Test Plan Report

By

Burak KATI

Ömer Mekin KARTAL

Ege TELLİ

Özgür ÖZERTURAL

Contents

1. Introduction

This document contains the explanation of the methodologies we are going to use until the end of the implementation process. We are going to test the components of our web application from the backend to the front end. The backend components that we are going to test are mainly the microservices that handles the connectivity between the frontend to backend, then backend to blockchain through API endpoints. Also, we are going to test our smart contracts on an online Solidity IDE called 'Remix'. Further explanations are below.

2. Workload Distribution

For now, we are separated into two groups for backend development and frontend development. Ege and Burak are responsible for backend development, Özgür and Mekin are responsible for frontend development and design. Once the minimal functionality is implemented in both sides – such as Add Wallet button connects a user to our system or Create Campaign Button takes input from the user and stores it in the database- we all are going to dive into smart contract development with solidity. For testing, same groups are responsible for the parts that they are developing.

3. Testing Schedule and Testing plan

Since we are using compartmentalized architectures for both frontend and backend development, we test our code as long as it is implemented. We try to abide by Agile Testing methodology which concentrates on delivering a working product first, then re-delivering according to the changes requested until there is no change to implement.

Of course, this comes with risks such as:

- Non-clear end
- Difficulties in measurements of the software
- Limited documentation
- Poor resource planning

But the advantages of Agile Testing such as:

- Ability to test the component as soon as it is implemented
- Testing is crucial before deployment
- Testers are involved from the beginning of implementation

overcomes the disadvantages of it, since we have to implement a fully functional web application in a limited time.

4. Testing Environment and Tools

We are going to test our application's components in several environments such as, Visual Studio Code, Remix and Postman, QTP/UFT.

4.1 Backend

- Visual Studio Code IDE enables us to utilize the debugger to test the code manually in data centric scope. (i.e does the variables contain the data it is supposed to hold?)
    - NPM allows us to run specific scripts while running the app such as 'npm start' or 'npm test' which runs the specified command that is triggered by the called script (i.e 'npm start' triggers the command 'node app.js' or any other specified file), allowing us to automate the testing process.
    - We will utilize NPM's command structure while running our automated tests and monitor our test cases.

  We will test our RESTful API endpoints using REST Clients such as Postman and an extension called "Rest Client" in Visual Studio Code.

- Rest Client:
    - It is an extension that let's us send http requests to a specified URL inside of Visual Studio Code.
    - It is a light weight bare bones software that can be imported to the editor and offers minimal functionality like sending http requests and editing request headers.
- Postman:
    - It is a standalone desktop app that has similar functionality to the Rest Client extension with some additional functionalities such as saving request history, saving response history, switching workspaces for testing different API endpoints.

We mainly prefer Rest Client because it provides easy to test the API responses

- JEST:
    - Jest is a javascript unit testing framework which is designed for ensuring the correctness of javascript codes and monitor the errors.
    - We will use it to test our methods and functions to ensure they are working as intended.

4.2 Frontend

- TBD
- QTP/UFT
- Selenium

4.3 Smart Contracts

- REMIX is an online solidity IDE that compiles and deploys the smart contracts to a specified test network.
    - It exposes the public methods or contract data that we specify to the editor and we can see their return values momentarily. (Please see Figure 1)
    - It also allows us to create unit tests for our smart contracts which lets us to test our contracts before deploying them.
    - We will use REMIX IDE for our smart contract implementation and test them using REMIX.

5. Testing

5.1 Unit Testing

A software testing approach known as unit testing involves evaluating individual software units, such as groups of computer program modules, use processes, and operating procedures, to see if they are acceptable for use. Also, unit testing is the first level of testing which is before integration testing. The objectives of using unit testing are;

- -to separate a code segment
- to verify the correctness of those segments
- to evaluate each process and function.
- to help developers understand the code much more easily and for giving developers a chance to make code changes quickly.

In our project, the skeleton of our backend is done and of course we are testing our units to create a clean and well working backend. For backend, the backend team implemented the minimum functionality parts such as connecting our project to Mongodb and creating controllers, models and services. In controller and route, we used unit testing for wallet, campaign and user files. First, we used automated unit testing tool JEST. We wrote some test cases to our code. After running yarn test or npm test in a command line, it executes all the test cases and returns the number of passed or failed test cases. It allows us to easily find what is wrong and correct them easily.

Also, we use javascript debugger function to monitor our project. We can follow the progress of our project step by step and find what needs to be fixed in a very simple way with debugger function. It is absolutely a very easy way to proceed single step through the code and stop execution at a given point of code to investigate the values and connections of that part.

