International Journal of Research in Engineering, Science and Management
Volume 5, Issue 12, December 2022
https://www.ijresm.com | ISSN (Online): 2581-5792

Traversing Serverless Architecture

Vaani Garg*
Assistant Professor, Department of Computer Science and Engineering, Maharaja Surajmal Institute of Technology, Delhi, India

***Abstract*: For a long time, developers have been spending a big part of their time and efforts in managing and caring for the server infrastructure of their application or website. Moreover, they have had to tend to the operating system and web server hosting process required for their application. They have had to divide their attention away from the main product their organisation offers. There has to be an easier way. A serverless architecture is a way to build and run applications and services without having to manage the backend infrastructure. Your application still runs on servers, but all the server management is done by the cloud provider. The purpose of this project is to completely explore the serverless architecture in its whole depth and breadth. It contains the comparison of serverless way of architecting applications with the traditional way of deploying applications on provisioned servers. It includes the comparison of cost, dependency on code quality, type and domain of project and the correct way of architecting applications for serverless. Serverless applications are made to run pods for each process request and these pods kill themselves or are killed by a master process in a master-slave architecture. A completely separate microservice based on NodeJS was created as the NodeJS based microservice is acting as a storage location server which stores all data that is required to be saved.**

***Keywords*: Serverless architecture, AWS, Persistent storage, Traditional server, Web crawler.**

1. Introduction

Servers are the backbone of any real-world, production-grade application. If you want to maintain such an application on the internet, you need to have servers that handle user traffic and requests. Traditionally these things have been managed through manually-maintained, offline databases and servers. Even though they may do the job in a lot of cases, they are a nuisance even in the best of scenarios. The larger an organisation gets, the bigger does its traffic gets. This means in the long-term you need more and more people to maintain and upgrade your backend servers, the very same people who could have been productive in different ways at different places.

Thus, serverless architecture offers a lot of incentives to companies which desire a better way to manage their backend. Companies can now offload the burden of managing their complex databases to cloud providers like Amazon and Google, who offer services like AWS and GCP respectively. Users of these services can opt for their paid plans, even though free plans are also usually available. Users pay for having cloud servers and computers on a pay-as-you-use basis, and therefore all the traffic and data management is managed by these cloud providers in lieu of a fee for their services.

2. Literature Review

Quite a bunch of work has already been done on Building serverless applications. Salesforce.com started the first popular SaaS cloud computing services in 1999. Today, Amazon, Google and Microsoft lead the pack in terms of providing cloud computing services. Cost-effective- Serverless provides a pay-per-use model where businesses don't need to invest in extra servers for handling an estimated workload.

The inherent convenience and scalability of serverless cloud providers has enticed quite a few big companies and startups. A serverless implementation of a core banking system is presented Pu et al. [1]. Goli et al. [2] presents a case study of migrating to serverless in the FinTech industry.

Big cloud providers like Amazon and google often offer complex, multi-layered solutions like Cache and Ram on the cloud too. Amazon for example offers Elasti Cache, which is an in-memory cache and data store. It is around 700x more expensive than Amazon's standard storage service. Infini Cache [3] is another in-memory object caching system based on stateless cloud functions. Cloud Burst [4] also proposes a caching mechanism in its architecture. Other latest advancements in the area are elaborated in [5]-[7].

3. Objectives

The primary objective of our work is to design an effective, optimized, and automated pipeline of the complete serverless architecture. The secondary objective is to create applications with different scenarios and to compare the cost, dependency on code quality, traffic dependency, and fit of different kind of projects in serverless.

We approach this problem:

1. Create a project on serverless architecture.
2. Compare it with the traditional server approach in terms of cost, code quality, and flexibility.
3. Solve the problem of persistent storage on serverless architecture.

4. Methodology

A. Tools And Techniques Used

Tools and techniques we used for "Traversing Serverless Architecture" project include:

1. React JS

*Corresponding author: vaani.garg@msit.in

2. Node JS
3. AWS

1) React JS

React JS is a JavaScript library to build online Single page applications in a quick and efficient manner. It is a component-based library which accomplishes state management via hooks. We will use React JS in tandem with Material UI to build a beautiful frontend interface for our website in an efficient manner.

2) Node JS

NodeJS is an open-source JS framework that is designed to build scalable network applications. NodeJS is a server-side platform built on Google Chrome's V8 JavaScript Engine.

3) AWS

AWS is the world's leading cloud database/server offering company with more than 100 distinct services available. More and more businesses are increasingly choosing to go online without having to deal with the hassle of independently managing a server. That is where AWS comes in.

B. Implementation

For the proposed website, we developed the entire application on React JS and NodeJS, while using AWS to host and power our website. We used beautiful Soup to write web crawlers in python to fetch data from our chosen websites and arrange the information on our webpage. We hosted our crawlers on AWS and they will run every day at a selected time. The resources needed to run the crawlers along with the cloud space needed to store our data will incur cost to us, and we compared how the costs vary with various software tweaks, and how they compare with a local server architecture. We noted how different parameter variations lead to cost fluctuations in our cost.

