IT 314

Software Engineering

PUBLIC RESTAURANT CONTROLLER UNIT TESTING

Fork & Feast

Group 28

➢ Tool Used:

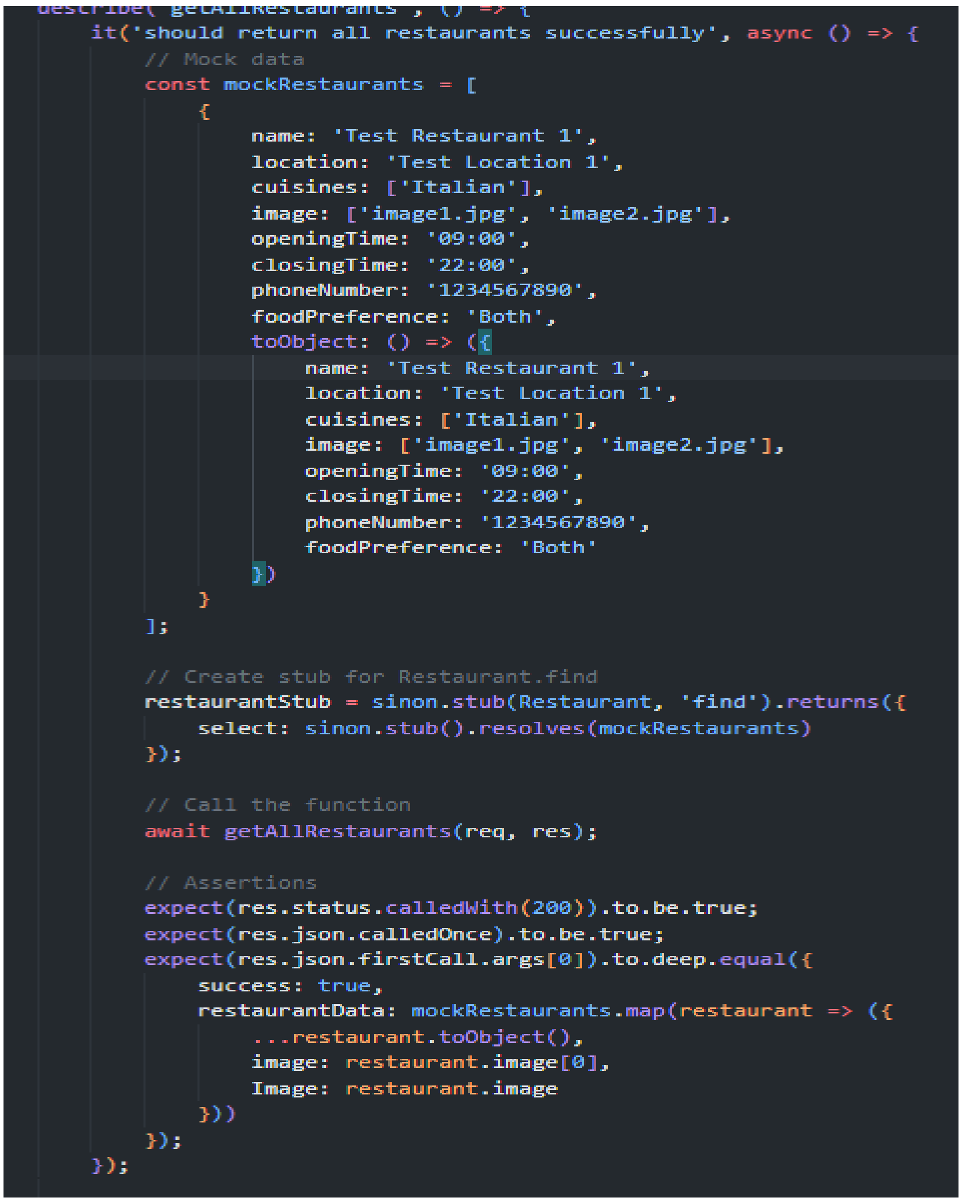
Mocha @10.8.2
Chai @4.3.4
Other – Sinon @19.0.2

