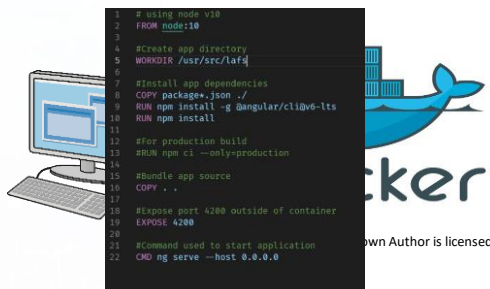


-Today we're going to talk about containerization and orchestration (using tools such as docker). We'll also talk about serverless architecture in the cloud,(including S3 storage, AWS lambda functions, and DynamoDB), as well as their associated costs and cost saving propositions posed to an organization considering using them. And finally, we'll discuss some of the security best practices surrounding securing a cloud application on AWS.



Containerization

- Containers provide a versatile solution to deploying in the cloud
- Tools required for containerization



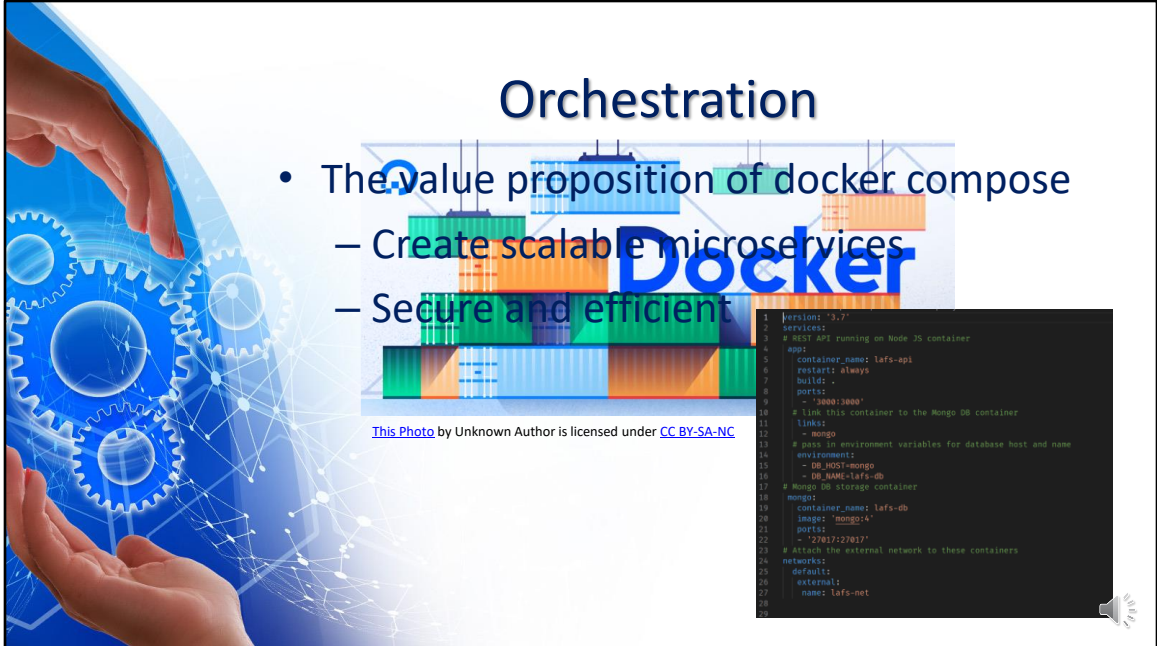
```
1 # Using node via
2 FROM node:10
3
4 # Create app directory
5 WORKDIR /usr/src/app
6
7 # Install app dependencies
8 COPY package*.json ./
9 RUN npm install --g @angular/cli&npm-lts
10 RUN npm install
11
12 # For production build
13 RUN npm ci --only=production
14
15 # Bundle app source
16 COPY . .
17
18 # Expose port 4200 outside of container
19 EXPOSE 4200
20
21 # Command used to start application
22 CMD ng serve --host 0.0.0.0
```

own Author is licensed

-The first thing on the agenda today is containerization and orchestration. Containers present a huge value proposition to developers by saving us time and cutting down on complexity while simultaneously allowing us to build more modular and secure applications. With containers we're able to develop our applications within a standardized environment that allows us essentially to write our code once and run it anywhere.

