Test plan

|  |  |  |  |
| --- | --- | --- | --- |
| Requirement | Tests to ensure this requirement is met | Reasoning |  |
| Endpoint responds correctly to REST requests (R1). | I will black-box test the endpoint with a large collection of pre-defined valid and invalid test orders and assert that the result is as expected. | This aligns with the principles of TTD which enhances coverage and efficiency by separating logic from test data. | System Tests |
| The endpoint avoids no-fly zones (R2). | I will make post requests to the endpoint with test orders using Mockito, loop through each node of the path and check that it is not in a no-fly zone. | Mocking external dependencies like the no-fly zone service allows for isolated testing of the endpoint's logic. Verifying each node ensures that the entire path adheres to the no-fly zone restrictions. |
| The endpoint delivers to the correct location (R3). | I will make post requests to the endpoint with test orders using Mockito and assert that the final node of the path is near Appleton. | Similar to the previous test, mocking ensures focused testing. Asserting the final node's proximity to Appleton Tower validates successful delivery. |
| The endpoint runs under 60 seconds (R4). | I will make post requests to the endpoint with test orders using Mockito and assert that the time taken is < 60 seconds (Performance testing). | This directly addresses the performance requirement. Mocking dependencies helps to isolate the endpoint's performance and avoid external factors influencing the results. |
| The REST controller interacts seamlessly with other services used by the endpoint (R5). | I will test the controller-to-FlightPathService integration by mocking the Flightpath and orderValidation services. | Mocking these services allows me to test the controller's logic independently, focusing on the data flow and interaction between the controller and its dependencies. | Integration Tests |
| The flight path logic within the FlightPathService works as intended, considering factors such as drone movement constraints, no-fly zones (R6). | I will use white-box testing to cover the logic of the FlightPathService using Junit. I will use concepts such as equivalence partitioning, boundary testing, and | White-box testing is appropriate here as I’m testing the internal logic of the FlightPathService. Equivalence partitioning and boundary testing ensure that different input ranges and edge cases are covered. Path coverage aims to test all possible execution paths within the logic. | Unit Tests |
| The FlightPathService retrieves restaurant information from the external API correctly (R7). | I will run unit tests that check the exception handling of the method getNoFlyZones as well as getCentralArea. | This focuses on the service's ability to handle potential errors during API calls. Testing exception handling ensures the service's resilience in case of network issues or unexpected API responses. |
| The endpoint handles invalid orders or requests gracefully (R8). | I will run a unit test on the validateOrder method by looping through a large collection of pre-defined valid and invalid test orders and check that the result is not an error and matches the expected orderStatus. | Using a large collection of test orders provides comprehensive coverage of different invalid order scenarios. Verifying the validation results ensures that invalid orders are identified and handled correctly. |