WeatherWise – Cloud Computing

Mattia **Piazzalunga**, Davide **Soldati**, Matteo **Severgnini** & Nicolò **Urbani**

Table of contents

Introduction

Introduction

WeatherWise is much more than just a weather application. It is an innovative platform designed to provide users with a comprehensive and engaging weather consultation experience. In a saturated market of weather applications, WeatherWise stands out for its intuitive and user-friendly interface, allowing users to quickly access the most relevant weather information. **WeatherWise is simplicity!**

One of the distinctive features of WeatherWise is its ability to provide not only precise and up-to-date weather forecasts thanks to data provided by certified sources but also advice and reviews on points of interest in a given location. This allows users to plan their activities based not only on weather conditions but also on attractions and activities available in the area. **WeatherWise is all in one app!**

But what truly makes WeatherWise unique is its approach to the accuracy of weather forecasts. The WeatherWise team understands that even the most reliable forecasts can prove to be wrong, and for this reason, they have introduced an innovative system that takes into account user feedback to calculate the likelihood of forecast accuracy. In this way, users not only receive weather information but also become active participants in the success of WeatherWise. **WeatherWise is by users, for users!**

Ultimately, WeatherWise offers users much more than just a simple weather forecast. It is a reliable companion for planning daily activities and discovering new experiences, all within one intuitive app. **WeatherWise is your weather app!**

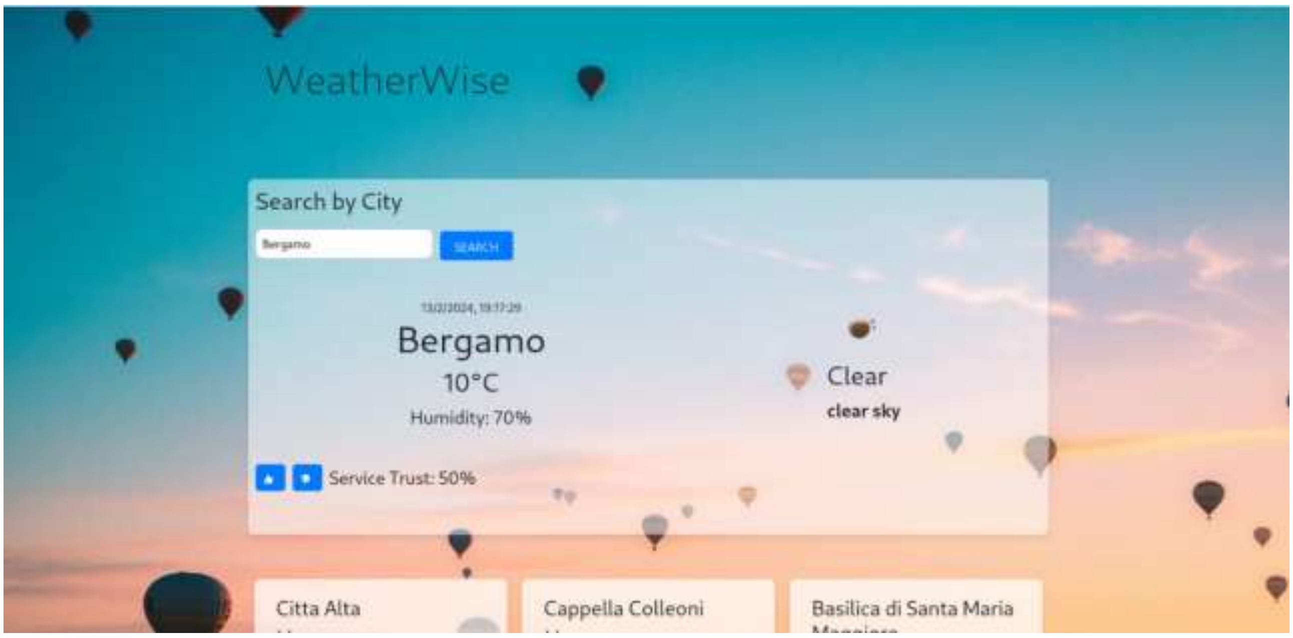
Functionality and implementation details

The **first release of WeatherWise** introduces some intuitive features for easy and user-**friendly weather consultation for a given city**, along with some interesting additional functionalities:

- Retrieval (GET) of **real-time weather for the city**, including temperature and humidity.
- Possibility to **provide feedback** (POST) for the current city weather forecast. The system allows associating a **probability of accuracy for the forecast at that location** (GET), based on feedback from the last 7 days.
- Possibility to **provide advice** (POST) on a "**point of interest**" to visit in the city, as well as getting an overview of all the places to visit in that location.
- The system allows controlled insertion of **cities** by **providing an updated list** of them (GET).

