# Lab 8

Distributed Software Systems
UNIBO - 2024/25

Anouk Wuyts - 1900124167

Davide De Rosa - 1186536

# Mashup

A **mashup** is a web application that uses content from **more than one source** to **create a single new service** displayed in a *single interface.*

The term implies **easy**, **fast integration**, frequently using **open application programming interfaces** (**open API**) and data sources to produce enriched results that were not necessarily the original reason for producing the raw source data.

For example, a user could combine the addresses and photographs of their library branches with a *Google Map* to create a **map mashup**.

The main drivers of a mashup are **data aggregation** and **visualization**.

# Mashup - Key Points

1. **Integration of Content**: mashups involve combining content and services from multiple sources to create a new, integrated web application. This could include data from various websites, like maps, social media, news, or other APIs

2. **User-Focused**: mashups are typically user-centric and are designed to provide a seamless and personalized experience for end-users. They are used to share new business ideas, to increase agility, and to speed up development reducing costs

3. **Client-Side**: most mashups are implemented on the *client side*, meaning the integration and presentation of data occur in the user's browser. JavaScript and HTML are commonly used to create mashup applications

A classic example of a mashup is a map that combines data from Google Maps and real-time traffic information.

# Weather Forecast Mashup

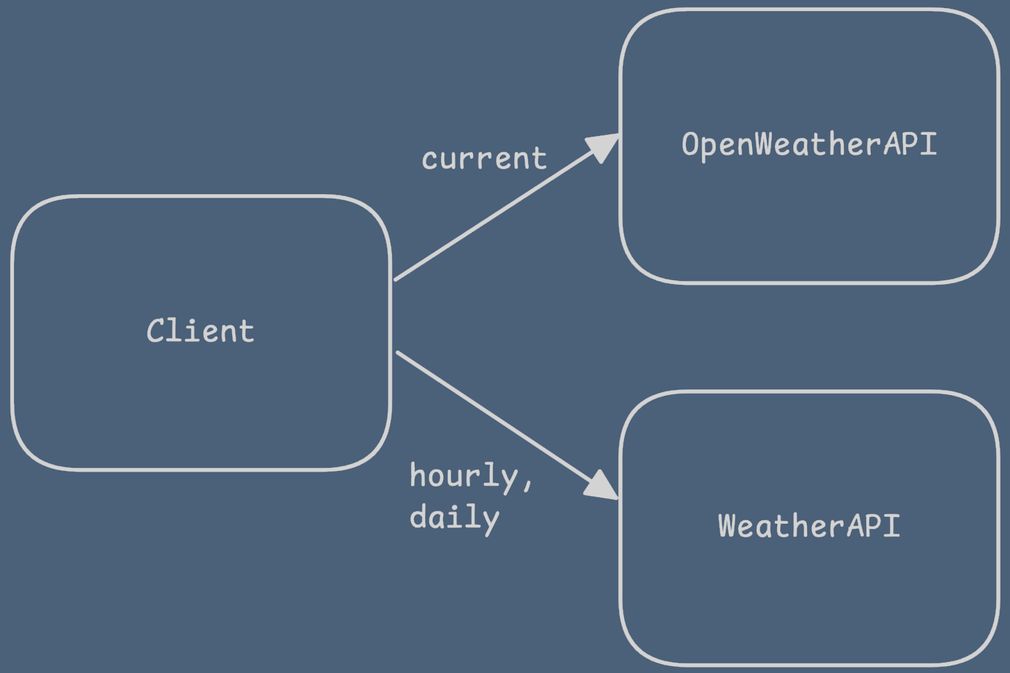
We created a simple **Weather Forecast Mashup** by combining 2 *different API sources* (*OpenWeatherMap* and *WeatherAPI*) for the city of **Bologna**.

We used **HTML** and **CSS** to create the blank web page.

We used **JavaScript** to get the data from the API’s and display it on the page.