-One of the benefits of containers is their simplicity. In terms of tool required to run a container, we only need a computer with a copy of our source code as well as a container software running on that machine such as docker.

-Once we've got that taken care of, we can go to our project directory and create a dockerfile. You could think of this file as a recipe for the docker image that you want to create for your application. The main steps here are defining our image, creating a working directory, installing our dependencies, copying our application files, and defining commands to run our application. From here, we use the docker build command to build the container, and docker run to run it on our host machine. That's all we need to do to get our container up and running.



Orchestration

- The value proposition of docker compose
 - Create scalable microservices
 - Secure and efficient

[This Photo](#) by Unknown Author is licensed under [CC BY-SA-NC](#)

```
1 version: '3.3'
2 services:
3   # REST API running on Node JS container
4   api:
5     container_name: lafs-api
6     restart: always
7     build: .
8     ports:
9       - '2000:2000'
10    # Link this container to the Mongo DB container
11    links:
12      - mongo
13    # pass in environment variables for database host and name
14    environment:
15      - DB_HOST=mongo
16      - DB_NAME=lafs-db
17    # Mongo DB storage container
18    mongo:
19      container_name: lafs-db
20      image: 'mongo:4'
21      ports:
22        - '27017:27017'
23    # Attach the external network to these containers
24    networks:
25      default:
26        external:
27          name: lafs-net
28
```

-Container orchestration takes what we've just discussed a step further by allowing us to define how individual containers interact with one another. Docker's orchestration tool is known as docker compose. The value proposition in using docker compose is that we can provide for separation of concern between the different microservices found within our application. In other words, we can handle our frontend logic, backend logic, and database within different containers that work together as one as opposed to creating a single monolithic container. This allows us to create secure and efficient applications.

-Similar to the docker file, we use a docker-compose.yml file within each container to define our individual containers and how they interact with one another

The Serverless Cloud

Serverless

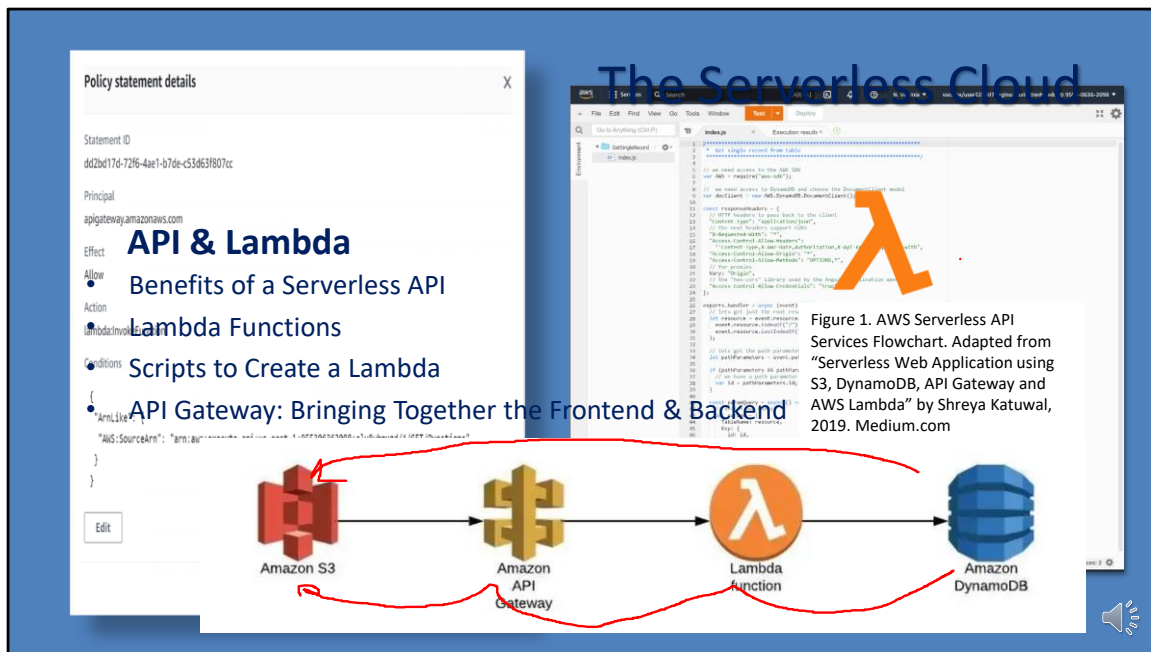
- What is “serverless”?
- S3 Storage
- S3 versus Local Storage
 - Secure by default
 - Options for fault tolerance and backup



