Building AWSome Serverless Solutions

Cod(H)erSpace workshop handbook

thoughtworks

Table of contents

Serverless Computing

Serverless isn't being devoid of servers. In fact, the servers exist but are managed by the cloud provider

What is serverless computing?

Serverless computing is a way of providing services on a need basis. These technologies feature auto-scaling, built-in high availability, and a pay-for-use billing model to increase agility and optimized costs. The providers offer on-demand delivery of compute power, database storage, applications, and other IT resources which can be accessed almost instantly. You can provision exactly the right type and size of computing resources you need without worrying about infrastructure management tasks like capacity provisioning and patching. Hence, you can focus on writing code that serves your customers.

Amazon Web Services is one such provider who offers technologies for running code, managing data, and integrating applications, all without managing servers. In the workshop, we use services like Lambda (as the backbone of application), DynamoDB (for storing the data), API Gateway (for creating and managing APIs), S3 (for hosting the application) and CloudFront (for distribution of content across network)

Serverless vs Serverful

To first understand the difference, let's use a car rental analogy. Serverful computing is similar to a regular car rental. You pick up a car, drive it, fill the fuel, maintain it in a good condition for as long as you are using it and pay for entire days of use. Serverless is more like an Uber setup. You hitch a ride and get out of the vehicle when you have reached your destination. You only pay how long you traveled. So, in order to call something a serverless service, it would have to follow four principles:

- Servers need to be hidden
- Pay only for what you use
- Have high availability out of box
- Be able to scale on demand

Serverless Computing	Serverful Computing
Automatically scale server instances up and down to handle load.	Does not scale up or down. It has a capacity that cannot be exceeded, and its resources stay available even if they're not being used
Require no maintenance. The cloud provider handles all these details of managing the underlying hardware. You just write and deploy code using tools provided by the cloud vendor.	Requires maintenance. If you run a server, you might have to monitor it, install software, install patches, tune it, and other operations. You have to figure out how to deploy your code to it.
Billed per function invocation. When you deploy code to a serverless backend, you will be charged for the resource's uses (invocation, memory, bandwidth). If you use nothing, you are charged nothing.	Has some ongoing cost associated with it. Typically costs are paid on an hourly, daily, or monthly basis just to keep the server up and running, even if it's not being used.
Event-driven by nature. You deploy code that runs in response to events that occur in the system. These can be things like database triggers that respond to changes, or HTTP requests that serve an API. Code does not run outside of the context of handling some event, and it is often constrained by some time limits.	Allows you to deploy services that run on an ongoing basis. You can typically login and run whatever programs you want, whenever you want, for as long as you want.

Table 1: Serverless vs Serverful

Advantages of serverless computing

- Move from idea to market faster: Eliminate operational overhead. Teams can release quickly, get feedback, and iterate to get to market faster.
- Lower your costs - With a pay-for-value billing model, resource utilization is automatically optimized and you never pay for over-provisioning.
- Adapt at scale - With technologies that automatically scale from zero to peak demands, you can adapt to customer needs faster than ever.
- Build better applications, easier - Serverless applications have built-in service integrations, so you can focus on building applications instead of configuring it.

Serverless services by AWS

Among the number of serverless services offered by AWS, we will be going through DynamoDB, Lambda, API Gateway, S3 and CloudFront.

DynamoDB

DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale. DynamoDB offers built-in security, continuous backups, automated multi-region replication, in-memory caching, and data export tools. It is a fast and flexible database service for all applications that need consistent, single-digit-millisecond latency at any scale. The flexible data model and reliable performance make DynamoDB a great fit for mobile, web, gaming, advertising technology, Internet of Things, and other high-performance applications.
In an Amazon DynamoDB table, the *primary key* that uniquely identifies each item in the table can be composed not only of a *partition key*, but also of a *sort key*.
Well-designed sort keys have two key benefits:

- They gather related information together in one place
- Tables can be queried efficiently.

