Application: <https://openweathermap.org/>

**Feature: search weather in your city**

**PART I: Test Design and Bug Challenge**

1. Create a Test approach for the given feature based on your Exploratory Testing skill (Different Test Types, Test Level, Test Coverage) (Please provide your assumption in the requirement as long as they are reasonable).

**Assumption in the requirement**:

* This web application could be run on latest browser: Chrome, Firefox, MS Edge, IE11
* This application is supported on mobile devices of OS: iOS and Android
* User enters a valid city then all matched cities will be displayed with few basic information
* User clicks on name of city in the result list then detail page of city is displayed

Suggestion Test approach:

All tests should be leveraged automation approach as much as possible and they should be integrated on CI servers to trigger by schedule or something happened (code merged, fast lane…).

* Unit testing: intent to detect and fix bug at the early stage. The earlier a defect detected, the lower the cost of its correction. Unit tests are in general quite cheap to automate and can be run very quickly. These tests will be integrated on continuous integration server to run whenever a merge request is performed.
* API testing: beside Web Application, OpenWeather also support open APIs. The same with Unit testing, this test type can be run very quickly. Tester could use tools: Postman, SOAP UI, Katalon Studio, … or using REST Assured – a Java library. API tests could be integrated into a CI server also.
* Functional testing: tests focus on the business requirements. These tests could be automation by a popular framework (e.g. Selenium Web Driver) to use later. Of course, they could be integrated with CI. However, these tests should be run manually few times when function/feature is under the development period and developer is encouraged to do this test. QA and developer could work together to compose test cases of the function/feature.
* Integration tests: take a smaller unit of unit testing and test their behavior as the whole.
* Compatibility testing: to test the web application on cross browsers
* End-to-end tests: test focus on users flows. These tests are expensive to perform and can be hard to maintain. However, it should have a few key end-to-end tests to increase the confident.
* Performance testing: we should have these tests to check behaviors of the system when it is under significant load.
  + Single User Tests: testing with one active user yields the best possible performance and response times can be used for baseline measurements.
  + Load tests: under average load, including the expected number of concurrent users/requests performing a specific number of transactions within an average hour
  + Peak load tests, Endurance tests, Stress tests, High Availability tests.
* Security testing: use tools to scan the system or manually do few tests kind of: XSS, SQL Injection, …
* Localization testing: test application when launch on difference regions
* Smoke tests: add basic tests of this feature into the smoke test suite which could integrate into build servers to check new builds. Smoke tests helps to decide whether we can run more expensive tests or simply to check that the application is running properly in the newly deployed environment.
* Regression tests: add tests of this feature to regression suites to check the feature still work when something changes happened.
* More on Exploratory testing: can reveal bugs that would go undiscovered during structured phase of testing and help to find new test scenarios to enhance the test coverage.
* Mobile testing: beside mentioned test types, application has developed on mobile devices: iOS and Android. This testing will be considered in another scope.

1. Design test cases for given feature by using different testing techniques (E.g. black box, white box…) for UI/API testing

**Equivalence Class Partitioning**

Checking input of search field by city name. We can divide it into 2 partition:

* Valid input: supported city in the attached list. E.g. Ho Chi Minh, London, …
* Invalid input:
  + No support city e.g. invalid
  + Presence of blanks before and after search string
  + Special letters: ?, ~, …
  + XSS, SQL injection e.g. <script>alert(1) </script>

**Boundary Value Analysis**

Checking API: Call for several city IDs



Requirement: The limit of locations is 20.

Valid case: we can check with number of ID: 1, 20

Invalid case: we will check with number of ID: 0, 21

**UI Test case**

First, I use Requirement Traceability Matrix to manage requirements. This table will break big requirements into smaller ones and linking relevant test cases to each requirement. This is base to measure the test coverage.

|  |  |  |  |
| --- | --- | --- | --- |
| **REQ Level 1** | **REQ Level 2** | **REQ Level 3** | **Test case ID** |
| Search cities | Search a valid city | Search by City name | TC001 |
| Search by City ID | TC002 |
| Search by zip code | TC003 |
| Search by geo coordinates | TC004 |
| Search an invalid city |  | TC005 |