Some WeatherWise screenshots and explanation

The **front-end design** of Kubernetes is shown in the following figures. In a very "simple" screen, currently created ad-hoc for an initial demo, WeatherWise displays **weather information** for the city of interest, including day, time, temperature, humidity, and sky condition. Along with these forecasts, WeatherWise provides an **indicator of the reliability of the service/forecast in the city of interest**, allowing also to **provide real-time feedbacks** on the accuracy of the forecast.



Bergamo

13/2/2024, 19:17:29

Bergamo

10°C

Humidity: 70%

Clear

clear sky

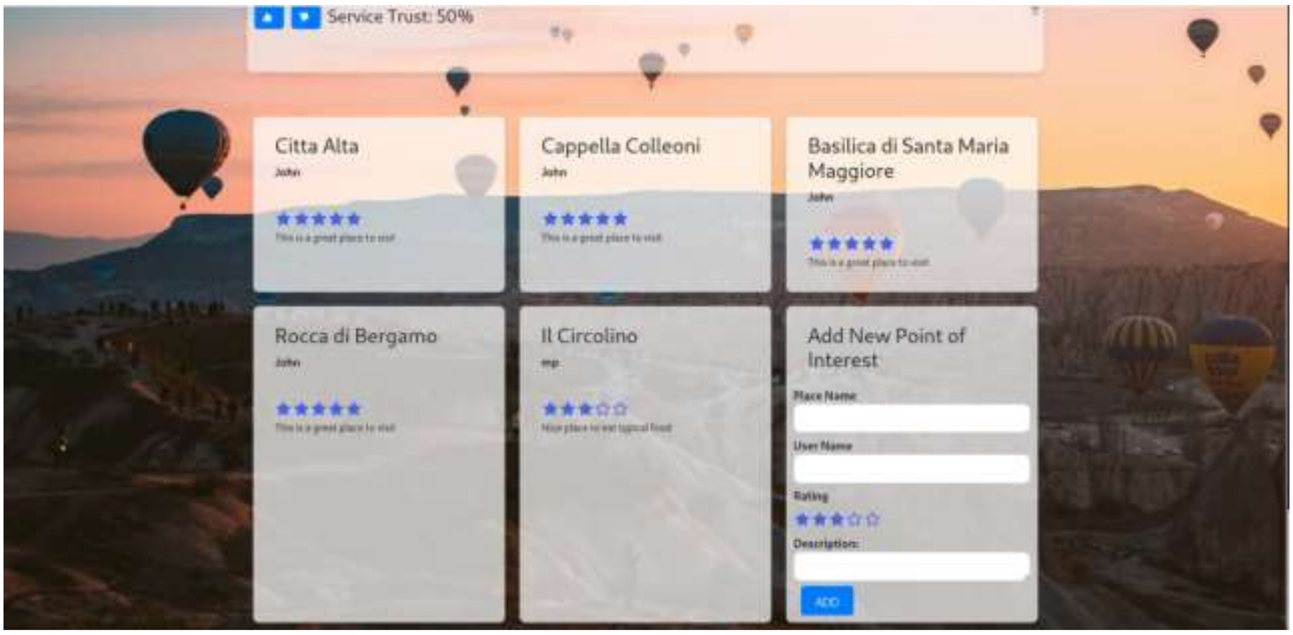
Service Trust: 50%

Citta Alta

Cappella Colleoni

Basilica di Santa Maria

Furthermore, WeatherWise positions itself as a **comprehensive application**, providing an **overview of points of interest** in the input city, ideally offering the ability **to organize an outing based on weather conditions.** Users indicate the points of interest.


Service Trust: 50%

Citta Alta
John
★★★★★
This is a great place to visit

Cappella Colleoni
John
★★★★★
This is a great place to visit

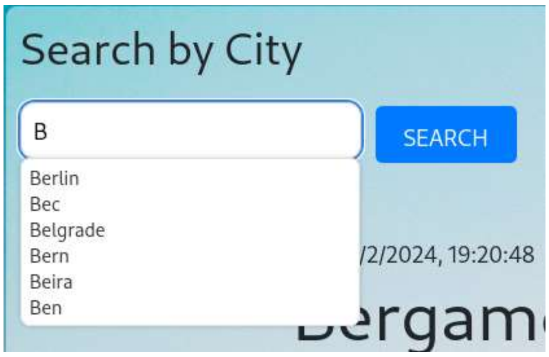
Basilica di Santa Maria Maggiore
John
★★★★★
This is a great place to visit

Rocca di Bergamo
John
★★★★★
This is a great place to visit

Il Circolino
mp
★★★☆☆
Nice place to eat typical food

Add New Point of Interest
Place Name
User Name
Rating
★★★☆☆
Description:

Finally, WeatherWise supports the user in **typing/selecting cities**.