Primary Key is like a combination of the *Partition Key* (PK) and the *Sort Key* (SK) in any DB. PK is mandatory. One table can have multiple Partitions, and each partition would have the same PK. To Uniquely identify records in a partition we use the key called SK.
Global Secondary Index (GSI) is a global index, where all the records are indexed based on completely other attribute(s) different from PK and SK. GSI can be added to a table even after creation.
Local Secondary Index (LSI) is a local index, local to a partition. It is used to index items within a partition. LSI cannot be added after creating a table
DynamoDB offers two types of models for capacity provisioning. It could be
Provisioned capacity - reserving some read and write capacity units all the time. Provisioned capacity is useful when we for sure know the expected DB usage capacity. We will be billed even if we use it or not.
On demand capacity - pay per call basis. It is useful when an application has less load or nondeterministic loads.

Pricing

- On Demand model - 1.25$ per million write operations and 0.25$ per million read operations
- Provisioned capacity - 0.00013$ per read capacity units/hour and 0.00065$ per write capacity units/hour

Lambda

Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. You can trigger Lambda from over 200 AWS services and software as a service (SaaS) applications, and only pay for what you use.
In case of servers, you will have to make assumptions for the CPU and RAM requirements. It will keep running continuously which is going to produce a long bill. You would also have to take care of scaling which is nothing but adding or removing servers as and when load changes. Lambda provides *Function as A Service.* It allows you to run code without thinking about servers. We need to pay only for the compute time that is consumed. There is no charge when code is not running. With Lambda, you can run code virtually for any type of application or backend service, all with zero administration.
Lambda runs code without provisioning or managing infrastructure. Simply write and upload code as a .zip file or container image. It automatically responds to code execution requests at any scale, from a dozen events per day to hundreds of thousands of events per second. Lambda saves costs by paying only for the compute time you use - by per-millisecond - instead of provisioning infrastructure upfront for peak capacity. Lambda optimizes code execution time and performance with the right function memory size. It can also respond to high demand in double-digit milliseconds with Provisioned Concurrency.

Pricing

- Total cost = No. of requests * Time taken to execute
- The AWS Lambda free tier includes one million free requests per month and 400,000 GB-seconds of compute time per month
- For non-free tier, the user is billed based on the architecture (x86/arm) and the memory size (128/256/512/...)

- Asia Pacific (Mumbai) region pricing
    - x86 architecture charges $0.0000166667 for every GB-second and $0.20 per 1M requests
    - Arm architecture charges $0.0000133334 for every GB-second and $0.20 per 1M requests

API Gateway

API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. APIs act as the front door for applications to access data, business logic, or functionality from your backend services. API Gateway supports containerized and serverless workloads, and web applications. It supports *integration* with services like Lambda, Kinesis, EventBridge, other HTTP services etc. In order for the APIs to invoke any of these services, the integration must be configured while creating the endpoints. We can add different stages also for a gateway. The endpoints can then be managed in these different stages like dev, stage, prod etc. If not needed, you always have the option of *$default* stage.
Using API Gateway, you can create RESTful APIs and WebSocket APIs. RESTful APIs are the usual architectural style that uses HTTP requests to access and use data. API Gateway provides 3 options for creating a RESTful API.

- HTTP APIs - low latency, supports only Lambda and HTTP services, cheaper
- REST APIs - advanced model which supports integration with larger set of services, options to cache the responses, costly
- Private REST APIs - another version of REST APIs which can only be accessed within a Virtual Private Cloud (VPC)

WebSocket APIs are different in the sense that they enable two way communication. Clients can send messages to servers and servers can send back responses or any other messages to clients. It comes pretty useful while developing chat applications, social feeds, collaborative editing/coding applications etc.

Pricing

API Gateway has no minimum fees or startup costs. You pay for the API calls you

receive and the amount of data transferred out and, with the API Gateway tiered pricing model, you can reduce your cost as your API usage scales.

