Code Testing Report

Investment Grade

Erika McCluskey, Hayden Jin

Table Of Contents

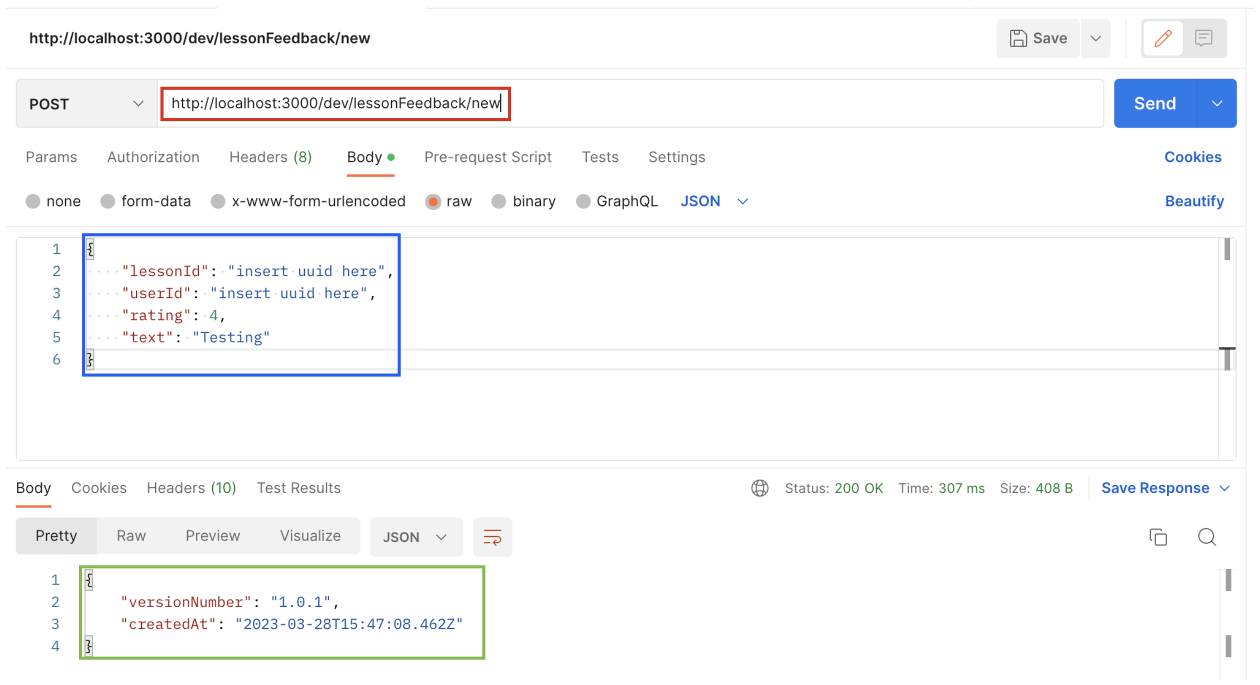
Introduction

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. The testing for each was done differently.

Backend Testing

For the REST API, the primary tools our team used to perform testing were Postman and AWS CloudWatch. Our API consists of multiple endpoints that are accessed by the frontend to fetch the necessary data to display on the frontend or to perform database updates. Endpoints are simply components of an API and each endpoint is accessed through a different path (or URL).

REST is a client-service architecture based on the request/response design. That means, the API listens for a request to then return a response. Prior to deploying the API to AWS, we ran the API locally on our machines to test the endpoints. In order to test our endpoints, we used Postman. Postman is a platform specifically designed to test APIs. Sending requests, such as POST or PUT requests with bodies, is very simple with their interface.