1) UI/UX

We have created a beautiful UI to present this functionality in a pleasing manner. We are using ReactJS as our frontend framework. Users can come to our website and see jobs from various big tech companies in a sorted manner, thus saving their time while job hunting.

2) Traditional Server vs Cloud Server

Fig. 1 and Fig. 2 depict how the data that our web crawler fetched from the Amazon website. The crawler is deployed on Amazon's cloud servers, thus saving us the need to set up our own. The data is fetched in JSON format, and can then be displayed using react JS in a pleasing manner. Table 1 shows the comparison of Traditional vs. Serverless architecture.

```
mongoose.connect(
  "mongodb+srv://admin:admin@cluster0-nbxxl.mongodb.net/jobsSkillUnga?retryWrites=true&w=majority",
  {
    //useMongoClient: true
    useNewUrlParser: true,
    useUnifiedTopology: true,
  },
  console.log("Database Connected")
);

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "*"); //'Origin, X-Requested-With, Content-Type, Accept, Authorization'
  if (req.method === "OPTIONS") {
    res.header("Access-Control-Allow-Methods", "PUT, POST, PATCH, DELETE, GET");
    return res.status(200).json({});
  }
  next();
});

const emailRoute = require("./backend/api/email");
const jobsRoute = require("./backend/api/jobs");
const emailSendRoute = require("./backend/api/functions/sendEmail");

app.use("/api/email", emailRoute);
app.use("/api/jobs", jobsRoute);
app.use("/api/email", emailSendRoute);
```

Fig. 1. Code for web crawler

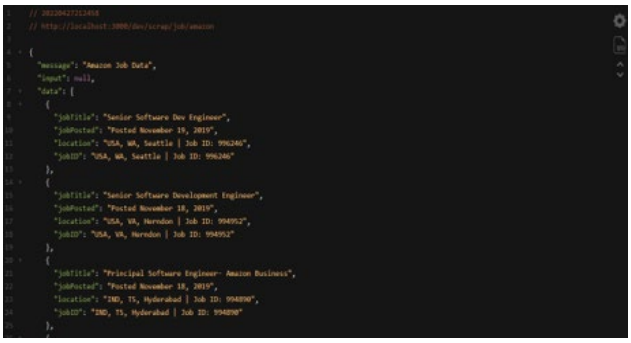

Fig. 2. Data fetched by web crawler

```
const mongoose=require('mongoose');
const express= require('express');
const router= express.Router();

const jobData=require("../../../models/HiristJob");

router.get('/hirist/backend',async(req,res)=>{
    jobData.find({UID:"hirist_engineering_Backend"},async(err,found)=>{
        if(err){
            return res.status(500).json({
                status: 500,
                message: err,
            })
        }
        else{
            if(found.length){
                return res.status(200).json({
                    status: 200,
                    message: found,
                })
            }
            else{
                return res.status(404).json({
                    status: 404,
                    message: "No jobs found"
                })
            }
        }
    })
```

Fig. 3. Traditional server

Table 1
Traditional vs. Serverless architecture

Topic	Traditional On-premise Server	Serverless
Cost on production	Preferred for applications with huge traffic	Preferred for applications with moderate and less traffic
Use Case	Works best for heavy applications connected with multiple services running together	Works best for APIs and trigger-based systems.
Speed of Development	No initial setup required.	Initial setup is required but load balancing is not required
Speed	Depends on both the intermediate services and the resources of the hosted system	Depends majorly on the code quality

Fig. 3 shows the implementation of a traditional JSON server using NodeJS. It is powered by our local machine, and hence is less powerful, even if we retain more control.

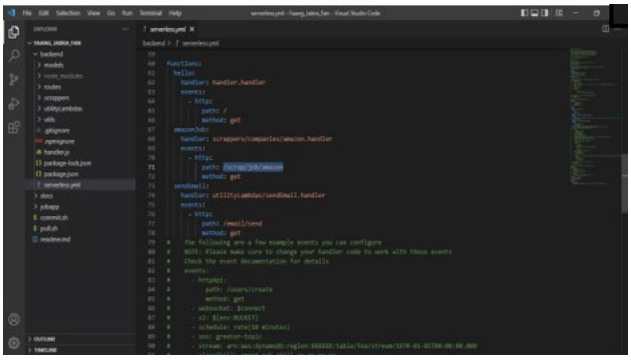

Fig. 4. Code to fetch data and send emails

We call the functions deployed on Amazon's lambda bucket as shown in Fig. 4. We ask our web crawler to go through the Amazon page and fetch the Jobs data, while also sending a reminder E-mail to everyone who has signed up for our mailing services. The mail service is also powered by AWS. There are many research advancements in the area [8], [9].

3) Persistent Storage

Serverless applications are made to run pods for each process request and these pods kill themselves or are killed by a master process in a master-slave architecture. Thus, serverless applications cannot persist storage or maintain a permanent storage point for files. While databases help store structured data and similarly S3 buckets and other static storage services help us store images, permanent native file storage is required for storage while processes run. This problem is solved in the following manner: A completely separate microservice based on NodeJS was created which was hosted separately.