Link to the source code: publicRestaurantController
Link to the test file: publicRestaurant_test

1. Get All Restaurant

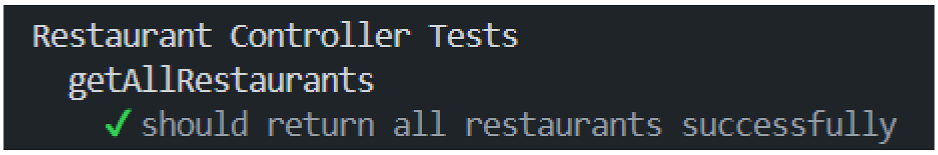
The getAllRestaurants function retrieves all restaurant data from a MongoDB database using Mongoose. It queries the Restaurant collection and selects specific fields such as name, location, cuisines, and others. The function then formats the data, ensuring the first image from the image array is included as image, while the full array is preserved as Image. Finally, it sends the transformed data as a JSON response with a success status. If any error occurs, it returns a failure response with a relevant message.

a) Should return all restaurants successfully


```
describe('getAllRestaurants', () => {
    it('should return all restaurants successfully', async () => {
        // Mock data
        const mockRestaurants = [
            {
                name: 'Test Restaurant 1',
                location: 'Test Location 1',
                cuisines: ['Italian'],
                image: ['image1.jpg', 'image2.jpg'],
                openingTime: '09:00',
                closingTime: '22:00',
                phoneNumber: '1234567890',
                foodPreference: 'Both',
                toObject: () => ({
                    name: 'Test Restaurant 1',
                    location: 'Test Location 1',
                    cuisines: ['Italian'],
                    image: ['image1.jpg', 'image2.jpg'],
                    openingTime: '09:00',
                    closingTime: '22:00',
                    phoneNumber: '1234567890',
                    foodPreference: 'Both'
                })
            }
        ];

        // Create stub for Restaurant.find
        restaurantStub = sinon.stub(Restaurant, 'find').returns({
            select: sinon.stub().resolves(mockRestaurants)
        });

        // Call the function
        await getAllRestaurants(req, res);

        // Assertions
        expect(res.status.calledWith(200)).to.be.true;
        expect(res.json.calledOnce).to.be.true;
        expect(res.json.firstCall.args[0]).to.deep.equal({
            success: true,
            restaurantData: mockRestaurants.map(restaurant => ({
                ...restaurant.toObject(),
                image: restaurant.image[0],
                Image: restaurant.image
            }))
        });
    });
```

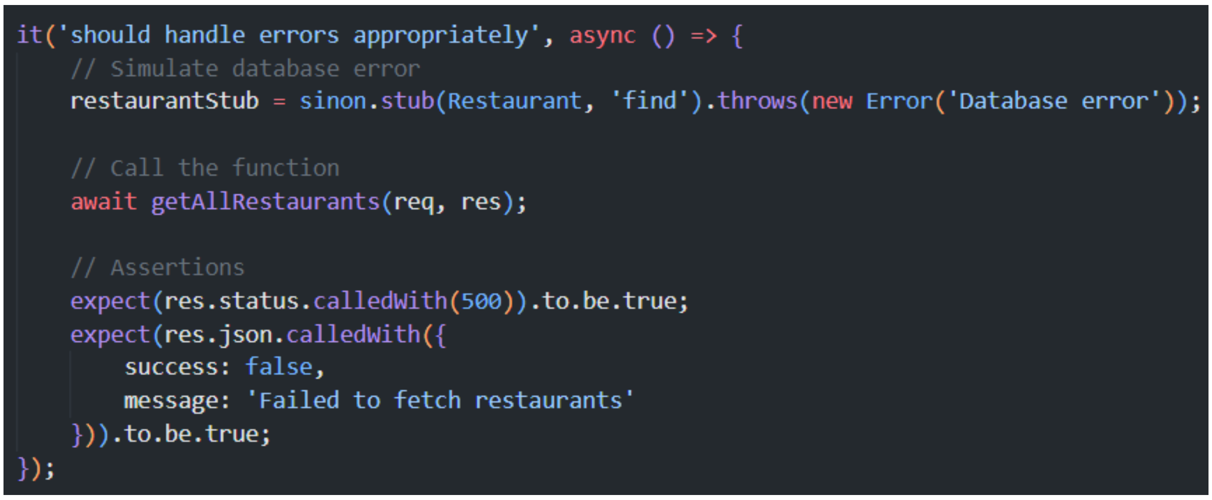
The test ensures that the getAllRestaurants function retrieves restaurant data correctly from a mocked database query using Restaurant.find. It checks that the function formats the data properly by including the first image as image and the full image array as Image. The test also verifies that a 200 status code is returned, along with a JSON response containing success: true and the correctly structured restaurantData. This validates the function's ability to handle and transform the data as expected.