- Free tier includes 1 million API calls for REST API calls, 1 million messages and 750,000 connection minutes for Websocket.
- HTTP APIs (per month)
    - 1.05$ for 1st 300 million requests
    - 0.95$ for above 300 million requests
- REST APIs (per month)
    - 3.50$ for 1st 333 million
    - 2.80$ for next 667 million
    - 2.38$ for next 19 billion
    - 1.51$ for above 20 billion

Simple Storage Service (S3)

Simple Storage Service (Amazon S3) is an object storage service offering industry leading scalability, data availability, security, and performance. Customers of all sizes and industries can store and protect any amount of data for virtually any use case, such as data lakes, cloud-native applications, and mobile apps. With cost-effective storage classes and easy-to-use management features, you can optimize costs, organize data, and configure fine-tuned access controls to meet specific business, organizational, and compliance requirements.
You can get started with Amazon S3 by working with buckets and objects. A *bucket* is a container for objects. An *object* is a file and any metadata that describes that file. To store an object in Amazon S3, you create a bucket and then upload the object to the bucket. When the object is in the bucket, you can open it, download it, and move it. When you no longer need an object or a bucket, you can clean up your resources. S3 doesn't support folder structure even though the console gives us an experience of folder structure.
S3 scales storage resources to meet fluctuating needs with 99.999999999% of data durability. It can store data across Amazon S3 storage classes to reduce costs without upfront investment or hardware refresh cycles. Protect your data with unmatched security, compliance, and audit capabilities and enables you to easily manage data at any scale with robust access controls, flexible replication tools, and

organization-wide visibility. For making your s3 secure, you can either use IAM policies or resource based policy. In resource based policies one will use bucket policies to make s3 secure.
S3 provides different ways to store your data. The major classifications are:

- S3 Standard - General purpose storage for any type of data, typically used for frequently accessed data
- S3 Intelligent - Tiering - Automatic cost savings for data with unknown or changing access patterns
- S3 - Infrequent Access - For long lived but infrequently accessed data that needs instant access
- S3 Glacier - For long-lived archive data accessed once a quarter

Pricing

- For Standard type (per month)
    - 0.025$ per GB for first 50TB
    - 0.024$ per GB for next 450TB
    - 0.023$ per GB for above 500TB
- Infrequent access
    - 0.0138$ per GB for long lived data and 0.011$ per GB for re-creatable data
- Glaciers ranging from 0.005-0.002 per GB

CloudFront

CloudFront is a web service that speeds up distribution of your static and dynamic web content, such as .html, .css, .js, and multimedia files, to your users. CloudFront delivers your content through a worldwide network of data centers called edge locations. When a user requests content that you're serving with CloudFront, the request is routed to the edge location that provides the lowest latency (time delay), so that content is delivered with the best possible performance.
If the content is already in the edge location with the lowest latency, CloudFront delivers it immediately. If the content is not in that edge location, CloudFront retrieves it from an origin that you've defined—such as an Amazon S3 bucket, a MediaPackage

channel, or an HTTP server (for example, a web server) that you have identified as the source for the definitive version of your content.
CloudFront speeds up the distribution of your content by routing each user request through the AWS backbone network to the edge location that can best serve your content. Typically, this is a CloudFront edge server that provides the fastest delivery to the viewer. Using the AWS network dramatically reduces the number of networks that your users' requests must pass through, which improves performance. Users get lower latency—the time it takes to load the first byte of the file—and higher data transfer rates.
CloudFront reduces latency by delivering data through 310+ globally distributed Points of Presence (PoPs) with automated network mapping and intelligent routing. It improves security with traffic encryption and access controls, and uses AWS Shield Standard to defend against DDoS attacks at no additional charge. CloudFront cut costs with consolidated requests, customizable pricing options, and zero fees for data transfer out from AWS origins. Also, it customize the code you run at the AWS content delivery network (CDN) edge using serverless compute features to balance cost, performance, and security.

Pricing

- Free tier
    - 1TB of data transfer out
    - 10,000,000 HTTP or HTTPS requests
- Pricing for India
    - 0.109$ for first 10TB
    - 0.085$ for next 40TB
    - 0.082$ for the next 100TB ...