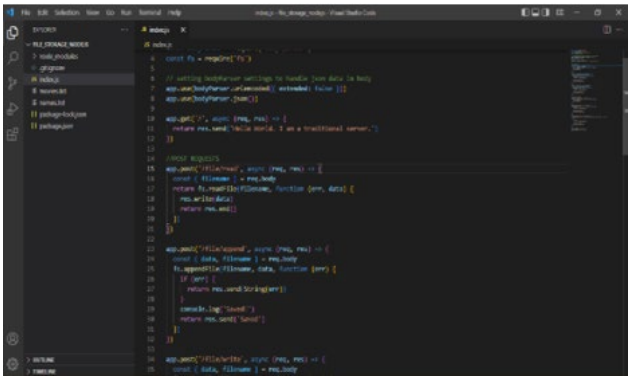

Fig. 5. NodeJS code

5. Results

Serverless applications cannot persist storage or maintain a permanent storage point for files. While databases help store structured data and similarly S3 buckets and other static storage services help us store images, permanent native file storage is required for storage while processes run.

This problem is solved in the following manner: A completely separate microservice based on NodeJS was created which was hosted separately. This microservice was capable of handling incoming request which has data that needs to be stored or a request to read existing data. This microservice helps store data permanently. Whenever the serverless architecture requires to store some data permanently, it sends a request to the microservice, which in turn creates mapping of that particular request and the file it creates and stores data in it. This data can then be later accessed by making another request from the serverless architecture. Thus, the NodeJS based microservice is acting as a storage location server which stores all data that is required to be saved.

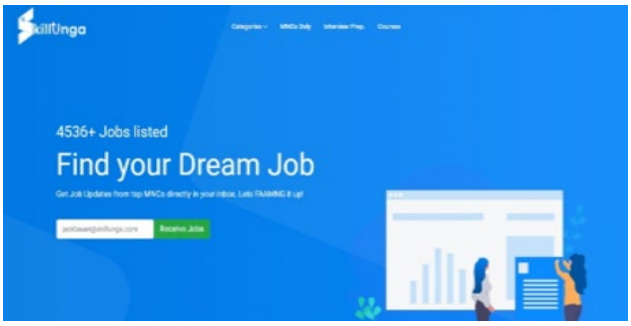

Fig. 6. UI screen

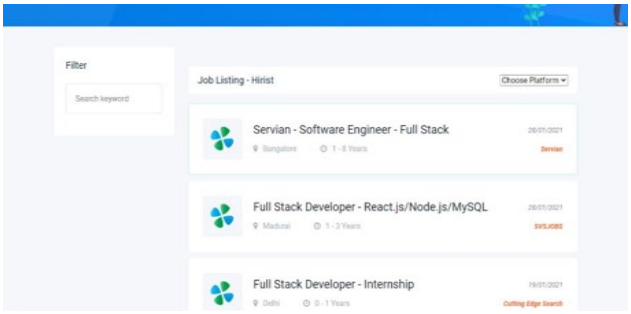

Fig. 7. Dashboard

6. Conclusion

Our comparisons in data storage and processing between local NodeJS servers and AWS cloud servers have yielded the expected results. While we retained more instantaneous control over our data and privacy while using persistent storage, we found it easy to scale our application when the backend was hosted on AWS, as in when we were using serverless architecture.

7. Future Scope

In the future, we could experiment with expanding the scope of our testing. We can use different types of frameworks and SDKs in one project and host them separately on local and cloud servers. We can try to create separate buckets for individual users, thereby enhancing the security, scalability, and privacy of our application.

We could compare how two different cloud service providers (For Ex- Google and Amazon) fare against each other when we compare the latency and load-bearing capacity of their servers.

References

[1] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. “Shuffling, fast and slow: Scalable analytics on serverless infrastructure”, 16th USENIX Symposium on Networked Systems Design and Implementation, 2019.

[2] Alireza Goli, Omid Hajihassani, Hamzeh Khazaei, Omid Ardakanian, Moe Rashidi, and Tyler Dauphinee. “Migrating from monolithic to serverless: A Fintech case study”, 2020.

[3] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. “Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache”, 18th USENIX Conference on File and Storage Technologies (FAST 20), 2020.

[4] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tamanol, “Cloudburst: Stateful functions-as-a-service”, 2020.

[5] Lam Phuoc Huy, Saifullah Huey, Marcel Sahillioglu, and Christian Baun. “Crypto Currencies Prices Tracking Microservices Using Apache OpenWhisk”, 2021.

[6] Jens Kohler. “A Serverless FaaS-Architecture: Experiences from an Implementation in the Core Banking Domain.”, 2021.

[7] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. “Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache”, 18th USENIX Conference on File and Storage Technologies (FAST 20), 2020.

[8] Jatana, Nishtha, and Kapil Sharma. "Bayesian spam classification: Time efficient radix encoded fragmented database approach." 2014 International Conference on Computing for Sustainable Global Development (INDIACom). IEEE, 2014.

[9] Dhand, Geetika, and S. S. Tyagi. "Survey on Data-Centric protocols of WSN." International Journal of Application or Innovation in Engineering & Management, 2.2 (2013): 279-284.