# Mashup Architecture

By combining the two different API's, we show on the Client side the data returned from the requests made. The *current weather* request is made to the *OpenWeatherAPI*, while the other two are made to the *WeatherAPI* service.



# Mashup Implementation

Using **JavaScript** we fetched the data from the different API's and displayed it inside the HTML page.

The example shows the code for the *Current weather* part.

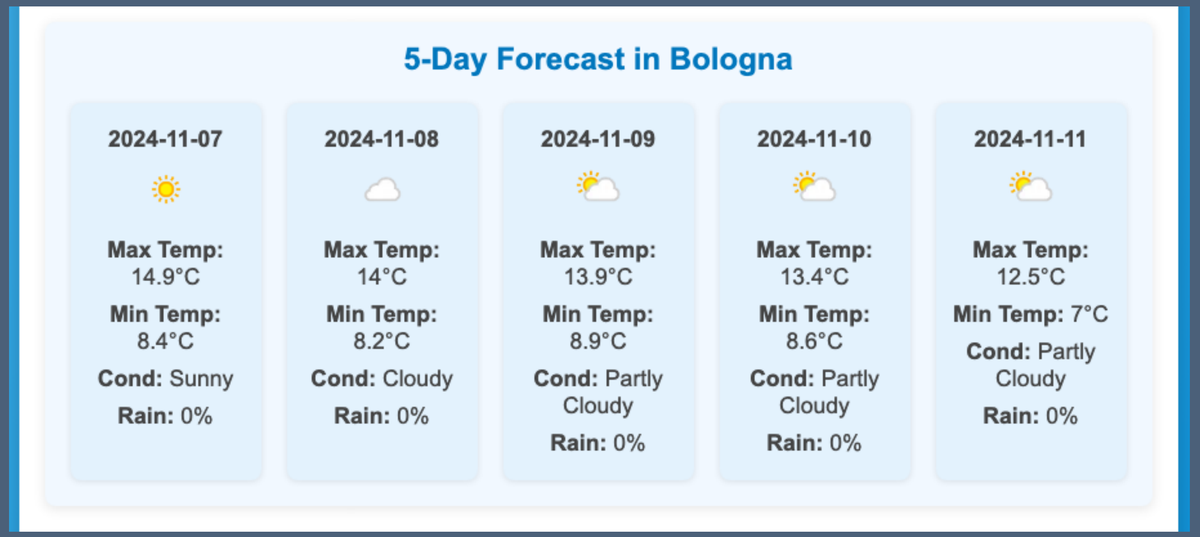

```
async function fetchCurrentWeather() {
    try {
        const response = await fetch(API_URL_CURRENT);
        const data = await response.json();
        displayCurrentWeather(data);
    } catch (error) {
        console.error("Error fetching current weather:", error);
    }
}
```

```
function displayCurrentWeather(data) {
    const temperature = data.main.temp;
    const feelsLike = data.main.feels_like;
    const weatherCondition = data.weather[0].main;
    const description = data.weather[0].description;
    const iconUrl = `http://openweathermap.org/img/wn/${data.weather[0].icon}@2x.png`;
    const humidity = data.main.humidity;
    const windSpeed = data.wind.speed;
    const windDirection = data.wind.deg;
    const visibility = data.visibility / 1000;
    const rainLastHour = data.rain ? data.rain["1h"] : 0;

    // Render HTML content
    currentWeatherData.innerHTML = `
        <img src="${iconUrl}" alt="${description}" />
        <p><strong>Temperature:</strong> ${temperature}°C</p>
        <p><strong>Feels Like:</strong> ${feelsLike}°C</p>
        <p><strong>Condition:</strong> ${weatherCondition} (${description})</p>
        <p><strong>Humidity:</strong> ${humidity}%</p>
        <p><strong>Wind:</strong> ${windSpeed} m/s, ${windDirection}°</p>
        <p><strong>Visibility:</strong> ${visibility} km</p>
        <p><strong>Rain (last hour):</strong> ${rainLastHour} mm</p>
    `;
}
```

# Mashup Implementation

The same logic is being applied for the *Hourly* and *Daily weather* forecast.