Deploying a full stack application in AWS

Detailed steps for deploying a serverless application

Application architecture

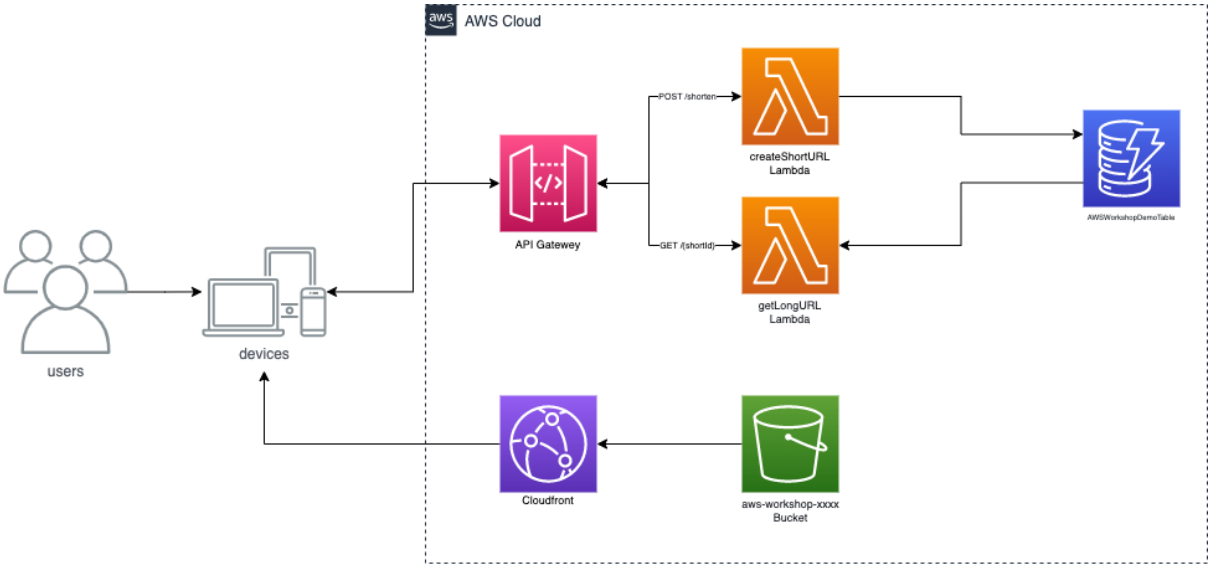

Figure 1: Application Architecture for URL Shortner

We would be deploying a small *URL Shortner* application in AWS. The API Gateway acts as the controller in the backend. All the requests are routed to Lambda which forms the service layer. We will be having two major functionalities.

- When a long url is passed as *urlToShorten* data in body, one of the endpoints would trigger a lambda. Lambda would create a unique ID, store it in DB and return the same to the user.
- When a short ID is passed as path parameter name *shortId*, another endpoint would trigger another lambda which would query the table and retrieve the original URL. This is then returned to the user.

The data is read and written to DynamoDB which acts as the repository layer. In the case of the front end, we would be using S3 for website hosting and it would be distributed via CloudFront to the end users.

Setting up local environment

- Installing Nodejs (version *14.x*). (Refer - Downloading and installing Node.js and npm | npm Docs (npmjs.com))
- The code for back end and front end can be found here . Make sure you have *cloned the repo* (or download as zip in case you don't have git in your system).
- Once the above steps are done, open the UI and backend folders *as separate projects* in any IDE of your choice and run *npm install in both the projects*.
- *Postman* - for testing the APIs we create during the deployment. You can choose any way you prefer (postman, curl commands etc.) to test.
- Last but not the least, an AWS account (if you don't have an AWS account, create one within free tier)

Note: Throughout the demo our AWS region would be ap-south-1 (Asia Pacific - Mumbai), which you could toggle from the drop down available at top right corner of AWS console.

Creating a table in DynamoDB

Navigate to the DynamoDB service in console

- Choose *Tables* option from the left side panel and click on *Create Table.*
- Fill in table name as *AWSWorkshopDemoTable* and partition key as *PK.*
- Choose *Customize settings* and select *On-demand* under the *Read/write capacity setting*.
- Leave everything else to default and click on *Create Table*.

