JavaFX Weather Forecast App

Complete Implementation Guide & Documentation

Table of Contents

Project Overview

Description

A modern desktop weather application built with JavaFX that provides real-time weather information for any city worldwide using the OpenWeatherMap API.

Key Features

- **Real-time Weather Data**: Current temperature, humidity, wind speed, and conditions
- **Modern UI**: Clean, responsive interface with gradient backgrounds and weather icons
- **Error Handling**: Comprehensive error management for network issues and invalid inputs
- **Asynchronous Operations**: Non-blocking API calls to maintain UI responsiveness
- **MVC Architecture**: Clean separation of concerns for maintainable code

Technologies Used

- **Java 11+**: Core programming language

- **JavaFX 17**: User interface framework
- **Maven**: Build and dependency management
- **Gson**: JSON parsing library
- **OpenWeatherMap API**: Weather data provider
- **CSS**: Custom styling for modern appearance

Architecture & Design Patterns

Model-View-Controller (MVC) Pattern

The application follows the MVC architectural pattern for clean separation of concerns:

Model Layer

- `Weather.java`: Core data model representing weather information
- `WeatherData.java`: Data Transfer Object (DTO) for API responses

Responsibilities:

- Data representation and validation
- Business logic for data transformation
- API response mapping

View Layer

- `WeatherView.java`: JavaFX UI components and layout management
- `weather-app.css`: Styling and visual appearance

Responsibilities:

- User interface rendering
- User input collection
- Visual feedback and animations

Controller Layer

- `WeatherController.java`: Business logic and user interaction handling

Responsibilities:

- Event handling and user input processing

- Coordination between Model and View
- Application flow control

Service Layer Architecture

Service Classes

- **WeatherService.java**: API communication and business rules
- **HttpUtil.java**: HTTP operations utility
- **JsonUtil.java**: JSON parsing utility

Benefits:

- Reusable components across different parts of the application
- Centralized error handling
- Easy unit testing and mocking

Project Structure

```
weather-app/
├── pom.xml                          # Maven configuration
├── README.md                        # Project documentation
└── src/
    └── main/
        ├── java/
        │   └── com/
        │       └── weather/
        │           └── app/
        │               ├── WeatherApp.java          # Main application class
        │               ├── controller/
        │               │   └── WeatherController.java # User interaction logic
        │               ├── model/
        │               │   ├── Weather.java         # Weather data model
        │               │   └── WeatherData.java     # API response DTO
        │               ├── service/
        │               │   └── WeatherService.java  # API service layer
        │               ├── util/
        │               │   ├── HttpUtil.java        # HTTP utility functions
        │               │   └── JsonUtil.java        # JSON parsing utilities
        │               └── view/
        │                   └── WeatherView.java     # JavaFX UI components
        └── resources/
            └── styles/
                └── weather-app.css                  # Application styling
```

Implementation Details

1. Main Application (`WeatherApp.java`)

java

```
public class WeatherApp extends Application {
    private static final String API_KEY = "dced2373c5f4ce53e966660cc5d2bff0";

    @Override
    public void start(Stage primaryStage) {
        // Initialize MVC components
        WeatherView view = new WeatherView();
        WeatherController controller = new WeatherController(view, API_KEY);

        // Configure and display stage
        Scene scene = view.createScene();
        primaryStage.setTitle("Weather Forecast App");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Key Responsibilities:

- Application lifecycle management
- MVC component initialization
- JavaFX stage configuration
- Error handling for startup failures

2. Data Models

Weather Model (`Weather.java`)

java

```
public class Weather {
    private String cityName;
    private double temperature;
    private int humidity;
    private double windSpeed;
    private String condition;
    private String description;
    private String iconCode;

    // Constructors, getters, setters, and toString()
}
```

API Response DTO (`WeatherData.java`)

```java
public class WeatherData {
    @SerializedName("name")
    private String cityName;

    @SerializedName("main")
    private Main main;

    // Nested classes for API structure
    public static class Main {
        @SerializedName("temp")
        private double temperature;
        // ... other fields
    }

    // Conversion method to Weather model
    public Weather toWeather() {
        // Transform API response to domain model
    }
}
```

Design Benefits:

- Clear separation between API structure and domain model
- Gson annotations for automatic JSON mapping
- Type safety and validation

3. Service Layer Implementation

Weather Service (`WeatherService.java`)