This test cases is successfully covered which is confirmed with the result shown in the terminal.



```
Restaurant Controller Tests
  getAllRestaurants
    ✓ should return all restaurants successfully
```

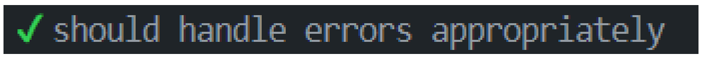
b) Should handle errors correctly



```
it('should handle errors appropriately', async () => {
    // Simulate database error
    restaurantStub = sinon.stub(Restaurant, 'find').throws(new Error('Database error'));

    // Call the function
    await getAllRestaurants(req, res);

    // Assertions
    expect(res.status.calledWith(500)).to.be.true;
    expect(res.json.calledWith({
        success: false,
        message: 'Failed to fetch restaurants'
    })).to.be.true;
});
```

The test ensures that the getAllRestaurants function handles errors appropriately when the database query fails. It uses a Sinon stub to simulate an error in the Restaurant.find method and verifies that the function responds with a 500 status code. Additionally, it checks that the JSON response contains success: false and an appropriate error message, "Failed to fetch restaurants". This validates the function’s error-handling mechanism and its ability to provide a clear response to the client in case of failures.

This is also confirmed by the confirmation message in the terminal.



```
✓ should handle errors appropriately
```