Next, test case design

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case ID** | **Name** | **Pre-condition** | **Test Steps** | **Test Data** | **Expectation** |
| TC001 | Search by City name |  | 1. Launch OpenWeather application in browser 2. Typing into search field 3. Press Enter | London | Matched cities are listed in the result list |
| TC005 | Search an invalid city |  | 1. Launch OpenWeather application in browser 2. Typing into search field 3. Press Enter | Invalid city, special letters, blanks | “Not found” alert should be displayed |

E.g. For search city by Name. We could apply Equivalence Partitioning technique to design tests. Divide data into 2 partition: valid (data is in defined list) and invalid (data is not in defined list)

**API Test cases**

For each endpoint we could consider test cases:

|  |  |  |
| --- | --- | --- |
| Positive tests | Test with valid required fields only | Check response code, response body |
| Test with valid required fields and optional fields |
| Negative tests | Missed required fields |
| Invalid required fields |
| Invalid optional fields |
| Unsupported methods |
| Invalid values in HTTP headers |

api.openweathermap.org/data/2.5/weather?q={city name}&appid=[{API key}](https://home.openweathermap.org/api_keys)

|  |  |  |  |
| --- | --- | --- | --- |
| TCID | Test Name | Test Data | Expectation |
| TC01 | Send request with valid city name and valid access code | q=London,  appid= b48987aa8224924b027bd9d3b15e4731 | {  "coord": {  "lon": -0.1257,  "lat": 51.5085  },  "weather": [  {  "id": 601,  "main": "Snow",  "description": "snow",  "icon": "13d"  }  ],  "base": "stations",  "main": {  "temp": 274.97,  "feels\_like": 270.45,  "temp\_min": 274.82,  "temp\_max": 275.37,  "pressure": 1021,  "humidity": 87  },  "visibility": 6000,  "wind": {  "speed": 3.6,  "deg": 160  },  "snow": {  "1h": 2.39  },  "clouds": {  "all": 90  },  "dt": 1610784647,  "sys": {  "type": 1,  "id": 1414,  "country": "GB",  "sunrise": 1610783903,  "sunset": 1610814107  },  "timezone": 0,  "id": 2643743,  "name": "London",  "cod": 200  } |
| TC02 | Send request with valid city name and invalid access code | q=London,  appid= invalid | {  “cod”: 401,  “message”: “Invalid API key. Please see http://openweathermap.org/faq#error401 for more info.”  } |
| TC03 | Send request with invalid city name and valid access code | q=invalid,  appid=b48987aa8224924b027bd9d3b15e4731 | {  "cod": "404",  "message": "city not found"  } |

1. Find bugs of the application/feature and report them in your desired bug template. Explain how you triage your defect in terms of priority/severity bug, the high score you get.

