Ecolution Testing Plan

1.0 Test Strategy

1.1 Objectives

The aim of this testing plan is to ensure the product, “Ecolution”, developed meets all expectations and requirements as outlined by the client. Testing will cover the testing of the product’s individual components to the functionality of the product’s parts as a whole. This will also including usability testing, ensure that the user experience is intuitive and engaging, as expected through the client’s request for the implementation of gamification elements within the product.

1.2 Scope

This product will be tested using a variety of testing types during its development. During the prototype phase, unit testing and integration testing will be used to ensure that development is progressing as expected, and to ensure that all baseline expectations from the client are fulfilled, or planned for. After this phase, systems and user acceptance testing will be used, alongside prior test suites, to check for issues with the product as it becomes more complex and complete.

1.3 Approach

Testing will be done through a combination of manual and automated testing to ensure the product meets expectations regarding functionality, usability, and quality.

1.3.1 Manual Testing

Manual testing will be employed for usability testing during the product’s prototype phase to ensure that development is progressing as expected by the client. This testing will be done to ensure the user interface is easy and intuitive to use and understand, and to ensure that the user experience is enjoyable, and fulfils the client’s requirement for an application using gamification.

1.3.2 Automated Testing

Automated testing will be used in unit testing in the prototype phase to ensure that individual components of the product function as expected. In later stages of development, regression testing will be implemented to ensure that new features added do not affect base functionality.

2.0 Test Cases: Unit Tests

2.1 Strategy

The tests detailed use Django's built-in test class, `TestCase`, due to the advantageous features offered compared to Python's standard unittest class. A test database is created and destroyed for every test run, ensuring that test data does not exist in the actual product's database. Every database transaction is rolled back, ensuring that the database is a clean slate for every unit test. Assertions are used to check the correct use and behaviour of models and views. All unit tests for the product exist within the `tests/unit` package.

2.2 Settings

2.2.1 setUp()

This function creates a new user, and then logs the user in. The user is then navigated to the "Settings" page.

2.3 Tasks

2.3.1 setUp()

This function creates a user, and then logs in that user. Three tasks are created and added to the test database; the first two tasks are initialised with the task's name and description, and the third task includes an amount of xp given upon completion. The user's pet is also initialised with 0 XP.

2.3.2 test_user_adds_tasks()

This test checks that the user is able to add predefined tasks to their "Current Tasks" list. The task is added to the user list, and then an `assertTrue` statement checks that that task exists in the user's list.

2.3.3 test_user_removes_tasks()

This test checks that the user is able to remove tasks from their "Current Tasks" list. The task is added to the user list, and then an `assertTrue` statement checks that that task exists in the user's list. The user then removes that task from their list, and then an `assertFalse` statement checks that the task no longer exists in the user's list.

2.3.4 test_user_completes_tasks()

This test checks that the user is able to mark tests as complete, and that tasks are then added to user's "Completed Tasks" list. The task is added to the user's list, and then an `assertTrue` statement checks that the task now exists in the user's "Current Tasks" list, and is marked as "Completed = False". The user then marks the test as completed, and saves this change. An `assertFalse` statement checks that the task no longer appears in the user's "Current Tasks" list. Then, an `assertTrue` statement checks that the task does not appear in the user's "CompletedTasks" list.

2.3.5 test_user_earns_points()

This test checks that the user is able to earn points from completing tasks. The user's points at the beginning of the test are saved in a variable for comparison later on. The third task created in 2.3.1 is then added to the user's list, and validated. The user then marks the task as complete, and checks that the task no longer appears in the current tasks list, instead appearing in the completed tasks list. The user's points after completing the task are then stored in another variable, and are compared within an `assertTrue` statement that checks that the user's points are now greater than their points before completing the task.

2.4 Events

2.4.1 setUp()

This function creates a new user, and then logs that user into the site. The user is then redirected to the "Events" page.

2.4.2 test_view_events()

This test checks that the user can view the events page. The test navigates the user to the "Events" page using a GET request. An `assertEqual` statement checks that the user was successfully redirected by checking the status code of the GET request, which should equal 200 when successful.

2.5 Logout

2.5.1 setUp()

This function creates a new user, and then logs that user into the site.

2.5.2 test_logout()

This function checks that a user can logout of their account. The test first checks that the user is logged in by navigating them to the home page. The user then navigates to the "Settings" page (where the log-out button is located). The user then selects the log-out button. An `assertFalse` statement checks that the user is logged out by trying to access a protected page, “Home” (which can only be accessed by a logged in user). An `assertNotIn` statement checks that the user session is no longer active.

2.6 Homepage

2.6.1 setUp()

This function creates a new user, and logs them in. The user is then directed to the “home” page.