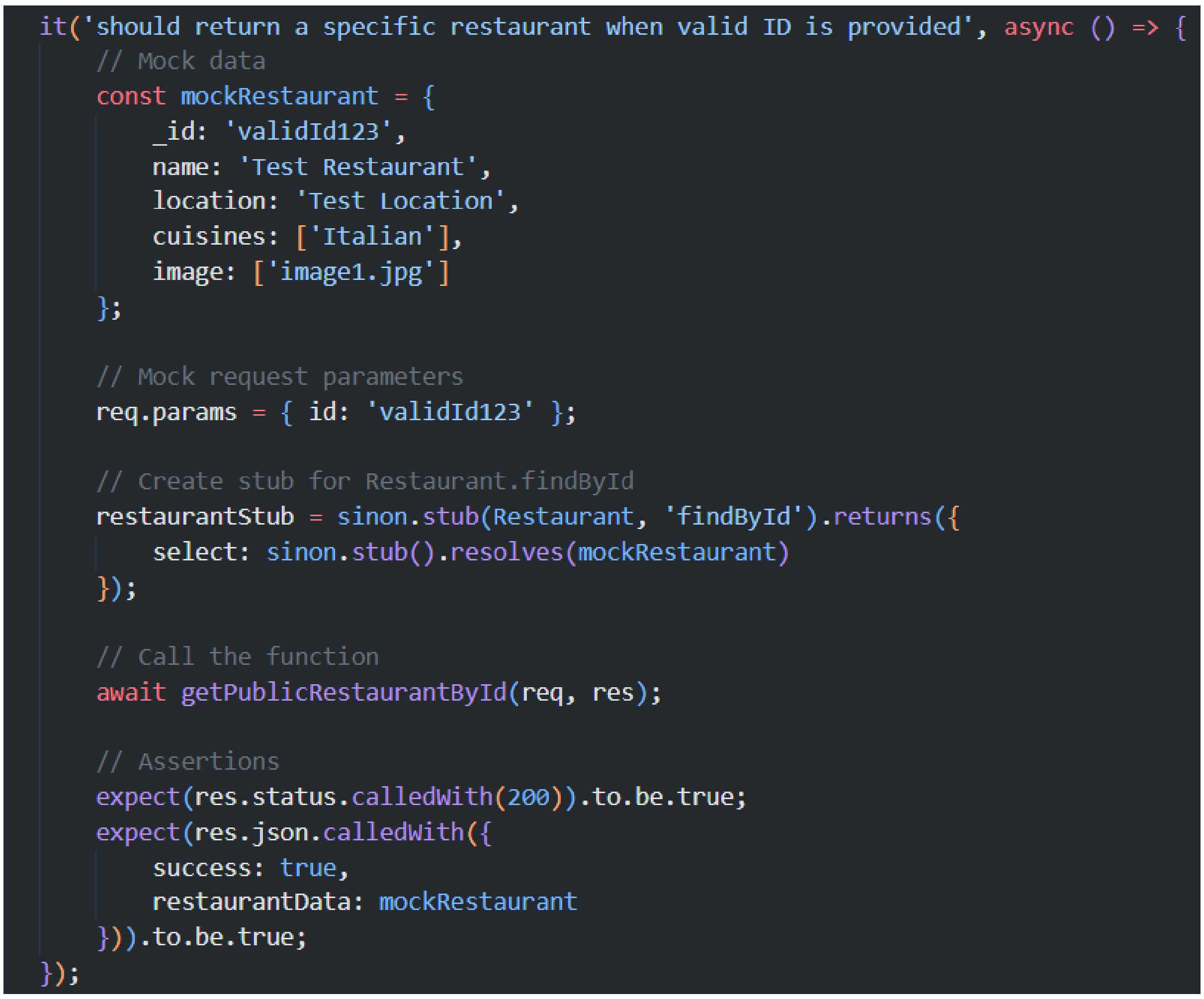
2. Get Public Restaurant by ID

The getPublicRestaurantById function retrieves public details of a restaurant based on its unique ID from the request parameters. It queries the database using the findById method while excluding sensitive data such as ownerId to ensure privacy.

If the restaurant is not found, the function responds with a 404 status code and a message stating, "Restaurant not found". If the restaurant is found, it sends a 200 status code along with the restaurant's data in a JSON response.

