Test Documentation

For project AFP1-WEATHER

Tester: Bujpal Dorian Mano

Test document creation date: 2025. 11. 30.

Latest git pull date: 2025.11.30.

(Yesterday, we had an agreement to merge our frontend and backend branches, so that I can properly test the application and its workings)

Purpose of the test:

The goal of the testing phase is to verify that the weather application functions correctly from both the frontend and the backend sides. The test focuses on checking whether the system can accept a city name, communicate with the OpenWeatherMap API, process the returned data, and present the user with the forecast without errors. The intention is to catch functional issues early, to highlight potential bugs and to ensure a smooth user experience.

Scope of the test:

The test covers all core components of the current implementation:

- Frontend: input handling, refresh logic
- Backend: API request handling, JSON processing, error handling
- Data output: basic layout, attribute display, debug output (peldaAttributum.php)

Anything outside this (e.g. UI design, styles, mobile compatibility) is not included in this test round.

General notes:

Testing is done using the provided backend and frontend files, in their current state and form. The system uses a live API key, so responses may vary based on network conditions and the API availability. The goal is not to validate the correctness of the data, but to confirm the program reacts properly to different inputs and possible API responses.

Test Script / Test Procedure

Preparation:

- Ensure internet connection is active
- Have the necessary files accessible on the server
- Confirm that the OpenWeatherMap API key is active
- Open the weather app in a browser

Basic Functionality Test

Valid City Input

1. Locate the city input field
2. Enter a real city name (e.g. Budapest)
3. Click the refresh / search button or press Enter

Observe whether:

- The page reloads with the new city parameter
- The backend fetches data without errors
- The forecast is displayed correctly (dates, temperature, description)

Expected result:
Forecast data appears cleanly and in the correct format.

Input Validation Tests

Invalid City

1. Enter a random invalid string
2. Trigger the refresh
3. Observe how the system reacts

Expected result:
Either an error message is shown or no forecast data is displayed, but the system should not crash or output broken HTML.

Empty Input

1. Clear the input field completely
2. Trigger the refresh

Expected result:
The backend should not make an API request and should return either no data or a controlled error state.

Backend Behavior Test

peldaAttributum.php Debug Output

1. Open peldaAttributum.php in the browser
2. Check whether raw API data is printed correctly
3. Confirm fields like list, main, weather, and timestamps appear

Expected result:
Decoded JSON structure is visible and matches OpenWeather's format.

Error Handling Test

Forced Connection Error

1. Temporarily disable internet OR change the API URL in backend.php
2. Trigger a search

Expected result:
The system should show a controlled error message without breaking layout.

UI Response Test

1. Enter a valid city
2. Trigger refresh

Observe whether:

- The page updates smoothly
- No input fields break or disappear
- No console errors appear in the browser dev tools

Expected result:
Frontend handles the update without JS exceptions.

Code Behavior Test (JS)

1. Open Developer Tools -> Console
2. Enter a city and click refresh

Check whether:

- The JS function reads the input correctly
- URL parameters update as intended
- No undefined errors occur

Expected result:
The script logs no errors and updates the URL correctly.

Repeatability Test

1. Run three different valid cities in a row (e.g. Berlin, London, Budapest)

Confirm that:

- Each request returns correct results
- No leftover data from previous searches appears

Expected result:
The app consistently shows accurate results for each city.

Important notes about this test document:

Current system status

At the current state of development, the application is not yet fully integrated. The backend and the frontend components exist and function individually, but they are not fully connected. This results in several warnings and incomplete data flows during testing. The team has been focusing on building and validating the core logic before final integration, so these issues are expected at this point in the project. The purpose of these tests is not to evaluate the final user experience, but to verify that the core mechanics (input handling, API communication, data processing) behave as intended, in isolation. Although the system generates errors in its current state, the work completed so far demonstrates clear progress and provides a solid foundation for the upcoming integration tasks.

Reason for early testing

Even with the unfinished state of the site, early testing is necessary to identify functional errors before the components are fully merged. Running these tests now helps highlight issues that would be more difficult to identify or trace back later (e.g. invalid responses, inconsistent data formatting). Documenting this stage is also important for accountability, as it shows that development is ongoing, concrete progress has been made and that the remaining problems are already being tracked and prepared for resolution. This ensures that the team can move forward efficiently and avoid major blockers during the final integration phase.

First testing

Error: file_get_contents(city.list.json): Failed to open stream

During testing, an error occurred when the system attempted to load the city.list.json file using file_get_contents(). The error message indicated that the file could not be opened because the path was incorrect, even though the file existed in the project directory. This issue happened because PHP resolves relative paths based on the location of the executing script, not the working directory or project root (according to ChatGPT). Since index.php was located in the frontend folder, the relative path city.list.json pointed to frontend/city.list.json, which did not exist. As a result, the function failed, and PHP reported that it could not locate the file.

Error: Object of class stdClass could not be converted to string

While testing the frontend, the system produced a fatal error stating that an stdClass object could not be converted to a string. This occurred when the frontend attempted to output the entire $data object directly using “echo $data;”. Since $data is a decoded JSON object (PHP stdClass), it cannot be implicitly converted into a string for display. PHP therefore raised a fatal error.