CoderByte Food Delivery
Backend Design and Architecture

Chris Nelson
cnelson0641@gmail.com
v1.0
8/16/2023

Overview of Requirements:

The CBFD (CoderByte Food Delivery) system backend is an AWS hosted application that accepts food orders from customers, processes payment, and passes the order to restaurants. The main functions of the application are:

- List restaurants, including by advanced filtering (distance, opening hours, cuisine, food item, etc.)
- Place orders for food items from a single restaurant
- Facilitate communication between the customer and the restaurant
- Process refunds
- Cancel orders
- Process payment
- Track order status

Hosting:

All infrastructure will be hosted on AWS for the following reasons:

- Fully-managed services: the services needed from AWS are fully-managed and do not require manual administration.
- Scalability: the services needed from AWS scale from proof-of-concept small deployments to large web-scale applications, with the help of the AWS CDN and their availability zones across the globe.
- Metrics: AWS CloudWatch provides metrics for all services to provide insight into trends and resource inefficiency.
- Tracking: AWS CloudTrail provides a single source for logging all actions across the different subsystems and users.
- Cost-efficiency: AWS Lambda functions are serverless, which means you only pay for the time the service is in use. For the initial ramping-up stage, where the workload is low, this will save a lot of money.

Technologies and Infrastructure Needed:

As mentioned above, all infrastructure will be hosted on AWS. This includes:

- AWS API Gateway: provides the API that the frontend (not covered here) will use to interact with backend services.

- AWS Lambda: executes the Docker container that has our business logic, in the form of a Java .jar file.
- AWS ECR: simple and easy to use Docker registry.
- AWS RDS: we will use a simple relational database like mysql to store our data.
- AWS CodePipeline: this CI/CD system will manage the process of automating the development, building, and deployment of changes.
- AWS CodeCommit: host our git repository.

Other Technologies:

Outside of AWS services above, we will use:

- Docker: easy to containerize applications to control the runtime environment, no matter where it is deployed.
- Terraform: to facilitate tracking and development of the infrastructure deployed, Terraform will be used. This allows easy deployment of staging and dev systems for testing, updates of production systems, and easy scaling.
- Maven: a Java build tool that manages dependencies, build artifacts, and unit testing.

APIs by User:

Customer:
- Order: create, get_status, request_cancellation
- Restaurant: get_by_filter
- Review: create

RestaurantAdmin:
- Order: create, get-status, update, cancel
- Restaurant: update

CbfdAdmin:
- Order: create, get_status, update, cancel
- Restaurant: update, get_by_filter
- Review: create, get_status, update, delete

Build Steps, Packing, and Deployment:

To facilitate automation of the build and deployment of the application, the following workflow will be available in a few flavors (dev, staging, production, etc.), all triggered manually for dev or automatically by staging and production when a branch is updated:

- Create .jar artifact: using maven (‘mvn package’), compile, create a .jar, and run tests.
- Build Docker image: using the Dockerfile checked into code, create a Java and Lambda compliant Docker image. This Docker image is the main artifact created.
- Push Docker image to ECR: docker push to the registry for access from Lambda.
- Update Lambda function: for production and staging, the Lambda function will need to be updated. For dev, a Lambda function should be created.

Library Classes to Create:

Restaurant:

- Attributes: is_open, is_accepting_orders, address, hours{}, cuisines[], food_items[], reviews[], rating, photos[]
- Functions: basic CRUD operations

Order:

- Attributes: status, Restaurant, food_items[], Payment
- Functions: basic CRUD operations, placeOrder(), getOrderStatus()