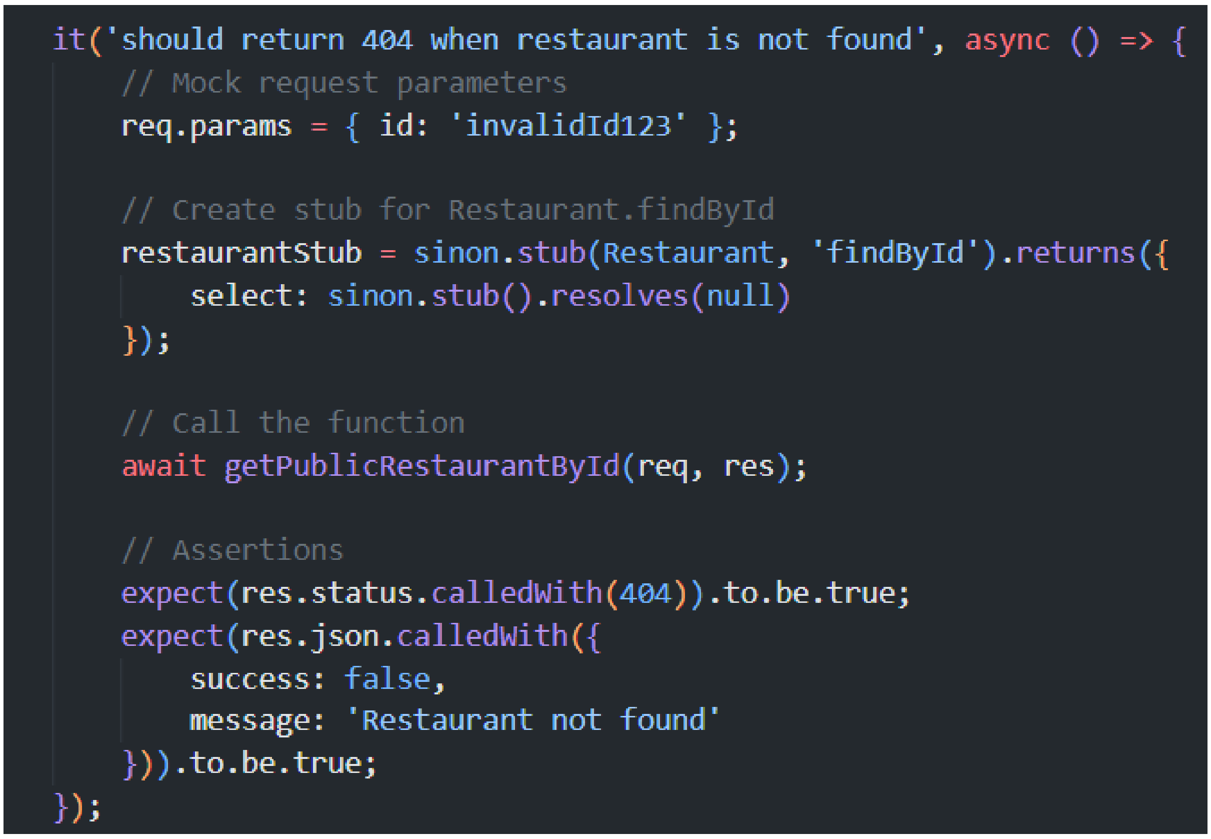
In case of any error during the process, the function logs the error and returns a 500 status code with the message, "Failed to fetch restaurant details".

a) Should return a specific restaurant when valid ID provided



```
it('should return a specific restaurant when valid ID is provided', async () => {
    // Mock data
    const mockRestaurant = {
        _id: 'validId123',
        name: 'Test Restaurant',
        location: 'Test Location',
        cuisines: ['Italian'],
        image: ['image1.jpg']
    };

    // Mock request parameters
    req.params = { id: 'validId123' };

    // Create stub for Restaurant.findById
    restaurantStub = sinon.stub(Restaurant, 'findById').returns({
        select: sinon.stub().resolves(mockRestaurant)
    });

    // Call the function
    await getPublicRestaurantById(req, res);

    // Assertions
    expect(res.status.calledWith(200)).to.be.true;
    expect(res.json.calledWith({
        success: true,
        restaurantData: mockRestaurant
    })).to.be.true;
});
```

This test checks if the getPublicRestaurantById function returns the correct restaurant when a valid ID is provided. It mocks the restaurant data and the Restaurant.findById method to simulate fetching the restaurant details. The test asserts that the response status is 200 and the returned JSON contains the correct restaurant data with success: true.