```java
```

```
public class WeatherService {
    private static final String API_BASE_URL = "https://api.openweathermap.org/data/2.5/weather";

    public Weather getWeather(String cityName) throws IOException {
        validateCityName(cityName);
        String url = buildApiUrl(cityName);
        String jsonResponse = HttpUtil.get(url);
        WeatherData weatherData = JsonUtil.fromJson(jsonResponse, WeatherData.class);
        return weatherData.toWeather();
    }

    public CompletableFuture<Weather> getWeatherAsync(String cityName) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                return getWeather(cityName);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        });
    }
}
```

Features:

- Synchronous and asynchronous API calls
- Input validation and sanitization
- URL construction and parameter encoding
- Error handling and custom exceptions

HTTP Utility (`HttpUtil.java`)

```java
```

```
public class HttpUtil {
    private static final int TIMEOUT = 10000; // 10 seconds

    public static String get(String urlString) throws IOException {
        URL url = new URL(urlString);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();

        // Configure connection with timeouts and headers
        connection.setRequestMethod("GET");
        connection.setConnectTimeout(TIMEOUT);
        connection.setReadTimeout(TIMEOUT);

        // Handle response codes and read data
        int responseCode = connection.getResponseCode();
        if (responseCode == HttpURLConnection.HTTP_OK) {
            return readResponse(connection);
        } else {
            throw new IOException("HTTP Error: " + responseCode);
        }
    }
}
```

Features:

- Configurable timeouts
- Proper resource management
- HTTP status code handling
- UTF-8 encoding support

4. User Interface Implementation

View Layer (`WeatherView.java`)

java

```
public class WeatherView {
    private VBox root;
    private TextField cityInput;
    private Button getWeatherButton;
    private VBox weatherDisplay;

    public WeatherView() {
        initializeComponents();
        layoutComponents();
        styleComponents();
    }

    private void layoutComponents() {
        // Create responsive layout with proper spacing
        HBox inputBox = new HBox(10);
        inputBox.setAlignment(Pos.CENTER);

        weatherDisplay.setAlignment(Pos.CENTER);
        root.setAlignment(Pos.TOP_CENTER);
        root.setPadding(new Insets(20));
    }
}
```

UI Features:

- Responsive layout with proper spacing
- Professional typography and fonts
- Loading states and visual feedback
- Weather icon integration
- Error message display

Controller Logic (`WeatherController.java`)

```
java
```

```
public class WeatherController {
    public WeatherController(WeatherView view, String apiKey) {
        this.view = view;
        this.weatherService = new WeatherService(apiKey);
        initializeEventHandlers();
    }

    private void handleGetWeather() {
        String cityName = view.getCityInput().getText().trim();

        setLoadingState(true);

        Task<Weather> weatherTask = new Task<Weather>() {
            @Override
            protected Weather call() throws Exception {
                return weatherService.getWeather(cityName);
            }

            @Override
            protected void succeeded() {
                Platform.runLater(() -> {
                    displayWeather(getValue());
                    setLoadingState(false);
                });
            }
        };

        new Thread(weatherTask).start();
    }
}
```

Controller Features:

- Event-driven architecture
- Asynchronous task management
- UI state management
- Error handling and user feedback

API Integration

OpenWeatherMap API Configuration

API Endpoint

```
https://api.openweathermap.org/data/2.5/weather
```

Request Parameters

- **q**: City name (required)
- **appid**: API key (required)
- **units**: Temperature units (metric for Celsius)

Sample API Call

```
https://api.openweathermap.org/data/2.5/weather?q=Mumbai&appid=YOUR_API_KEY&units=metric
```

Response Structure

```json
{
  "name": "Mumbai",
  "main": {
    "temp": 27.1,
    "humidity": 84
  },
  "weather": [
    {
      "main": "Clouds",
      "description": "overcast clouds",
      "icon": "04d"
    }
  ],
  "wind": {
    "speed": 9.2
  }
}
```

Error Handling Strategy

Network Errors

- **Connection Timeout**: 10-second timeout with retry suggestion

- **HTTP 404**: City not found error with user-friendly message
- **HTTP 401**: Invalid API key error
- **Other HTTP Errors**: Generic service unavailable message

Data Validation

- **City Name**: Minimum 2 characters, non-empty validation
- **API Response**: JSON structure validation
- **Data Types**: Type safety with proper casting

User Interface Design

Design Principles

Modern Material Design

- Clean, minimalist interface
- Gradient backgrounds for visual appeal
- Consistent spacing and typography
- Professional color scheme

User Experience Features

- **Loading States**: Visual feedback during API calls
- **Error Messages**: Clear, actionable error information
- **Input Validation**: Real-time form validation
- **Responsive Layout**: Proper component sizing and alignment

CSS Styling (`weather-app.css`)