B

Berlin
Bec
Belgrade
Bern
Beira
Ben

/2/2024, 19:20:48

Technologies, languages and tools of WeatherWise

Microservices

The idea, compared to monolithic applications, is to divide the application into a series of smaller and interconnected services, autonomous and independently distributed software components (**independent units of deploy**). These components are called **services.** A service typically implements a set of distinct features or functionalities and exposes APIs (the interaction with a service occurs through its API, which encapsulates its implementation details) for other microservices or a Web UI.

At runtime, **each instance of a microservice is associated with a container** or a cloud VM. The use of containers is one of the strongest predictors of success with microservices. The crucial aspect is that we can rethink the concept of an application as a collection **of loosely coupled** and **independently distributable** services.

In particular:

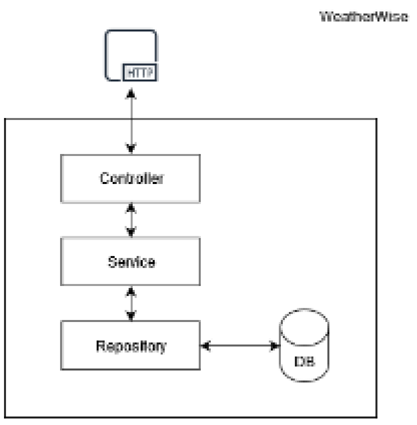
- **Microservices architecture addresses the problem of complexity**. The application is divided into manageable parts or services, each with a well-defined boundary. Individual services are much quicker to develop and much easier to understand and maintain.
- **It allows each service to be developed independently by a team focused on it**. Developers are free to choose any technology that makes sense, as long as they adhere to the "contract" with the APIs. Additionally, the fact that services are relatively "small" enables the possibility of rewriting them to use new technologies and frameworks.

The principles and practices of Software Engineering on which microservices architecture is based include:

- **Low coupling and high cohesion**.
  Microservices architecture promotes the concept of "loose coupling" between services. The principle of loose coupling is one of the fundamental pillars of a microservices-based architecture. It means that services should be designed to be independent of each other and not tightly dependent on internal details of other services.
- **Asynchronous is better than synchronous**.
- **Choreography is better than orchestration.**
- **Single Responsibility Principle.**
  Each module or class should have responsibility for a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

The use of microservices also **impacts** the **database-application relationship**: each service has its own database schema; although it goes against the idea of enterprise-level data and often involves data duplication, it **allows loose coupling.**

In short, the introduction of microservices-based development is a **true cultural shift**; but, how are the individual microservices of WeatherWise structured?



In detail:

- The **controller** receives HTTP requests from the outside;
- Business logic is executed in the **service** layer;
- To save information in the database, we use a **repository** layer.

Spring Cloud

Spring Cloud is a project within the **Spring ecosystem** that focuses on creating **reliable and robust microservices** applications. It is an **open-source framework** developed in the Java environment and is widely used in enterprise **Java** application programming. The use of Spring Cloud **simplifies the development of microservices**-based applications (which is why it was chosen for WeatherWise), providing a set of tools and libraries that address typical challenges of microservices architecture, such as configuration management, service discovery, and communication between components.

Programming Languages, Frameworks and API

Regardless of the microservices implementation, we can still see two fundamental parts that constitute our application: front-end, backend & databases.

The **front-end** founds its basis in the following languages and frameworks:

- **HTML** which is the standard markup language for documents view able by a web browser;
- **CSS** which is a style sheet language used to describe the presentation of a document based on a markup language such as HTML;
- **Javascript** for the dynamic and control part of the front-end of web pages.
- **JQuery** which is a library used to extend the functionality of Javascript and simplify its use;
- **Bootstrap** which is a collection of free tools for creating Web sites and applications;
- **React** which is a JavaScript library for building user interfaces, primarily developed by Facebook. It allows developers to create reusable UI components and efficiently manage the state of their applications. React utilizes a declarative approach to describe how the UI should look, making it easier to understand and maintain complex UIs.

- **Axios** which is another JavaScript library used for making HTTP requests from web browsers and Node.js. It provides an easy-to-use interface for performing asynchronous operations, such as fetching data from APIs.

The back-end founds its basis in the **following languages and frameworks**:

- **Spring** which is an Open Source web application framework based on Java. In particular, Spring Cloud and many of its associated libraries were used (many of which are described in other parts of this report).
- **Java** which is a high-level object-oriented programming language designed to have as few implementation dependencies as possible;
- **JPA** (Java Persistence API), is the Java specification that defines how to persist java objects. It is considered as a link between an object-oriented model and a relational database system;
- **Hibernate**, is the standard implementation of JPA, a framework that provides an object-relational mapping (ORM) service, that is, it manages the persistence of data on the database by representing and maintaining on relational DBMS a system of Java objects;
- **Spring Data JPA** is an abstraction that makes it easier to work with a JPA provider like Hibernate which is used by default. Specifically, Spring Data JPA provides a set of interfaces for easily creating data access repositories;
- **Javax Validation**, an annotation system for expressing entity validation criteria in Java;
- **Lombok**, an annotation-based Java library that offers various annotations aimed at replacing Java code that is well known for being boilerplate, repetitive, or tedious to write.

