CSIT6000O Group Project Report

Reimplement an AWS Serverless Application: Serverless To-do List

Group 2 Cloud Explorers

PAN Han 20881280 hpanan@connect.ust.hk

CHEN Chen 20881450 cchencu@connect.ust.hk

LONG Shiyao 20932120 slongaa@connect.ust.hk

1 Introduction

In this project, we re-implemented an AWS serverless application using open-source alternatives for AWS components. Our goal was to demonstrate our ability to work with different technologies and platforms and our knowledge of cloud computing concepts. We chose to re-implement a serverless to-do list application, which is a simple but useful application. Our to-do list web application offers the following features:

- Create new todos
- View all todos
- Marking a todo as complete
- View complete/incomplete todos
- Delete todos

The original application relied heavily on AWS services, such as AWS Lambda, DynamoDB, and API Gateway, for its functionality. We replaced all the AWS components used in the original application with open-source alternatives, including OpenFaaS for AWS Lambda, MongoDB for DynamoDB, and Nginx Ingress Controller to replace AWS API Gateway. Nginx acts as a reverse proxy and router, handling incoming HTTP/HTTPS requests and directing them to the appropriate service based on the URL suffix. Furthermore, the project was containerized and can be deployed to any Kubernetes cluster, improving the ease of using and deploying our application.

Our final deliverable includes all the necessary components packaged as Docker images, which can be deployed directly to a Kubernetes cluster. In addition to the backend components, we also developed a frontend web application using JavaScript React framework. The frontend provides a user-friendly interface for creating and managing tasks and communicates with the backend via OpenFaaS functions.

In this report, we provide a detailed description of our project, including the infrastructure, and implementation of frontend, backend, gateway and database. Finally, we discuss the lessons learned from the project and potential future work to enhance the application.