http://localhost:3000/dev/lessonFeedback/new

POST http://localhost:3000/dev/lessonFeedback/new

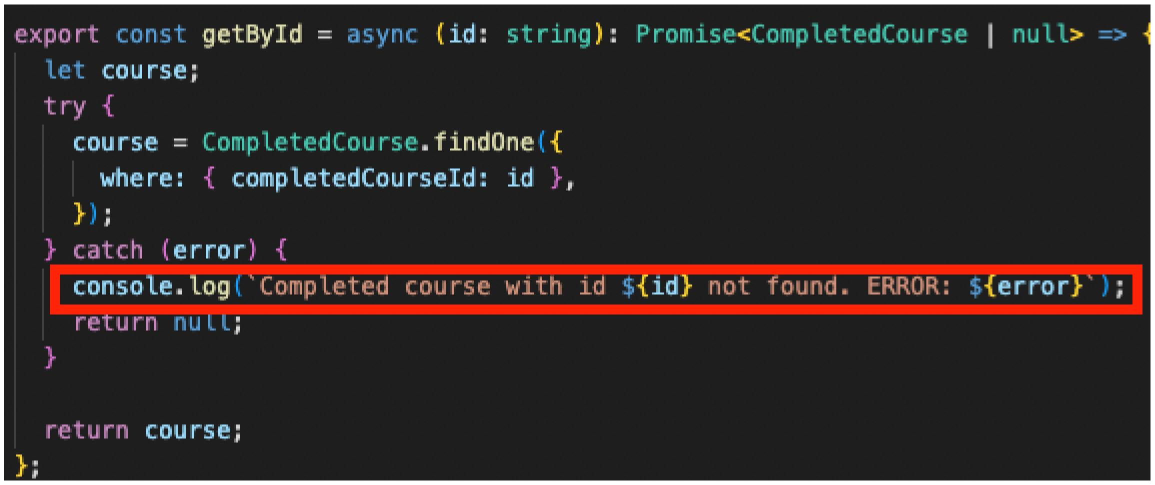
```
{
    "lessonId": "insert uuid here",
    "userId": "insert uuid here",
    "rating": 4,
    "text": "Testing"
}
```

Status: 200 OK Time: 307 ms Size: 408 B

```
{
    "versionNumber": "1.0.1",
    "createdAt": "2023-03-28T15:47:08.462Z"
}
```

Figure 1. Postman Interface

Shown in Figure 1 is the interface of Postman. In red is the endpoint to test with the type of request to the left (ie. GET, POST, PUT, etc.), in blue is the optional request body in JSON format, and in green is the response returned. The response also has a status which is helpful to determine the cause of errors. For example, status 502 bad gateway indicates that the URL cannot be found: either there's a mistake in the URL, it's the wrong type of request, or something is wrong with the router. Every endpoint added to our API was tested this way prior to deploying it to AWS. Once the endpoint was tested and returned the desired response, the endpoint was deployed to the AWS REST API.



```
export const getById = async (id: string): Promise<CompletedCourse | null> => {
  let course;
  try {
    course = CompletedCourse.findOne({
      where: { completedCourseId: id },
    });
  } catch (error) {
    console.log(`Completed course with id ${id} not found. ERROR: ${error}`);
    return null;
  }

  return course;
};
```

Figure 2. Output to the console

A useful testing strategy we employed was to output to the console in case of errors (shown in Figure 2). When testing the endpoints locally, the message is simply outputted to the terminal on the machine. However, when the endpoint errors out on AWS, the error is outputted to CloudWatch. CloudWatch is an AWS service to monitor and troubleshoot systems such as lambda functions (ie. our REST API). Since code does not always act the same in the production environment than it does in the dev environment, having output messages with the specific error message when an endpoint fails makes finding errors a lot easier. The alternative is to attempt to find the error by searching through the backend code for what could have gone wrong.

Frontend Testing

Investment Grade runs through React Native which is a framework that enables cross-platform development. The way our team approaches testing was to have one person in charge of iOS and another person in charge of Android. Since React Native allows hot refreshing, we ran the emulators as we were writing code to see the effects.

Similar to testing the API by outputting specific error messages to the console, having the application run while we were writing code made it quite easy to pinpoint error locations in the code since it was obvious that if it broke suddenly, it was due to something we had just added. To ensure the correct data structures/content, we used the technique to output data to the console and make sure it contains everything we think it contains. This was very helpful to identify and fix undefined data structures as these often cause the application to crash if left undefined.

In terms of our testing plan, we used an Excel spreadsheet with a list of tests that needed to pass before deploying to the app store. As soon as we added a new feature, we added the list of tests for that feature to the spreadsheet and performed testing. That being said, we tested our application throughout the entire development process; we did not wait until the end to perform tests. Choosing this approach made it a lot easier to fix bugs since we would be testing small chunks at a time which made it easier to pinpoint the location of the bug.