**Project Title: Full Stack Weather Forecast App**

**Problem Statement**

Build a full stack web application that allows users to search for any city and view a 3-day weather forecast with actionable suggestions (e.g., carry umbrella if rain is predicted, use sunscreen if temperature is high, etc.).  
The solution must be robust, production-ready, secure, and support automated CI/CD from development to deployment.

**Tech Stack Used**

**Frontend:**

* React (with Hooks)
* React Router
* Vite
* CSS

**Backend:**

* Flask (Python)
* Flask-CORS
* python-dotenv
* requests

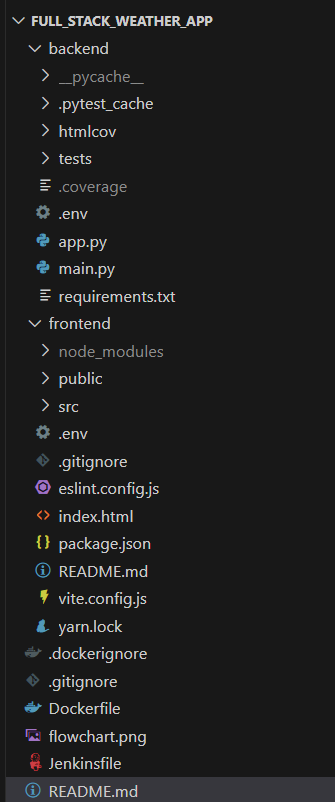
**Testing:**

* pytest
* pytest-cov

**CI/CD & Deployment:**

* Jenkins (automated build, test, and deployment)
* Docker (containerization)
* Render (cloud hosting)

