**ATS CloudinityInc.**

**Architecture 1: Serverless Architecture**

**Key Components:**

* AWS Lambda: To run backend code without provisioning or managing servers.
* Amazon API Gateway: To create, publish, maintain, monitor, and secure APIs.
* AWS DynamoDB: As a NoSQL database service for applications.
* AWS S3: To store static assets, including the React frontend and any uploaded files.
* AWS Cognito: For user authentication and authorization.

**Flow:**

* Frontend (React App): Hosted on AWS S3, accessible to users via a web browser.
* User Authentication (Optional): AWS Cognito is used if user authentication is required.
* API Calls: The React app makes API calls to AWS API Gateway.
* Serverless Backend: API Gateway routes requests to the appropriate AWS Lambda functions.
* Data Storage: Lambda functions interact with AWS DynamoDB for data persistence.
* File Storage: Any file uploads are handled by Lambda functions and stored in AWS S3.

**Architecture 2: Spring Boot on AWS Fargate**

**Key Components:**

* AWS Fargate: For running containers without managing servers or clusters.
* Amazon RDS or DynamoDB: As the database service.
* Elastic Load Balancer (ELB): To distribute incoming application traffic.
* Amazon ECR (Elastic Container Registry): To store Docker images.
* CI/CD.

**Flow:**

* Spring Boot Application: Packaged as a Docker container.
* Docker Image Storage: Images stored in Amazon ECR.
* Deployment: Docker containers are deployed on AWS Fargate.
* Database Connection: The Spring Boot app connects to either Amazon RDS (for SQL) or DynamoDB (for NoSQL).
* Load Balancing: ELB distributes incoming traffic to ensure scalability and reliability.
* CI/CD Pipeline

Estimated sprint time for the project micro services:

* Creating Git and AWS account and managing IAM roles – **1 day**
* Building React-app -> **2 weeks**
* Lambda functions for DynamoDB and S3 -> **2 weeks**
* Creating REST API for Lambda and react app -> **2weeks**
* Integrating React with Api gateway and Lambda **-> 1 week**
* Full Cycle deployment **-> 1 week**

AWS

GITHUB

VS CODE

JENKINS

DOCKER

JIRA

**Frontend (React.js):**

* The React application will serve as the UI for the job application form.
* Users fill in their details and upload their resume here.
* The application makes **HTTP API calls** to interact with the backend.

**Backend (AWS Lambda & API Gateway):**

* AWS Lambda functions will handle the business logic of your application.
* Use AWS API Gateway to create **HTTP APIs** that the React app will call.
* The APIs will be responsible for processing requests from the frontend, such as saving data to DynamoDB and uploading files to S3.

**Storage:**

* **AWS S3:** Store the resumes uploaded by the users. Lambda functions can upload files to S3 buckets.
* **AWS DynamoDB:** Store the details of the applicants. Lambda functions will interact with DynamoDB to store and retrieve data.

**CI/CD Pipeline (Jenkins, Docker, GitHub):**

* **GitHub:** Host your code repository for the React app, Lambda functions, and any other services you create.
* **Docker:** Use Docker to containerize your application, especially useful if you have services other than Lambda.
* **Jenkins:** Automate your deployment pipeline. Jenkins can pull code from GitHub, run tests, build Docker images (if needed), and deploy to AWS.
* **AWS SAM (Serverless Application Model):** Utilize SAM templates to define your serverless resources (Lambda, API Gateway, DynamoDB) and deploy them as a stack.

**Testing and Quality Assurance (SonarQube):**

* Integrate **SonarQube** into your Jenkins pipeline for continuous code quality checks and security scanning.

**Security and Best Practices**:

* Ensure that your AWS resources (like S3 buckets and DynamoDB tables) have proper access controls and policies.
* Implement authentication and authorization mechanisms if necessary.
* Use **environment variables** and **AWS Secrets Manager** for managing sensitive data like API keys.

**Monitoring and Logging:**

* Implement logging in your Lambda functions.
* Use **AWS CloudWatch** for monitoring and logging the performance of your Lambda functions, API Gateway, and other AWS resources.

