Test Plan – Web API Testing

Revision History

Version	Date	Author	Description of Change
1.0	2021-03-1	Avery Cameron, Raymond Knorr	Original Document
2.0	2021-03-20	Avery Cameron, Raymond Knorr	Update with additional endpoints. Revisions.

Table of Contents

1.0 Introduction

The URStreamSight API provides endpoints for clients to access organization data and recycling route and infraction data. The API is used with the front-end for access. The API uses a sequelize database and contains Role Based Access Control (RBAC). SSIO (StreamSight Input/Output) is our onboard computer used for bin detection and item collection.

2.0 Objectives

These are the objectives of our team in relation to testing:

1. Define the QA strategy.
2. Develop and maintain an overall test plan
3. Define. create, and maintain a master repository of test scripts/cases to cover all systems and areas that may require QA testing.
4. Execute the test scripts/cases and record the results of the execution for each project/requirement
5. Monitor bugs and other issues to be fixed
6. Create and maintain a test suite for each project/requirement

3.0 Testing Strategy

Overall, the API will be tested using Integration Testing to ensure the quality of each major feature. Using Integration Testing allows for the testing of groups of code, reducing test development time. Function testing will be used to test the inputs of individual endpoints.

SSIO will be tested using Unit Tests. The overall codebase is smaller and better suited to Unit Tests compared to the highly connected API and front end.

3.1 Features to be Tested

Each endpoint of the API will be tested, with tests organized under the main endpoints. The tests will cover the controllers and services of each These endpoints are:

- Organizations
- Reviews
- Routes
- Service Addresses
- Service Events
- User Actions
- User Roles
- Users
- Users Service

And as well, RBAC and database seeding will also be tested through the integration tests. These features will not be directly tested though.

3.2 Features Not to be Tested

We are not testing token validation in the auth folder. We are not testing the API deployed Lambda, and SQS to DB function. The camera module of SSIO was intentionally not heavily tested, in favor of thorough user testing while actively used in the field. This was mainly chosen as Prairie Robotics wanted SSIO to be run through multiple research and development iterations and test the software during these phases.

3.3 Unit Testing

SSIO will be tested using Unit Testing. These tests cover our bin manager, gps, serial, camera, and bin detection modules and focus on path testing to ensure code branches are reliable. We especially thoroughly tested improper data formats and bad readings from SSIO's connection to its onboard gps.

3.4 Integration Testing

Integration testing will be the main focus of the testing for the API. The tests will cover the controller calls and the internal calls to the service for each endpoint. The main tests are based off of function and branch testing and the outline here:

#	Test Scenario Category	Test Action Category	Test Action Description
1	**Basic positive tests (happy paths)**		
	Execute API call with **valid required parameters**	Validate status code:	1. All requests should return 2XX HTTP status code 2. Returned status code is according to spec: – 200 for GET requests – 201 for POST or PUT requests creating a new resource – 200 for a DELETE operation

		Validate Body:	1. Response is a well-formed JSON object (Automatic) 2. Response structure is according to data model (Body returns expected values)
2	**Positive + optional parameters**		
	Execute API call with **valid required parameters AND valid optional** parameters Run same tests as in #1, this time including the endpoint's optional parameters (e.g., filter, sort, limit, skip, etc.)		
		Validate status code:	As in #1
		Validate Body:	Verify response structure and content as in #1. In addition, check the following parameters: – offset: ensure the specified number of results from the start of the dataset is skipped

3	**Negative testing – valid input**		
	Execute API calls with **valid input** that attempts illegal operations. i.e.: – Attempting to create a resource with a id that already exists (e.g., serviceAddress with the same id) – Attempting to delete a resource that doesn't exist (e.g., serviceAddress with no such ID) – Attempting to update a resource with illegal valid data (e.g., rename a serviceAddress to an existing name) – Attempting illegal operation (e.g., delete a serviceAddress without permission.)		

		Validate status code:	1. Verify that an erroneous HTTP status code is sent (400, 403, 404) 2. Verify that the HTTP status code is in accordance with error case as defined in spec
		Validate payload:	1. Verify that error response is received 2. Verify that there is a clear, descriptive error message/description field 3. Verify error description is correct for this error case and in accordance with spec

4	**Negative testing – invalid input**		
	Execute API calls with invalid input, e.g.: – Missing or invalid authorization token – Missing required parameters – Invalid value for endpoint parameters, e.g.: – Invalid UUID in path or query parameters – Body with invalid model (violates schema) – Body with incomplete model (missing fields or required nested entities).		
		Validate status code:	As in #1
		Validate Body:	As in #1

4.0 Tools

Our API tests are run using Jest. Jest allows for easy, configurable TypeScript tests. The tests send calls to our API which executes the request which returns a response and is validated.

In SSIO, we are testing using DocTests from PyCharm and the pytest library. We also included the pytest libraries for generating testing components, which are like a lite version of mocked objects.

5.0 Dependencies

Our API tests can be run with little configuration but our tests use a local MySQL server instance. This requires a local MySQL server instance to be installed and running on the test machine. This configuration is easily completed with MySQL Workbench for a GUI instead of using the command line interface to set the database up. The MySQL server needs a Database created with any name, this could be URSS_DB_TEST. The tables do not need to be created as the tests will generate them automatically.

The API also expects a .test.env file in the root directory of the app. The contents of the file are specific to your MySQL setup but in general they are:

```
ENV=test
MYSQL_HOST=localhost
MYSQL_PORT=3306
MYSQL_USER=your DB user
MYSQL_PASS=your DB password
MYSQL_DB_NAME=URSS_DB_TEST
```

6.0 Risks/Assumptions

The tests don't test token validation, and the Lambda function and SQS to DB Function. This requires additional testing in the staging stage before being deployed to prod in the future.

Additionally, load and performance testing will need to be implemented as the amount of use increases. With the low load on the database and experienced performance the system handles well but in the future the testing plan could be expanded to include this in the future.