The FaceApp web application, designed for social media interactions, utilizes a Node.js for frontend and has no backend. The entire application is containerized using Docker and hosted on AWS. The deployment process incorporates continuous integration and continuous deployment (CI/CD) practices to ensure a smooth and automated workflow. Dependencies include testing libraries like Jasmine, Karma, Protractor, and tools such as TSLint for static analysis and Sass for CSS preprocessing. The pipeline also integrates monitoring through AWS CloudWatch, enhancing the overall development and deployment process.

Architecture

Frontend

Angular (Frontend): The application's frontend is built using Angular, providing a robust and dynamic user interface.

Backend (No Backend for this Application)

The application does not have a traditional backend. Instead, it relies on serverless or other backend-as-a-service (BaaS) solutions for any necessary backend functionality.

Deployment

Docker: The application is containerized using Docker, ensuring consistency and portability across different environments.

Hosting

AWS: The containerized application is hosted on AWS, leveraging cloud services for scalability and reliability.

Deployment Pipeline

Continuous Integration (CI)

Version Control:

Developers commit code changes to a version control system (e.g., Git).

Automated Build:

An automated build process is triggered upon code changes, compiling the Angular frontend and ensuring all dependencies are in order.

Static Code Analysis:

Static code analysis is performed using tools like codelyzer for Angular and TSLint for TypeScript, enforcing coding standards and identifying potential issues.

Unit Testing:

Jasmine and Karma are utilized for unit testing the Angular components, ensuring individual parts of the frontend function as expected.

Continuous Deployment (CD)

Continuous Deployment Trigger:

Successful completion of the CI process triggers the continuous deployment pipeline.

End-to-End Testing:

Protractor, an end-to-end testing framework, validates the functionality of the Angular application from a user's perspective.

CSS Preprocessing:

Sass is employed for CSS preprocessing, enhancing the styling capabilities of the Angular application.

Containerization:

The application is containerized using Docker, ensuring consistency and ease of deployment.

Deployment to AWS:

The Dockerized application is deployed to the AWS infrastructure, leveraging cloud services for hosting.

Monitoring:

AWS CloudWatch is integrated for monitoring, providing insights into metrics, logs, and performance indicators.

Dependencies

codelyzer (6.0.2): Performs static code analysis for Angular applications.

jasmine-core (3.8.0): Core library for Jasmine, used for unit testing.

jasmine-spec-reporter (7.0.0): Provides enhanced reporting for Jasmine tests.

karma (6.3.5): Test runner for JavaScript and TypeScript code.

karma-chrome-launcher (3.1.0): Launcher for Chrome browser in Karma tests.

karma-cli (2.0.0): Command-line interface for Karma.

karma-coverage-istanbul-reporter (3.0.3): Generates code coverage reports for Karma tests.

karma-jasmine (4.0.1): Integrates Jasmine with the Karma test runner.

karma-jasmine-html-reporter (1.7.0): HTML reporter for Jasmine tests in Karma.

protractor (7.0.0): End-to-end testing framework for Angular applications.

sass (1.32.13): CSS preprocessor for enhanced styling capabilities.

ts-node (10.1.0): TypeScript execution environment for Node.js.

tslint (6.1.3): Linter for TypeScript code.

typescript (4.3.5): TypeScript language compiler.

Pipeline Components

1. Source Code Repository

The source code for the FaceApp web application is stored in a version control system, such as Git. Developers push their changes to the repository, triggering the deployment pipeline.

2. Continuous Integration (CI)

The CI system is responsible for automating the build and testing processes whenever changes are pushed to the source code repository. The CI pipeline ensures the code quality and compatibility before proceeding to deployment.

CI Workflow:

Code Build: Pulls the latest code from the repository.

Dependencies Installation: Installs Node.js dependencies using npm.

Build and Test: Compiles the application code and runs automated tests.

Artifact Generation: Creates a Docker image containing the application and its dependencies.

Artifact Storage: Stores the Docker image in a container registry (e.g., Amazon ECR).

3. Continuous Deployment (CD)

The CD pipeline deploys the Dockerized FaceApp application to the AWS infrastructure.

CD Workflow:

Artifact Retrieval: Pulls the Docker image from the container registry.

Infrastructure Provisioning: Creates or updates the necessary AWS resources (e.g., EC2 instances, Load Balancers) using Infrastructure as Code (IaC) tools like AWS CloudFormation.

Container Deployment: Deploys the Docker image to the provisioned infrastructure.

Health Checks: Performs health checks to ensure the application is running correctly.

DNS Update: Updates DNS records to point to the new deployment.

4. Monitoring and Logging

The pipeline includes monitoring and logging components to ensure the health and performance of the FaceApp application.

Monitoring:

AWS CloudWatch: Monitors metrics such as CPU usage, memory utilization, and request latency.

Application Insights: Gathers insights into application behavior and performance.