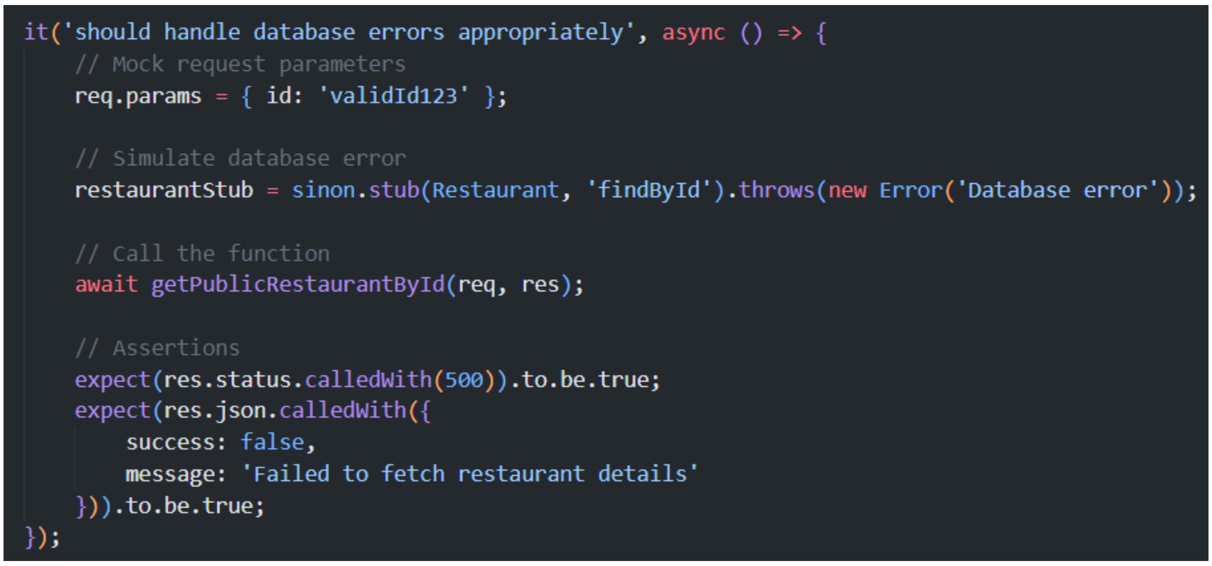


```
it('should return 404 when restaurant is not found', async () => {
    // Mock request parameters
    req.params = { id: 'invalidId123' };

    // Create stub for Restaurant.findById
    restaurantStub = sinon.stub(Restaurant, 'findById').returns({
        select: sinon.stub().resolves(null)
    });

    // Call the function
    await getPublicRestaurantById(req, res);

    // Assertions
    expect(res.status.calledWith(404)).to.be.true;
    expect(res.json.calledWith({
        success: false,
        message: 'Restaurant not found'
    })).to.be.true;
});
```

b) Should return 404 when restaurant is not found

This test checks if the getPublicRestaurantById function correctly returns a 404 error when a restaurant is not found. It simulates an invalid restaurant ID ('invalidId123') in the request parameters. The Restaurant.findById method is stubbed to return null, indicating no restaurant was found. The test then asserts that the response status is 404 and the JSON response contains the error message 'Restaurant not found'.

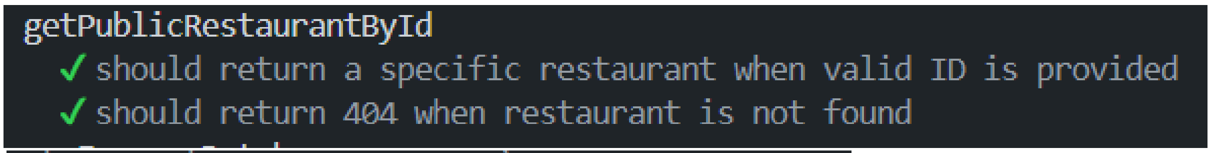
c) Should handle database errors appropriately



