# Software Requirements Specification (SRS)

## Project Title: Python Weather Web App

#### 1. Introduction

**1.1 Project Purpose**

* The purpose of this document is to outline the complete software requirements for a Python-based Weather Web Application.
* This document serves as a guide for all stakeholders including developers, testers, project managers, and clients.
* The Python Web Weather App is designed to provide real-time weather updates to users based on their selected or current location.
* It fetches weather data from a reliable third-party API (such as OpenWeatherMap) and displays it through a user-friendly web interface.
* The goal is to make weather information easily accessible, responsive, and visually intuitive.

**1.2 Project Scope**

* This Weather Web App allows users to input a city name and retrieve real-time weather information such as temperature, humidity, weather condition, wind speed, and more.
* The application fetches data from the OpenWeatherMap API and displays it in a user-friendly format.
* Optionally shows forecast (hourly/daily).
* Uses Python (Flask or Django) for backend logic.
* Uses HTML, CSS, JavaScript for frontend interface.
* Retrieves weather data using an external API.
* Can be extended to include geolocation support and personalized weather suggestions.

**1.3 Abbreviations & Definitions**

* API: Application Programming Interface
* GUI: Graphical User Interface
* HTTP: Hypertext Transfer Protocol
* JSON: JavaScript Object Notation
* SRS: Software Requirements Specification
* UI: User Interface
* REST: Representational State Transfer
* CRUD: Create, Read, Update, Delete

**Glossary & References**

* API – Application Programming Interface
* UI – User Interface
* JSON – JavaScript Object Notation
* REST – Representational State Transfer
* OpenWeatherMap – A weather data provider service
* Reference APIs: <https://openweathermap.org/api>

*2. Overall Description*

**2.1 Product Perspective**

* The Weather Web App is a standalone application using a client-server architecture.
* The client side is developed using HTML/CSS/JavaScript, and the server-side logic is implemented in Python using Flask.
* The app interacts with the OpenWeatherMap API to fetch weather data.

**2.2 Product Functions**

* Input a city name
* Fetch and display current weather details:

Temperature

Feels like

Weather condition

Humidity

Wind speed

* Error handling for invalid or empty inputs
* Secure API key management

**2.3 User Classes and Characteristics**

* General Users: Interested in checking weather conditions. No technical knowledge required.
* Administrator (optional): Responsible for monitoring API usage and system health.

**2.4 Operating Environment**

* Client: Web browsers (Chrome, Firefox, Edge, Safari)
* Server: Flask running on Linux/Windows/Mac OS
* Cloud Hosting: Heroku, AWS, or PythonAnywhere

**2.5 Design and Implementation Constraints**

* Must use Python 3.x and Flask
* Must connect to OpenWeatherMap API
* API key must be stored securely using environment variables
* Responsive design for desktop and mobile devices

*3. Specific Requirements*

**3.1 Functional Requirements**

* FR1: The system shall allow users to input a city name.
* FR2: The system shall validate the input field.
* FR3: The system shall fetch data from OpenWeatherMap API using the city name.
* FR4: The system shall display weather parameters.
* FR5: The system shall handle and display errors (e.g., invalid city name, no response).

**3.2 Non-Functional Requirements**

* NFR1: The system should respond within 3 seconds.
* NFR2: The app must be mobile responsive.
* NFR3: API keys must not be exposed.
* NFR4: The code must follow PEP8 standards.
* NFR5: Availability should be 99.9% if hosted in production.

**3.3 External Interface Requirements**

* User Interface: Form for input and area to display weather results.
* API Interface: Connects to OpenWeatherMap API.
* System Interface: Interaction between frontend (HTML/CSS/JS) and backend (Flask).

**3.4 Maintainability**

* Use of modular and well-commented code.
* Proper documentation and README for setup.
* Use of environment variables for configuration.

**3.5 Database Requirements**

* No database required for core weather data features.
* Optional logging of requests using SQLite.

**3.6 Document Validation**

* Document reviewed and validated by developers and clients.
* Requirements are verified against OpenWeatherMap API capabilities.

*4. Appendices*

**4.1 Sample API Response**

{

"weather": [{ "main": "Clear", "description": "clear sky", "icon": "01d" }],

"main": {

"temp": 298.15,

"feels\_like": 297.15,

"humidity": 40

},

"wind": { "speed": 3.6 },

"name": "New York"

}

**4.2 Project Structure**

weather\_app/

├── app/

├── static/

├── templates/

├── welcome.html

├── city.html

└── main.html

├── \_init\_.py

└── routes.py

├── database/

└── weather.db

├── preprocess.py

├── config.py

├── run.py

├── requirements.txt

└── README.md

**4.3 Useful Links**

* Flask Docs: https://flask.palletsprojects.com/
* OpenWeatherMap API: https://openweathermap.org/api
* Python-dotenv: <https://pypi.org/project/python-dotenv/>

*5. Requirements Validation*

**5.1 Requirements Review**

* Conduct peer reviews and walk-throughs
* Compare with customer expectations

**5.2 Inspection**

* Formal verification of each requirement with test cases
* Review input validation, error messages, and data accuracy

**5.3 Test Case Generation**

* TC1: Verify correct weather data retrieval for valid city
* TC2: Verify error message on empty input
* TC3: Verify error on invalid city input

**5.4 Reading**

* Independent reading by testers and developers to validate understanding

**5.5 Testability**

* Each function will be tested individually and in integration
* Output of each API call will be validated

**5.6 Modifiability**

* Use of environment configs, modular code, and comments for future updates

**5.7 Traceability**

* Each requirement is traceable to specific code modules and test cases
* Maintain traceability matrix in README or documentation