**Login Flow details:**

**In the first scenario**, AWS Cognito is used directly for authentication, and there's no need for API Gateway or Lambda unless you need to perform additional processing or customization beyond what Cognito offers.

**In the second scenario**, you involve API Gateway and Lambda for more complex or customized authentication flows, which gives you more control over the authentication process, such as adding custom logic before or after the authentication step.

when new users sign up in your React app using AWS Cognito for authentication, their data is stored in Cognito itself. AWS Cognito is not just an authentication provider but also a user directory service. It securely manages and stores user data.

Here's what happens during the sign-up process:

* Sign-Up: The user provides their information (like username, password, email, phone number, etc.) on the sign-up form in your React app.
* Data Storage: This information is sent to AWS Cognito, where a new user record is created. Cognito stores the user's data, including custom attributes you've defined if any.
* Confirmation & Verification: Cognito can also handle email and phone verification by sending a confirmation code to verify the user's contact information, depending on your configuration.

Once the user is signed up and their email or phone is verified (if you have set up verification), they are able to log in. The authentication (login) requests are then checked against the user information stored in Cognito.

Cognito manages this user data securely and provides features like user authentication, user management, password reset, and multi-factor authentication (MFA) out of the box. You don't need to provision a separate database to store user credentials; Cognito handles all of this for you.

**Why Cognito?**

Storing user data like emails and passwords directly in a database like DynamoDB is technically possible, but it's generally not recommended due to security considerations. Managing user authentication data requires implementing best practices for security, like hashing and salting passwords, which services like AWS Cognito handle automatically.

**Integrating Cognito with Amplify and React App:**

Integrating your React login page with AWS Cognito and hosting it on AWS Amplify involves several steps. Here's a guide to walk you through the process:

**1. Set Up AWS Cognito User Pool**

First, you need to create a user pool in AWS Cognito:

**Create a User Pool**:

Go to the AWS Cognito console.

Click on “Manage User Pools” and create a new user pool.

Follow the prompts to set up the pool. Make note of the Pool Id and Pool ARN.

**Configure App Client:**

In the user pool, create a new App Client (without client secret).

Note down the App Client Id.

**Set Up Domain:**

Under “App integration” > “Domain name”, set up a domain for hosting your sign-up and sign-in pages managed by Cognito (if you want to use Cognito hosted UI).

**Configure User Pool Policies:**

Set password policies, multi-factor authentication (MFA), and other security features according to your requirements.

**2. Integrate Cognito with Your React App**

Use the AWS Amplify library to integrate the Cognito User Pool with your React application:

**Install AWS Amplify:**

npm install aws-amplify

**Configure Amplify:**

In your React project, configure Amplify with the Cognito User Pool and App Client Id.

import Amplify from 'aws-amplify';

Amplify.configure({

Auth: {

mandatorySignIn: true,

region: 'your-region',

userPoolId: 'your-user-pool-id',

userPoolWebClientId: 'your-app-client-id',

}

});

**Implement Authentication:**

Use the Auth class from AWS Amplify to sign in, sign up, and sign out users.

Example of signing in a user:

import { Auth } from 'aws-amplify';

async function signIn(username, password) {

try {

const user = await Auth.signIn(username, password);

console.log(user);

} catch (error) {

console.log('error signing in', error);

}

}

Implement similar functions for sign-up and sign-out.

**3. Host Your App on AWS Amplify**

*Finally, host your React app on AWS Amplify:*

**Push Your Code to a Git Repository – GitHub**

**Connect Your Repository to AWS Amplify:**

Go to the AWS Amplify Console.

Click “Connect App” and follow the prompts to connect your repository.

**Configure Build Settings:**

AWS Amplify will try to detect your build settings automatically. Review and adjust them if necessary.

**Deploy:**

Once your build settings are configured, trigger a build and deploy your app.

AWS Amplify will provide a URL where your app is hosted.

**Set Up Redirection and Rewrites (if necessary):**

If you're using React Router or similar, set up rewrites so that all URLs fall back to index.html.

By following these steps, your React app should be integrated with AWS Cognito for authentication and hosted on AWS Amplify, providing a secure and scalable solution for your user management and application hosting needs.