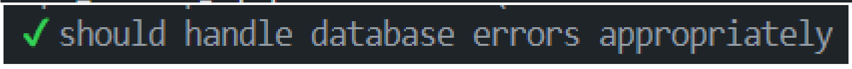
```
it('should handle database errors appropriately', async () => {
    // Mock request parameters
    req.params = { id: 'validId123' };

    // Simulate database error
    restaurantStub = sinon.stub(Restaurant, 'findById').throws(new Error('Database error'));

    // Call the function
    await getPublicRestaurantById(req, res);

    // Assertions
    expect(res.status.calledWith(500)).to.be.true;
    expect(res.json.calledWith({
        success: false,
        message: 'Failed to fetch restaurant details'
    })).to.be.true;
});
```

This test checks if the getPublicRestaurantById function handles database errors correctly. It simulates a database error by making the Restaurant.findById method throw an error. The test asserts that, when the error occurs, the function responds with a 500 status code and a JSON message 'Failed to fetch restaurant details', indicating that the system gracefully handles database issues.

These test cases are confirmed by the confirm message in the terminal:



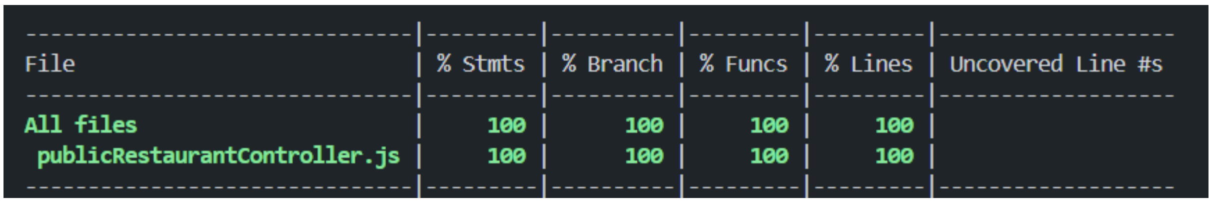
```
getPublicRestaurantById
  ✓ should return a specific restaurant when valid ID is provided
  ✓ should return 404 when restaurant is not found
```



```
✓ should handle database errors appropriately
```

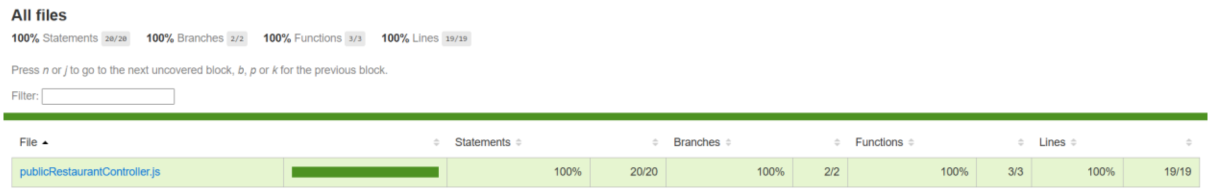
- Coverage Report

  For this we used the “nyc --reporter=text” command which gave us the following output:



```
-------------------------------|---------|----------|---------|---------|-------------------
File                           | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------------|---------|----------|---------|---------|-------------------
All files                      |     100 |      100 |     100 |     100 |
 publicRestaurantController.js |     100 |      100 |     100 |     100 |
-------------------------------|---------|----------|---------|---------|-------------------
```

For in depth report we used “nyc --reporter=lcov” which gave us an html file highlighting the number of times a particular line ran during the entire testing.