The **databases are** basis in the following languages and frameworks:

- **SQL**, a standard relational database language that allows for 360-degree management, creation, editing, and querying;
- **MySQL** is an open-source relational database management system (RDBMS) that is widely used for managing structured data. It is developed, distributed, and supported by Oracle Corporation.
- **PostgreSQL** is an open-source RDBMS known for its robustness, reliability, and feature richness. It is developed and maintained by a global community of developers;
- **FlyWay,** is an open-source database migration tool. It helps developers automate and manage the process of evolving a database schema over time.

Note*. In particular the microservices feedbacks and places each make use of a different RDBMS.*

Maven was used, instead, for Java software project management in terms of code compilation, deployment, documentation, and development team collaboration.

HTML
CSS
JS
B
React JS
Java
spring
PostgreSQL
Maven
SQL
HIBERNATE
Flyway
Project Lombok
MySQL
AXIOS

CRUD

CRUD is an acronym that refers to the **four main operations for manipulating entities in a relational database**:

- **Create**: creating tables, data, and relationships;
- **Read**: reading what was created;
- **Update**: modification, by updating, of what was created;
- **Delete**: deleting what was created.

These operations were implemented in our project specifically for **DB management.**

API and API Rest interface

APIs (Application Programming Interface) are a set of **definitions and protocols** by which application software is built and integrated. A **REST API**, on the other hand, is a **programming interface that uses HTTP to handle remote data**.

Due to the fact that almost all devices support HTTP, they can easily work with the REST interface without further implementation. The result is web services that are particularly valued for a high degree of platform **independence, scalability, performance, interoperability**, **and flexibility**. For the reasons just mentioned, we decided to use the **REST architecture to implement our platform**, also in anticipation of a possible, future expansion of functionality.

In particular:

- **REST** stands for Representational State Transfer and it's an architectural style for distributed systems. It's, therefore, an abstraction, a design pattern, a way of discussing architecture without worrying about its implementation.
- **HTTP,** acronym for "Hypertext Transfer Protocol," is a communication protocol used for data transmission over the Internet. It's one of the fundamental protocols that enables clients (like web browsers) to request web resources from servers and receive them reliably.

For this reason, we also designed, in this sense, **APIs that were built ad-Hoc** to differentiate HTTP calls.

Microservice 1 – Weather Consulting

Base path: /meteo

Path	Method	Description

/{cityName}	GET	Return the weather forecast for the city, also providing information about humidity and temperature.

Microservice 2 – Feebacks

Base path: /feedbacks

Path	Method	Description
/percentage/{cityName}	GET	Return a probability regarding the accuracy of the weather based on the feedback provided in the last 7 days
/	POST	Add feedback on the effectiveness of the city's weather forecast.

Microservice 3 – Places of interest

Base path: /places

Path	Method	Description
/cities/{cityName}	GET	Return all the points of interest in a given city.
/	POST	Add a point of interest for a city based on the city's name, the name of the location, the user's name recommending it, a rating, and a brief description.

Microservice 4 – City consultation

Base path: /cities

Path	Method	Description
/{prefixCityName}	GET	Return 5 cities that start with a given prefix.

A microservices pattern: API-Gateway

To access microservices, we **cannot directly call the ports associated with them**, as the microservices environment may have multiple instances of the same microservice, functioning on different ports. Therefore, relying on hard-coded port values is not feasible.

It becomes essential, therefore, to add a component at the beginning of our architectural landscape: the **API gateway**.

The **API Gateway pattern** is a common pattern in the realm of microservices, often used to **manage client requests to a distributed architecture of microservices**. In this pattern, an API Gateway serves as a **unified entry point for all client requests**. But what are the key points?

- **Request aggregation.**

The API Gateway can aggregate client requests and, if necessary, make multiple calls to different microservices to fulfill a single request. This reduces the number of network calls between clients and microservices and improves the overall system performance.

- **Request routing**.

  The API Gateway can route client requests to the appropriate microservices based on the information contained in the request. This allows hiding the complexity of microservice distribution from the client and simplifying the evolution of the underlying architecture.

- **Load balancing and failover**.

  The API Gateway can implement load balancing between microservice instances to evenly distribute traffic and improve system reliability. In case of a microservice failure, the API Gateway can also perform failover to alternative instances to ensure service continuity.

More information: https://microservices.io/patterns/apigateway.html

A microservices pattern: Service Registry

The "**Service Registry Pattern**" is an pattern widely used in the realm of microservices. This pattern is employed to enable distributed services to **dynamically discover** and communicate with each other in a scalable and highly distributed environment.

