ExpressTest

Automatic Test Case Generation for Express.JS Applications

In General

I created a Node.JS CLI application called “ExpressTest”. ExpressTest is used to analyze Express.JS applications and create a set of base test cases which the developer can modify to suit their needs. ExpressTest generates test cases for three different strategies: Random Input, Boundary Scenario, and Invalid Input. ExpressTest then generates test files which can be run by the library Jest and analyzed for code coverage.

The Problem

Rest APIs can take a very long time to develop, and for some users, writing test cases which have good code coverage can take a very long time. Automating part of the process in generating test cases in Express applications will save the developer time. However, it is extremely difficult to fully automate the process. This is due simply to the fact that programs are not good at determining proper output of code. There also exists the possibility of the code being incorrect, and if the system were able to programmatically determine the proper outcome from the input code, it would be incorrect anyways.

The Scope

I aimed to create an application that *assists* in the creation of test cases, and in no way serves as a replacement for defining your own test cases. This is for a few reasons:

- The code can never determine the full context of the application.
- As discussed, computers are bad at estimating or predicting proper outcome of unrun code.
- If this project were to be larger, it would take longer than the deadline for this project.
- Only query parameters are supported (i.e. example.com/?query=value)

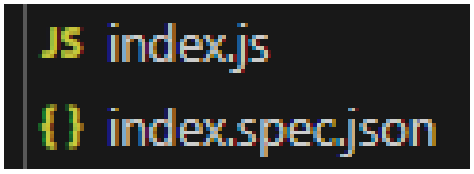

Figure 1. Index.js and Corresponding Specification File

The Process

Before generating any test cases, Specification files need to be set up. The specification for each test must be specified in a .spec.json file (Figure 1). These files must be created as neighbors and named the same as any .js or .ts file where there are Express.JS route definitions present. Inside of each Specification file, every route that the user wants tested must be defined. Figure 2 shows an example configuration. ExpressTest generates test cases for three different strategies: Random Input, Boundary Scenario, and Invalid Input. Each test scenario performs the following:

- **Random Input:** Randomizes the inputs of the API endpoint while maintaining type and specification (string length, number bounds, etc.) accuracy.
- **Invalid Input**: Randomizes the inputs of the API endpoint while explicitly not maintaining valid type.
- **Boundary Scenario**: Analyzes the limits allowed by specification. For example, if a number is only valid between 0 and 100, the Boundary Scenario tests 0 and 100.

Figure 2. Example Specification File

After generating these test scenarios, ExpressTest saves them to their corresponding directory in the target project and automatically runs an initial test. If any test fails, it outputs to the console. An example of an output file can be found in Figure 3.

```
// RANDOM INPUT SCENARIOS

it('should handle random input scenario', async () => {
    try {

        const res = await request(app).get('/users/')
            .query({
                id: "accomplished impolite guest"
            });


        // Customizable assertions
        expect(res.statusCode).toBe(200);
        // Add more specific assertions based on expected response

    } catch (error) {
        // Error handling and logging
        console.error('Test failed');
        throw error;
    }
});
```

Figure 3. Example Test CaseOutput of ExpressTest

ExpressTest utilizes Supertest as a wrapper to analyze the output of Express.JS routes. Supertest is needed to analyze the code coverage of each test scenario. Code coverage is included as a feature of Jest, a popular unit testing library. By running the test manually from the target project, the total code coverage is shown as in Figure 4.

```
-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |   84.61 |       50 |      75 |   84.61 |
 express-app       |      80 |       50 |      50 |      80 |
  index.js         |      80 |       50 |      50 |      80 | 19-20
 express-app/route |     100 |      100 |     100 |     100 |
  route.js         |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-------------------

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.856 s
Ran all test suites.
Done in 2.82s.
```

Figure 4. Code Coverage Report

The Results

While the time saved by the developer can not be quantified, the benefits of using this program for hobby projects and smaller applications is palpable. Firstly, it reduces the total amount of manual work needed to be done by a developer - just create endpoint specifications and use the baseline test cases generated to create unit tests that have maximum coverage. While this package is not perfect, it achieves its goal of assisting developers in this tedious process.

The Future

In the future, this program could be expanded to generate more precise and code-informed test cases. Currently, it only generates random inputs. However, using the TypeScript library to analyze each node within a function, it is possible to find equalities (i.e. x == 2) and use those to inform the random input generator to result in higher code coverage or possibility of detecting bugs. Additionally, route parameters can be supported instead of just query parameters. If route parameters could be included, type support can be extended from just numbers and strings if the target is a TypeScript project.