5 Testing

Our project will make use of many types of testing to test the various software components and subcomponents. For each of our components, our goal is to have at least 75% code coverage and 100% of tests pass. Each component is responsible for containing unit, integration, and system tests that verify its behavior internally, while additional integration and system tests will be outside of these components to test their interaction.

When changes are made in one component, it will be required to run all tests in that component in addition to all integration and system tests. New tests should be added to verify the behavior of newly added logic. Our main instruments for testing will be the Python testing framework unittest for our components written in Python and the testing framework jest for our website component written in TypeScript. This will work to run unit tests on individual components and integration and system tests across components.

The team's overall testing philosophy is to test early, test often, and use it as a tool. Testing can be a wonderful tool in a Test-Driven Development (TDD) framework because expected inputs and outputs can be specified first, then the tests' results indicate to the developer whether they've implemented the desired functionality.

5.1 Unit Testing

Scraper

The scraper tests and code coverage should be run after every commit to ensure no breaking changes have been made. The testing plan is as follows:

1. Open terminal
2. Change directory to `scraper`
3. Run `make coverage` command
4. Verify no failures in tests
    a. If all tests pass, the text "OK" will show at the bottom of the output.
    b. If one or more tests fail, the bottom of the output will read "FAILED" with the number of failures in parenthesis (e.g., "FAILED (failures=1)")
5. Open generated coverage file `htmlcov/index.html`
6. Ensure test coverage is greater than or equal to 75%.

Parser

The parser tests and code coverage should be run after every commit to ensure no breaking changes have been made. The testing plan is as follows:

1. Open terminal
2. Change directory to `parser`
3. Run `make coverage` command
4. Verify no failures in tests
    a. If all tests pass, the text "OK" will show at the bottom of the output.
    b. If one or more tests fail, the bottom of the output will read "FAILED" with the number of failures in parenthesis (e.g., "FAILED (failures=1)")
5. Open generated coverage file `htmlcov/index.html`
6. Ensure test coverage is greater than or equal to 75%.

Website

The tests and code coverage for the website component should be run after every commit to ensure no breaking changes have been made. The testing plan is as follows:

1. Open terminal
2. Change directory to `frontend`
3. Run `yarn test` command
4. Verify no test failures
5. Open generated coverage file `htmlcov/index.html`
6. Ensure test coverage is greater than or equal to 75%.

5.2 Interface Testing

Our interfaces include the source list for the scraper, the entity-relationship output from the parser, and the graph API queried by the frontend to be rendered. To test these interfaces, we have tests in place that send data through the interface and verify its integrity. In addition, the parser entity-relationship output can be tested by inputting mock article data and ensuring the parser output is as expected. For most of our existing and future tests we use the Python testing library such as unittest or mocking tools in a test class that asserts the scraper and parser are functioning properly. The graph API is provided by Neo4j and thus isn't tested in our project, although any bindings that we write for this API will be.

5.3 Integration Testing

The critical integration paths in our project begin with the human-developed source list being handled correctly by the scraper. The next is the scraper properly storing article information in the MongoDB database. Then after this is completed, the parser retrieves the MongoDB data and constructs the list of entities and relationships. Lastly this entity-relationship data is stored in the Neo4J graph database and then queried by the frontend for rendering. All the above paths are essential to the project operating correctly and can be tested by using Docker Compose to build and run all the containers on a local or cloud server, then use a script or library to verify the container are communicating properly with data expected by that test case.

5.4 System Testing

By using source lists with inputs for which the expected outputs are known we can test that the output of our system falls within our defined parameters. Our first step to test output we will use *portions* of documents we annotated manually to ensure the machine learning program is outputting what we expect it to and compare it to the accuracy requirements of the annotation. As the machine learning portion gets fed more documents and learns how to annotate with more accuracy, we will begin to give it larger portions of a document until it can annotate a full document within the accuracy percentage requirement we defined. Once this point has been reached, we will begin to give it full documents until the machine learning has been refined enough to where it is no longer required for us to test each output for accuracy. Our system testing will use all the tools mentioned in the unit, interface, and integration test sections to test the whole system, as each section needs to perform to our standards for the whole program to perform well. If one piece is not working up to standards, the output may contain unexpected behavior.

5.5 Regression Testing

Our strategy to implement Regression Testing is supported by two rules: Have a separate, clean, working copy of the code on our main branch and run unit, integration, and system tests on every pull request to this branch. The pull request is not able to be merged until all tests are passing. This will ensure existing functionality has not been broken, plus having the main branch be a clean working copy allows us to rollback changes that break existing functionality. Critical features that our project needs to ensure do not break are the scraper fetching sources and the parser processing these sources and outputting expected entities and relationships.

5.6 Acceptance Testing

Acceptance testing of our functional requirements will be performed by running unit, integration, and system tests to provide verifiable evidence that our test cases are passing. Most of our project's non-functional requirements can be summed up as a satisfactory level of code quality, libraries being open-source and a license allowing free usage, and documentation of code. These non-functional requirements can be verified by reviewers of pull requests. For example, if a new library is added as a dependency, the reviewer should be double checking the license allows us to use it. Our client will be involved in acceptance testing by being made available the number of passed tests and code coverage, ensuring we've met the expectations set in the requirements and metrics/evaluation criteria. They will also be delivered our iterative beta builds after each sprint to evaluate our progress and correctness in our design implementation.

5.7 Security Testing

Security Testing will consist of testing that malicious queries from the frontend website are not able to perform any sort of remote code execution (RCE) on the backend database. To ensure this doesn't happen, all inputs from the user need to be sanitized. Testing plans for security testing include performing various types of penetration testing on the frontend, including NOSQL injection and cross-site scripting, by performing queries with malformed input to execute malicious code on the server or in the user's browser.

5.8 Results

The unit testing on the scraper had two testing requirements associated with it. All tests (100%) had to pass, and the test coverage of scraper Python source code had to be at least 75%. Our unit testing successfully achieved those goals; all unit tests passed, and the coverage is 76%. This was able to show that the scraper is working exactly as intended taking input from the source list and outputting the article data. The following figure shows an output of test coverage in our current implementation of the scraper. Note the total coverage is shown at the top.

Coverage report: 76%

coverage.py v6.5.0, created at 2022-11-10 19:05 -0600

Module	*statements*	*missing*	*excluded*	*coverage ↑*
src/scrapy_core/scrapy_core/spiders/source_list_scraper.py	58	32	0	45%
src/scrapy_core/__init__.py	0	0	0	100%
src/scrapy_core/parsers/AllTextOfChildParser.py	23	0	0	100%
src/scrapy_core/parsers/ChildOfClassParser.py	18	0	0	100%
src/scrapy_core/parsers/__init__.py	10	0	0	100%
src/scrapy_core/parsers/abstract.py	12	0	5	100%
src/scrapy_core/scrapy_core/__init__.py	0	0	0	100%
src/scrapy_core/scrapy_core/spiders/__init__.py	0	0	0	100%
src/scrapy_core/types.py	15	0	0	100%
Total	**136**	**32**	**5**	**76%**

coverage.py v6.5.0, created at 2022-11-10 19:05 -0600

The scraper testing serves as a proof of concept for each component containing unit, integration, and system tests. Future work includes creating tests for the other components, and then integration and system tests between components. All the testing described in this section will ensure that our implementation matches both our chosen design and verify our requirements are being met.