srs on web app on amazon web services

direito corporativo (ACE Engineering College)

Software Requirement Specification

For

Serverless Web Application On AWS

Prepared By:-

Manish Kumar
B.Tech CSE
IV Year Sec. B
1810910902

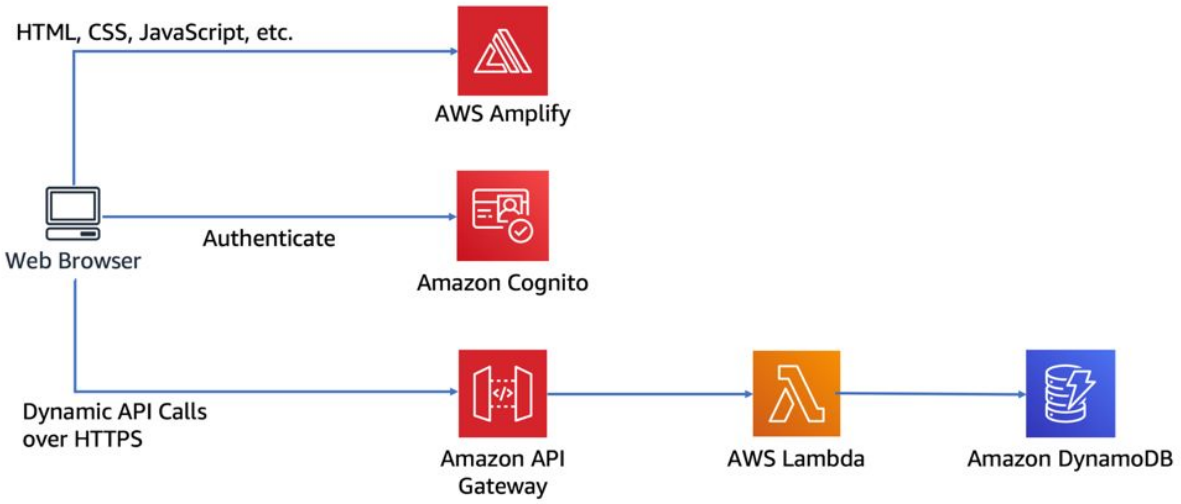
CONTENT

Overview & Purpose

In this Project, I'll create a simple serverless web application that enables users to request rides from the MyRide fleet. The application will present users with an HTML based user interface for indicating the location where they would like to be picked up and will interface on the backend with a RESTful web service to submit the request and dispatch a nearby ride. The application will also provide facilities for users to register with the service and log in before requesting rides.

Application Architecture

This application architecture uses **AWS Lambda**, **Amazon API Gateway**, **Amazon DynamoDB**, **Amazon Cognito**, and **AWS Amplify** Console. Amplify Console provides continuous deployment and hosting of the static web resources including HTML, CSS, JavaScript, and image files which are loaded in the user's browser. JavaScript executed in the browser sends and receives data from a public backend API built using Lambda and API Gateway. Amazon Cognito provides user management and authentication functions to secure the backend API. Finally, DynamoDB provides a persistence layer where data can be stored by the API's Lambda function.



Service Requirements & Usage

1. **AWS Amplify** - AWS Amplify hosts static web resources including HTML, CSS, JavaScript, and image files which are loaded in the user's browser.

2. **Amazon API Gateway & AWS Lambda** - JavaScript executed in the browser sends and receives data from a public backend API built using Lambda and API Gateway.

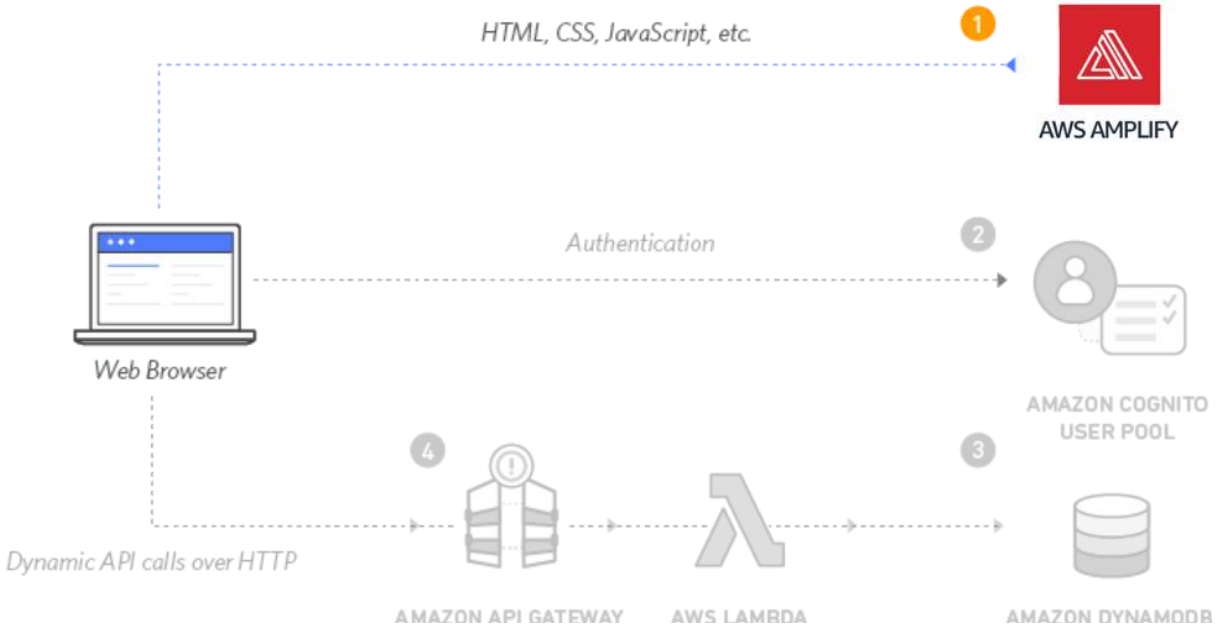
3. **Amazon Cognito** - Amazon Cognito provides user management and authentication functions to secure the backend API.

