TED UNIVERSITY

CMPE 492 Test Plan Report

CryptooFun

Team Members:

Kayra POLAT - 1000306178

Baturalp KIZILTAN - 4456996054

Emrecan ERBAY - 4221160055

Can ŞENGÜN - 1179712534

Supervisor:

Yücel ÇİMTAY

Jury Members:

Tolga Kurtuluş ÇAPIN

Emin KUĞU

Table of Contents

1 INTRODUCTION 1
2 INTEGRATION TESTING 1
2.1 Cash Wallet Service 1
2.2 Leaderboard Service 1
2.3 Portfolio Service 1
2.4 Progression Service 2
2.5 Trade Butler Service 2
2.6 Lobby Service 2
3 END-TO-END TESTING 2
4 LOAD TESTING 3

1 INTRODUCTION

Cryptocurrencies have become the most used investment tool in recent years. According to 2022 data, there are currently 300+ million users actively investing in cryptocurrencies. The global crypto market cap is $1.06 trillion as of August 1, 2022. It is in the hands of users to use this investment tool, which has a very bright future, correctly and safely. It is not right to invest in crypto money without understanding the risks and dynamics of the market. At this point, the CryptooFun application appears.

CryptooFun is ensure that people make minimal losses from their future investments by practicing with our application before investing their money in cryptocurrency exchanges. It will also provide real stock market experience by receiving the data of cryptocurrencies live. Thus, the experiences that people will have, will be more suitable for real life. The feature that distinguishes our project from other projects in the market is that we will enable people to compete while improving their knowledge with practice. Thus, we will encourage people to use our app.

In this report, we will indicate which testing processes our application will go through. Being able to test an application is important. For the application to be usable and workable in every aspect, test processes that fit the structure of the application must be applied. The test types that we see suitable for our application are integration testing, end-to-end testing, and load testing. We will implement these test structures as mentioned below.

2 INTEGRATION TESTING

In the integration stage, database connectivity and integrity & functionality of internal service components are tested via HTTP traffic. We are planning to use Postman, an automated API testing and documentation platform. Throughout the project, we used Postman to document both gRPC and HTTP APIs. Now, we will test those documented API definitions in a plain way.

2.1 Cash Wallet Service

- Get Wallet: Sends an HTTP GET request with an access token. Expects status code of 200 and user's wallet balance to be a number.

2.2 Leaderboard Service

- Get Top 100: Sends an HTTP GET request without authorization. Expects a leaderboard table with 100 rows.

2.3 Portfolio Service

- Get Portfolio: Sends an HTTP GET request with an access token. Expects status code of 200 and user's portfolio to be an instance of array data structure.

2.4 Progression Service

- Get User Level: Sends an HTTP GET request without authorization. Expects status code of 200 and user's level to be a number >= 0.

2.5 Trade Butler Service

- Create Market Order: Sends an HTTP POST request with an access token. Expects status code of 200 and the response body to have a field called "orderId".

2.6 Lobby Service

- Get Lobby By ID: Sends an HTTP GET request without authorization. Expects status code of 200 and check a lobby's details with proper data types.
- Search Lobbies: Sends an HTTP GET request without authorization. Expects status code of 200 and array of lobbies with proper data types.
- Join to Lobby as User: Sends an HTTP POST request with an access token. Expects status code of 200.
- Get Details of Lobby User: Sends an HTTP GET request with an access token. Expects status code of 200 and check lobby details for the user.

3 END-TO-END TESTING

Tests system functionalities from user's perspective. Tests start from the browser and get executed across multiple microservices. It allows us to verify business capabilities similar to real world scenarios, comprehensively. Our web app is a Next.js application and Cypress (an open-source JavaScript E2E test framework) will be used for automated E2E tests. Example test scenarios are given as the following:

Trading for buy action:

Browser → Authenticate → Visit "Wallet Page" → Read wallet balance and remember → Visit "BTC Trading Page" → Buy 1 BTC with market order → Visit "View Orders Page" → Check if order's created successfully → Wait 10 seconds and refresh page → Check if order's executed successfully and remember cost → Visit "Wallet Page" and compare with the expected value → Visit "Portfolio Page" and check if 1 BTC's added.

Trading for sell action:

Browser → Authenticate → Visit "Wallet Page" → Read wallet balance and remember → Visit "BTC Trading Page" → Sell 1 BTC with market order → Visit "View Orders Page" → Check if order's created successfully → Wait 10 seconds and refresh page → Check if order's executed successfully and remember cost → Visit "Wallet Page" and compare with the expected value → Visit "Portfolio Page" and check if portfolio is empty.

Find and Join a Lobby:

Browser → Authenticate → Visit “Lobbies Page” → Search for a lobby with keyword by title → Visit a lobby → Try to join the lobby → Check whether user joined successfully.

4 LOAD TESTING

Load tests are a subset of performance tests and allow us to benchmark our services. An open-source load testing framework called Locust will be used for simulating high user load. We are planning to conduct our tests in a public cloud environment. We create Kubernetes clusters for infrastructure needs of our system in Google Cloud Platform. The example test scenarios are given as the following:

Test Trading Subsytem under heavy load: Trading subsystem consists of Trade Butler service, Trade order stream (Kafka events), and Order Processing service components. A trade order starts its journey as an HTTP POST request in Trade Butler service from user. Locust will generate HTTP user load from this façade and will report response times specifically for Trade Butler service. In the meantime, we will check system stability across Trading subsystem components.

Test Lobby service’s search/listing functionality under heavy load: We expect that Lobby service to be one of our most used services, especially search/listing functionality. Therefore, we are planning to generate high user traffic against it similar to the Trading Subsystem scenario.