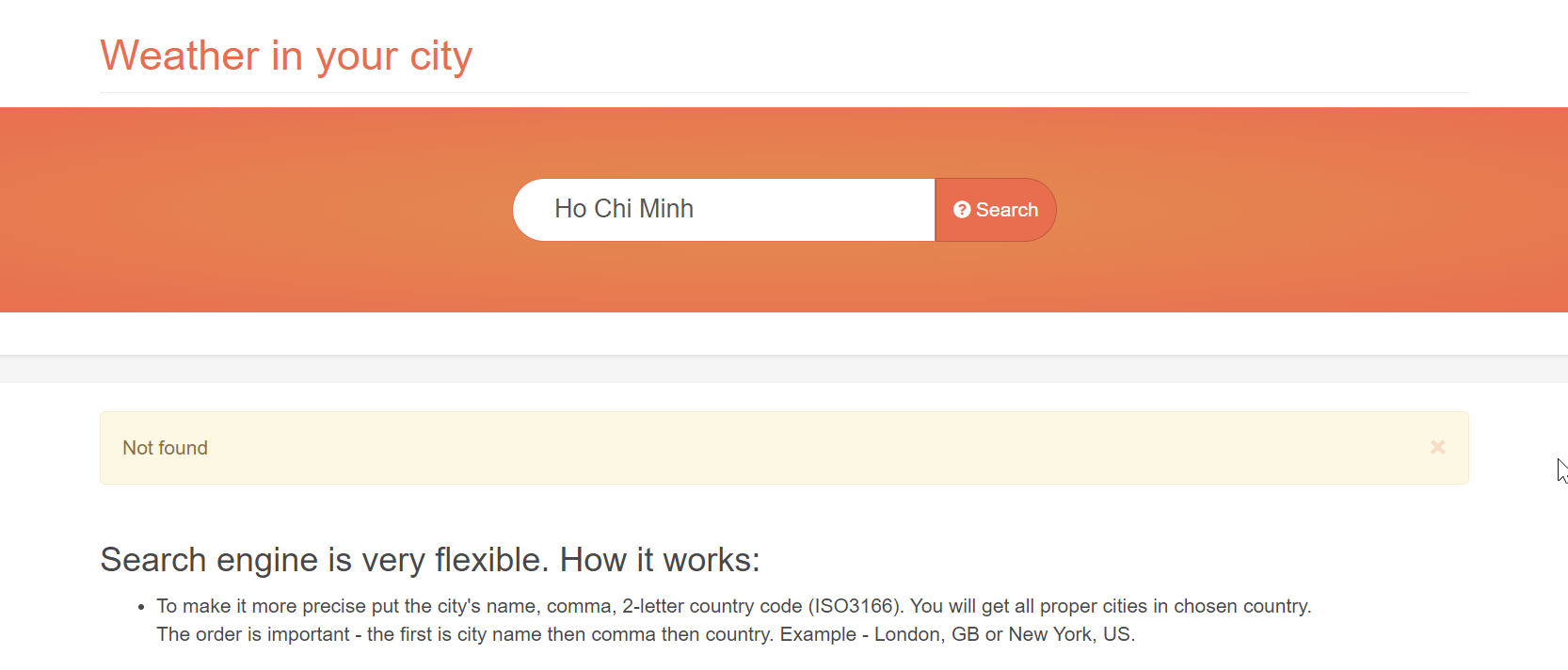
**Summary**: Should trim blanks which presence before and after of search field

**Description**:

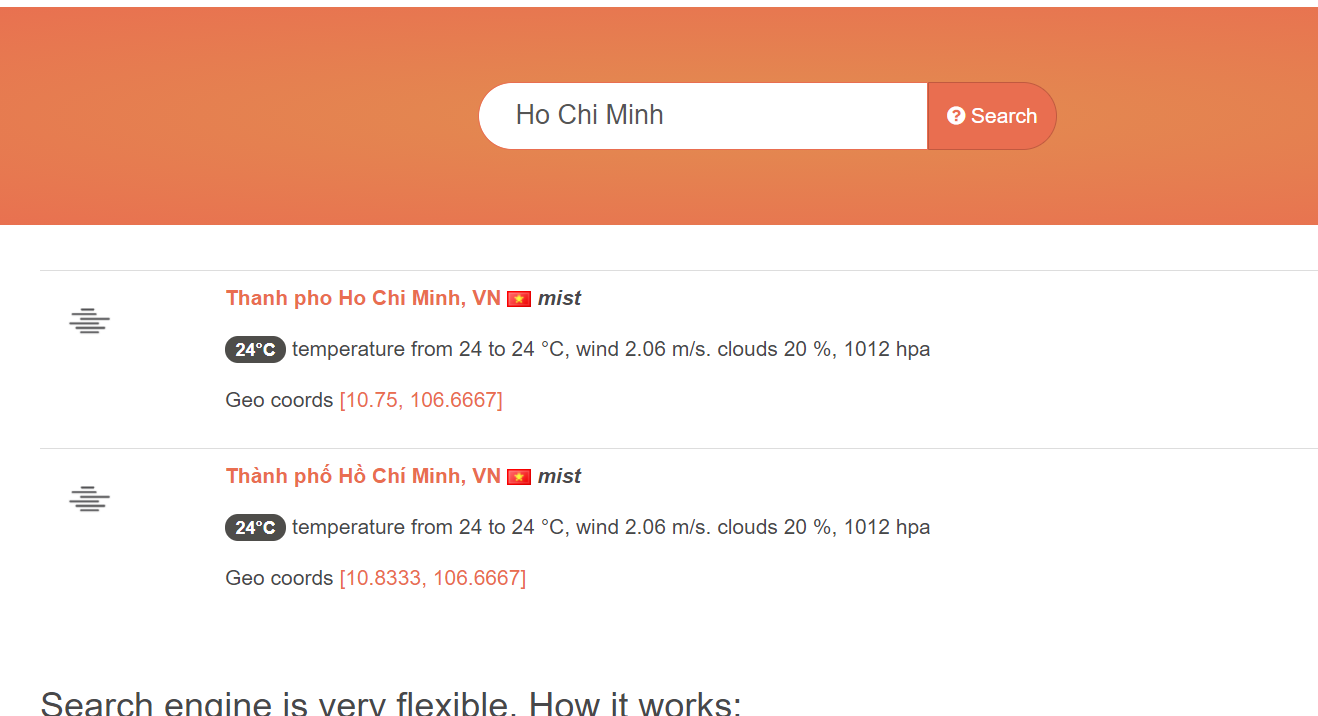
Steps:

1. Open “Open Weather” application
2. Fill Search field with data: “ Ho Chi Minh ”
3. Press Enter

Actual result: Not Found



Expectation result: displaying 2 rows of result as finding with “Ho Chi Minh”



**Reporter**: Quang Tran

**Priority**: Minor

**Environment**: all browsers

**Relevant Test case ID**: <Test case ID>

**Relevant Test script**: <Test script name>

The priority is the order in which team should resolve a defect. To decide a correct priority for a defect, I need the consultation of Product Owner/PM/client or tracking tools (e.g. Google Analytic) about pages/functions which user touch. Here are levels of the priority:

* Critical: should be fixed as soon as possible.
  + User can see this issue frequently;
  + It affects the system severely and system cannot be used. No work around for it.
* Major: it could be fixed during the normal development.
  + Big feature is broken completely, but it is still under the feature flag
  + Small features broken without workaround
* Minor: it could be fixed once the more serious defects have been fixed.
  + User rarely see it.
  + Small features broken with workaround

**PART II: UI Automation**

1. UI tests

TC01: Verifying the search result for multi types of input on Search field

TC02: Verifying that user could see forecast detail at n day in 8-day forecast

1. The automation framework

* Programming languages: Java (JDK 1.8)
* Maven 3.6.3 to build and manage project
* TestNG 6.14.3 to manage tests
* ReportNG 1.1.4 for local report
* Log4j 1.2.12 for logging
* Support to manage the run configuration: src/resources/setup.properties
* Support: data driven, multiple browser and cloud integration

1. Static code check result

We can use SonarQube to scan code for coding issues. Sorry I didn’t setup it to get result for now. Besides that, we could use SonarLint – an IDE extension to scan code during coding period.

1. Prevent flaky tests

Beside the execution speed of tests, flaky tests also a key factor for success of an automation framework. This framework applies few solutions:

* Using explicit waiting techniques before performing any actions through wrap commands on Selenium commands.
* Trying to repeat an action the second time if it is failed on the first time through try catch mechanism to prevent few lost DOM situations.
* Re-run the test in the case it was failed on the first time through RetryListener and TestListener. (No implement yet)

1. Test Report

* Report is generated through ReportNG. To get report please refer to README.md
* The framework just supports local report for now. To manage reports of runs in one place, we can consider integrating few tools: Report Portal (ReportPortal.io) or Allure report. E.g. Report Portal, starting report portal server from docker and adding the listener into testng file as:

<listener class-name="com.epam.reportportal.testng.ReportPortalTestNGListener" />

1. Data Driven

The framework supports data driven for tests through TestNG (DataProviderCenter class). It also support to share test data between tests through datamap.properties. Refer to README.md for some more information.

1. Multiple browser

The framework supports to run tests on some browsers:

* Mac os: chrome, firefox
* Windows os: chrome, firefox, internet explorer
* Cloud: chrome, firefox, internet explorer, ms edge, safari

1. CI/CD integration solution

Consider using Jenkins, Bamboo, … to trigger tests run whenever code merged, build or deploy. Combine with configure pipelines to visualize the process.

1. Cloud integration solution

The framework supports to run tests with the cloud service provider: Sauce Labs

1. Ability to run tests in parallel

The framework support to run tests in parallel through testNG configuration:

<suite name="Open Weather" verbose="2" parallel="methods" thread-count="1" data-provider-thread-count="4">

Beside that we could consider running tests in parallel through distribution systems: Jenkins, Selenium Grid.

1. Distribution execution

Configure Jenkins jobs to execute tests on multiple configurations through the cloud or Selenium Grid.