4. **Amazon DynamoDB** - Amazon DynamoDB provides a persistence layer where data can be stored by the API's Lambda function.

Design & Implementation

Step 1:- Static Web Hosting with Continuous Deployment

In this step I'll configure AWS Amplify to host the static resources for your web application with continuous deployment built-in. The Amplify Console provides a git-based workflow for continuous deployment & hosting of full stack web apps. In subsequent modules I'll add dynamic functionality to these pages using JavaScript to call remote RESTful APIs built with AWS Lambda and Amazon API Gateway.

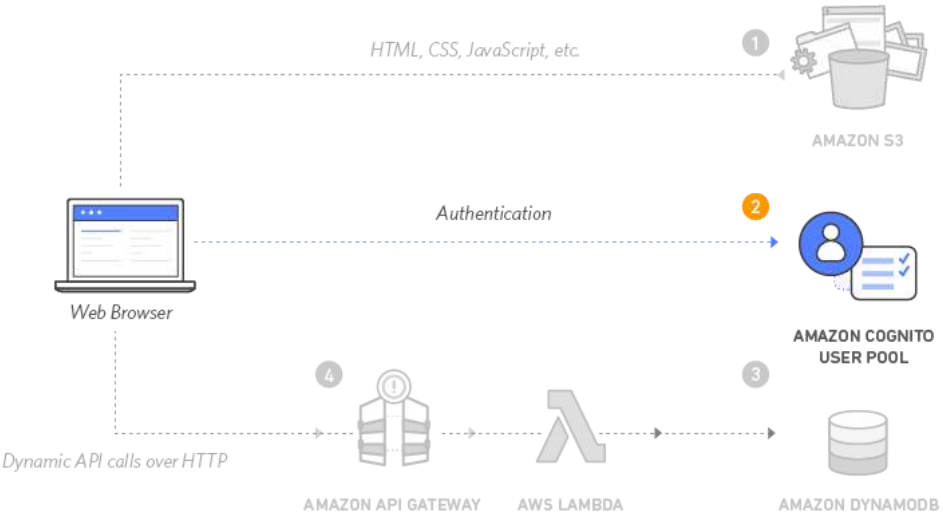


(Architectural Diagram)

The architecture for this step is very straightforward. All of the static web content including HTML, CSS, JavaScript, images and other files will be managed by AWS Amplify Console. The end users will then access the site using the public website URL exposed by AWS Amplify Console. We don't need to run any web servers or use other services in order to make this site available.

Step 2:- User Management

In this step, I'll create an Amazon Cognito user pool to manage your users' accounts. I'll deploy pages that enable customers to register as a new user, verify their email address, and sign into the site.

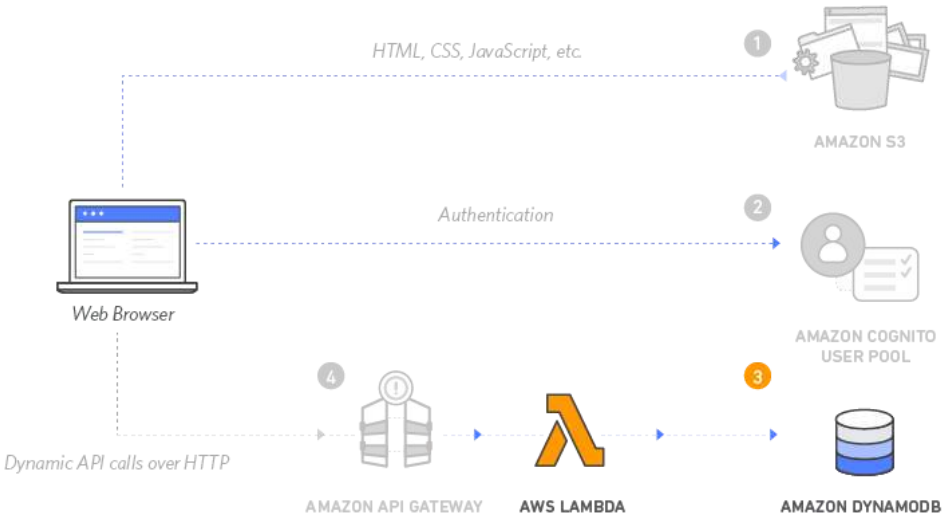


(Architectural Diagram)

