Code Quality Review Report

Investment Grade

Erika McCluskey, Hayden Jin

Table Of Contents

Table Of Contents **2**
Introduction **3**
Design Patterns **3**
Model-View-Controller (MVC) 3
REST 4
API Gateway 4
Best practices **4**
Environment Variables 4
Types 5
Readability 5
Naming Convention 5
DRY 5
Improvements **6**

Introduction

The Investment Grade application has two working parts: the frontend and the backend. The backend is a REST API which is hosted on Amazon Web Services (AWS) with Amazon API Gateway and Amazon Lambda. The frontend is written in React Native. This short report will outline the design patterns, best practices, and other techniques we employed to maintain good code quality.

Design Patterns

Model-View-Controller (MVC)

Our system architecture is based on the MVC design pattern. This means that all interaction between the frontend and the backend passes through the controller beforehand. The controller (ie. our API) performs type checking on data it receives from the frontend via path parameters and on data it returns to the frontend. Type checking via the controller helps us ensure no bad/incorrect data is getting passed to the frontend or to the database. Using Typescript over Javascript made this possible which is why we chose to go with Typescript. In addition, we used an object-relational mapping tool called Sequelize to make calls to our database. This is also on the controller side and helps protect us against SQL injection attacks.

REST

REST stands for representational state transfer and is an architectural design pattern. The REST pattern allows our frontend and backend to be implemented independently. This is described as statelessness and means that one entity does not need to know the state of the other. Not only is this modularity good for separation of concerns (frontend vs. backend), it also makes our backend quicker and more scalable. Our REST API uses HTTP as the application protocol which is quite standard for REST APIs.

API Gateway

Our backend is a REST API that is hosted on AWS and uses the API Gateway service. While this is the name of the AWS service, it is also the name of a design pattern. The API gateway design pattern is a single entry point for backend APIs. Each function within the API has its own public endpoint which expands upon that single entry point URL. The API gateway routes incoming requests to the correct endpoint in the API. Using API gateway is useful for scalability and high availability which is what we were striving for when designing our application. It is very simple to add new endpoints to the API and the API is always on and listening for incoming requests.

Best practices

Environment Variables

All of our sensitive data such as API keys and authentication data is stored in an environment file and ignored by source control. This is to prevent anyone who should not have access to our services to take this data form GitHub. Nowadays, there exist

programs that clone all new repos added to GitHub specifically to collect data such as API keys and passwords; protecting this data is important.

Types

Our backend has a type defined for each model and the data that gets returned to the frontend must be of this type. This helps with unexpected data types being returned to the frontend and potentially crashing our app. In addition, it keeps everything consistent and predictable given that every single endpoint related to a model will return exactly the same type every time.

Readability

Our code contains little to no comments as we prioritized naming our variables with good names that give context as to what they do. Similarly, we tried to name our functions as best as possible to make the code more readable. This was critical for our team given we all had the same responsibilities in terms of coding which means we often worked on the same files.

Naming Convention

All of our variables and functions use the camelcase naming convention.

DRY

As best as possible, we tried to abide by the Don’t Repeat Yourself principle. We did this by extracting the logic for commonly repeated functions (such as fetching data from the API) into separate files and exporting those functions.

```
export const postRequest = async (
 route: string,
 body: string | undefined = undefined,
) => {
 try {
   const postResult = await fetch(`${apiBaseURL}/${route}`, {
     method: 'POST',
     headers: {
       Accept: 'application/json',
       'Content-Type': 'application/json',
     },
     body: body,
   });
   return postResult;
 }
```

The code block above shows an example of this. Instead of having this function logic everywhere that calls a POST request to our API, we wrote this function as an exportable function that is used by multiple files. We did this with many functions that repeated themselves across the app.

Improvements

While we did try our best to stick to coding best practices and employ design patterns, there is a lot of room for improvement. Our error handling if an asynchronous function fails is quite poor as we are not handling unsuccessful promises. While our API is rather reliable, every function that calls the API should handle the case that the function call

fails. In addition, there are some places in the app where functions call other functions that call other functions and that may make some logic slightly difficult to understand. That being said, these are refactors that are in the plans for future work on the app.