**Better Than Waze**

**Test Plan**

**Submitted by**:

**Adam Elgressy**

**Anat Tetroashvili**

**Guy Rephaeli**

**Orel Kanditan**

**Shay Michaeli**

**Sharon Hadar**

**Supervised by:**

## Tanya Brokhman-Linder

Table of Contents

[1. Introduction 3](#_Toc499375966)

[1.1 Purpose 3](#_Toc499375967)

[1.2 Project overview 3](#_Toc499375968)

[2. Test strategy 3](#_Toc499375969)

[2.1 Test objectives 3](#_Toc499375970)

[2.2 Test assumptions 3](#_Toc499375971)

[2.3 Scope and Levels of Testing 4](#_Toc499375973)

[2.3.1 Functional Test 4](#_Toc499375974)

[2.3.2 User Acceptance Test (UAT) 5](#_Toc499375975)

[3. Validation and Defect Management 6](#_Toc499375976)

[4. Test Environmente 7](#_Toc499375977)

[5. Test cases 7](#_Toc499375978)

[5.1 Function tests 7](#_Toc499375979)

[5.1.1 Data-Base Tests 7](#_Toc499375980)

[5.1.2 Location Tests 9](#_Toc499375980)

[5.1.3 Navigation Tests 10](#_Toc499375980)

[5.1.4 City Generation Tests 11](#_Toc499375980)

[5.1.5 Geo-Json Handling Tests 11](#_Toc499375980)

[5.2 UAT tests 12](#_Toc499375981)

[5.2.1 Usability Tests 12](#_Toc499375980)

[5.2.2 Performance Tests 13](#_Toc499375980)

# Introduction

## Purpose

This test plan describes the testing approach and overall framework that will drive the testing of the BTW Project. The document introduces:

* Test Strategy: rules the test will be based on, including the givens of the project (e.g.: start / end dates, objectives, assumptions); description of the process to set up a valid test (e.g.: entry / exit criteria, creation of test cases, specific tasks to perform, scheduling, data strategy).
* Execution Strategy: describes how the test will be performed.

## Project overview

Better Then Waze (BTW) is a geographic system project. BTW designed to help planning an efficient roads system, and to find the best paths between locations in the system.

To achieve these goals, BTW is giving the user the ability to:

1. **Simulate a city** – a real roads system with junctions, streets and locations. The user will give the specification, and BTW will simulate the desired city.
2. **Pick the fastest way between two locations**. By keeping information about heavy traffic for each junction and road – BTW will know how to find the best path between two points and display the directions to the user.

All cities are first represented through the Geo-Json file format, enabling a uniform yet robust representation of the city and traffic information. The input\simulated city are then stored in SQL DB, from which information can be easily extracted when needed.

Using the information stored in the DB, a city can be represented as a very detailed graph, on which graph algorithms can be applied to navigate efficiently.

# Test strategy

## Test objectives

The objective of the test is to verify that the functionality of BTW works according to the specifications. The final product of the test is twofold:

* A production-ready software;
* A set of stable test scripts that can be reused for Functional test execution.

## Test assumptions

The unit tests will assume each module is an independent piece of software and test them accordingly. The integration tests will assume each module works independently and will assume we have a working internet connection for database communication.

## Scope and Levels of Testing

User level: check all interfaces are correctly represented. Check all desired paths are received

Map display representation level: check all maps are correctly represented and sent to be displayed. Check a calculated path is also represented correctly on the map. Check the possibility to update the display when there is a change in the database content.

Data base level: check all the data is consistent with the map input, and with the current state of the roads. Check the possibility to update the information in the data base.

Logic level: check all the calculated paths are correct, check all the information collected from the data base is represented in a sufficient way in java. Also check the possibility to update this representation over time.

### Functional Test

**PURPOSE**: Functional testing will be performed to check the functions of application. The functional testing is carried out by feeding the input and validates the output from the application.

**METHOD**: The test will be performed according to Functional scripts/Test procedures with a well-defined PASS/FAIL criteria.

TestRandomizedMap: check a random map can be created by demand. check it can calculate several random paths on it. Check that information can be added and removed from it. Try to display it on the screen. If all previous actions can be performed, then return SUCCESS otherwise return fail.

TestGivenMap: check a given map can be readed from the input. check it can calculate several random paths on it. Check that information can be added and removed from it. Try to despaly it on the screen. If all previous actions can be performed, then return SUCCESS otherwise return fail.

TestDataBaseConnection: check that a connection to the database can be established. If yes, return a SUCCESS, else return FAIL.

TestDataBaseQuerryCorrectness: try query several simple examples from a small and well known database. Check if the results are consistent with the small data base content. The test receives as an input the final state expected from the data base and compare the test results to it. If consistency exists, return SUCCESS, else return FAIL.

TestDataBaseUpdateCorrectness: try update several predefined examples from a small and well known database. Try to make the result look like a well known expected transform. The test receives as an input the final state expected from the data base after the transformation and compare the test results to it. If consistency exists, return SUCCESS, else return FAIL.