A custom role for Lambda

Since we are going to create more than one lambda, we can create an IAM role and then associate that role for each lambda. To create one, navigate to the IAM service in console.

- Choose *Roles* from the left side panel and select *Create Role.*
- Select *AWS Service* under *Trusted Entity Type* and *Lambda* under *Use case*.

- On clicking *Next*, you would see a list of policies which can be attached to the role. As a basic policy, choose *CloudWatchFullAccess** so that lambda can create logs.
    - Make sure whenever you attach the policy, you select the one that just satisfies the need (least privilege)
- After selecting the policies, click *Next*, fill in the role name as *aws-workshop-lambda-role*.
- Verify the policies and click on *Create role*.

*CloudWatchFullAccess - CloudWatch is a service that when integrated with any service would log the execution results. It is useful for debugging and checking the status of your application.

Adding lambda functions

Navigate to Lambda service in the console

- Click on *Create Function* and choose the *Author from scratch* option.
- Enter the function name as *createShortURL*.
- Choose *Node.js 14.x* as the runtime.
- Go to *Change default execution role* and select Use an existing role. Select the *aws-workshop-lambda-role* that would be appearing in the drop down.
- Click on *Create Function*.
- Repeat all above steps and create a new function name *getLongURL*.
- To test whether you could see the logs
    - Select any function you created and add *console.log('Hello there!')* before the return statement.
    - Click on *Test* option and create a new test event.
    - Click on Test again and you would be able to see the execution results next to the code.
    - To see the logs in CloudWatch, select the *Monitoring* option above the code section and select *View logs in CloudWatch.*
    - The latest log will be the one corresponding to the latest execution.

Creating APIs

Navigate to the API Gateway service in the console.

- Select *Create API* option.
- Click on the *Build* button of *HTTP API* from the types given in the dashboard.
- Click on *Add integration* and choose *Lambda* from the options.
- Set the region and choose the *createShortURL* lambda you created.
- Similarly, add one more integration for *getLongURL* lambda.
- Fill *aws-workshop-api-gateway* as the API name and click on *Next*
- In the *Configure Routes* section
    - Select the *POST* method from the drop down for *createShortURL* integration.
    - Rename the route to */shorten*
    - Select the *GET* method from the drop down for *getLongURL* integration.
    - Rename route to */{shortId}* in the route name (note: the routes that need *path parameters* specify the parameter inside the curly braces).
- Leave the stages setting to default values ($default with auto deployment enabled).
- Click on *Next*, review the data and select *Create* option.
- Testing in Postman
    - Go to the api gateway you created and copy the url.
    - Add a new request in *Postman* and paste the url there.
    - Select the method as *POST* and append */shorten* to the url and click on send (will get *Hello from lambda* as reply with status 200).
    - Add one more request and select *GET* as the method.
    - Append */some-id* to the url and click on send (will get the same response).
- Configuring CORS in API Gateway
    - Even though you got a response from Postman, actual applications would face CORS issues.
    - To resolve this, navigate to the API gateway and select the API you created.
    - Select the *CORS* option from the left side panel.

- Select *Configure* option and add
  - *Access-control-allow-origin* as * (could specify the origin explicitly for more security).
  - *Access-control-allow-method* as * (we might need all types of API calls).
  - *Access-control-allow-headers* as *content-type.*
- Select 'Save'.

Deploying actual code to Lambda

We can deploy lambda code by uploading a ZIP file directly. If the size exceeds more than 10MB, we could upload the ZIP file to S3 and then use the S3 location to deploy the code. Follow the below steps to generate the ZIP files needed for the deployments.