5.2 Performance Testing

We will use Rest Clients and Jest for testing the constraints of our currently modelled system for our test cases. These constraints can be considered as concurrent connections and requests to the server, very large or incompatible inputs and connectivity issues and handling.

5.2.1 Data Constraints

Our system relies on MongoDB for data storage and it is

For example, if a user has entered a very long campaign description -which is a text field on the frontend- that MongoDB cannot handle, we can catch it before it happens and ask for the user to re-enter the description or any other field in the same way.

Same goes for incompatible inputs. For example, while creating a donation campaign, if a user enters a random string into the campaign goal amount, our system shall catch it and prompt user to re-enter the amount in digits.

5.2.2 Connectivity Constraints

Our modelled system is planned to handle concurrent operations by multiple users. For this we will use a custom tester that sends multiple requests from different threads and mock ip addresses to our server trying to mimic different concurrent users. We can test our server's limitations using this approach, enabling us to build a more scalable web app.

5.3 System and Integration Testing

We will use a top-down breath-first approach to test the system validity. We will start testing from the main control module then other modules are integrated by moving downside of the control hierarchy. Normally all modules that are directly subordinate to the main control module are replaced with stubs, and the main control module is utilized as a test driver but because we already have stubs instead the actual modules, we will be testing those stubs. Due to this we will be facing some challenges like delaying some tests until stubs are replaced with actual modules because stubs have limited functionalities.

Our system is software based so the best system testing method we found is functional Blackbox Testing. The tool we will use for this called QTP/UFT (Unified Functional Testing). This is an automated functional testing tool that we can create and execute automated tests for the purpose of finding the errors, defects or gaps.

6. Responsibilities and Risks

6.1 Responsibilities

Main responsibilities which effects this process are:

- to live up to expectations of the stakeholders,
- ensure the safety of the information and data which assembled from users,
- ensure the transactions being done safely with encryptions,
- keeping server integrity,

In sight of these factors, risks can be determined easily.

6.2 Risks

Main risks that may appear from the responsibilities and cases given above are losing the data collected from the users, losing connection to eth servers during transactions, losing the users just because of the interface presented and experience they have in the software.

To handle those risks, servers should undergo maintenance frequently since there is 3 step connection. Also these connections should be encrypted and even developers should not be able to see the wallet id which is connecting to eth servers for the sake of integrity. In case of ui and ux, main source of improvement should be user feedbacks yet there can't be any solid answer to this since "person factor" cannot be controlled.