Let's delve into the details. A **service registry** is a central component that **keeps track of available services in the network**. Each service, upon starting up, registers itself with the Service Registry, providing its network information (e.g., IP address and port number) and metadata (e.g., service name, version, etc.).

Other services wishing to communicate with a registered service can **query the Service Registry** to obtain the contact information of the target service. This allows services to dynamically discover other services in the network **without relying on static configurations or fixed IP addresses**. Once the contact information of the target service is obtained from the Service Registry, **services can communicate directly with each other**. This may involve sending HTTP requests, making RPC (Remote Procedure Call) calls, exchanging data messages via a messaging bus, or other distributed communication methods.

More information: https://microservices.io/patterns/service-registry.html

A microservices pattern: Circuit breaker

The "**circuit breaker pattern**" is a pattern that aims to **improve the resilience and stability of the system.** It is applied to manage calls between services, especially when there are external dependencies such as third-party APIs, databases, or other microservices. The pattern is implemented through **three main states**:

- **Closed**.
  In this state, the circuit is "closed," and calls are forwarded normally to dependent services. The circuit monitors responses and keeps track of any errors or abnormal response times.
- **Open**.

If the number of errors or response times exceeds a certain predefined threshold, the circuit "opens." In this state, calls to dependent services are not forwarded directly but are instead handled differently, such as returning fallback responses or predefined errors without attempting to contact the external service.

- **Half-Open**.
  After a certain period of time or after a specific number of attempts, the circuit can transition to the "half-open" state. In this state, some test calls are sent to the dependent service to evaluate if it has come back online and is functioning correctly. If these test calls are successful, the circuit returns to the "closed" state and resumes forwarding calls normally. Otherwise, the circuit remains "open" to avoid further overloading the service.

To implement circuit breaker we use **Resilience4j** which is a lightweight, fault tolerance library.

More information: https://microservices.io/patterns/reliability/circuit-breaker.html

A microservices pattern: Config Server

The "**config server pattern**" is a pattern that allows for **efficient and scalable management of configuration for distributed applications**. In general, this pattern involves the use of a dedicated service called the "**config server**" that manages and provides configuration for different instances of microservices within the architecture.

Application configuration is therefore "**externalized**," meaning it is not directly embedded in the application code but is dynamically loaded from the config server either at startup or during application runtime. A key aspect is that the config server supports **dynamic configuration refresh**. This means that changes made to the configuration can be applied without needing to restart or reload the entire application.

More information: https://microservices.io/patterns/externalized-configuration.html

An external API: OpenWeather

OpenWeather is a **weather forecasting service** that provides real-time weather information and short- and long-term forecasts. It utilizes an extensive network of weather stations, satellites, and other sensors to gather meteorological data from around the world. These data are then processed using **advanced meteorological models** to generate accurate weather forecasts, including information such as temperature, humidity, wind speed, precipitation, and more. OpenWeather also **offers APIs**, used by us in WeatherWise, to allow developers to easily integrate weather forecasts into software projects.

More information: https://openweathermap.org/

An external API: Geonames

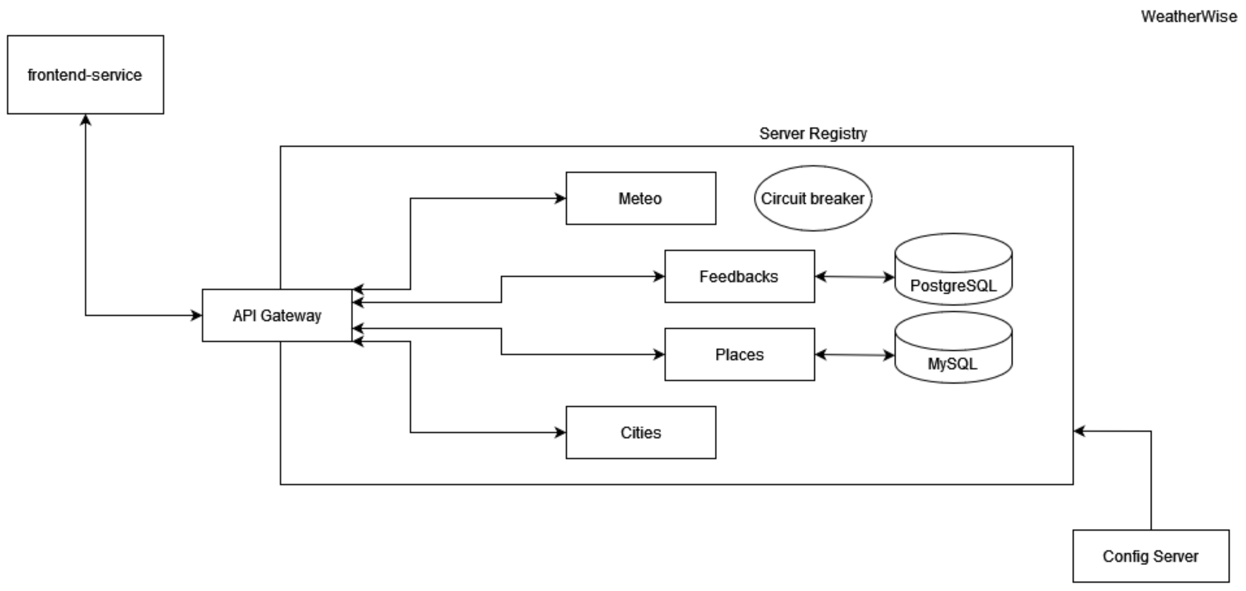
Geonames is a **geographic database** that provides information about **geographical features of the world**, such as city names, countries, rivers, mountains, lakes, and other geographic elements. Geonames also **provides some APIs** those allows developers to access the data and is used in WeatherWise for **autocompleting the city name** for which to make forecasts.