- Open the *url-shortner-backend* project in IDE (or navigate in terminal)
- Run the shell script available in the directory.
  - Mac/linux users - *sh package-lambdas.sh*
  - Windows users
    - Run the script *package-lambdas-win.bat*.
    - Once the *build* folder is created, zip the *nodejs* folder inside that and rename it as *lambdalayer.zip*.
    - Move inside *babel-src* folder inside build folder and zip both *createShortURL* and *getLongURL* folders.
- In case the shell script isn't working
  - Create a folder named *nodejs.*
  - Copy the *node_modules* folder to the *nodejs* folder
  - Zip the nodejs folder and rename it to *lambdalayer.zip*.
  - Create two folders with the same name as lambdas and add an 'index.js' file in both.
  - Run *npx babel src/<lambda-name>* and copy the contents to the index file of the corresponding folder
  - Zip both folders.

In order for the lambdas to execute, it should be provided with the dependencies we add in the project. Take the case of NodeJS. We would be having node_modules which contain all the dependencies. The size of the folder is too high and providing it to every lambda, every time is just redundant work. It would also take up the size we can utilize for the code. To eliminate such problems, lambda offers a feature called *Layer*. Lambda layer when attached to lambda would act as a dependency layer. So, this service can be utilized to manage the dependencies across all lambdas.
We have packaged the dependencies in local (lambdalayer.zip). In order to create the layer

- Navigate to Lambda service and select *Layers* from the left panel.
- Click on *Create Layer*.
- Fill in the name as *aws-workshop-lambda-layer*.
- Select the *Upload a .zip file* option and click on *Upload* button.
- Choose the zipped nodejs folder (*lambdalayer.zip*).
- Select *Node.js 12.x* and *Node.js 14.x* from the *Compatible runtimes* drop down.
- Click on *Create*.

Now that we have our zipped codes and lambda layer all set, let's start deploying the code.

- Navigate to the *Lambda* functions.
- Select one of the lambda you created.
- Click on *Upload from* option in the left of *Code source* and choose *.zip file* from the drop down.
- Select the corresponding zipped folder from your local and upload.
- Once deployed, scroll down to bottom and select *Add a layer* from the *Layers* section.
- Select *Custom Layers* option while choosing the layer and select the layer you created.
- Select the version (latest) from the next drop down and click *Add*.
- Repeat the same steps for the other lambda

In order for the lambda to read and write to the table, it needs permission to access the table. For that, we need to attach the permissions to the role we have assigned to lambda.

- Navigate to IAM services in the console and then to the *Roles* section.

- Select the role *aws-workshop-lambda-role* (the role you created for lambda) and click on *Add permissions*.
- Select DynamoDB full access permission by filtering and attach the policy.
  For fine grained accesses, use different roles for each lambda.
  It gives you flexibility to attach the least amount of permissions to each role.

Once the code is deployed, we can test it out in Postman and see the working

- Once lambdas are deployed, open Postman again and select the *POST* call.
- Add a body (raw, JSON) and enter the below content there

```
{
    "urlToShorten": "<paste a long url>"
}
```

- Hit *Send*. You should get a success response with an id.
- Copy the id (without quotes), move to the *GET* request you created earlier.
- Paste the id you copied instead of *some-id* and hit *Send*.
- You should get a success response with the URL you pasted on the *POST* call.

Hosting website in S3

For hosting the content in S3, we need the code that can be deployed. To get the production ready code

- Open the *url-shortner-ui* project in IDE.
- Open the .env file and paste the URL of your API gateway (replacing the quotes, no need to add quotes for the url)
- Run the command *npm run build* in the terminal. You would be able to see a *dist* folder on the root of the project.

To deploy the code in S3, first navigate to S3 service in the console.

- Click on *Create Bucket* and provide the bucket name as *aws-workshop*. Append your name or any unique number to make the bucket name unique (S3 bucket names should be globally unique).
- Click on *Create Bucket* button (leave all other settings to be default).