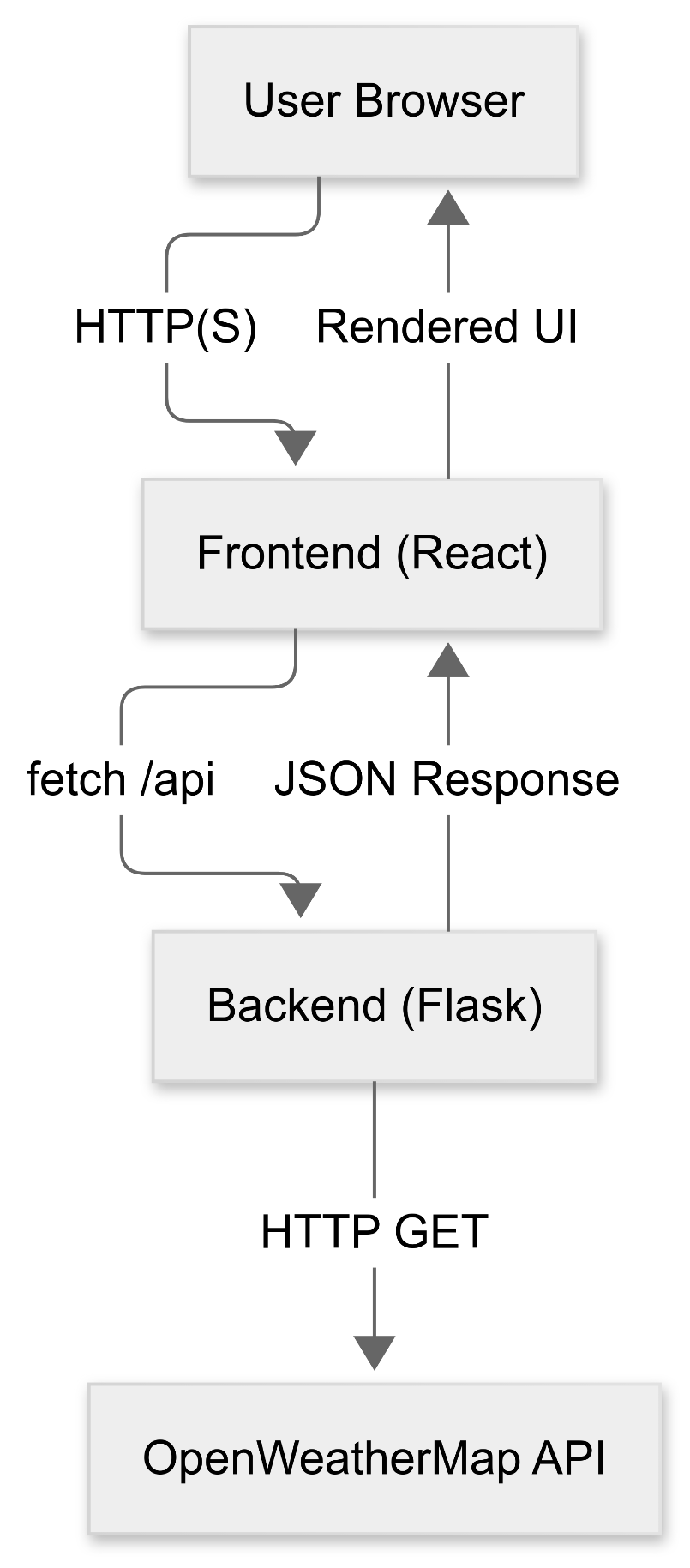
**Project Structure**



**Development to Deployment Flow**

1. **Development**
   * Developers build the frontend in React and the backend in Flask.
   * The backend /api endpoint fetches weather data from OpenWeatherMap, processes it, and returns a 3-day summary with smart suggestions (umbrella, sunscreen, warm clothes, wind, thunderstorm, etc.).
2. **Version Control**
   * All code is pushed to GitHub.
3. **Continuous Integration (CI)**
   * A GitHub webhook notifies Jenkins of new commits.
   * Jenkins pulls the latest code, builds the Docker image, and runs backend tests using pytest.
4. **Continuous Deployment (CD)**
   * If tests pass, Jenkins triggers a deploy hook on Render.
   * Render pulls the latest Docker image and deploys the app.
5. **Production**
   * The app is live on Render.
   * Users can search for cities and get weather forecasts with actionable suggestions.

**Flowchart:**



**Key Features**

* **Smart Weather Suggestions:**
  + "Carry umbrella" if rain is expected
  + "Use sunscreen lotion" if temperature > 40°C
  + "It's cold, wear warm clothes" if temperature < 10°C
  + "It's sunny, wear sunglasses" if clear
  + "It’s too windy, watch out!" if wind > 10 mph
  + "Don’t step out! A Storm is brewing!" if thunderstorm
* **Automated Testing:**
  + Backend logic is covered by pytest tests, including edge cases and error handling.
* **Secure & Production Ready:**
  + API keys and secrets are managed via environment variables and never exposed to the frontend.
  + Dockerized for consistent deployment.
  + Automated CI/CD pipeline ensures code quality and fast delivery.

**Non-Functional Requirements (NFRs) Covered**

**Performance:**

* Fast API responses and efficient data processing.
* React frontend is built and served as static files for quick load times.
* Dockerization ensures consistent, optimized runtime environments.

**Scalability:**

* Stateless backend (Flask) can be horizontally scaled.
* Frontend and backend can be scaled independently.

**Security:**

* API keys and secrets are stored in environment variables, never exposed to the frontend or committed to source control.
* CORS is controlled via Flask-CORS.
* Backend acts as a proxy to prevent direct exposure of third-party API keys.

**Reliability & Availability:**

* Graceful error handling for missing/invalid input and external API failures.
* Automated CI/CD ensures only tested, working code is deployed.

**Maintainability:**

* Clear separation of concerns (React for UI, Flask for API/business logic).
* Modular codebase, easy to extend.
* Automated tests ensure code changes do not break existing functionality.

**Observability:**

* Print statements for debugging (can be upgraded to structured logging).
* Test coverage with pytest-cov.

**Design Patterns, SOLID, 12 Factor, and HATEOAS Principles**

**Design Patterns:**

* MVC (conceptual): Flask backend (Controller/Model), React frontend (View).
* Service Layer: /api endpoint as a service layer.
* Adapter/Proxy: Backend adapts and secures external API data for frontend.
* Singleton: Flask app instance.
* Test Client & Mocking: Used in pytest for isolated, repeatable tests.
* Pipeline: Jenkinsfile implements the CI/CD pipeline pattern.

**SOLID Principles:**

* Single Responsibility: Each function/route has a single responsibility.
* Open/Closed: Logic is open for extension, closed for modification.
* Interface Segregation: Minimal, focused API endpoints.
* Dependency Inversion: External APIs are accessed via requests and can be mocked.

**12 Factor App:**

* Codebase in Git, dependencies declared, config via env vars, stateless processes, port binding, disposability, dev/prod parity, logs, and admin processes.

**Production Readiness**

* Dockerized for deployment on any cloud/container platform.
* Environment variables for all sensitive and environment-specific config.
* Jenkins pipeline automates build, test, and deployment.
* Graceful error handling and scalable, stateless backend.
* Can be extended with health checks and advanced logging.

**Testing, TDD, BDD & Quality**

* TDD: Backend logic is covered by pytest unit tests.
* BDD: Tests describe behavior (can be extended with pytest-bdd).
* Quality: Automated tests in CI pipeline, test coverage with pytest-cov, and code can be linted/formatted.

**Sensitive Information Protection**

* API keys and secrets are stored in .env files and environment variables, never in source control.
* Jenkins secrets are managed as credentials, not in the Jenkinsfile.
* Backend acts as a proxy, so API keys are never sent to the frontend.

**How to Run Locally**

**Backend:**

* cd backend
* pip install -r requirements.txt
* python main.py

**Frontend:**

* cd frontend
* npm install
* npm run dev

**Deployment**

* Push code to GitHub.
* Jenkins automatically builds, tests, and deploys the app to Render using Docker and deploy hooks.
* The app is live and accessible via the Render URL.

**Conclusion**

This project demonstrates a modern, production-ready, full stack weather app with automated CI/CD, robust error handling, secure secret management, and actionable weather insights for users.  
It follows best practices for architecture, security, testing, and deployment, and is easily extensible and maintainable for future growth.