GeoNames

More information: https://www.geonames.org/

Organization of the application: the architecture of WeatherWise

"We showcase, at a high level, the **architecture of WeatherWise**.



The **config server** provides all configurations for WeatherWise microservices, including the **Server Registry**, which is the heart of WeatherWise, enabling scalability through automatic service discovery.

The application services available are **Meteo**, **Feedbacks**, **Places**, and **Cities**, each with multiple possible instances and accessible through the **API Gateway**, serving as the single access point to the **WeatherWise backend**.

The **frontend** is the only externally accessible part and communicates with the backend through the API Gateway.

All detailed information about the services is available in the specific sections.

A Cloud-Native Application

Introduction

WeatherWise is a **cloud-native application** designed to provide users with real-time, up-to-date, and accurate weather information. Being cloud-native means that the application has been developed using **architectures and technologies specific to cloud computing**, such as microservices, containerization, and container orchestration. This allows WeatherWise to be **highly scalable, resilient, and easily deployable**. But let's delve into the details of the cloud technologies used.

Container

Containers provide an additional layer of abstraction and automation of operating system-level virtualization on Linux. The goal is to provide a **consistent packaging and execution of software.**

Containers and virtual machines have similar advantages in resource isolation and allocation, but they operate differently because **containers virtualize the operating system rather than the hardware.** Containers are more portable and efficient.

Within a container, **all necessary executables**, binary code, libraries, and configuration files should be **present to run the application**. However, multiple containers can run on the same machine, sharing the operating system kernel with other containers, each of them running as isolated processes in user space.

Each microservice is associated with a container in WeatherWise. This allows:

- **Isolation**. Containers provide an isolated environment for running the microservice, ensuring that dependencies and necessary resources are contained within the container itself.
- **Portability**.
  Containers are portable and can be run on any operating system or environment that supports the container engine. This facilitates consistent deployment of the microservice across different environments, including development, testing, and production environments.
- **Scalability**.
  Containers allow for quickly scaling the number of instances of a microservice based on workload demands. Container orchestration, such as via Kubernetes, simplifies instance management and load distribution efficiently among them.
- **Resource Management**.
  Containers allow for clear specification of resources (CPU, memory, disk space, etc.) allocated for running the microservice. This enables more efficient resource utilization and better management of application performance.
- **Agility in Development and Deployment**.
  Using containers, you can speed up the development and deployment process of the microservice. Developers can create replicable development environments locally, and operations teams can easily deploy microservices into production without manually managing application dependencies.
- **Security**.
  Containers provide a certain level of isolation and protection for the microservice.
- **Version Management**.

Container images can be versioned and managed through version control systems like Git. This simplifies rollback to previous versions in case of issues and facilitates management of application dependencies over time.

Docker Hub

Docker Hub is a **container hosting service** that allows developers to distribute, manage, and collaborate on containerized applications. It provides both **public and private repositories** of container images that can be used as a foundation for creating and deploying containerized applications. It was used by the WeatherWise team to **remotely store built images of services** during the CI process.

Kubernates

Kubernetes (K8s) is an **open-source system**, managed by the Cloud Native Computing Foundation, for **automating the deployment, scaling, and management of containerized applications across multiple machines** (called nodes).

Kubernetes makes **global decisions about the cluster** and detects and responds to cluster and application events. The **control plane** constantly monitors the cluster's state and reconciles differences between the **current state and the desired state** in response to events. K8s performs a number of tasks automatically, such as starting or restarting containers (or pods).

Why do we use Kubernates?