# Web Service

A **web service** is a software system identified by a **URI** and designed to support **interoperable machine-to-machine interaction over a network**.

It has an **interface** described in a machine-processable format (specifically **WSDL**).

Other systems interact with the web service using **SOAP-messages**, typically conveyed using **HTTP** with an **XML** serialization in conjunction with other web-related standards.

Companies use these technologies as a mechanism for allowing their applications to cross network boundaries and communicate with those of their partners, customers and suppliers.

# Web Service - Key Points

1. **Interoperable Data Exchange**: web services are designed for interoperable data exchange between different software systems. They are not focused on user interfaces but rather on providing machine-to-machine communication. They are the key components in web mashups

2. **Standardized Protocols**: web services often use standardized communication protocols like **SOAP** (Simple Object Access Protocol) or **REST** (Representational State Transfer) for data exchange

3. **Server-Side**: web services are typically implemented on the server side, exposing APIs that can be accessed by various clients, including web applications, mobile apps, and other software systems. They are essential so that other applications can use the functionality of your program once it is exposed to the network

Examples of web services include weather data APIs, payment processing APIs, or social media APIs like Twitter's API, which allows developers to access and interact with the data.

# Weather Forecast Web Service

We created a simple **Weather Forecast Web Service** that allows users to retrieve weather forecasts for specific locations by sending a request to the service.

We implemented the *web service* using **Python** with the use of **Flask** and **requests** as the main libraries.

*Flask* is used to expose **RESTful API** endpoints, *requests* is used to make HTTP requests to the *weather service*.


```python
def fetch_weather_data(endpoint, params):
    params['key'] = API_KEY
    response = requests.get(f"{API_URL}/{endpoint}.json", params=params)

    return response.json() if response.status_code == 200 else None
```

# Web Service Architecture

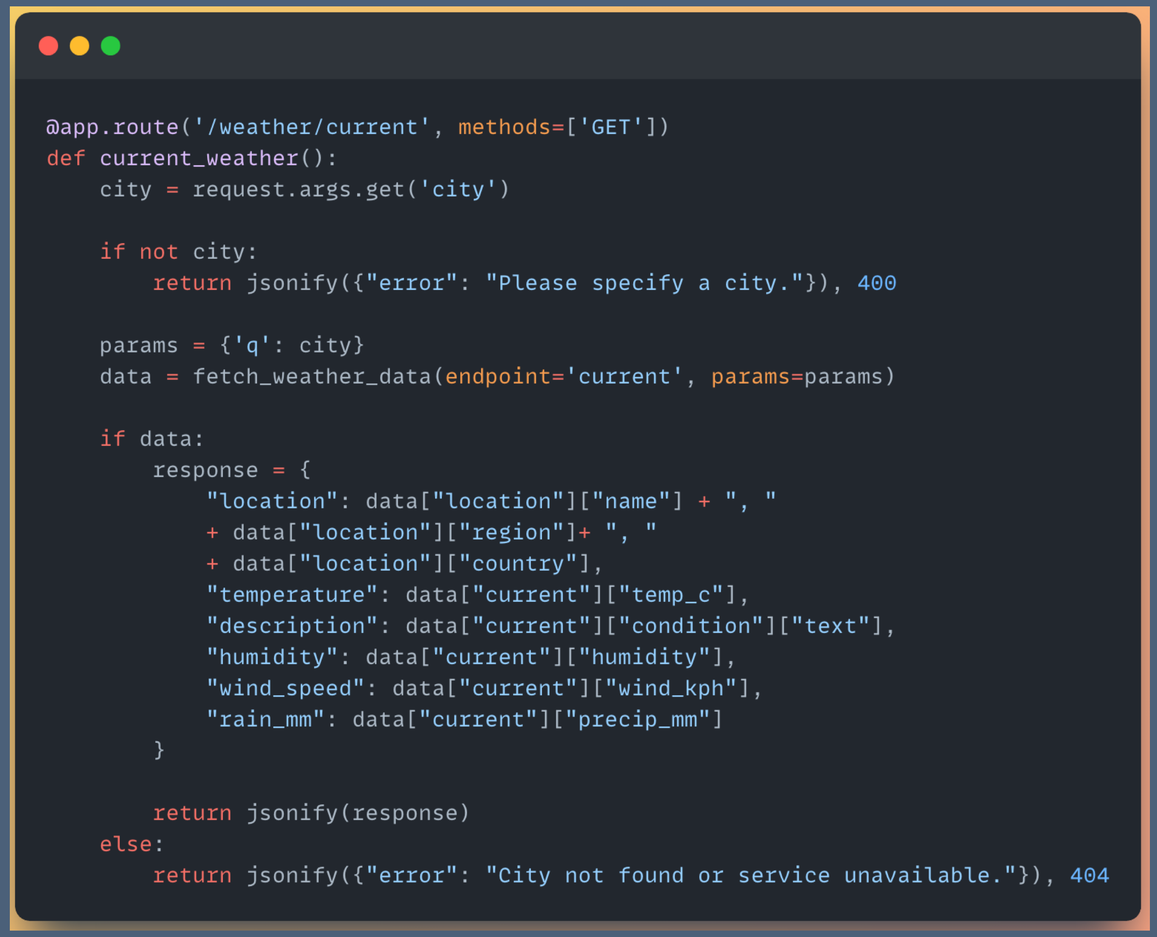
The *web service* acts as a **wrapper** for the **WeatherAPI** API's, allowing users to call our service and get the weather forecast requested.


