Ulm University | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Developing an API to Supply Third-party Applications with Environmental Data

Master's thesis at Ulm University

Submitted by:
Fabian Widmann
fabian.widmann@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

Supervisor:
Johannes Schobel

2018

Version: 30th January 2018

Satz: PDF-LaTeX 2_ε

Abstract

In healthcare, weather-sensitivity and the effect of environmental factors on various diseases were subject to extensive research in the last decades. Mostly without discovering statistically significant relationships between diseases and environmental parameters. This is often attributed to a lack of scale for existing studies.

Currently, there are no openly available solutions that can support surveys in this regard. Such solutions should be easy to integrate with an existing study platform. In turn, environmental data needs to be fetched for multiple users. This fact led to studies restricting participants in terms of their location or other factors. Consequently, this also meant, that the size of the studies was limited due to the placed constraints. Through the advance of technology, it is now possible to easily retrieve additional information from participants via their mobile smart devices which can be used to fetch various other types of data.

These circumstances led to the creation of an environmental data API described in this thesis. It provides functionality to retrieve environmental data from various data sources for a given tuple of latitude, longitude, and timestamp. The API facilitates adding new data sources by simply extending the provided examples. There are no restrictions in terms of spatial or temporal resolution or even source of the data. The resulting API fetches environmental data from multiple sources. It also facilitates obtaining data from other data sources and querying by researchers - including options to filter the data by various parameters. Finally, the API also supports converting between different units.

Contents

1 Introduction

Many patients hold the belief, that the weather bears an influence on the perceived symptoms of their disease. This, in turn, led to research on the influence of the environment on various diseases. Therefore, over the years, researchers performed a number of studies to examine those claims. Earlier on, researchers had to rely mostly on paper-based questionnaires filled out by the participants (cf. [1], [2], [3], [4], [5]). In addition, researchers were able to only obtain environmental data from few select areas. Consequently, data was obtained from mostly one weather source (e.g. around specific areas) during the study (cf. [6], [7], [8], [9], [10], [11], [12]).

Through advances in modern technology, smart mobile devices, such as smartphones or tablets, are more prevalent than ever before. These devices can potentially contain a multitude of sensors that may provide contextual information about the user such as acceleration, air pressure, gyroscope, magnetometer, location (via GPS), temperature and more. Accordingly, the collected data can be integrated in various surveys by retrieving environmental data for the specific point in time. This has already been done by some studies, which did no longer place strict limitations in terms of residence or locations on participants (cf. [13], [14]).

Combined with the increasing prevalence of publicly available data, location and time can be used to obtain a multitude of environmental parameters from various data sources. Accordingly, this results in new possibilities for general research on the influence of weather on diseases. This, in turn, leads to the goal of this thesis: providing an extensible API that allows researchers to easily integrate environmental data querying and retrieval into their existing study platforms.

Thus, to create a useful tool that can be adapted into various surveys, different data sets need to be evaluated and compared. Afterwards, fitting data sets should be chosen to serve as an example on how to integrate data sources into the API.

Consequently, the result of this thesis will be an API that can be integrated into already existing study platforms. It will provide a means to collect environmental data from various data sources based on a given location and timestamp. Depending on the data sources used, this also removes restrictions regarding the residence of participants. Additionally, it should be possible for the researchers to include and adapt data sources that fit their needs regarding the study goal. Furthermore, the collected data needs to facilitate being queried by researchers in various ways. This includes, for example, applying filters to queries (e.g. filter by participants, data source, parameters, etc.) In addition, support for converting between units or even shaping the output to the needs of the researchers' needs be added.

1.1 Structure

First, the thesis introduces related work in Chapter 2. This section describes the methodology and results of several conducted studies pertaining to the influence of environmental factors on diseases. Chapter 3 provides an analysis of possible scenarios and illustrates the design for the proposed API. Specific use-cases for the participants, researchers and technicians are introduced, followed by important principles this work adheres to. The following 4th chapter provides an overview of various data sources and their limitations. It then focusses on the *DWD hourly* and the *ECMWF Copernicus Atmospheric Monitoring Service* data sets. Each of the mentioned data sets is introduced, including limitations and access to the data. The chapter concludes with an outlook on additional data sets and challenges that can arise when combining data from different data sets or sources. Afterwards, the architecture of all created components is explained in Chapter 5. It further explores various aspects of the architecture, ranging from the design phase and used software architectural patterns to used tools, frameworks and the specific implementations that were done in the scope of this thesis. Implemented components include the environmental API itself, but also various utility projects that retrieve data from the specific services or help with unit conversions. Chapter 6 starts with a small section that explores the current API and data sets by proposing a scenario that, in turn, is further examined by looking at the numbers the API must be able to handle. Afterwards, it describes the current status and provides a look into the future. The chapter ends with a conclusion that reiterates important aspects of this work.