```css
```

```css
.root {
    -fx-background-color: linear-gradient(to bottom, #87CEEB, #E0F6FF);
    -fx-font-family: "Segoe UI", Arial, sans-serif;
}

.weather-button {
    -fx-background-color: #3498DB;
    -fx-text-fill: white;
    -fx-font-weight: bold;
    -fx-background-radius: 5px;
    -fx-effect: dropshadow(gaussian, rgba(0,0,0,0.2), 3, 0, 0, 2);
}

.weather-display {
    -fx-background-color: rgba(255, 255, 255, 0.9);
    -fx-background-radius: 10px;
    -fx-padding: 20px;
    -fx-effect: dropshadow(gaussian, rgba(0,0,0,0.3), 5, 0, 0, 3);
}
```

Styling Features:

- Gradient backgrounds
- Drop shadows for depth
- Rounded corners for modern look
- Hover effects for interactivity
- Professional color palette

Setup & Installation

Prerequisites

Required Software

1. **Java Development Kit (JDK) 11 or higher**
   - Download from Oracle
   - Verify installation: `java --version`
2. **Apache Maven 3.6 or higher**
   - Download from Maven

- Verify installation: `mvn --version`

3. **OpenWeatherMap API Key**
   - Sign up at OpenWeatherMap
   - Get free API key (1,000,000 calls/month)

Optional Tools

- **IntelliJ IDEA** or **Eclipse** for development
- **Git** for version control

Installation Steps

Step 1: Create Project Structure

```bash
# Create main project directory
mkdir weather-app
cd weather-app

# Create Maven directory structure
mkdir -p src/main/java/com/weather/app/{controller,model,service,util,view}
mkdir -p src/main/resources/styles
```

Step 2: Copy Project Files

1. Create `pom.xml` in the root directory
2. Copy all Java files to their respective packages
3. Copy `weather-app.css` to `src/main/resources/styles/`

Step 3: Configure API Key

Update `WeatherApp.java`:

```java
private static final String API_KEY = "YOUR_ACTUAL_API_KEY_HERE";
```

Step 4: Build and Run

```bash
```

```
# Clean and compile
mvn clean compile

# Run the application
mvn javafx:run
```

Commands Reference

Maven Commands

Project Management

```bash
# Clean build artifacts
mvn clean

# Compile source code
mvn compile

# Run all tests
mvn test

# Package as JAR
mvn package

# Clean and package
mvn clean package
```

JavaFX Specific

```bash
# Run JavaFX application
mvn javafx:run

# Debug mode
mvn javafx:run -Djavafx.args="--debug"

# Run with system properties
mvn javafx:run -Djavafx.args="-Dprism.verbose=true"
```

Dependency Management

```bash
# Download dependencies
mvn dependency:resolve

# Display dependency tree
mvn dependency:tree

# Update dependencies
mvn versions:use-latest-versions
```

Java Commands (Alternative)

Direct Execution

```bash
# Compile manually (if Maven not available)
javac --module-path "javafx/lib" --add-modules javafx.controls,javafx.fxml -cp "gson.jar" src/main/java/com/weather/ap

# Run manually
java --module-path "javafx/lib" --add-modules javafx.controls,javafx.fxml -cp "src/main/java;gson.jar" com.weather.app.
```

IDE Commands

IntelliJ IDEA

```bash
# Import Maven project
File → Open → Select weather-app folder

# Run application
Right-click WeatherApp.java → Run 'WeatherApp.main()'

# Debug application
Right-click WeatherApp.java → Debug 'WeatherApp.main()'
```

Eclipse

```bash
```

```
# Import project
File → Import → Existing Maven Projects

# Run application
Right-click project → Run As → Java Application
```

System Commands

Verify Installation

```bash
# Check Java version
java --version

# Check Maven version
mvn --version

# Check JavaFX modules
java --list-modules | grep javafx
```

Testing Guide

Manual Testing Scenarios

Valid City Names

```bash
Test Cases:
- "London" → Should display UK weather
- "New York" → Should display US weather
- "Tokyo" → Should display Japan weather
- "Mumbai" → Should display India weather
- "Paris" → Should display France weather
```

Invalid Inputs