By exposing **RESTful API's**, every user can access the web service by making HTTP requests to the service.

# Web Service - Current Weather

The **Current Weather API** asks for a *city* as a query parameter and returns a **JSON Object** with the current weather data of the city.

```
{
    "description": "Mist",
    "humidity": 88,
    "location": "Bologna, Emilia-Romagna, Italy",
    "rain_mm": 0.0,
    "temperature": 12.4,
    "wind_speed": 3.6
}
```



```python
@app.route('/weather/current', methods=['GET'])
def current_weather():
    city = request.args.get('city')

    if not city:
        return jsonify({"error": "Please specify a city."}), 400

    params = {'q': city}
    data = fetch_weather_data(endpoint='current', params=params)

    if data:
        response = {
            "location": data["location"]["name"] + ", "
            + data["location"]["region"]+ ", "
            + data["location"]["country"],
            "temperature": data["current"]["temp_c"],
            "description": data["current"]["condition"]["text"],
            "humidity": data["current"]["humidity"],
            "wind_speed": data["current"]["wind_kph"],
            "rain_mm": data["current"]["precip_mm"]
        }

        return jsonify(response)
    else:
        return jsonify({"error": "City not found or service unavailable."}), 404
```

# Web Service - Hourly Weather

The **Hourly Weather API** asks for a *city* as a query parameter and returns a **JSON Object** with the hourly weather data of the city for the current day.

```python
@app.route('/weather/hourly', methods=['GET'])
def hourly_forecast():
    city = request.args.get('city')

    if not city:
        return jsonify({"error": "Please specify a city."}), 400

    params = {'q': city}
    data = fetch_weather_data('forecast', params)

    if data:
        hourly_data = [
            {
                "time": data["forecast"]["forecastday"][0]["hour"][i]["time"],
                "temperature": data["forecast"]["forecastday"][0]["hour"][i]["temp_c"],
                "description": data["forecast"]["forecastday"][0]["hour"][i]["condition"]["text"],
                "humidity": data["forecast"]["forecastday"][0]["hour"][i]["humidity"],
                "wind_speed": data["forecast"]["forecastday"][0]["hour"][i]["wind_kph"],
                "rain_mm": data["forecast"]["forecastday"][0]["hour"][i]["precip_mm"]
            }
            for i in range (24)
        ]

        response = {"location": data["location"]["name"] + ", "
                                + data["location"]["region"]+ ", "
                                + data["location"]["country"],
                    "date": data["forecast"]["forecastday"][0]["date"],
                    "hourly_forecast": hourly_data}

        return jsonify(response)
    else:
        return jsonify({"error": "City not found or service unavailable."}), 404
```


```json
{
    "date": "2024-11-06",
    "hourly_forecast": [
        {
            "description": "Partly Cloudy ",
            "humidity": 88,
            "rain_mm": 0.0,
            "temperature": 10.4,
            "time": "2024-11-06 00:00",
            "wind_speed": 4.0
        },
        {
            "description": "Partly Cloudy ",
            "humidity": 88,
            "rain_mm": 0.0,
            "temperature": 10.2,
            "time": "2024-11-06 01:00",
            "wind_speed": 4.0
        },
        .
        .
        .,
        {
            "description": "Clear ",
            "humidity": 83,
            "rain_mm": 0.0,
            "temperature": 9.8,
            "time": "2024-11-06 23:00",
            "wind_speed": 5.4
        }
    ],
    "location": "Bologna, Emilia-Romagna, Italy"
}
```