2.6.2 test_homepage_view_xp()

This test checks that the user is able to view their XP points on the homepage. The user redirects back to the homepage. An `assertContains` statement checks that the homepage contains the text “XP”, confirming that the user’s XP appears as expected.

2.6.3 test_homepage_view_all()

This test checks that the user is able to view all their current tasks by selecting the “View all” button on the homepage by their “Current Tasks” list. The test first checks that the “View all” button appears using an `assertContains` statement. The user then selects the button through a POST request. An `assertRedirects` statement then checks that the user is redirected to the “Tasks” page.

3.0 Test Cases: Integration Tests

3.1 Strategy

The tests detailed use Django’s built-in test class, `TestCase`, due to the advantageous features offered compared to Python’s standard unittest class [see: 2.1]. These tests interact with the product database, to ensure that all database operations and interactions with the front-end function as expected. All unit tests for the product exist within the `tests/integration` package.

3.2 Admin

3.2.1 setUp()

This function creates a new admin user, and instantiates the admin page URL.

3.2.2 test_admin_login()

This test checks that an admin user can login to the admin page. The user is logged in using credentials data defined in 3.2.1. An `assertRedirects` statement checks that the user is redirected to the intended page. An `assertTrue` statement then checks that the user is logged in using the Django session key functionality.

3.2.3 test_admin_login_invalid_user()

This test checks that a non-admin user cannot login to the admin page. The user attempts to login using an invalid username (a different user name compared to the one specified in 3.2.1). An `assertNotEqual` checks that the user is not logged in not redirected to the main Admin page.

3.2.4 test_admin_login_invalid_pwd()

This test checks that an admin user cannot login to the admin page with an invalid password. The user attempts to login using an invalid password (a different password compared to the one specified in 3.2.1). An `assertNotEqual` checks that the user is not logged in not redirected to the main Admin page.

3.3 Homepage

3.3.1 setUp()

This function creates a new user, and logs them in. A task is created and then added to the database. The task is then added to the user's "Current Tasks" list. The user's pet is then created, and initialised with a starting XP of 0.

3.3.2 test_homepage_view_pet_name()

This test checks that the user is able to view their pet on the homepage. The user is logged in and navigates to the homepage. An `assertContains` statement checks that the pet's name is present in the response content, confirming that the pet's name is displayed correctly.

3.3.3 test_homepage_current_tasks()

This test checks that the user is able to view their current tasks on the homepage. The user is logged in and navigates to the homepage. Two `assertContains` statements check that the task name and description are present on the page, verifying that the tasks added appear on the homepage.

3.3.4 test_homepage_points_increase()

This test checks that the user can earn points from completing tasks. The user's points at the start are saved to a variable for later comparison in order to determine whether the user's points have increased as expected. The user then completes the flow for adding and completing tasks.

The user then returns to the homepage, and the user's points post-completing the task are saved to another variable. An `assertTrue` statement then compares the values of the user's points before and after completing the task. Several debug statements are present to easily track the user's points as the test runs.

3.4 Login

3.4.1 setUp()

This function creates a new user, and instantiates the login and signup urls used in the tests.

3.4.2 test_login_valid_creds()

This test checks that a user can login with a valid username and password. The user attempts to sign in through a POST request, using the valid credentials specified in 3.5.1. An `assertRedirects` statement checks that the user is successfully redirected to the site homepage.

3.4.3 test_login_invalid_pwd()

This test checks that a user cannot login with a valid username but invalid password. The user attempts to sign in through a POST request, using the username specified in 3.4.1, but a different password. An `assertNotEqual` statement checks that the user has not been redirected to the homepage (as they should not be logged in, and therefore able to view the homepage).

3.4.4 test_login_invalid_user()

This test checks that a user cannot login with an invalid user but valid password. The user attempts to sign in through a POST request, using the password specified in 3.4.1, but a different username. An `assertNotEqual` statement checks that the user has not been redirected to the homepage (as they should not be logged in, and therefore able to view the homepage).

3.5 Settings

3.5.1 setUp()

This function creates a new user, and logs in the user.

3.5.2 test_settings_change_pwd()

This test checks that the user is able to change their account password. The test checks that the user can navigate to "/settings/" page by using an `assertEqual` statement to check the status code.

3.5.3 test_settings_delete_acc()

This test checks that the user is able to delete their account. The user first navigates to the "settings" page. The user then selects the "Delete Account" button through a POST request. An `assertEqual` statement checks that the button was successfully selected. The user then selects the "confirm" button on the confirmation screen, checked with another `assertEqual` statement. The user should then be logged out, tested with an `assertNotIn` statement that checks the user session no longer exists. A "try-except" statement checks that the user no longer exists in the database by trying to search for the user in the database. If the user no longer exists, they are then deleted from the database.

