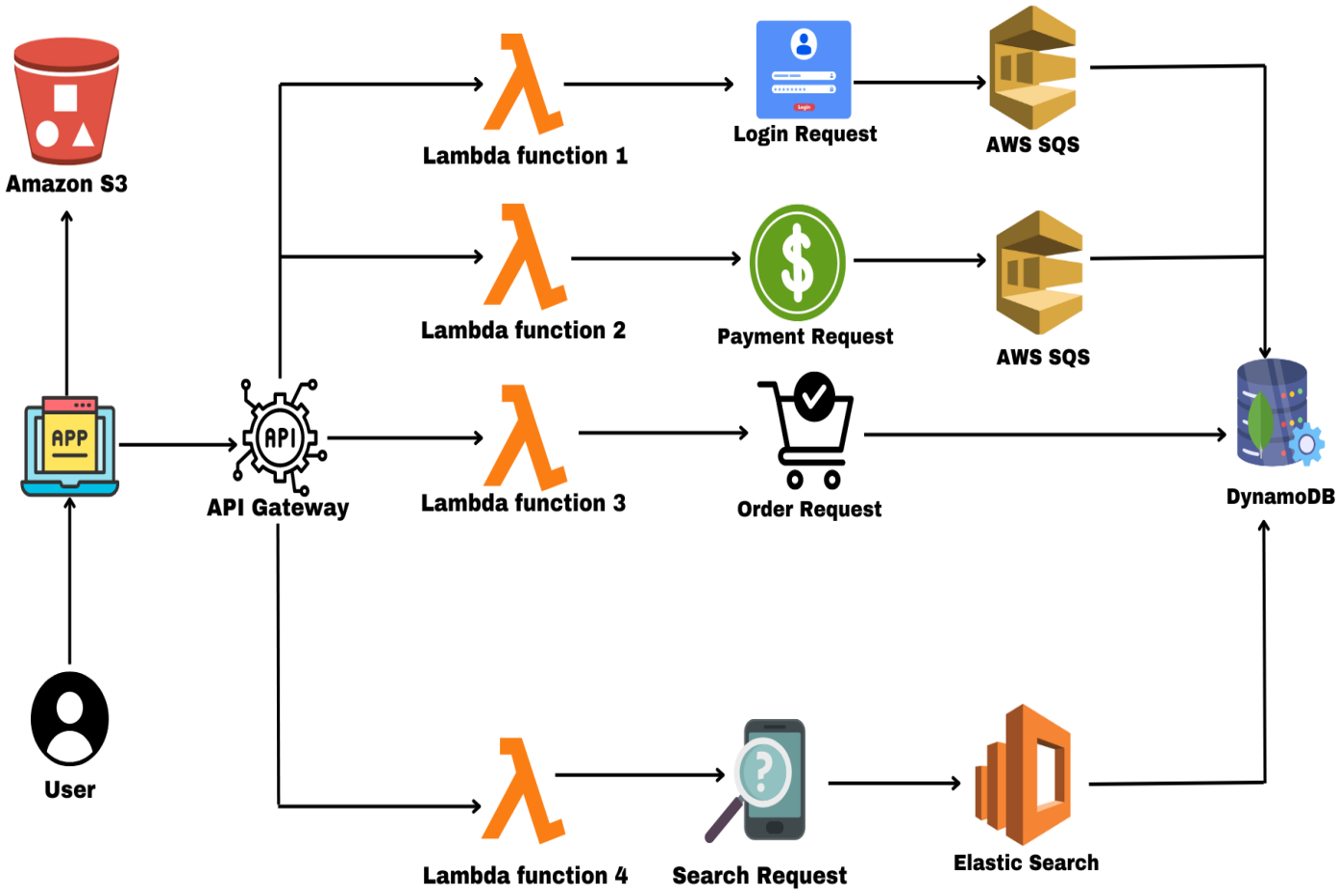
DevOps Shack

SERVERLESS ARCHITECHTURE

Serverless architecture allows for the development of applications without managing servers or infrastructure. In this document, we will explore a serverless architecture that uses Amazon SQS (Simple Queue Service) and AWS Lambda functions to process search queries.



What is Serverless Architecture?

Serverless architecture is a cloud computing model where applications are built without worrying about the underlying infrastructure. Instead of provisioning and managing servers, developers can focus on writing code that responds to requests without worrying about the underlying infrastructure. This approach provides several benefits, including:

Scalability: Serverless architectures can scale up or down automatically to meet changing demand.

Cost-effectiveness: Serverless architectures only charge for the compute time consumed, which can lead to significant cost savings.

Reliability: Serverless architectures are designed to be highly available and fault-tolerant, with built-in features such as automatic scaling and load balancing.

Architecture Overview

The architecture consists of the following components:

Login Request

The web application sends a login request to the API Gateway.

The API Gateway receives the request and triggers Lambda Function 1.

Lambda Function 1 authenticates the user using credentials (e.g. username and password) and checks if the user exists in the DynamoDB Database.

If the user exists, Lambda Function 1 generates a unique session ID and stores it in DynamoDB along with the user's data.

The API Gateway receives the response from Lambda Function 1 and sends it back to the web application.

The web application receives the response and updates the user's session information.

Search Request

The web application sends a search request to the API Gateway.

The API Gateway receives the request and triggers Lambda Function 2.

Lambda Function 2 retrieves the search query from the request and sends it to Elasticsearch.

Elasticsearch indexes the query and returns a list of matching results.

Lambda Function 2 receives the results from Elasticsearch and returns them to the API Gateway.

The API Gateway receives the results and sends them back to the web application.

The web application receives the results and displays them to the user.