TestGraphCreationForLogic: get as input a predefined small database, and make a graph representation of the data structures in java. Print the data structures created and check the consistency to the input. If consistency exists, return SUCCESS, else return FAIL.

TestLogicFindRightPaths: get as input a graph representation in java of a small well known database. Try to find predefined shortest paths by sing the graph representation. If found all paths correctly return SUCCESS, else return FAIL.

### User Acceptance Test (UAT)

**PURPOSE**: this test allows the end users to complete one final review of the system prior to deployment. The user will enter a well known map (of his own city) and will search a well known path (for example from his home to work) but with an alternative path existing on the map.to ensure only the correct rout is available, the user will give all the road on the map not on the desired path a higher traffic load factor. Then he will check if the rout returned from the system is the correct one, which mean the shortest.

Another test will be preformed with the above conditions, but now the well known path will get a high traffic load factor, and the rest of the map will get a significantly lower traffic load. Here we want to receive the alternative path, and not the shortest.

After setting the correct traffic load for each test, more tests will be to take off manually some roads from the shortest and the alternative paths on the map and see if the received path are changed according to the roads taken. The possibilities are:

* When the traffic load on the shortest path is lowest, erase from the map a road from this path and see if the new path calculated is making sense, or even that the new path is not considering the road that was taken.
* When the traffic load on the shortest path is lowest, erase from the map a road not from this path, and see that the result stay the same.
* When the traffic load on the shortest path is highest, erase from the map a road from this path and see if the new path calculates is still the alternative path.
* When the traffic load on the shortest path is highest, erase from the map a road from the alternative path and see that the received path is changed' or cannot be calculated.

One more test is to see it recognize when the source and destination are the same. Should just say the user has arrived. An error will be a path the require the user to move from his location.

Check all the buttons provided by the main menu can be pressed and lead to the right menu. After that check that every menu can resume the main menu by the "resume" button.

**METHOD**: Will be performed manually by team members according to written test cases.

# Validation and Defect Management

It is the responsibility of the tester to open the defects, link them to the corresponding script, assign an initial severity and status,

It is the responsibility of the developer to retest after a fix is provided and close the defect.

Defects will be categorized according to the following severity status:

|  |  |
| --- | --- |
| Severity | Impact |
| 1 (Critical) | * This bug is critical enough to crash the system, cause file corruption, or cause potential data loss * It causes an abnormal return to the operating system (crash or a system failure message appears). * It causes the application to hang and requires re-booting the system. |
| 2 (High) | * It causes a lack of vital program functionality with workaround. |
| 3 (Medium) | * This Bug will degrade the quality of the System. However there is an intelligent workaround for achieving the desired functionality - for example through another screen. * This bug prevents other areas of the product from being tested. However other areas can be independently tested. |
| 4 (Low) | * There is an insufficient or unclear error message, which has minimum impact on product use |
| 5 (Cosmetic) | * There is an insufficient or unclear error message that has no impact on product use. |

# TEST ENVIRONMENT

The test running environment is a working computer connected to the internet (for approaching the database server), with enough memory to hold a current zoom of the map representation. The code will be first checked on the IntelliJ IDE using JUnit, and afterwards on the Technion server. requires a virtual machine running at least java 8 and above.

# Test cases

## Function tests

### Data-Base Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| Insert Road information | Use SQL query to insert new road with legal values. | Road table affected, new tuple appears. | Critical – tuple isn't created, wrong values appear. |
| Insert Crossroads information | Use SQL query to insert new Crossroads with legal values. | Crossroads table affected, new tuple appears. | Critical – tuple isn't created, wrong values appear. |
| Insert Passageway information | Use SQL query to insert new Passageway with legal values. | Passageway table affected, new tuple appears. | Critical – tuple isn't created, wrong values appear. |
| Insert Weight information | Use SQL query to insert new Weight with legal values. | Weight table affected, new tuple appears. | Critical – tuple isn't created, wrong values appear. |
| Insert Place information | Use SQL query to insert new Place with legal values. | Place table affected, new tuple appears. | Critical – tuple isn't created, wrong values appear. |
| Can’t create illegal Crossroads | 1. Use SQL query to insert new crossroads. 2. Supply illegal Passageways id. | Crossroads table isn’t affected. | Critical – Old data table affected.  High – tuple is added to the table. |
| Can’t create illegal Passageway | 1. Use SQL query to insert new Passageway. 2. Supply illegal crosses roads id. | Passageway table isn’t affected. | Critical – Old data table affected.  High – tuple is added to the table. |
| Can’t create illegal Place | 1. Use SQL query to insert new Place. 2. Supply illegal road id. | Place table isn’t affected. | Critical – Old data table affected.  High – tuple is added to the table. |
| Retrieve Road Information | Use SQL query to get road data by road id. | Road table isn’t affected.  Correct information received. | Critical – Old data table affected.  High – received wrong information. |
| Retrieve Crossroads Information | Use SQL query to get Crossroads data by id. | Crossroads table isn’t affected.  Correct information received. | Critical – Old data table affected.  High – received wrong information. |
| Retrieve Passageway Information | Use SQL query to get Passageway data by id. | Passageway table isn’t affected.  Correct information received. | Critical – Old data table affected.  High – received wrong information. |
| Retrieve Place Information | Use SQL query to get Place data by Place id. | Place table isn’t affected.  Correct information received. | Critical – Old data table affected.  High – received wrong information. |
| Can’t get information with wrong id | Use SQL query to receive road information, with unknown id. | Road table isn’t affected.  Error message – unknown road. | Critical – Old data table affected.  High – received wrong information. |
| DB knows how to get information from JSON | 1. Create legal json file with system information. 2. Use DB functionality to keep the information from the json file. | The json file is recognized correctly, all data spread and inserted into the correct tables by columns. | Critical – Old data table affected.  High – received wrong information. |
| Get maps from DB | Use SQL query to receive map information by map id. | The correct json file describing the map id received. | Critical – Old data table affected.  High – received wrong information. |