2 Infrastructure

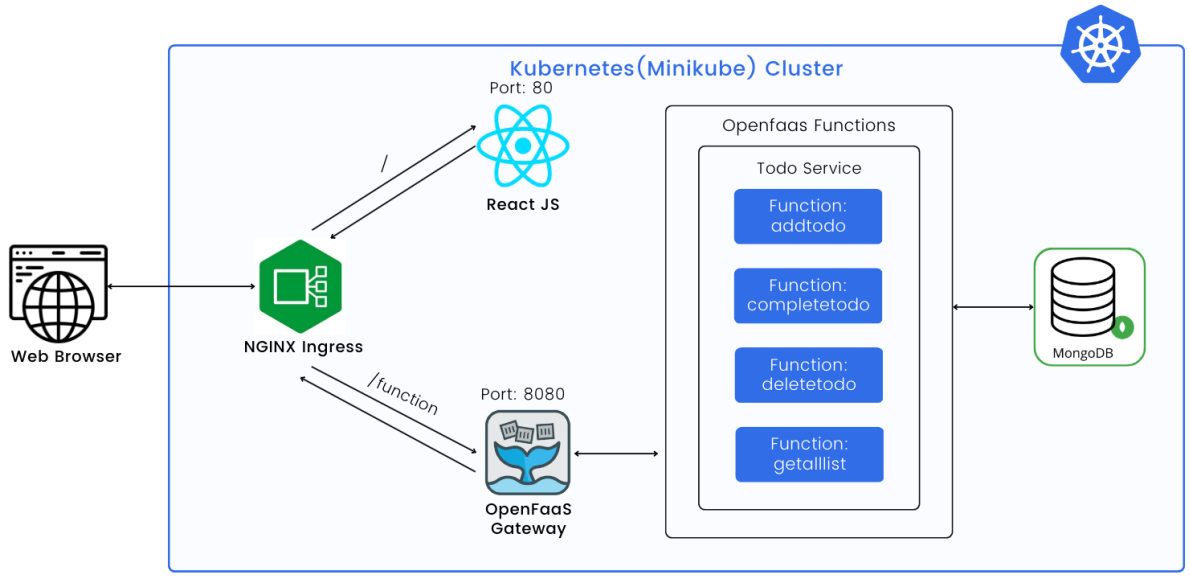

Figure 1. Infrastructure

The re-implemented serverless application consists of several open-source components that replace the AWS components in the original application. The application is fully containerized using Docker images and is ready to be deployed to any Kubernetes cluster. Our deployment utilizes a Minikube cluster, which is a lightweight Kubernetes implementation that contains only one node. To access the application, requests are first sent to the Nginx Ingress, which matches the URL path of the request against the configured rules and forwards the request to the appropriate Kubernetes services. The frontend service exposes an port 80 and the gateway service exposes a port 8080 within the cluster. OpenFaas is used to replace AWS Lambda in the original application, allowing for easy deployment of functions to Kubernetes clusters. The database service is implemented using MongoDB. To simplify the communication of resources within the cluster, we deploy ingress, frontend service and openfaas gateway service in the same namespace named "openfaas". The database service and openfaas function services are deployed within another namespace, named "openfaas-fn".

3 Implementation

3.1 Backend

Our backend implementation uses OpenFaaS functions to replace the Lambda functions provided by AWS. OpenFaas is an open-source serverless framework that allows us to build and deploy our serverless functions. We deployed four serverless functions on OpenFaas, which are addtodo, completetodo, deletetodo, and getalllist.

Each OpenFaas function contains three components, the shared.py, handler.py, and requirement.txt. The shared.py file contains the common functions and constants that are used across the openfaas functions. The handler.py file is where the main logic of the openfaas function is implemented. It contains the function that will be triggered when the function is invoked. The requirements.txt file lists all the required packages and their versions for the openfaas function to run successfully. This file

is used to install the necessary dependencies during the Docker image build process. When the Docker image is built, the shared.py, handler.py and requirements.txt files are all included in the image so that the function can be run correctly.

Function - addtodo

addtodo is a function that is used to add an item to a todo list. It first imports the necessary modules and functions from shared.py to handle exceptions and get the user ID and headers. Then it connects to the MongoDB database using the URL from the environment variables. The function handles the request by first getting the content of the item from the request payload and the user ID from the request headers. It then aggregates the collection to get the maximum item ID for the user and increments it to create a new item ID. The item is then created with the ID, content, and other necessary fields, and inserted into the database. Finally, the function returns a JSON response with a 200 status code, headers, and a message confirming that the item has been added to the list along with its ID.

Function - completetodo

completetodo function is used to mark an item as completed in the todo list. When a user presses the "complete" button on a specific item, this function is triggered with the item ID in the request payload. The function first retrieves the user ID from the request headers, and then uses this ID and the item ID to update the corresponding item in the MongoDB collection. The ‘complete’ field of the item is set to True, indicating that the item is completed. The function then returns a success response with the updated item ID and a message indicating that the item has been completed. This function is implemented using the Python Flask framework and is built into a Docker image using the OpenFaaS python3-http template.

Function - deletetodo

The deletetodo function is used to delete a todo item from MongoDB database. The function uses the pymongo library to connect to a MongoDB instance running on a specified URL and authenticate using a user ID. The handler function is called when the function is invoked and it retrieves the item ID from the request payload and the user ID from the headers of the request using functions defined in a shared.py module. Then it constructs a MongoDB query using the item ID and user ID to delete the item from the database using the delete_one method of the mycol collection object. Finally, it returns an HTTP response with a status code of 200, headers, and a JSON payload indicating the deleted item ID and a message saying the item was deleted.

Function - getalllist

The getalllist function is used to retrieve data from MongoDB database. The handler function receives a request through its event parameter and uses the get_user_id and get_headers functions to extract user authentication information from the request headers.

If the user_id is generated (meaning that the user has not yet been created in the database), an empty list is created for item_list. Otherwise, a query is executed to retrieve all items in the database with a partition key of "user#user_id".

Each item in item_list is then formatted into a dictionary with three keys: "id", "item", and "completed", which are derived from the item's "sk" value, "content" value, and "complete" value respectively. The formatted items are then added to a new list named formatted_items.