-Now lets talk a little about serverless cloud architecture. The term serverless refers to the principle of using compute resources on an as needed basis. The biggest advantage to be had here is the fact that serverless architecture allows us to pay only for what we use whereas with a traditional server, we'd either have to pay a server host or incur the costs associated with running our own server. AWS provides several serverless resources, but we'll discuss a few of the most popular ones today as well as how they can be combined to create a scalable and efficient web application.

-S3 (Simple Storage Service) is an AWS service that allows us to store data on the cloud including large files such as videos and even static website content

-S3 has the advantage of being secure by default and providing convenient tools for fault tolerance and backup. While we could do this on premise using local storage, S3 is much cheaper and easier to implement in the long run.



-As I previously mentioned, the benefits of a serverless API is that they allows us to create scalable, efficient, and cost effective web applications. At the foundation of this idea, we have the Lambda function. Lambdas are bits of logic that are run on cloud servers as they are called. For instance, we may send a request to a lambda function to request access to data stored on our database.

-We can create a lambda function in our AWS console by selecting our runtime (such as node.js) and writing the script. For example, we may write a GET request to get server data. We must also create policy documents that allow for role based access control to our lambda, which you can see here.

-The following diagram shows the logical flow of a serverless application from frontend to backend. We host our frontend on S3 which allows the browser to call API Gateway. From here our request is routed to the proper corresponding Lamba functions, which can performs some action such as accessing our database and return subsequently the result.

The Serverless Cloud

Database

- DynamoDB
- Differences from MongoDB
 - Single table design improves performance
- Necessary scripts
- Queries
 - findAll, deleteOne, upsert, and more

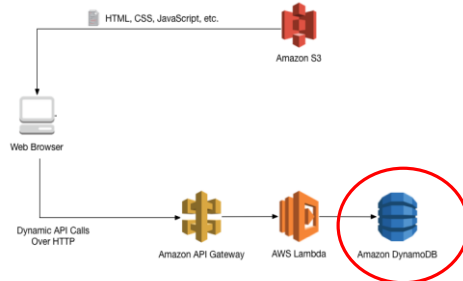


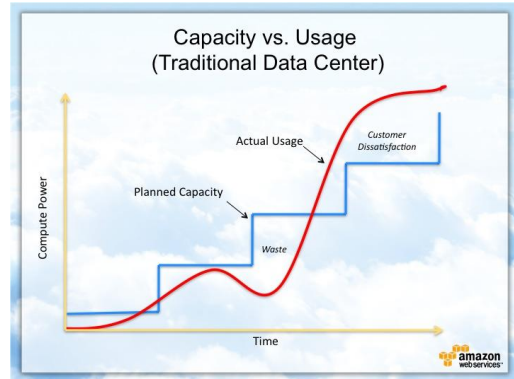
Figure 2. Serverless API Flowchart. Adapted from "Tutorial for building a web application with Amazon S3, Lambda, DynamoDB and API gateway" by Connor Leech, 2017, Medium.com.