- Once created, open and upload the folders and files in the *dist* folder of your url-shortner-ui project (choose one by one or drag and drop all files).
- Navigate back to the bucket contents again and select the *Properties* option from the top.
- Scroll down to the end and select the *Edit* option of *Static website hosting*.
- Click on the *Enable* option and fill in *index.html* in the place of the *Index Document* field. Save the changes.
- You could get the website url in the static website hosting section of *Properties*.
- On hitting the URL, you get a 403 error. In order to host a static website in S3 accessible to everyone, follow the below steps
    - Click on the options symbol on top left of the page and select *Block public access settings for this account*.
    - Edit the setting and turn off *Block all public access* and save.
    - Navigate back to the bucket and select the *Permissions* section.
    - Edit the *Block public access (bucket settings).* There also and turn off the *Block all public access*.
    - On the permissions section, scroll down and edit the *Bucket policy*.
    - Add the following bucket policy (edit the bucket name to your bucket name. For example, if the bucket name is *aws-workshop-codher*, resource would be “arn:aws:s3:::aws-workshop-codher/*”)

```
{
        "Version": "2012-10-17",
        "Statement": [
                {
                        "Sid": "Statement1",
                        "Principal": "*",
                        "Effect": "Allow",
                        "Action": [
                                "s3:GetObject"
                        ],
                        "Resource": [
                                "arn:aws:s3:::{bucketName}/*"
                        ]
                }
        ]
}
```

- Refresh the URL again and you should be able to see the website.

Now the web application is hosted and users can use the application. But in order to distribute the content across the network, we need to create a CloudFront distribution. Also, ideally the buckets should not be publically accessible as there might be other data also that you may choose to store there. So, once you have played around with the application, turn on the *Block public access* in both bucket level and account level.

Distributing content via CloudFront

Navigate to CloudFront service in the console.

- Select *Create distribution*.
- In the *Choose domain name* section, select the s3 domain that you created earlier.
- In *S3 bucket access,* select *Yes, use OAI* option and click on *Create new OAI*.
- Verify the name and select *Create* option.
- Choose that OAI from the dropdown (if not selected already) and select *No, I will update the bucket policy* in the *Bucket policy* section.
- Leave everything for the default value except the *Default root object* under the *Settings* section. Fill in *index.html* as the value there.
- Click on *Create distribution* button (creation of distribution will take a few minutes).

Since we have selected the option for Bucket policy as *No, I will update the bucket policy*, we need to modify the policy that we have added for the bucket.

- Navigate to S3, choose the bucket and go to *Permissions* section.
- In the existing bucket policy, edit the Principal as shown below

```
"Principal": {
        "AWS": "arn:aws:iam::cloudfront:user/CloudFront Origin Access Identity
<oai-id>"
}
```

  You can find the Origin Access Identity ID by going to the *Origin access identities* section on the left panel

Once you have updated the bucket policy, you would be able to hit the cloudfront url and see the website.

Question and Answer session

Questions raised during the session and the answers to it

When do we need a sort key(which kind of applications)?

In AWS, for each resource we need to assign a role. Each role could have a set of policies which will specify the permissions.

In our example, we had Lambda. We assigned a role for it and in the policies we specified that it could use CloudWatch. Later, when we needed to, we added the access to DB.

For More visit the AWS documentation.

When do we need a sort key(which kind of applications)?

DB schemas in DynamoDB are a bit different. We try to follow the single table design, where we would try to use a single table for the entire application.

Eg: Let's say there are Products and orders for each product, then the schema would be

PK - ProductID

SK- OrderID

PK	SK	OtherOrderAttributes...
Product#1	Order#1	
	Order#2	
Product#2	Order#1	
	Order#2	
	Order#3	

For more (with use cases) visit the AWS blog.

References

1. AWS Documentations
    - Serverless computing - https://aws.amazon.com/serverless/
    - DynamoDB - https://aws.amazon.com/dynamodb/
    - Lambda - https://aws.amazon.com/lambda/
    - API Gateway - https://aws.amazon.com/api-gateway/
    - S3 - https://aws.amazon.com/s3/
    - CloudFront - https://aws.amazon.com/cloudfront/
2. Serverless vs serverful analogy - Techno-Analogy: Serverless | LinkedIn
3. Quasar framework - Quasar CLI | Quasar Framework

Keerthana Kasthuril
Consultant

keerthana.kasthuril@thoughtworks.com

Kinnari Sutaria
Consultant

kinnari.sutaria@thoughtworks.com