APPENDIX

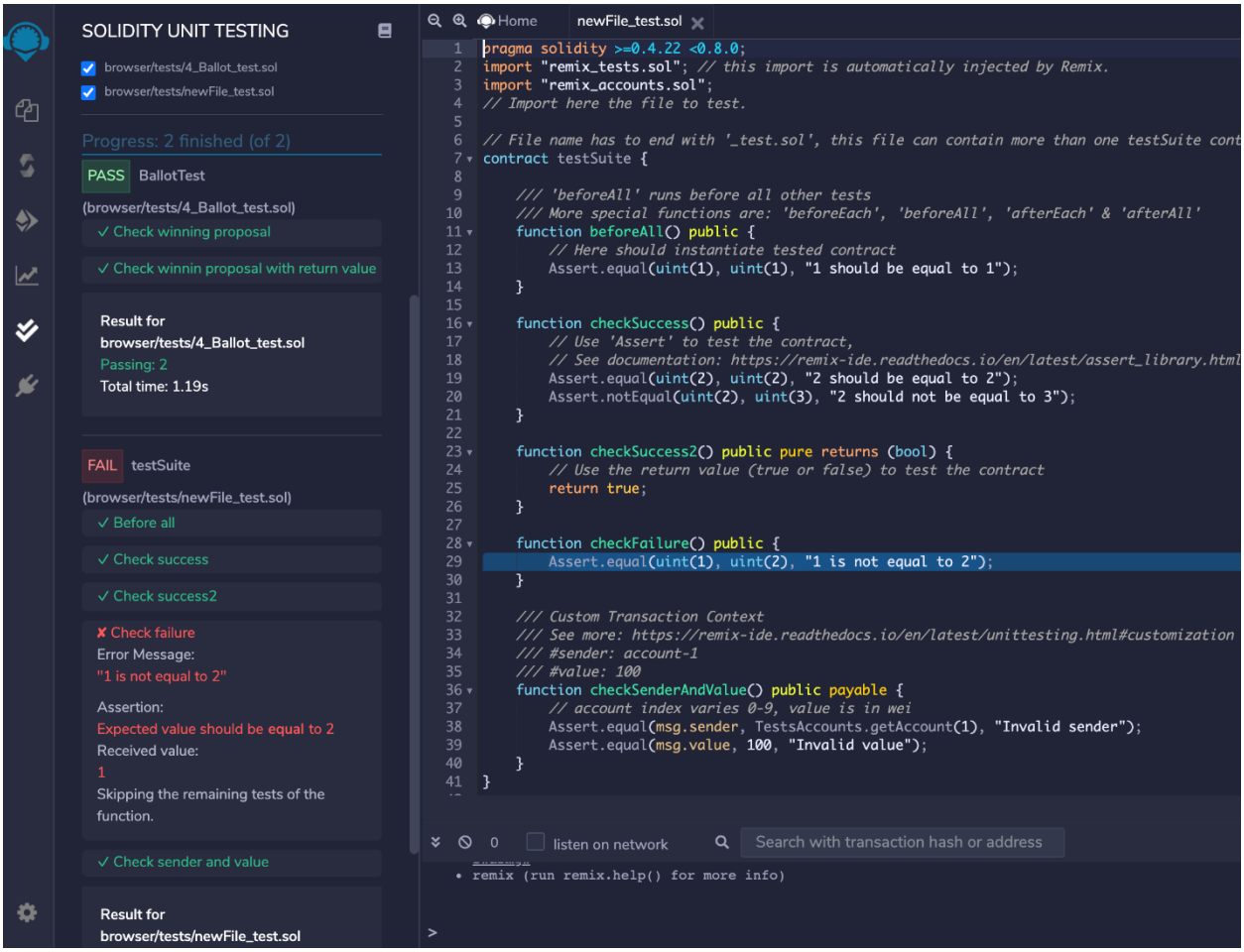

Figure 1 REMIX IDE Test output

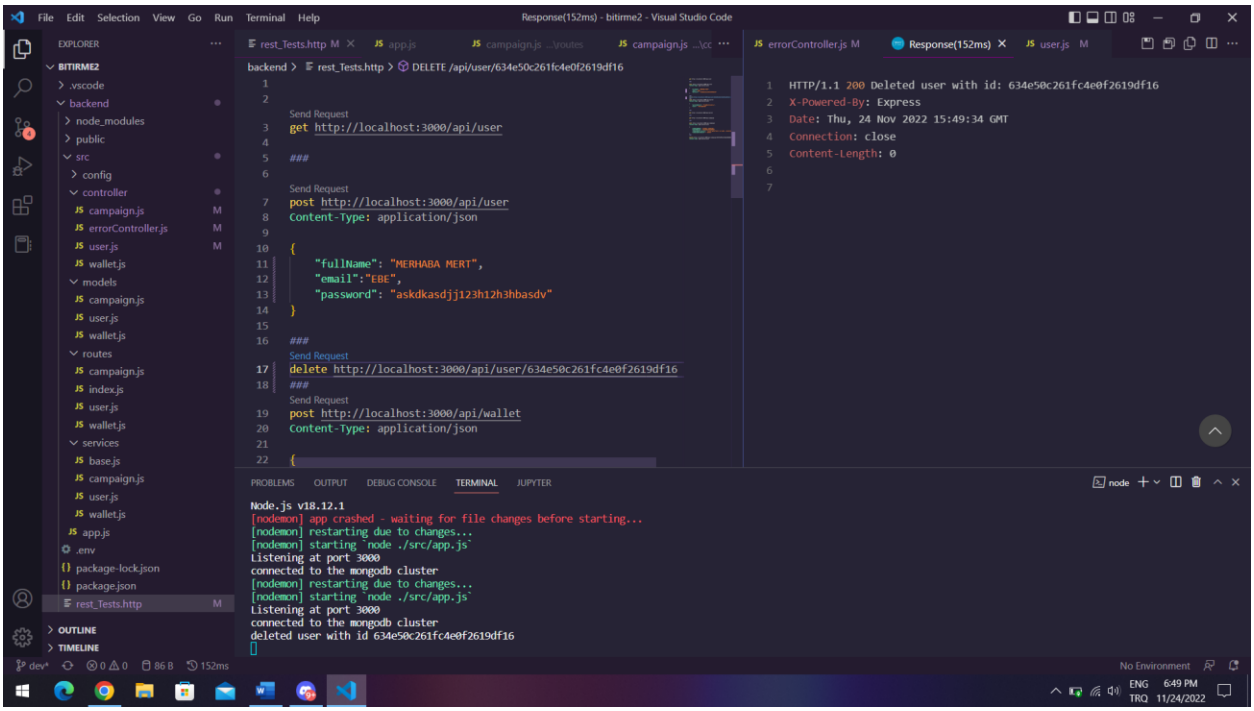

Figure 2 Rest Client Extension

REFERENCES

[1]: Remix IDE: https://remix-ide.readthedocs.io/en/latest/unittesting.html

[2]: JEST: https://jestjs.io/