Fluency Accounts API Test Plan

Objective and Scope

The purpose of this document is to provide an abbreviated plan for manual and automated testing of the Accounts endpoints of Fluency's V2 API. For the sake of brevity, only two endpoints will be gone into any depth; GetCustomerAccountById and UpdateCustomerAccount. These endpoints were chosen because they are relatively basic forms of the most common verbs used in the API; **GET** and **PUT**, respectively.

Strategy

It is suggested most of the test cases or scenarios mentioned in this document are first performed manually. This provides a much quicker turnaround time and will provide better context and clarity to the person testing.

The following are the types of testing to be performed. They are not necessarily listed in order, but it is logical to behind by testing the documentation so requirements and expectations are clear. Following the acceptance of this strategy, these high-level items should be developed into individual test cases and stored in the company's tool of choice.

1. **Review documentation**. All endpoints are included and for each, methods, parameters, request bodies, and error codes are included and correct.
2. **Authentication and authorization**. Ensure the user has access to everything they would expect to have access to, but no more.
3. **Endpoint functionality**. Based on the documentation review above, the functionality of all positive, negative, and boundary cases should provide correct responses or gracefully error when appropriate.
4. **Data validation**. Updates or additions should be performed with invalid data types being sent, data that is not a member of a particular enumeration, data that is outside of certain bounds, and incomplete data. Appropriate errors responses should be validated.
5. **Performance testing**. The API should respond to requests within the SLA without load, with load, and the data should be validated to be correct under these conditions. Additional testing should be done with requests containing large volumes of data.
6. **Security testing**. The most common forms of API attacks, included, but not limited to SQL injection should be verified to be innocuous against the API.
7. **Error handling**. The various types of error codes the API can return should all be tested.

Environment

It is assumed whomever is responsible for performing this work will have access to a development environment containing certain preconfigured accounts and data scenarios. It is imperative there be a method to clean this environment during teardown. In some circumstances, tests may be long-running and dependent on a stable environment; it will be important to have an architecture that supports both of these types of testing.

Tools

For manual testing, it is suggested Postman be used. It is near ubiquitous in the API testing world, is cross-platform, and the ability to share and sync workspaces is invaluable. There is some amount of automation made available with Postman and there is CI integration, but anecdotally, these tools are generally not considered to be mature enough for serious use.

For automated testing, it is suggested REST Assured (Java) be used. While other languages and frameworks may be better candidates for automated API testing, the fact that Java is the primary back-end language at the company outweighs those advantages. REST Assured is mature, relatively easy to work with, and choosing a tool in a language that developers are familiar with goes a long way to promote shifting testing left. Some amount of performance and load testing can be done in this tool, but it should be relatively limited.

For performance and load testing, both JMeter and Gatling are compelling choices. Both support Java (JMeter natively, Gatling through a relatively new DSL) and are widely used. JMeter provides an easier means of getting started (has a GUI), while Gatling performs better and is easier to maintain since it is code-based. This decision should be made based on the technical ability of the people using the tools.

Integration and Reporting

All suggested automation and performance/load testing should be integrated with the CI process, provide alerts on failures, block merging of code when thresholds are met, and provide reports where applicable. These reports should be attached to each job and at the very least a baseline should be stored in a central location (a data warehouse?) for trend analysis.

Risks and Mitigations

- **Internal services unavailable**. This test plan relies on there being access to test environments. In the event these are unavailable, it may be feasible to perform a subset of tests on production. It will be important to not impact customers, especially when it

comes to performance or load testing so it is suggested any production tests of those be done in off-houses.

- **External tools unavailable**. The only external tool in this test plan is Postman. It is suggested Postman be installed locally (as opposed to being used web-based), and the shared workspace be backed up to a file repository at regular intervals. If this backup can be automated, it should be.
- **Data privacy**. In the event that production environments do need to be used, the tester should do their utmost to ensure data security. Additionally, any production data used to potentially seed a test environment should go through a fuzzing process.
- **API versioning and changes**. Should a new version of the API be released or changes simply be made, the decision will have to be made to continue supporting the old, continue running automated tests against it, and time will need to be invested to migrate to and support these changes.

Outside Scope

- **Comprehensive security testing**. Whole basic security testing is included in this plan, specialized tools and knowledge may be required.
- **Comprehensive data validation**. The testing outlined above should go a long way to ensuring bad data doesn't get into the system via the API, but it should be assumed the API is not the only means to enter data. In the event of unexpected data existing, the test above may not be sufficient to handle or detect it.
- **User experience testing**. This test plan strictly validates API functionality, not how end-users interact with it.
- **Third-party testing**. While extremely important given the context of the Fluency application, integrations with third-party services or systems are not included. Should the scope of this test plan increase, as it would when dealing with other endpoints of the API, this may no longer be outside of the testing scope.
- **Real-world load simulation**. Until the API is live, it may be beyond the capabilities or knowledge of the development team to perfectly simulate real-world traffic patterns. This may be possible in subsequent iterations of this plan and testing.