Finally, the function returns a response with status code 200, headers generated by the get_headers function, and a body consisting of a dictionary containing the formatted_items list and a "message" key with value "show all item", which indicates that the response contains all items in the user's to-do list.

Backend Test

The OpenFaas Gateway provides a web-based UI that allows us to test and invoke functions. Figure 2 below demonstrates one of our test cases. In this test case, we tested the addtodo function using the OpenFaaS UI. The input provided was a JSON object with the keys "item" and "completed", with the corresponding values of "Do my homework" and false. After executing the function, the response body contained a JSON object with the keys "itemId" and "message", and their corresponding values of 3 and "item added to list", respectively. This indicates that the function successfully added the new todo item to the database, and returned the expected response.

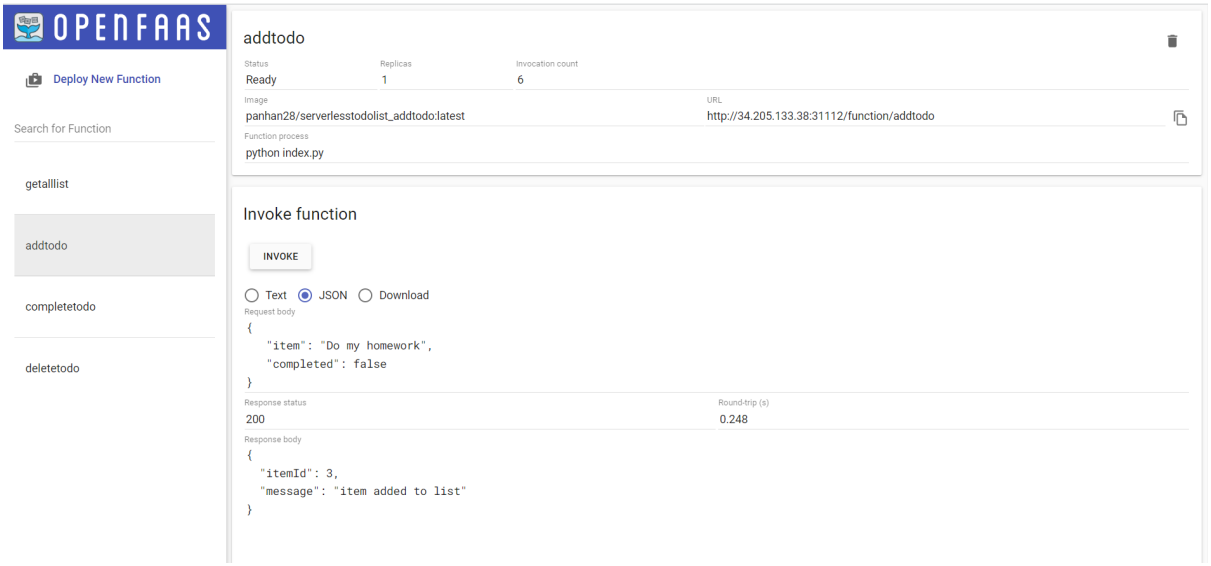

Figure 2. OpenFaaS UI test case on addtodo function

3.2 Database

We used MongoDB as our database. MongoDB is a popular NoSQL document database that stores data in a flexible, JSON-like format. In this case, the "test" database is being used, and there is a "list" collection that is storing the to-do items. The "pk" field is being used as a partition key to group the items by user, and the "sk" field is being used as a sort key to uniquely identify each item. The

functions are using the PyMongo driver to interact with the database, and there are functions for adding, updating, deleting, and retrieving items from the database. Overall, the database design seems simple and straightforward, and it should be easy to scale and maintain as the number of users and items grows.

3.3 Frontend

Description

The frontend of the application is built using ReactJS and uses the Bootstrap framework for styling. It makes API calls to the backend using the Axios library to retrieve, add, delete and complete todo items. We decoupled and containerized the frontend from specific cloud services, making it more portable and flexible. It can be deployed in any Kubernetes environment. Below Figure 3 is our user interface demonstration.