Payment Request

The web application sends a payment request to the API Gateway.

The API Gateway receives the request and triggers Lambda Function 3.

Lambda Function 3 retrieves the payment details (e.g. amount, currency) from the request.

Lambda Function 3 uses Stripe Payment Gateway to process the payment.

Stripe Payment Gateway processes the payment and returns a response to Lambda Function 3.

Lambda Function 3 receives the response and updates DynamoDB with the payment status.

The API Gateway receives the response and sends it back to the web application.

The web application receives the response and updates the user's account information.

Order Notification

When an order is processed, Lambda Function 4 is triggered by an SQS Queue message.

Lambda Function 4 retrieves the order details from DynamoDB and generates an order notification.

Lambda Function 4 sends an SQS Queue message with the order notification.

SQS Queue processes the message and triggers Lambda Function 4 again.

Lambda Function 4 receives the message and sends an email notification to customers.

File Storage

The web application uploads a file to S3 Storage using AWS SDK or AWS CLI.

S3 Storage stores the file in a bucket or folder specified by the web application.

Code

Lambda Function 1: login_request

```
const AWS = require('aws-sdk');
const jwt = require('jsonwebtoken');

exports.handler = async (event) => {
  const { username, password } = event;
  const userPool = new AWS.CognitoIdentityServiceProvider({ region: 'us-west-2' });
  const params = {
    AuthFlow: 'USER_PASSWORD_AUTH',
    ClientId: 'your_client_id',
    Username: username,
    Password: password
  };
  try {
    const response = await userPool.authenticateUser(params).promise();
    const userData = response.AuthenticationResult;
    const token = jwt.sign(userData, 'your_secret_key', { expiresIn: '1h' });
    return {
      statusCode: 200,
      body: JSON.stringify({ token })
    };
  } catch (error) {
    return {
```

```
    statusCode: 401,
    body: JSON.stringify({ message: 'Invalid username or password' })
   };
  }
};
```

This code uses the AWS SDK to authenticate the user with Cognito User Pools. It takes the username and password as input, and returns a JSON Web Token (JWT) if the authentication is successful. The JWT contains the user's data, such as their ID, email, and other attributes. The token is signed with a secret key, and has a one-hour expiration period.

You can test this function by sending a POST request to the Lambda function with the following JSON payload:

```
{
 "username": "your_username",
 "password": "your_password"
}
```

The function will return a JSON response containing the JWT token if the authentication is successful. If the authentication fails, it will return an error message with a 401 status code.

Lambda Function 2: payment request

```
const AWS = require('aws-sdk');
const stripe = require('stripe')('your_stripe_secret_key');

exports.handler = async (event) => {
 const { amount, currency, paymentMethod } = event;
 try {
  const paymentIntent = await stripe.paymentIntents.create({
   amount: amount,
   currency: currency,
   payment_method: paymentMethod
  });
  return {
   statusCode: 200,
   body: JSON.stringify({ paymentIntent })
  };
 } catch (error) {
  return {
   statusCode: 500,
   body: JSON.stringify({ message: 'Error processing payment' })
  };
 }
};
```

Lambda Function 3: Order Request

```
const AWS = require('aws-sdk');
const DynamoDB = new AWS.DynamoDB.DocumentClient();


exports.handler = async (event) => {
 const { orderDetails } = event;
 try {
  await DynamoDB.put({
   TableName: 'orders',
   Item: orderDetails
  }).promise();
  return {
   statusCode: 200,
   body: JSON.stringify({ message: 'Order placed successfully' })
  };
 } catch (error) {
  return {
   statusCode: 500,
   body: JSON.stringify({ message: 'Error placing order' })
  };
 }
};
```

Lambda Function 4: Search Request

```
const AWS = require('aws-sdk');
const Elasticsearch = new AWS.Elasticsearch();


exports.handler = async (event) => {
 const { query } = event;
 try {
  const searchResult = await Elasticsearch.search({
   IndexName: 'your_index_name',
   Query: query
  }).promise();
  return {
   statusCode: 200,
   body: JSON.stringify({ results: searchResult.Hits })
  };
 } catch (error) {
  return {
   statusCode: 500,
   body: JSON.stringify({ message: 'Error searching' })
  };
 }
};
```

SQS Queue Configuration

Here is an example configuration for the SQS queue:

```
{
 "QueueName": "my-queue",
 "QueueUrl": "https://sqs.us-west-2.amazonaws.com/123456789012/my-queue",
 "VisibilityTimeout": 300,
 "MessageRetentionPeriod": 3600,
 "RedrivePolicy": {
  "maxReceiveCount": 5,
  "deadLetterTargetArn": "arn:aws:sqs:us-west-2:123456789012:my-queue-dlq"
 }
}
```

Elasticsearch Configuration

Here is an example configuration for Elasticsearch:

```
{
 "index": "my-index",
 "type": "_doc",
 "body": {
  "query": {
   "match": {
    "text": "{{search_text}}"
   }
  }
 }
```

```
}
```

DynamoDB Configuration

Here is an example configuration for DynamoDB:

```
{
 "table_name": "my-table",
 "key_schema": [
  {
   "AttributeName": "id",
   "KeyType": "HASH"
  }
 ],
 "attribute_definitions": [
  {
   "AttributeName": "id",
   "AttributeType": "S"
  }
 ],
 "table_status": "ACTIVE"
}
```

Conclusion

In this document, we have explored an architecture that uses Amazon SQS and AWS Lambda functions to process search queries. This architecture provides a scalable, fault-tolerant, and flexible way to handle search queries, allowing for decoupling of individual components and enabling them to be developed, tested, and deployed independently.