2 Related Work

As of now, a multitude of links between environmental parameters and various diseases have been examined by researchers. The examined diseases contain, among other things, the emotional and mental health, headaches and migraines, but also various rheumatic diseases. As time went on, the methods of the researchers advanced from traditional questionnaires to acquiring data from specific weather stations to even more customized retrieval and evaluation tools in recent studies.

Influence of Environmental Parameters on Diseases

Lots of surveys that explored links between various diseases and environmental parameters have been conducted throughout the years. While the influence of cold weather on the common cold is well-established, other diseases are said to be influenced by environmental parameters as well. Amongst others, this includes headaches, migraines and various rheumatic diseases. Many patients that suffer from those diseases often complain about being *weather-sensitive*[1]. This led to the development of a questionnaire that tried to assess weather-sensitivity. Earlier studies had a limited number of participants and shorter timespans that have been monitored. The effect of weather conditions on rheumatic diseases was examined by conducting a study in 1990 with n=62 (50 women, 12 men) patients that suffer from various rheumatic diseases over one month in Israel [6]. Patients were asked to complete daily questionnaires that rated joint pain and swelling and the activity level on a three-point scale. Atmospheric pressure, relative humidity, temperature and rain were recorded by the staff during the time of the

[1] Also known as meteoropathy: "a health condition or symptom caused by certain weather conditions" - `https://www.macmillandictionary.com/dictionary/british/meteoropathy`, accessed: 2018-01-08

study. This study resulted in the fact, that women were being more sensitive to weather than men (62% vs 37%) and that the effect on perceived pain differed on the specific rheumatic disease but influences were noted between barometric pressure, temperature and rain.

Only a few years later, in 1992, a questionnaire was developed that provides a weather-sensitivity index with a five-point scale [1]. It was used to evaluate the influence of weather on chronic pain patients that suffered from musculoskeletal disorders (including low back, neck and shoulder pain). Afterwards, a study was conducted with n=70 patients at a university clinic in the USA. This resulted in the finding that 75% of their patients had reported that temperature, humidity, precipitation and sudden weather changes influenced their pain to some degree. Additionally, only three percent of the patients reported no link between their pain and the weather. However, the patients were unable to link specific symptoms which are consistently influenced by the weather over time. The researchers, in turn, suggested that this effect of the weather on pain might be mediated by psychological factors or the patient's mood. Finally, they concluded, that this does not minimize the need to assess patient believes about weather and their pain but in fact, rather increases the need to further investigate this matter.

In 1994, a study [2] examined relationships between weather, disease severity and symptoms for patients suffering from fibromyalgia[2]. They assessed the participants' beliefs about the weather affecting symptoms and examined differences between individuals reporting low and high weather-sensitivity by conducting a study with n=84 participants. In turn, participants completed various questionnaires assessing pain, arthritis impact, tender points and weather-sensitivity. Weather data was obtained from the National Oceanic and Atmospheric Administration and was evaluated every 2 hours from 14:00 to 00:00 on the day of the assessment. While participants reported, that weather affected their musculoskeletal symptoms predominantly, the strongest relationship they have found was between weather beliefs and self-reported pain scores. Participants with high weather-sensitivity tended to have a more functional impairment. The only other positive

[2]"Fibromyalgia is a common and complex chronic pain disorder that causes widespread pain and tenderness to touch that may occur body wide or migrate over the body" - `https://www.fmcpaware.org/aboutfibromyalgia.html`, accessed 2017-12-18

relationship that has been found was the wind speed affecting the self-reported pain. A modest negative relationship with the tender point index was also discovered.

Afterwards, as the previous studies have not shown clear indications, another study [7] examined the reports of rheumatoid arthritis patients claiming that their pain was influenced by the weather in a larger study in 1999. As previous studies were rather small and short, their conducted study consists of n=75 participants (living in the USA) that recorded their daily pain severity for 75 consecutive days. Specific weather parameters, such as pressure, relative humidity and percentage of sunlight were obtained from a local weather station. This study resulted in weak evidence for an association between pain and weather. The Pain was most severe on cold days and on days with less sunlight, and especially for patients that reported higher overall levels of pain. The magnitude of the effects found, however, are not statistically significant for all participants.

Modern Research

Almost every group of researchers to date either guessed that the sample size was too low to be able to find statistically significant links or that psychological factors might be the cause for the patient's belief, that weather has an influence on their disease. In addition, most studies found links, but they were mostly not statistically significant. In comparison to the researchers' approach, another study tried to assess the prevalence of weather-sensitivity in Germany [5]. It provided a basis for further research by finding data about the prevalence of weather-sensitivity and its symptoms in 2002. This was examined by conducting a survey with n=1064 citizens age 16 or older by embedding a questionnaire in a representative multiple topic survey that was held in the form of house interviews. As such, the results are representative of the population in Germany. The study has shown, that 19.2% of the populations believe that weather influences their health to a "high degree" and 35.3% believe, that weather has "some influence on their health". In addition, the authors of the study also found regional differences in weather-sensitivity. Results also showed that Northern Germany had higher weather-sensitivity when compared to other regions. This might be due to a more unsettled weather in these regions compared to other parts of Germany. The most reported symptoms have been headaches and migraines, lethargy, sleep disturbances, fatigue, joint pain, irritation, depression, vertigo, concentration problems and scar pain. About one-third of the weather-sensitive participants were incapable of doing their regular work because of the mentioned symptoms at least once in the past year.