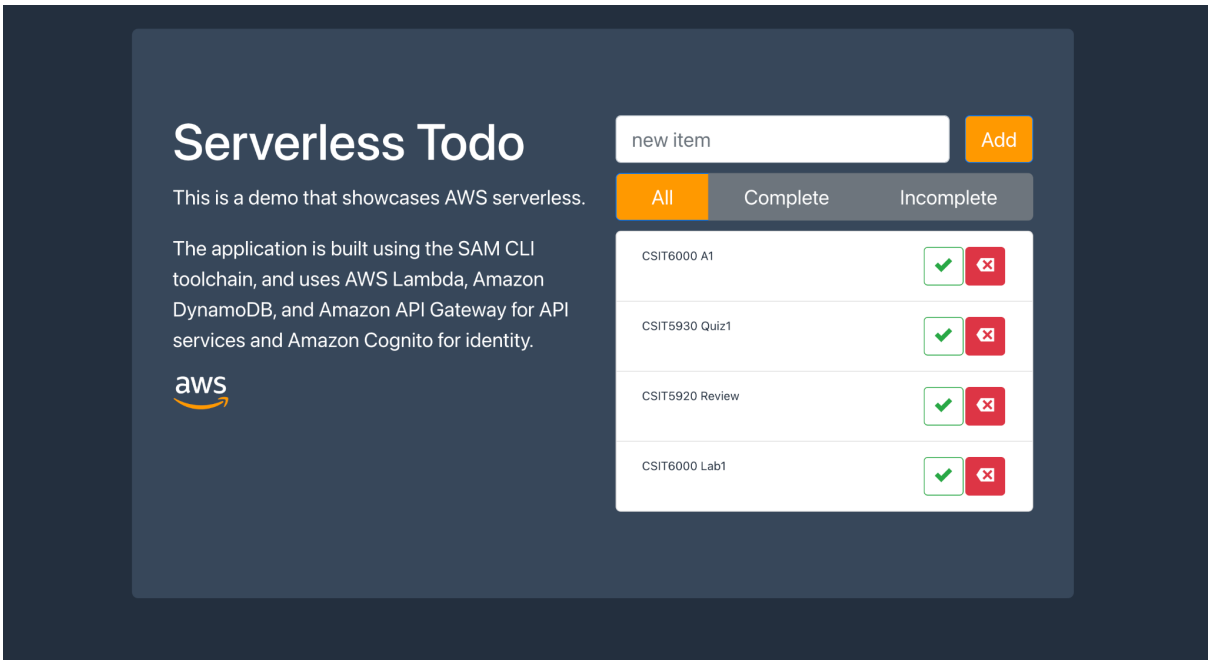

Figure 3. User Interface

Replace AWS Components

The first phase of updating the frontend involved replacing AWS components, specifically shifting the deployment from Amplify towards Kubernetes. Amazon Cognito was also removed from the equation, eliminating the dependency on AWS for user authentication and identity management. For the serverless functions, OpenFaaS was chosen. We invoked our re-implemented OpenFaas functions instead of the original AWS Lambda functions.

Build Docker Image

The second phase of the frontend update focused on containerizing the application with Docker. This approach offers several benefits, including increased development speed, consistency across different environments, and easier scalability. A Dockerfile was created to define the application's container environment. This file specified has two stages, the build stage, and the run stage.

In the build stage, a Docker image based on Node.js 14 is created. The application is copied into this image, and all dependencies are installed using npm. The application is then built using the npm run build command.

In the run stage, the built application from the build stage is copied into this new image. The http-server package is then globally installed in the image using npm install -g http-server. This package is a simple, zero-configuration command-line HTTP server that will be used to serve your application. The application is configured to run on port 8888.

3.4 Gateway

In the initial application, AWS API Gateway previously acted as the front door. It handled and routed incoming API calls and provided Cross-origin resource sharing (CORS) support, access control and monitoring. In our project, we leveraged the OpenFaaS gateway as a key component that functions as an API gateway. This component provides a unified HTTP access interface to the functions deployed on our Kubernetes cluster.