```bash
```

Test Cases:
- "" (empty) → Button should be disabled
- "XYZ123" → Should show "City not found"
- "A" → Should show validation error
- Special characters → Should handle gracefully

Network Scenarios

bash

Test Cases:
- No internet → Should show connection error
- Slow connection → Should show loading state
- API service down → Should show service error
- Invalid API key → Should show authentication error

UI Functionality

bash

Test Cases:
- Button states → Loading/enabled/disabled
- Input field → Clear after successful search
- Weather display → Show/hide based on results
- Error messages → Clear, actionable text
- Weather icons → Load correctly from API

Error Testing

API Errors

bash

 # Test invalid API key
Replace API_KEY with "invalid_key" → Should show authentication error

 # Test network timeout
Disconnect internet during API call → Should show timeout error

 # Test malformed response
Mock service with invalid JSON → Should show parsing error

UI Edge Cases

```bash
# Test window resizing
Resize application window → Layout should remain proper

# Test rapid clicking
Click "Get Weather" multiple times quickly → Should handle gracefully

# Test long city names
Enter very long city name → Should handle without UI breaking
```

Performance Testing

Response Times

- API calls should complete within 10 seconds
- UI should remain responsive during API calls
- Loading states should appear immediately

Memory Usage

- Application should not have memory leaks
- Weather icon images should be properly disposed
- HTTP connections should be closed properly

Troubleshooting

Common Issues and Solutions

Maven Issues

Problem: `mvn: command not found`

```bash
Solution:
1. Install Maven from https://maven.apache.org/download.cgi
2. Add Maven/bin to system PATH
3. Restart terminal and verify: mvn --version
```

Problem: `No plugin found for prefix 'javafx'`

```
bash

Solution:
1. Ensure you're in directory with pom.xml
2. Verify pom.xml contains javafx-maven-plugin
3. Run: mvn clean compile first
```

JavaFX Issues

Problem: `Module javafx.controls not found`

```
bash

Solution:
1. Verify JavaFX dependencies in pom.xml
2. Check Java version compatibility (11+)
3. Clear Maven cache: mvn dependency:purge-local-repository
```

Problem: CSS not loading

```
bash

Solution:
1. Verify CSS file location: src/main/resources/styles/
2. Check file path in WeatherView.java
3. Ensure CSS file is included in build
```

API Issues

Problem: `City not found` for valid cities

```
bash

Solution:
1. Check API key validity
2. Verify internet connection
3. Try different city name format (e.g., "London,UK")
4. Check OpenWeatherMap service status
```

Problem: `Authentication failed`

```
bash
```

```
Solution:
1. Verify API key is correct
2. Check if API key is activated (may take few minutes)
3. Ensure no extra spaces in API key
4. Check API key usage limits
```

Network Issues

Problem: `Connection timeout`

```bash
Solution:
1. Check internet connectivity
2. Verify firewall settings
3. Try different network
4. Check if corporate proxy blocks API calls
```

Problem: SSL/HTTPS errors

```bash
Solution:
1. Update Java to latest version
2. Check system date/time
3. Verify SSL certificates are up to date
```

Debug Mode

Enable Debug Logging

```bash
# Run with debug output
mvn javafx:run -X

# Java system properties for debugging
-Djavafx.verbose=true
-Dprism.verbose=true
-Dcom.sun.javafx.isLoggingEnabled=true
```

Log Analysis

```bash
# Common log patterns to look for:
- "HTTP Error" → API connectivity issues
- "JSON parsing" → Data format problems
- "JavaFX Platform" → UI thread issues
- "Connection timeout" → Network problems
```

Performance Optimization

Memory Management

```bash
# JVM memory options
-Xms256m -Xmx512m

# Garbage collection tuning
-XX:+UseG1GC -XX:MaxGCPauseMillis=200
```

JavaFX Performance

```bash
# Hardware acceleration
-Dprism.forceGPU=true

# Image caching
-Djavafx.animation.fullspeed=true
```

Future Enhancements

Phase 1 Enhancements (Short-term)

Extended Weather Information

```java
```

```
// Add to Weather model
private double feelsLike;
private int visibility;
private String sunrise;
private String sunset;
private double pressure;
```

5-Day Forecast

```java
// New service method
public List<Weather> getFiveDayForecast(String cityName) {
    // Implementation for forecast API endpoint
}
```

Location-based Weather