- **Container Orchestration**.
  Kubernetes excels at managing containers, such as those created with Docker.
- **Scalability**.
  Kubernetes allows you to easily scale our applications up or down based on demand. We can scale horizontally by adding more containers to handle increased traffic, or vertically by allocating more resources to existing containers.
- **High Availability**.
  Kubernetes provides features for ensuring high availability of your applications. It can automatically restart containers that fail, replace containers that don't respond to health checks, and distribute traffic among healthy instances.
- **Resource Efficiency**.
  Kubernetes optimizes resource utilization by scheduling containers onto nodes in a cluster based on resource requirements and available capacity. It can pack multiple containers onto the same node, maximizing resource usage.
- **Extensibility**.
  Kubernetes has a rich ecosystem of plugins and extensions that extend its functionality. We can integrate Kubernetes with other tools and platforms for logging, monitoring, networking, storage, and more.

Minikube

Minikube is an **open-source platform** that allows running a **local Kubernetes cluster** on a **single machine**. It is useful for developers and operators who want to test Kubernetes applications in an isolated and controlled development environment, without the need for a complete cloud infrastructure or a remote Kubernetes cluster. We used Minikube to run our application.

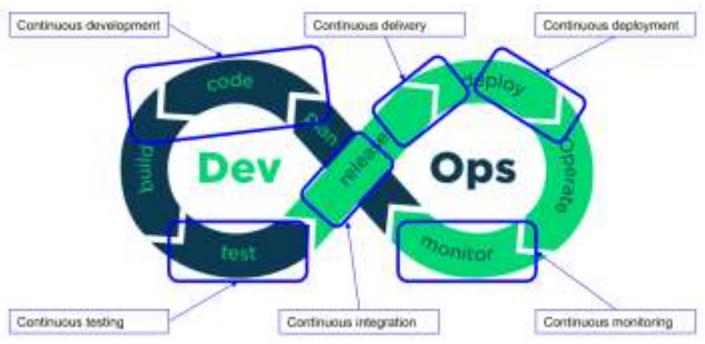
The DevOps Culture

The term "**DevOps**" doesn't have a truly concrete definition. It's a **philosophy**, a way of working, and means different things to different people. It's a software development method, a technology management approach that emphasizes communication, collaboration, integration, automation, and measurement of cooperation between software developers (dev) and operational staff (ops) in order to create and deliver software applications to their users.

Here are **six essential principles when adopting DevOps**:

- **Customer-centric action**.
  The DevOps team should take customer-centric action by consistently investing in products and services.
- **End-to-end responsibility**.
  The DevOps team should provide performance support until end-of-life, increasing accountability and the quality of engineered products.
- **Continuous improvement**.
  DevOps culture focuses on continuous improvement to minimize waste and continually accelerate product or service enhancement.
- **Automate everything**.
  Automation is a vital principle of the DevOps process, not only for software development but also for the entire infrastructure landscape.
- **Work as one team**.
  In DevOps culture, the roles of designer, developer, and tester are already defined. All they needed to do was work as one team with complete collaboration.
- **Monitor and test everything**.
  It's crucial for the DevOps team to have solid monitoring and testing procedures.

DevOps has its lifecycle:



- **Continuous Development**
  This is the phase of the DevOps lifecycle where software is continuously developed. Unlike the Waterfall model, the final software outcomes are broken down into multiple short development cycles, developed, and then delivered very quickly. Code can be written in any language but is managed using version control tools.
  Version control helps developers track and manage changes regarding code in a software project. Git is open-source distributed version control software, precisely the most widely used in the world, so much so that it is considered a standard (according to a Stack Overflow developer survey, more than 87 percent of developers use Git). It is critical for collaboration in project development: developers issue their work locally and then synchronize the repository copy with the copy in the server. Making it easier and more efficient for the team to use Git comes into play GitHub a hosting service for software projects, a true cloud-based implementation of Git.
- **Continuous Testing**
  This phase involves continuously testing the developed software to identify any bugs. In this phase, the use of Docker containers to quickly simulate the test environment is also a preferable choice. Once tested, the code is continuously integrated with the existing code.
- **Continuous Integration**
  This is the phase where code supporting new features is integrated with the existing code. Because software development is continuous, updated code must be continuously and seamlessly integrated with systems to reflect changes for end-users. The modified code should also ensure that there are no errors in the runtime environment, allowing us to test the changes. This phase is the cornerstone of the entire DevOps lifecycle.
- **Continuous Delivery**
  This is the phase where code changes are automatically prepared for release into production. A cornerstone of modern application development, continuous deployment expands on continuous integration by deploying all code changes to a test and/or production environment after the build phase. If implemented correctly, developers will always have a deployment-ready build artifact that has gone through a standardized testing process to ensure there are no errors in the runtime environment.
- **Continuous Deployment**

Continuous Deployment goes beyond continuous delivery. With this practice, every change that passes all stages of the production pipeline is released to all your customers. It is the phase where code is deployed into the production environment.

- **Continuous Monitoring**
  This is a crucial phase in the DevOps lifecycle aimed at improving software quality by monitoring its performance. This practice involves the operational team monitoring user activity to identify bugs or any improper system behavior. Any significant issues detected may be reported to the development team so they can be addressed in the continuous development phase.