Logging:

AWS CloudWatch Logs: Captures application logs for debugging and troubleshooting.

Pipeline Configuration

Environment Variables

The pipeline relies on environment variables for configuration. These variables include:

AWS_ACCESS_KEY_ID: AWS access key for deployment.

AWS_SECRET_ACCESS_KEY: AWS secret key for deployment.

AWS_REGION: AWS region for deployment.

DOCKER_REGISTRY_URL: URL of the Docker registry (e.g., ECR) for image storage.

ECR_REPOSITORY: Name of the repository in the Docker registry.

APPLICATION_PORT: Port on which the Node.js application will run.

DATABASE_URL: URL of the database for the FaceApp application.

Deployment Triggers

The deployment pipeline is triggered automatically on code pushes to the main branch of the source code repository. Manual triggers may also be configured for specific releases or updates.

Dependencies:

c-ares == 1.19.0

C-ares is a sub dependency within the NodeJs package.

Our faceapp application uses c-ares 1.19.0 NodeJs's sub dependency for asynchronous DNS requests. It is an asynchronous resolver library.

Libuv == 1.47.0

The libuv dependency is a multi-platform support library with a focus on asynchronous I/O. It is primarily developed for use by Node.js.

For our faceapp application, we utilize libuv version 1.47.0 to abstract non-blocking I/O operations, which allows Node.js to achieve high performance and handle a large number of concurrent connections. This version also provides cross-platform support for features such as networking, threading, and operating system-related functionality, making it an essential component of Node.js's event-driven architecture.

Llhttp == 16.19.0

The llhttp dependency is the http parser used by Node.js. LLHTTP (Low-Level HTTP) is a Node.js module that offers a lower-level interface for interacting with HTTP requests and responses.

In our faceapp application, LLHTTP version 16.19.0 bypasses the higher-level abstractions provided by the built-in http module, giving us more granular control over the HTTP protocol. This is beneficial for specific use cases where fine-grained control is necessary, such as: Customizing HTTP headers and bodies, Handling raw data, and Performance optimization.

Nghttp2 == 1.40.0

The nghttp2 dependency is a C library implementing HTTP/2 protocol. nghttp2 is an open-source implementation of the HTTP/2 protocol, designed to improve performance and efficiency compared to HTTP/1.1. It offers several key features such as Multiplexing: Allows multiple requests to be sent concurrently over a single TCP connection, reducing latency and improving resource utilization, Header compression: Uses HPACK header compression to reduce the size of HTTP headers, further enhancing performance, and Server push: Enables servers to proactively push resources to clients, potentially reducing the number of round trips required to load a page.

We use NGHTTP2 version 1.40.0 in our Node.js application, we typically wouldn't directly interact with it in JavaScript code. Instead, we would use a Node.js wrapper or binding that provides a JavaScript API for NGHTTP2. One such popular library is http2, which is included as part of Node.js core starting from version 8.8.0. Using the http2 module in Node.js allows us to leverage the features of the HTTP/2 protocol, such as header compression, multiplexing, and server push, in your applications. These features can help improve performance and efficiency, especially for applications that require low-latency communication or handle a large number of concurrent requests.

Openssl == 1.9.0

The openssl dependency is a fork of OpenSSL to enable QUIC. OpenSSL is toolkit for general-purpose cryptography and secure communication. Node.js currently uses the quictls/openssl fork, which closely tracks the main openssl/openssl releases with the addition of APIs to support the QUIC protocol. See maintaining-openssl for more information.

For our faceapp application, OpenSSL version 1.9.0 is used under the hood by the Node.js runtime to provide secure communication capabilities. However, we typically don't interact with OpenSSL directly in our Node.js application code. Instead, Node.js provides built-in modules such as crypto and tls, which utilize OpenSSL internally to provide

cryptographic and secure communication features. For example, the crypto module provides cryptographic functionality, such as encryption, decryption, hashing, and digital signature generation and verification. Where as the tls module provides an implementation of the TLS (Transport Layer Security) and SSL (Secure Sockets Layer) protocols for secure communication. We use it to create secure servers and clients for handling HTTPS requests, secure WebSocket connections, and more.

Undici == 5.28.2

Undici is a high-performance HTTP client built for Node.js. It's designed to be fast, efficient, and lightweight, making it well-suited for building scalable web applications that require making numerous HTTP requests.

In our application we utilize undici version 5.28.2 which is built on top of the HTTP parser in Node.js core and provides features like connection pooling, pipelining, and streaming. We install Undici using npm install command then we declare require function for instance const { Client } = require('undici');, after that we create a client instance such as const client = new Client('http://example.com'); and then make requests using client-request() const { body } = await client.request({ method: 'GET', path: '/api/data'}); and then we handle response for using await (const data of body) { console.log(data.toString());} then we finally close client using await client.close();