```java
// GPS integration
public Weather getWeatherByLocation(double lat, double lon) {
    // Use coordinates instead of city name
}
```

Phase 2 Enhancements (Medium-term)

Dark/Light Theme Toggle

```css
/* Add theme switching CSS classes */
.root.dark-theme {
    -fx-background-color: linear-gradient(to bottom, #2C3E50, #34495E);
}

.root.light-theme {
    -fx-background-color: linear-gradient(to bottom, #87CEEB, #E0F6FF);
}
```

Favorite Cities Management

```java
// New model for favorites
public class FavoriteCity {
    private String cityName;
    private String countryCode;
    private LocalDateTime lastUpdated;
}

// Persistence layer
public class FavoritesService {
    public void saveFavorite(String cityName);
    public List<FavoriteCity> getFavorites();
    public void removeFavorite(String cityName);
}
```

Weather Alerts and Notifications

```java
// Alert system
public class WeatherAlert {
    private String alertType;
    private String message;
    private LocalDateTime timestamp;
    private AlertSeverity severity;
}
```

Phase 3 Enhancements (Long-term)

Weather Maps Integration

```java
// Map component
public class WeatherMapView extends StackPane {
    private WebView mapWebView;

    public void showWeatherMap(String cityName) {
        // Integrate with mapping service
    }
}
```

Weather History and Trends

```java
// Historical data
public class WeatherHistory {
    private String cityName;
    private List<Weather> historicalData;
    private WeatherTrends trends;
}
```

Multiple Language Support

```properties
# resources/messages_en.properties
app.title=Weather Forecast App
button.getWeather=Get Weather
label.temperature=Temperature
label.humidity=Humidity

# resources/messages_es.properties
app.title=Aplicación de Pronóstico del Tiempo
button.getWeather=Obtener Clima
label.temperature=Temperatura
label.humidity=Humedad
```

Configuration Management

```java
// Settings model
public class AppSettings {
    private TemperatureUnit temperatureUnit;
    private String defaultCity;
    private boolean autoRefresh;
    private int refreshInterval;
    private ThemeMode themeMode;
}
```

Technical Improvements

Unit Testing Framework

```java
// Test structure
@Test
public void testWeatherServiceValidCity() {
    WeatherService service = new WeatherService("test-api-key");
    Weather weather = service.getWeather("London");
    assertNotNull(weather);
    assertEquals("London", weather.getCityName());
}

@Test
public void testInvalidCityThrowsException() {
    WeatherService service = new WeatherService("test-api-key");
    assertThrows(IOException.class, () -> {
        service.getWeather("InvalidCity123");
    });
}
```

Caching System

```java
// Weather cache implementation
public class WeatherCache {
    private final Map<String, CachedWeather> cache = new ConcurrentHashMap<>();
    private final Duration cacheExpiry = Duration.ofMinutes(10);

    public Optional<Weather> getCachedWeather(String cityName) {
        CachedWeather cached = cache.get(cityName.toLowerCase());
        if (cached != null && !cached.isExpired()) {
            return Optional.of(cached.getWeather());
        }
        return Optional.empty();
    }
}
```

Error Reporting System

```java
```

```java
// Error tracking
public class ErrorReporter {
    public void reportError(String component, Exception error, Map<String, String> context) {
        // Log error with context information
        // Optionally send to error tracking service
    }
}
```

Database Integration

```java
// SQLite database for local storage
public class WeatherDatabase {
    public void saveWeatherRecord(Weather weather);
    public List<Weather> getWeatherHistory(String cityName, LocalDate from, LocalDate to);
    public List<String> getSearchHistory();
}
```

Conclusion

This JavaFX Weather Forecast App demonstrates modern software development practices with:

- **Clean Architecture**: MVC pattern with proper separation of concerns
- **Professional UI**: Modern design with responsive layouts and visual feedback
- **Robust Error Handling**: Comprehensive error management and user feedback
- **Asynchronous Operations**: Non-blocking API calls for better user experience
- **Extensible Design**: Well-structured code ready for future enhancements

The application successfully integrates with external APIs, provides a professional user interface, and follows industry best practices for desktop application development.

Total Lines of Code: ~1,200 lines **Development Time**: ~8-12 hours for complete implementation **Skill Level**: Intermediate to Advanced Java/JavaFX

Document Version: 1.0
Last Updated: July 26, 2025
Author: JavaFX Weather App Development Team