While the OpenFaaS gateway itself does not support CORS functionality, we cleverly leveraged Ingress resources to configure CORS rules. In our Ingress configuration, we established CORS-related settings using annotations, thereby enabling the handling of CORS requests effectively. We also use Ingress to uniformly manage the routing of external access to the services within the cluster.

3.5 Deployment

The deployment of our web application is based on a Minikube cluster, which is a lightweight Kubernetes implementation containing only one node. We leveraged Kubernetes to deploy the various components of the application such as the frontend, backend, and database. The OpenFaaS platform is used to deploy and manage serverless functions. Nginx Ingress Controller is used as the Ingress controller to manage the routing rules and forward the requests to the correct backend services. We also utilized Docker containers to package and deploy the application components. Overall, the deployment process involved configuring various Kubernetes resources, creating Docker images, and deploying them on the Minikube cluster.

4 Challenges and Solutions

Throughout the project, we encountered several challenges and problems during implementation.

Understanding Kubernetes Concepts

As Kubernetes has a steep learning curve, we found it challenging to understand the different resources and how they work together. Specifically, creating and managing Kubernetes Deployments, Services, and Ingress proved to be areas of struggle. We had to invest a considerable amount of time and effort in researching, experimenting, and troubleshooting to get our Kubernetes infrastructure up and running smoothly.

Dockerization

Another challenge we faced was creating Docker images of the frontend, backend, and MongoDB that could be deployed on Kubernetes. We also had to consider how to handle data persistence and backups, as well as how to configure networking and authentication. This required significant knowledge of Docker, React and MongoDB, as well as some trial and error to get the image working as expected.

Debugging OpenFaaS

Finally, we encountered issues with debugging OpenFaaS. Debugging serverless applications can be challenging, as we may not have access to the underlying infrastructure or be able to reproduce issues easily. To overcome this, we had to use various debugging techniques, such as logging and tracing tools, debugging locally using the OpenFaaS CLI, and using Kubernetes tools like kubectl to inspect pods and containers.

CORS Issue

During the implementation of our project, we faced a notable challenge related to the CORS. CORS is a security feature that restricts HTTP requests from different domains and OpenFaaS doesn't natively support CORS. Hence, we utilized Ingress to handle the CORS issue. The initial configuration of Nginx Ingress Controller was challenging, especially with respect to defining the correct routing rules. There were issues with the routing of static files and API endpoints to the correct services within the Kubernetes cluster. It was difficult to ensure that all the routes were correctly captured and directed to the appropriate services.

5 Conclusion

In this project, we have successfully implemented a serverless web application using OpenFaaS, Kubernetes, and other open-source technologies. We have demonstrated that it is possible to build a highly scalable and cost-effective application without using proprietary cloud services like AWS. Our implementation includes a fully functional frontend, backend, and API gateway. Overall, we believe that this project provides a valuable alternative to using commercial cloud services for building serverless applications.

Reference

AWS Serverless Web Application Reference Architecture. Retrieved May 12, 2023, from https://github.com/aws-samples/lambda-refarch-webapp

ubernetes. (n.d.). Kubernetes - Production-Grade Container Orchestration. Retrieved from https://kubernetes.io/

Minikube. (n.d.). Minikube - Run Kubernetes locally. Retrieved from https://minikube.sigs.k8s.io/docs/

OpenFaaS. (n.d.). OpenFaaS - Serverless Functions Made Simple. Retrieved from https://www.openfaas.com/

React. (n.d.). React - A JavaScript library for building user interfaces. Retrieved from https://reactjs.org/

Set up ingress on Minikube with the nginx ingress controller. Kubernetes. (2023, March 23). https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/

Ingress. Kubernetes. (2023a, February 2). https://kubernetes.io/docs/concepts/services-networking/ingress/

Nginx Ingress Controller. (n.d.). Kubernetes Ingress with Nginx Example. Retrieved from https://kubernetes.github.io/ingress-nginx/examples/