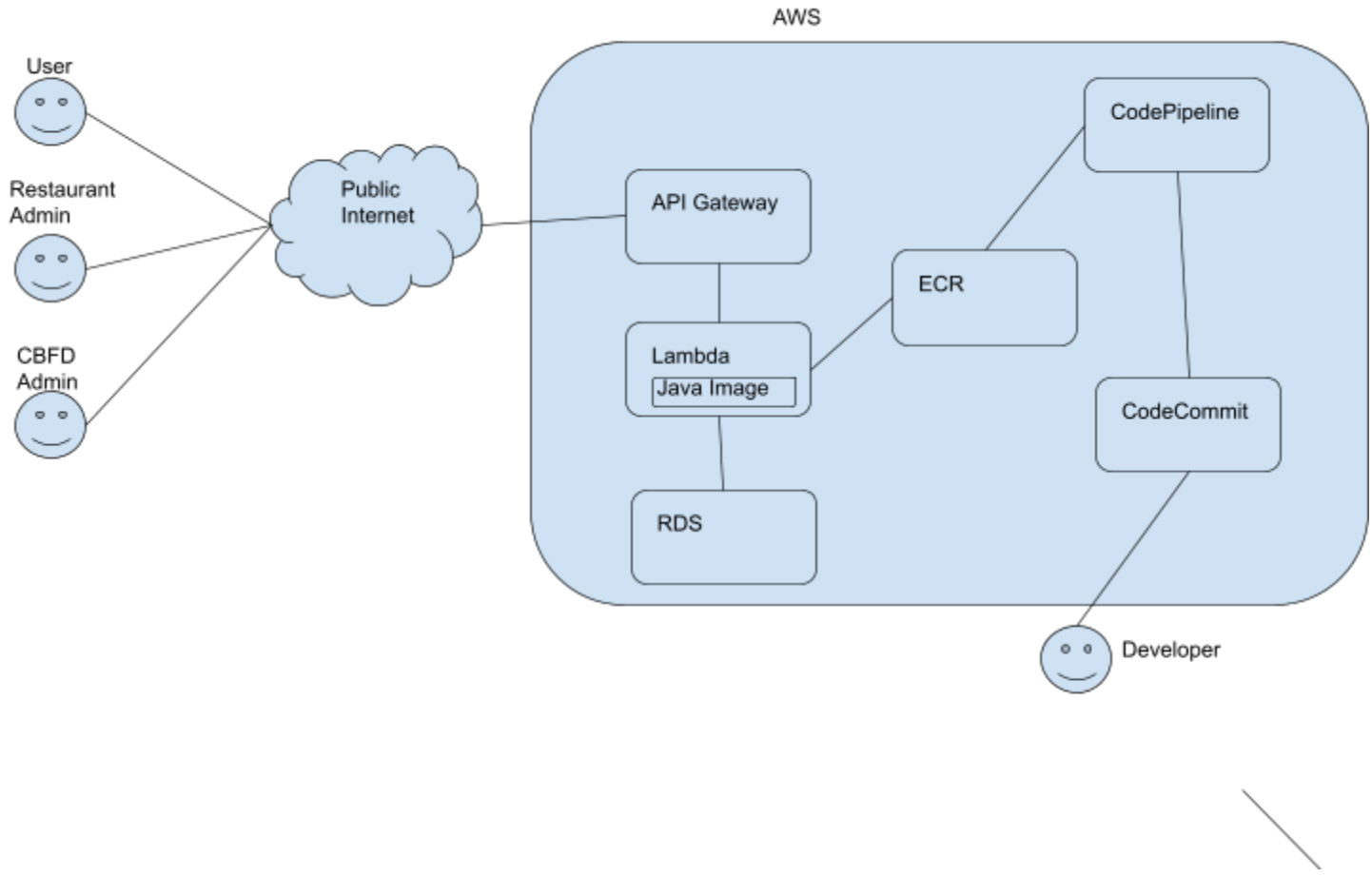
Payment:

- Attributes: square_api_obj
- Functions: basic CRUD operations, makePayment(), invokeRefund()

DB:

- Attributes: rds_object
- Functions: basic CRUD operations

Architectural Diagram:



Future Considerations:

Load balancing and content distribution:

- If we need to have higher availability, we can spin up more Lambda functions in different regions.
- We can put a load balancer (ELB) to distribute the load

Performance improvements and moving to a microservice-based application:

- As we scale many services (DB, logging, etc.) may need their own dedicated services.
- We can use either AWS built-in services, or develop each service in its own container and deploy it on its own.

Deploying on a k8s cluster:

- If each of the services are developed and run inside of its own docker container, we can create a Helm chart to map the entire application
- We can then deploy the application to EKS, and manage updates via the Helm chart.