At WeatherWise, **we have integrated the DevOps culture** starting from creating a CI/CD pipeline to automate all processes. GitHub Actions is the service we used in WeatherWise provided by GitHub. It allowed us to **automate workflows for our GitHub repositories**, from building to packaging.

GitHub Actions

Other information

README, License, Authors e .gitignore

To conclude with the analysis present in this report, we would like to highlight the task of **four files present in the project folder** and which have, despite often being underestimated, great importance: README, License, Authors, and .gitignore.

- **README** is a file that contains information about the files contained in an archive or directory and is commonly included in software packages;
- **License** is a file that contains the full text of the license chosen for the project without any modifications;
- **Authors** is a file that identifies who worked on a particular project which critical for copyright management;
- **.gitignore** is a file in the git system that contains a set of items to be ignored by the version control system.

Released version

In software development, **version** corresponds to a certain **state in the development of a software.** Conventions for numbering a software version normally involve a triplet of numbers in the form **X.Y.Z**, where X, Y, and Z:

- **X is the major version**: which should increase only as a result of radical changes in the product, such as those that make it in some way incompatible with its earlier versions;
- **Y is the minor version**: which increases with the introduction of small features to complement existing ones, but maintaining substantial compatibility;
- **Z is the patch version**: which augments usually only by correcting errors with the same functionality.

We release the **first version of WeatherWise** (v1.0.0).

LICENSE

WeatherWise is provided under **GPL-3.0 license.**

But what does it mean?

The GNU General Public License version 3.0 (**GPL-3.0**) is a software license developed by the **Free Software Foundation** (FSF). It is designed to ensure the **freedom of users to run, modify and share software.**

We **summarize the key features** of GPL licenses.

- **Each program with its own source**
  Should a developer wish to distribute the result of his labors under a GPL license, he is required to always publish the source code alongside the usual pre-compiled version.
- **Each program must be accompanied by a copy of the licensed**
  GPL requires that all works protected by it also include a copy of the full text from the license of use itself, in which the guaranteed freedoms and obligations to be respected are explained.
  To preserve maximum clarity over time, it is not permitted to replace the license with a simple link: the actual full text must be conveyed.
- **No constraints until you distribute**

Users are free to take any listing released under the GPL and modify it as they wish. As long as the work derived from it remains within a circumscribed scope, the user is not required to make the source code of the derived application available.
This right also extends to professional circles: it is therefore perfectly permissible to take, for example, management software released under the GPL, adapt it to the specific needs of one's own business, and then use the resulting version without being forced to publish the source code for it.

- **If you make it publicly available, you also need the source**
  The constraints of GPL become more stringent should one wish to make publicly available (and, therefore, also "sell") a program protected by that license or a derivative of it. In such a circumstance, the party is required to use the same GPL license, and is not allowed to impose more restrictive constraints.
  It does not matter whether the changes are substantial such as a profound rewriting of certain functions, as minimal as the variation of a few strings of text or, again, whether it is the same program "rebranded": what matters is that, should the decision be made to distribute it to the public, the source of any derived version must also be made available in the same manner as provided by GPL (**copyleft** concept).
- **As long as the source is there, it can also be sold**
  Nothing prohibits selling GPL-protected software: the important thing is that the presence of the source code is also guaranteed. It is clear, however, that the buyer can then choose to make that work available to third parties in a completely legal way.
  This is not a contradiction: by selling GPL-protected software, the merchant is not paid for the product itself, but rather for the service provided, i.e., the "making available" of the product.
- **No linking from closed applications**
  Another limitation that must be kept well in mind by those making derivative programs concerns the use of GPL-licensed libraries. So-called static linking (i.e., the copying, accomplished at compile time, of code contained in an external library within the main executable) is allowed only within programs that are later released under the same license.
  In other words, it is not possible to make a closed source program using GPL-protected libraries.

Thanks to: http://tinyurl.com/28f2bbzu.

Why did we choose it?

- **To preserve freedom**
- We want our software to remain open and free; GPL-3.0 is an excellent choice. It protects users' freedom and helps preserve the sharing and collaboration approach.
- **Open development community**
  GPL-3.0 is often associated with open source development projects and communities that promote collaboration. By using this license, you can take advantage of the support and contributions of a large community of developers.

Conclusion

Useful Links

We report **useful links** to WeatherWise here:

- Link a **GitHub** repository https://github.com/mattiapiazzalunga/WeatherWise;

The team

This report is maintained by the WeatherWise team, whose members are:

- Mattia **Piazzalunga** - 851931;
- Davide **Soldati** - 861178
- Matteo **Severgnini** - 851920;
- Niccolò **Urbani** - 856213.

We really care about the success of this project, so for any problems/understandings, we are available.