# Test Plan

## Overview

To ensure the quality of the tool it is vital to perform appropriate testing to ensure the quality of the tool is acceptable to others. Tests will be based on the specification / documentation found [here](https://github.com/DogRacer78/Abstract-App-To-Source-Code/blob/main/METHODS.md). As the tool uses an external library to transpile from Python to JavaScript that will not be tested here, but rather if my custom methods are correctly transpiled. The second item that will be tested is if the code that is produced from my custom methods functions correctly. As it is not feasible to test every combination of code that exists, I will choose to test each part independently and then have some more complex examples.

## Testing Custom Methods

To test the custom methods I will use a set of automated tests. This is because it will be easy to compare the output of a small code snippet to the expected result. This will allow a next tester to quickly test the transpiler aspect of the tool.

## Testing Code Functionality

Assuming that the correct code is produced, then the functionality needs to be tested. This will be a manual process. The output of these tests will have to be verified manually. I will provide a template to discuss the expected output of each method, as well as a small code snippet to use to use in the test.

## Custom Methods Test Cases

The below list of test cases aims to test the output from the transpiler. I have tried to include both valid and invalid test cases. The automated test cases can be found in the “Automated Test Cases” document. Some things to note for any future testers:

* Code output is compared using Abstract Syntax Trees, in these the location data is removed as it is not important for the program
* The code is tested with any helper methods added, this includes an electron back end and the web app front end
* Some methods will throw exceptions when an error is found in the apps syntax. Some tests check if the correct exception is thrown rather than looking for correct code output
* When running the test cases the output will be fed into a Test\_Out.log file as the standard out is filled with debug information, within this file all test cases will be shown and state if they pass / fail, upon failure it will show the expected result and the received result, if an exception is thrown unexpectedly then it will be shown
* Each test case has an output file than is defined in JSON (except HTML tests), they will either contain an AST for code or contain a single “msg” field for an expected exception method
* The test input is defined as python file (except HTML tests), this is used for the input for each test, there is some duplicated test data, but each test case should have its own input file regardless

### Running Automated Tests

To run the suite of automated test simply clone the repository from GitHub, enter the repository folder and run “npm test”. This will run the tests automatically and produce the Test\_Log.log file with the results.

## Manual Tests

As discussed earlier the manual tests will be testing the functionality of the code. They will also be used for some tests that cannot be carried out automatically.

* Manual tests cases can be found in the separate document entitled “Manual Test Cases”, there is 9 distinct test cases that can be run by following the method laid out
* Test cases 1 – 7 aim to test different aspects that cannot be tested automatically, these include
  + Testing if HTML from webflow.com can be correctly downloaded and used
  + Tests if the fake-form custom attribute works as expected
  + Testing if the correct error message is shown when invalid python apps are supplied
  + Testing if the correct files are created when running the gen-db command to create a server for the database
* Tests 8 and 9 focus on testing the functionality of the database methods
  + As outlined in the test method there is a supplied python and HTML file that can be used to generate a web and electron app

### Testing Database Methods

Graphical user interface, application

Description automatically generatedWhen generating the apps to test the database methods it should look as below:

The app contains functionality to test each aspect of the database methods, these include:

* Loading data, using the “Show all” button or the “Get Song Details” you can check if the dbLoadData method is working as expected
* Inserting data, using the “Add Song” button and input, new songs can be added to the database, this tests if the dbInsertData method works as expected
* Updating data, using the “Update Song” button and input boxes the artist can be changed on a song, this tests the dbUpdateData method
* Deleting data, using the “Delete Song” button and the input box you can delete a song, this tests the dbDeleteData method
* The change events can be checked by opening another client, either electron or web and making an insert, update or delete, a message will be shown indicating that a change was detected in the relevant dialogue box

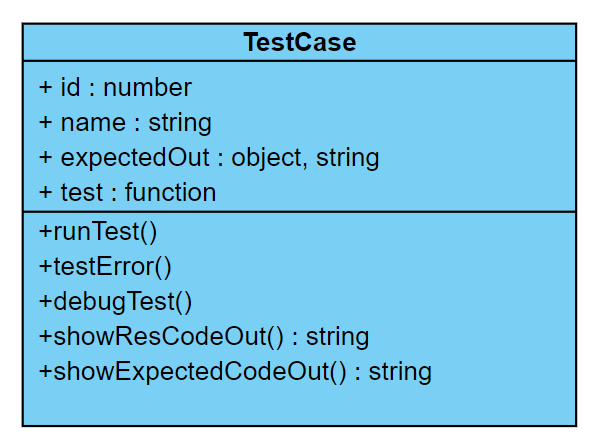
## Future Tests

Due to the nature of the tool it is impossible to test every possible combination that a user could do. I have only tested a subset of what I believe is necessary to have a functioning program.

### Automated Tests

The automated tests greatly speed up the testing process for checking the correctness of code produced, however at current they only test single method calls on their own. To solve this issue more complex test cases could and should be created to test the more obscure aspects of the tool. The goal is not to test the external transpiler libraries but rather to test how the custom methods handle different aspects in both Python and JavaScript.

#### Adding a New Test

When adding a new automated test case using the basic test framework I created should be used. Refer to the following UML of a TestCase class to understand how to add a new test case.

* id – the id number of the test
* name – A name or description of the test
* expectedOut – either an AST object or a string with an exception message
* test – the test function to carry out
* runTest() – call this when you want to run a test that does not expect an exception to be thrown. This will compare the return value of the test function to the expectedOut object, the test result will be output to a log file
* testError() – call this function when you want to test if the correct exception is thrown, it will check the expectedOut with the exception message thrown
* debugTest() – shows the test data
* showResCodeOut() – if you are testing an AST it will return the code version of the result, can only be called after the test is run
* showExpectedCodeOut() – if you are testing an AST it will return the code version of the result

#### Example

A screenshot of a computer

Description automatically generated with medium confidence

Above is an example of how a test case should be structured. Test 7 here is using an AST test, for its expected result. It is reading in a JSON file containing an AST and parsing it, the removeLocData helper function is included in the test program to remove the location data from an AST as it is not needed for comparison. The location data is automatically removed from a result AST. The result of a test is returned from the test function passed in, notice how they are async, this is because some test cases run async methods and need to be awaited. It is recommended that all your test case methods are async and awaited when being run.

Test 8 is an example of testing for an exception, as can be seen for the expected result, we are passing in the “msg” property of some JSON, this JSON incudes the exception message expected. You could just enter this as a string but using external JSON is a good practice for keeping tests organized.

Text

Description automatically generated

Above is an example of how to run tests. As can be seen test 7 is run normally, to check for an AST. Test 8 is run using testError() as it is looking for an exception. We have an example of the debugTest() method being used here for test 8. Notice how each test contains writeOutput(“\n”) after it is called. This is how to write to the log file and just adds some spacing in the output. The writeOutput() function is a helper function in the Test program.

### Manual Tests

The goal would be to have all the tests ran automatically, however that may not be possible. Manual test aim to test the functionality of the code produced rather than the correctness of transpilation. Manual tests should focus more on full applications and testing the code produced behaves as expected.

#### Adding a New Test

To add a new manual test case simply describe the method used and the expected output, these can then be appended to the Manual Test Cases document to reflect any new carried out.