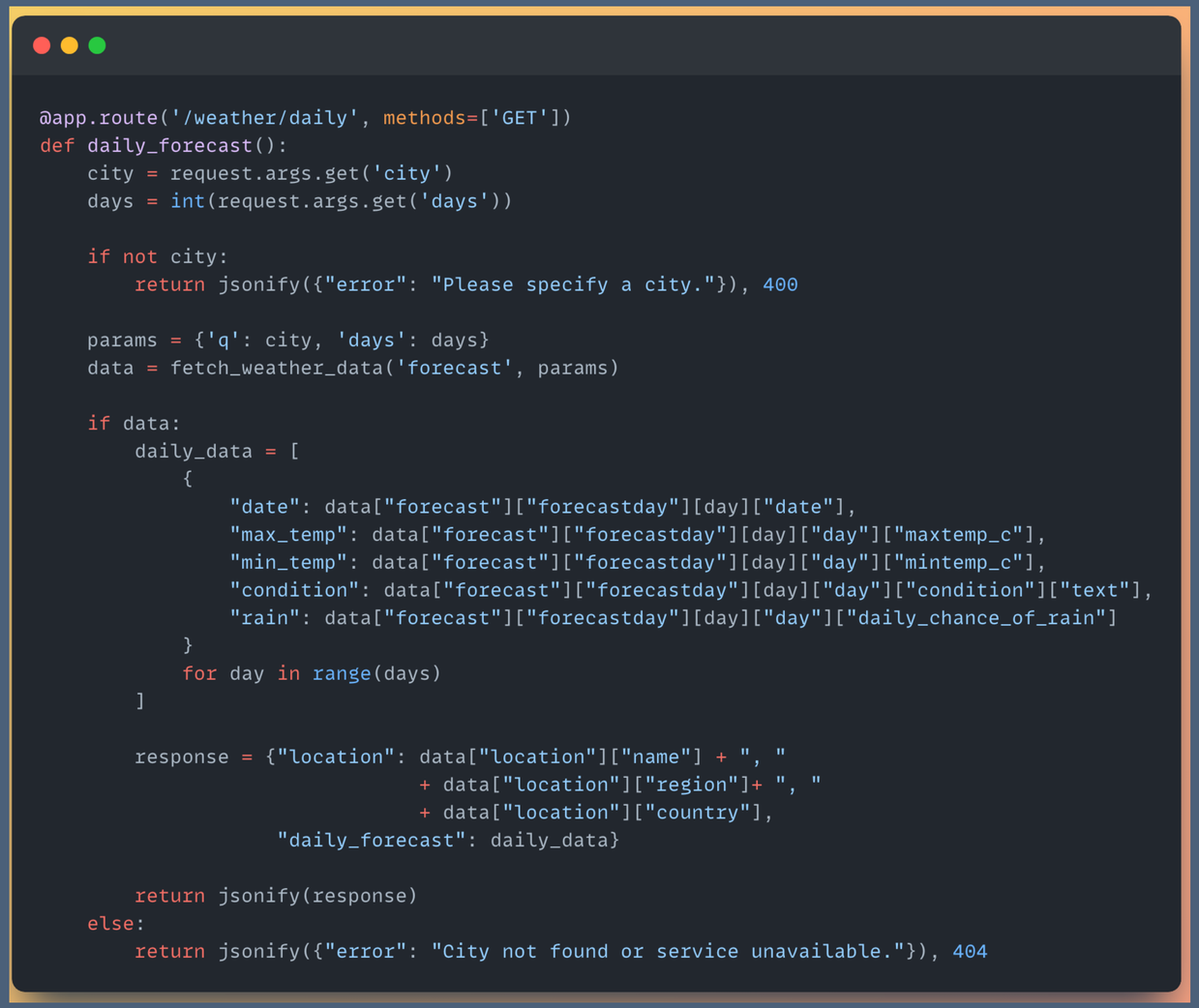

*The JSON screenshot is not really representative of the real data.*

*It has been cut for space purposes.*

# Web Service - Daily Weather

The **Daily Weather API** asks for a *city* and the number of *days* as query parameters and returns a **JSON Object** with the weather data of the city for the amount of days asked.

```
{
    "daily_forecast": [
        {
            "condition": "Overcast ",
            "date": "2024-11-06",
            "max_temp": 15.3,
            "min_temp": 9.1,
            "rain": 0
        },
        {
            "condition": "Sunny",
            "date": "2024-11-07",
            "max_temp": 14.9,
            "min_temp": 8.4,
            "rain": 0
        }
    ],
    "location": "Bologna, Emilia-Romagna, Italy"
}
```



```
@app.route('/weather/daily', methods=['GET'])
def daily_forecast():
    city = request.args.get('city')
    days = int(request.args.get('days'))

    if not city:
        return jsonify({"error": "Please specify a city."}), 400

    params = {'q': city, 'days': days}
    data = fetch_weather_data('forecast', params)

    if data:
        daily_data = [
            {
                "date": data["forecast"]["forecastday"][day]["date"],
                "max_temp": data["forecast"]["forecastday"][day]["day"]["maxtemp_c"],
                "min_temp": data["forecast"]["forecastday"][day]["day"]["mintemp_c"],
                "condition": data["forecast"]["forecastday"][day]["day"]["condition"]["text"],
                "rain": data["forecast"]["forecastday"][day]["day"]["daily_chance_of_rain"]
            }
            for day in range(days)
        ]

        response = {"location": data["location"]["name"] + ", "
                                + data["location"]["region"]+ ", "
                                + data["location"]["country"],
                    "daily_forecast": daily_data}

        return jsonify(response)
    else:
        return jsonify({"error": "City not found or service unavailable."}), 404
```

# Mashup vs Web Service

| | Weather Mashup | Weather Web Service |
|---|---|---|
| **Ease of use** | User friendly design.<br>Data is shown automatically. | Not usable in a visual way.<br>The service is easier to use<br>for programmers. |
| **Reusability of the code** | There is not really a way<br>to reuse the code. | High reusability.<br>It is designed specifically for<br>being used in different contexts. |
| **Control over shown data** | The developer decides what<br>to show to the end user. | The user can decide what<br>data to ask for.<br>There is more freedom. |
| **Architecture** | Usually Client sided. | Server sided. |

Thank you!