3.5.4 test_settings_change_pwd_different_newpds()

This test checks that the user cannot change their password if the passwords in the new password fields do not match. A POST request enters input into the password fields; the user's current password is entered, then two different passwords are entered into the new password fields. The user is then logged out, and then attempts to login with the first password entered into the field. An `assertNotEqual` statement checks that the user is not directed to the "home" page. This step is then repeated with the second password entered, checking that the user cannot login with either password.

3.5.5 test_settings_change_pwd_invalid_current_pwd()

This test checks that the user cannot change their password without entering the correct current password. A POST request simulates the user entering input into the "Change Password" fields; an incorrect current account password is entered, alongside two matching new passwords The user is then logged out, and then attempts to log back in using the new password. An assertNotEqual statement checks that the user is not directed to the homepage, and is therefore not logged in.

3.6 Signup

3.6.1 setUp()

Instantiates the signup and login page URLs to be used in the tests, and creates user data (email, username, a set of matching passwords).

3.6.2 test_signup_valid_creds()

This test checks that a user can sign-up for an account using valid credentials (specified in 3.9.1). After the user data is entered into the signup form, an `assertRedirects` statement checks that the user is redirected to the "/login/" page. The test then checks that the user is correctly added to the database, checked using a `assertIsNotNone` statement.

3.6.3 test_signup_invalid_email()

This test checks that signup is not possible with an invalid or blank email. The user data is entered into the signup form, and then an `assertNotEqual` statement checks that the user is not redirected to the “/login/” page (as this redirect happens after an account is successfully created). An `assertRaises` statement then checks that user has not been created and added to the database by trying to find the user in the database.

3.6.4 test_signup_invalid_pwd()

This test checks that signup is not successful with invalid passwords. The user data is entered into the signup form, with passwords that do not match the validation criteria. The `assertNotEqual` statement then checks that the user is not directed to the “/login/” URL, indicating that signup was not successful. An `assertRaises` statement then checks that user has not been created and added to the database by trying to find the user in the database.

3.6.5 test_signup_redirect()

This test checks that the user is redirected to the “/login/” page after successfully signing up for an account, and is then able to login to that account with the newly created credentials. User data is inputted into the signup form, and then an `assertRedirects` statement ensures that the user is redirected to the login page. An `assertTrue` statement checks that the user exists and has been added to the database. The user data is then inputted into the login form, then an `assertRedirects` checks that the user is logged in and successfully redirected to the “/home/” page by checking the status code.

3.6.6 test_signup_different_passwords()

This test checks that a user cannot sign-up for an account with the passwords entered into the input fields do not match. User data is inputted, with two different passwords, into the sign-up form. An `assertNotEqual` statement checks that the user is not directed to the “/login/” page. An `assertRaises` statement then checks that user has not been created and added to the database by trying to find the user in the database.

3.7 Tasks

3.7.1 setUp()

This function creates and logs in a test user. It then creates two tasks, with task names and descriptions.

3.7.2 test_user_creates_tasks()

This test checks that the user can create their own tasks. The test first initialises the task data, and then simulates the user inputting this data through a POST request. An `assertIsNotNone` statement checks that the task exists in the database. An `assertEqual` statement checks that the task is linked to the user, and an `assertTrue` statement checks that the task appears in the user's current list.

3.7.3 test_user_deletes_tasks()

This test checks that the user can delete their own created tasks. The test first creates a test for the user, and checks the tasks have been created using an `assertEqual` statement. An `assertTrue` statement then checks that the task then appears in the user's tasks list. The user then deletes the task — an `assertFalse` statement checks that the task no longer appears on the user's task list. An `assertIsNone` statement checks that the task no longer appears in the task database, as user-created tasks only exist for a given user and should not exist independently. An `assertEqual` statement checks that the task was successfully deleted. Several debug statements are present in order to keep track of the flow.

3.7.4 test_created_tasks_visibility()

This test checks that users cannot view other users' created tasks. The test creates a second user, and then creates tasks for each user. An `assertIn` statement checks that the task for each user appears in their task's list. An `assertNotIn` statement checks that users cannot view each other's created tasks in their lists. An

3.7.5 test_view_completed_tasks()

This test checks that the user can view their completed tasks page. The test directs the user to the "Completed Tasks" section of the "Tasks" page, and checks that the user was successfully able to view this through an `assertEqual` page comparing the response status code.

3.7.6 test_view_current_tasks()

This test checks that the user can view their current tasks page. The test directs the user to the "Current Tasks" section of the "Tasks" page, and checks that the user was successfully able to view this through an `assertEqual` page comparing the response status code.