All files

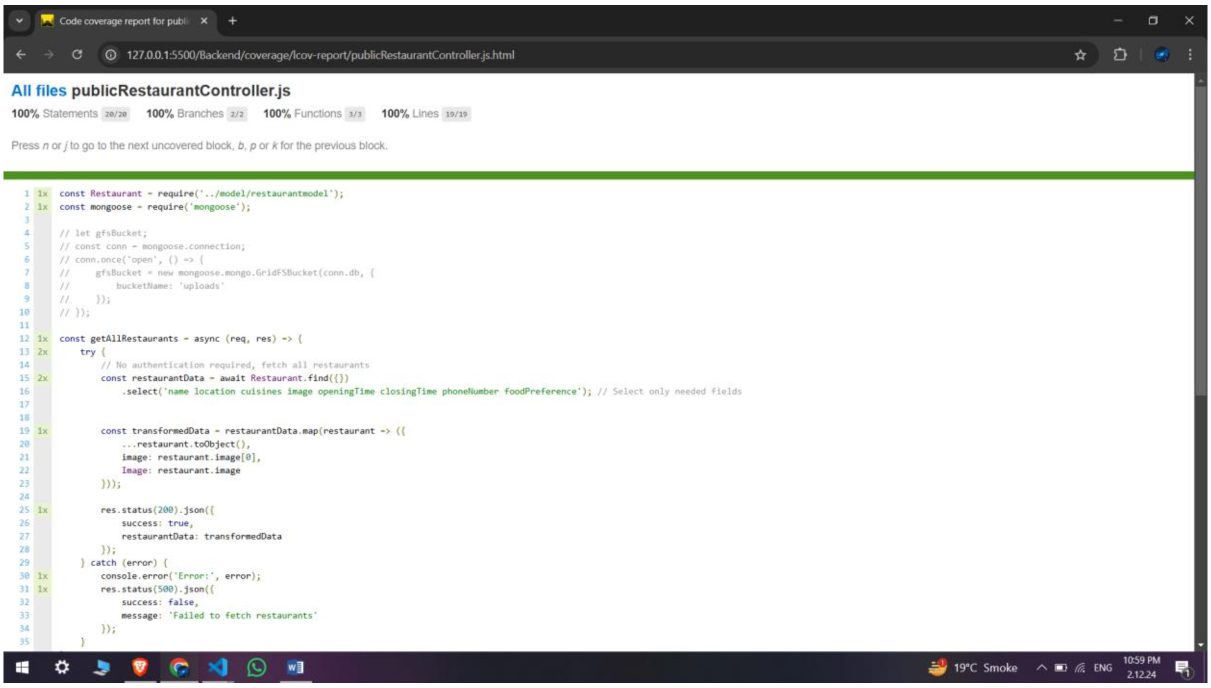
100% Statements 20/20 **100%** Branches 2/2 **100%** Functions 3/3 **100%** Lines 19/19

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines	
publicRestaurantController.js	🟩	100%	20/20	100%	2/2	100%	3/3	100%	19/19

In detail it looks like this:-



Code coverage report for publi
127.0.0.1:5500/Backend/coverage/lcov-report/publicRestaurantController.js.html
19°C Smoke
ENG
10:59 PM
2.12.24

All files publicRestaurantController.js

100% Statements 20/20 **100%** Branches 2/2 **100%** Functions 3/3 **100%** Lines 19/19

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1x  const Restaurant = require('../model/restaurantmodel');
1x  const mongoose = require('mongoose');

    // let gfsBucket;
    // const conn = mongoose.connection;
    // conn.once('open', () => {
    //     gfsBucket = new mongoose.mongo.GridFSBucket(conn.db, {
    //         bucketName: 'uploads'
    //     });
    // });

1x  const getAllRestaurants = async (req, res) => {
2x      try {
            // No authentication required, fetch all restaurants
2x          const restaurantData = await Restaurant.find({})
                .select('name location cuisines image openingTime closingTime phoneNumber foodPreference'); // Select only needed fields


1x          const transformedData = restaurantData.map(restaurant => ({
                ...restaurant.toObject(),
                image: restaurant.image[0],
                Image: restaurant.image
            }));

1x          res.status(200).json({
                success: true,
                restaurantData: transformedData
            });
        } catch (error) {
1x          console.error('Error:', error);
1x          res.status(500).json({
                success: false,
                message: 'Failed to fetch restaurants'
            });
        }
```

Note: The detailed HTML report can be seen from the index.html in the coverage part of our project folder.