-The final piece to this puzzle is DynamoDB, a NoSQL database service created by AWS. For those of you familiar with MongoDB, it's similar in structure, however it prioritizes the use of a single table design. This particular design can give us significant speed and compute advantages over MongoDB as it eliminates the need to do costly joins.

-In order to use DynamoDB, we import our data into the database and create a policy document within our Lambda function that defines our tables as resources. This is similar to the procedure we used to integrate Lambda with API Gateway. Once the Lambda has access to the table, we can perform any of the typical queries that you'd find in REST architecture. For instance, some of the queries we performed are findAll, findOne, deleteOne, and upsert, which searches for a particular record, inserts into the database if not found, and updates the record if it is found.

Cloud-Based Development Principles

- Pay-for-use-model
 - Pay only for what you use
- Elasticity
 - Automatically scales your architecture as needed



-As I alluded to earlier, one of the greatest benefits to be had in serverless computing comes in the form of paying only for what you use or the pay-for-use-model. If your serverless application is only being used minimally, you will pay a minimal amount. In fact, many services are covered under the AWS 'free-tier' and only incur costs once they reach a certain usage threshold. Naturally, as your application grows, the costs associated with usage will scale in a linear fashion.

- This brings us to another key benefit of serverless development which is elasticity. Serverless architecture gives us the option to automatically scale as usage increases or decreases, meaning that AWS will allocate additional resources to our applications as demand increases or scale them down as demand decreases. This is all done without any need for user intervention.

Securing Your Cloud App

Access

- How can we prevent unauthorized access?
 - IAM (Identity and access management)

Policies

- Roles vs Policies
- Custom policies

Bucket policy

The bucket policy, written in JSON, provides access to the objects

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::snhu-2024837/*"
    }
  ]
}
```

API Security

- Securing Lambda and API Gateway
- Securing Lambda and Database
- S3 Bucket



-As I've mentioned before, AWS does a pretty good job of making it's services secure by default. For example in our S3 storage, the default policies block all outside access, and it's up to us as users to decide who we allow in instead of the other way around.

-This begs the question though, how can we prevent unauthorized access? AWS has an answer for this as well with IAM or Identity and Access Management. IAM allows us to define roles for every class of user that accesses our services as well as policies that we can use to grant access to particular users or groups of users based on role.

-These policies come in the form of policy documents. Within these documents we may choose to grant access to particular resources to particular roles. And again, roles are used to provide access control over users based on what class of user they are.

-In securing an API, we can apply the principles of roles and policies on each individual service. In doing so, we can also define how each service may interact with other services. For instance, we create policies to allow our API gateway to access our Lambda functions as well as policies for our Lambdas to access our database.



CONCLUSION

- Containerization provides a versatile solution to developing applications.
- Serverless architecture provides a cost effective and convenient alternative to traditional/cloud servers.
- IAM role based access control provides fine grained control over our services.

Thank you for your time.



In summary:

- We've discussed containerization and its versatility within the context of application development
- We also talked about serverless architecture and its unique properties that present a cost effective and convenient alternative to both traditional and cloud servers
- Finally, we talked about securing our AWS services using Identity Access Management

I hope this presentation has left you with a better understanding of cloud services and motivates you to use them in future projects. Thanks for your time!



REFERENCES:

- Katuwal, S. (2019, December 10). *Serverless web application using S3, dynamodb, API gateway and Aws Lambda*. Medium. Retrieved December 11, 2022, from <https://medium.com/@shreyakatuwal/serverless-web-application-using-s3-dynamodb-api-gateway-and-aws-lambda-32a10b1a9d5e>
- Leech, C. (2020, June 5). *Tutorial for building a web application with Amazon S3, Lambda, DynamoDB and API gateway*. Medium. Retrieved December 11, 2022, from <https://medium.com/employbl/tutorial-for-building-a-web-application-with-amazon-s3-lambda-dynamodb-and-api-gateway-6d3ddf77f15a>