### Location Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| Create Graph | All graph information from the DB. | A graph that represents the city correctly. | Critical – graph isn’t created; created graph does not represent the city correctly. |
| Find Coordinates | Enter coordinates to get the crossroad corresponding to them. | Crossroad object located in the specified coordinates. | Critical – object isn’t returned, wrong object is returned. |
| Do Not Find Illegal Coordinates | Enter Illegal coordinates. | Error message, specifying the illegal coordinates | Critical – Crossroad object is returned. |
| Find Location Name | Enter location name to get the road corresponding to it. | Road object containing the specified location. | Critical – object isn’t returned, wrong object is returned. |
| Do Not Find Illegal Location | Enter nonexisting location name. | Error message, specifying the illegal location name. | Critical – Road object is returned. |
| Find Address | Enter address to get the road corresponding to it. | Road object containing the specified address. | Critical – object isn’t returned, wrong object is returned. |
| Do Not Find Illegal address | Enter nonexisting address. | Error message, specifying the illegal address. | Critical – Road object is returned. |

### Navigation Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| Create Route | Start and finish points. | A legit route between the two points. | Critical – route is not returned; output route is invalid  Medium – Route isn’t optimal |
| Do not Create nonexisting Route | Start and finish points. | Error message specifying the start and finish points | Critical – output route is returned |
| Calculate route timing | Start and finish points. | A legit route between the two points, and the time it takes to pass the route. | High – estimated time is not returned; output route and time are not compatible. |
| Find greedy optimal route | Start and finish points. | A legit and short route between the two points, and the time it takes to pass the route. | High – there exists a point on the route, from which the suggested route was not optimal. |

### City Generation Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| Generate random city | Default values | A city with all information nescesary | Critical – the city data is transferred to the next layer which deems it insufficient |
| Generate random city | User values | A city with all information nescesary, matching the user input | Critical – the city data is transferred to the next layer which deems it insufficient  High – the city doesn’t match the user input |

### Geo-Json Handling Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| Convert Geo-Json input to SQL Database | Geo-Json user input | Data transfers to SQL database and is searchable with queries | High – Database is missing some of the input |
| Convert simulated map data to Geo-Json | Generated map in app | Geo-Json file that can be exported | High – Geo-Json file is missing some of the information of the map |

## UAT Tests

### Usability Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| Input City Representation | Geo-Json file specifying a city | Graphic representation of the city. | Critical – no graphical representation is shown; graphic representation and input city are incompatible |
| City representation Error Handling | Incoherent Geo-Json file | Error message describing incoherencies. | Critical – the application gets stuck\shuts down; a graphical representation is shown |
| City Generation and representation | Parameters describing some desired city characteristics | Graphic representation of the generated city. | Critical – no graphical representation is shown; graphic representation and input parameters are incompatible |
| City Generation Error Handling | Incompatible city characteristics | Error message describing the incompatible parameters. | Critical – the application gets stuck\shuts down; a graphical representation is shown |
| City representation and navigation | Geo-Json file specifying a city, and start and end points for navigation | Graphic representation of the generated city and navigation route. | Critical – no graphical representation is shown; graphic representation and input parameters are incompatible |
| City representation and illegal navigation | Geo-Json file specifying a city, and illegal start and end points for navigation | Graphic representation of the generated city and error message specifying the illegal points for the navigation task. | Critical – no graphical representation is shown; a navigation route is shown; the application gets stuck\shuts down. |

### Performance Tests:

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective** | **Enter:** | **Exit:** | **Defect Categorization:** |
| City Creation Performance | Geo-Json file specifying a city | Graphic representation of the city | Medium – city creation takes more than an hour |
| City Generation Performance | Parameters describing some desired city characteristics | Graphic representation of the city | Medium – city generation takes more than a few hours |
| Navigation Performance | Start and finish points (city is already initialized) | Graphic representation of the navigation route in the city | Medium – route calculation takes more than a few minutes |
| Route Quality | Start and finish points (city is already initialized) | Graphic representation of the navigation route in the city | Medium – route takes much longer to drive than the optimal route |