When users visit this website they will first register a new user account. For the purposes of this project we'll only require them to provide an email address and password to register. However, I can configure Amazon Cognito to require additional attributes in this application. After users submit their registration, Amazon Cognito will send a confirmation email with a verification code to the address they provided. To confirm their account, users will return to your site and enter their email address and the verification code they received. We can also confirm user accounts using the Amazon Cognito console if you want to use fake email addresses for testing. After users have a confirmed account (either using the email verification process or a manual confirmation through the console), they will be able to sign in. When users sign in, they enter their username (or email) and password. A JavaScript function then communicates with Amazon Cognito, authenticates using the Secure Remote Password protocol (SRP), and receives back a set of JSON Web Tokens (JWT). The JWTs contain claims about the identity of the user and will be used in the next step to authenticate against the RESTful API I built with Amazon API Gateway.

Step 3:- Serverless Service Backend

In this step, I'll use AWS Lambda and Amazon DynamoDB to build a backend process for handling requests for your web application. The browser application that I deployed in the first module allows users to request that a ride be sent to a location of their choice. In order to fulfill those requests, the JavaScript running in the browser will need to invoke a service running in the cloud.

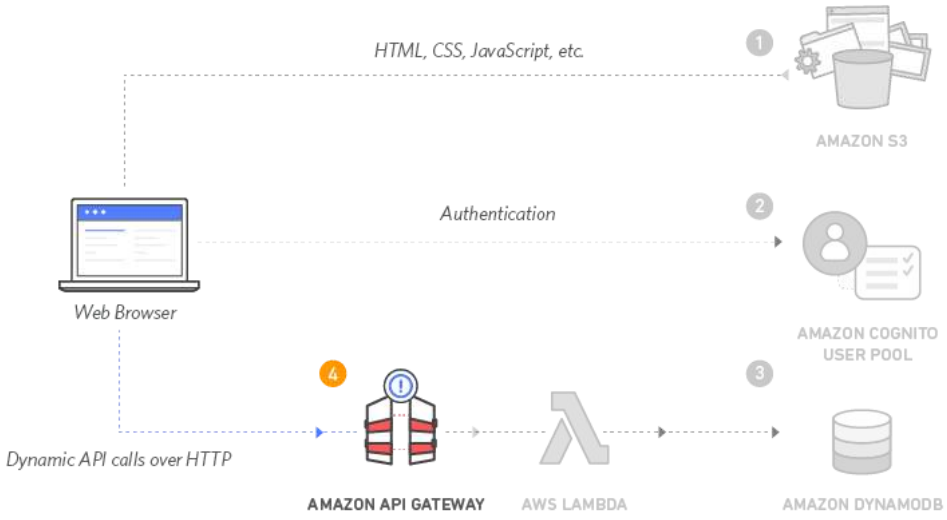


(Architectural Diagram)

I'll implement a Lambda function that will be invoked each time a user requests a ride. The function will select a ride from the fleet, record the request in a DynamoDB table and then respond to the front-end application with details about the ride being dispatched.The function is invoked from the browser using Amazon API Gateway.

Step 4:- RESTful APIs

In this step, I'll use Amazon API Gateway to expose the Lambda function that I built in the previous module as a RESTful API. This API will be accessible on the public Internet. It will be secured using the Amazon Cognito user pool I created in the previous module. Using this configuration I will then turn this statically hosted website into a dynamic web application by adding client-side JavaScript that makes AJAX calls to the exposed APIs.



(Architectural Diagram)

The diagram above shows how the API Gateway component I will build in this step integrates with the existing components I built previously. The grayed out items are pieces that I have already implemented in previous steps.The static website I deployed in the first step already has a page configured to interact with the API I'll build in this step. The page at /ride.html has a simple map-based interface for requesting a ride. After authenticating using the /signin.html page, the users will be able to select their pickup location by clicking a point on the map and then requesting a ride by choosing the "Request Ride" button in the upper right corner.

Project Scope

The main goal of this application is to use various kinds of technologies based on cloud to create a highly available, Scalable web application architecture that is fully serverless and hosted on cloud/ AWS to be more specific. This app will let users order rides.The services involved in this architecture are very crucial to build a serverless web app And by using such services and architecture we can build a fully functional serverless web app.