In addition to the prevalence in rheumatic diseases, asthma can also be influenced by air pollution. A study [15] examined the association between air pollution and admissions to children's hospitals in children under 15 years in Turkey. Data was obtained from a nearby meteorological station. The results showed, that n=2779 admissions occurred (14 children a day) with a significant association between admissions for asthma and respiratory outcomes for all fractions of particulate matter. The highest association noted was an 18% rise in asthma admissions correlated with a 10µg/m^3 increase in coarse *particulate matter* ($PM_{10\text{-}2.5}$[3]).

In 2011, an article came up about weather and migraine which raised the question, whether so many patients can be wrong about their beliefs regarding weather-sensitivity [16]. The author makes the point, that many patients report weather as a *trigger* for migraines and some even call them “human barometer”. In turn, he examined various studies regarding weather-sensitivity in patients and came up with potential reasons that might have an influence on the significance of the resulting data. For a migraine specifically, he listed the number of triggers that cause the migraines at about 6.7 on average per patient. This huge amount of possible causes of a migraine makes it difficult to pinpoint the specific trigger that caused a specific migraine instance. In addition, a specific migraine trigger may not precipitate an attack on each exposure. He also wagers, that the location of the study might influence the findings. This is done by citing a study reporting an increase in migraines in a hotter climate, while another one did not come to the same conclusion. This study was in Austria and was active during October to March, but the maximum temperature was 21.5℃, which might not have been hot enough to get the same results. He reasons, that another possible reason might be, that the mechanisms by which (environmental) trigger factors precipitate migraines are not well understood. It might be possible, that one factor is deemed to trigger a migraine

[3] "Particulate matter (PM), also known as particle pollution, is a complex mixture of extremely small particles and liquid droplets that get into the air. Once inhaled, these particles can affect the heart and lungs and cause serious health effects." - `https://www.epa.gov/pm-pollution`, accessed 2017-12-09

but the specific factor might have just influenced another one. The timing of weather changes is also another point that might have to be examined further, as they do not happen abruptly and may occur at different times in neighbouring locations. Finally, he also reasons, that migraine populations are not homogeneous, some triggers might only influence individuals but not others. This could even mean, that two individuals might be sensitive to opposite environmental factors. Which might lead to cancelling out the effect for the whole population. Due to the aforementioned reasons, the author proposes that it might be necessary to monitor single patients over longer timespans, instead of using larger groups. However, studies with even larger patient numbers and prolonged follow-ups might unravel possible relationships between environmental factors and migraine.

Recently Conducted Study

Summarizing, most studies either used questionnaires or required participants to be inside of a specific radius around either a chosen weather station or a zone around one hospital. A study from 2017, however, examined the relation between Ménière's Disease[4] and weather factors in the United Kingdom [14]. Participants (n=397) allowed researchers to map their GPS data to the closest available weather station. In turn, weather data has been collected from their nearest active station. This included parameters, like the air temperature, atmospheric pressure at the station level, atmospheric pressure at sea level, visibility and wind speed. The mapping was done by using the *Medical & Environmental Data Mash-up Infrastructure project (MEDMI)*[5]. This project allows users to link and analyse complex meteorological, environmental and epidemiological data by combining existing databases into a new framework. The study found strong evidence, that changes in atmospheric pressure and humidity can be associated with symptom exacerbation in Ménière's disease. Lower atmospheric pressure or high humidity were associated with higher odds of an attack and higher levels of vertigo, tinnitus and aural fullness.

[4]A disorder in the inner ear, characterized by vertigo, tinnitus and hearing loss - `https://www.nidcd.nih.gov/health/menieres-disease`, accessed: 2017-12-18

[5]`http://www.ecehh.org/research-projects/medmi/`, accessed: 2017-12-10

The findings and methodology of this study indicate that it is possible, that previously unrecognised links may be found with the help of now readily available technology. This includes high precision geolocation that can be logged by using common smart mobile devices. This can be achieved by providing potential participants of studies with apps that directly replace the paper-diary approach of earlier studies and are able to log the user's location during runtime. As a next step, the gathered location can then be used to query databases of various environmental data providers to retrieve a multitude of readily available parameters. MEDMI is one example for such a platform that can provide researchers with rich tools to obtain data, but due to its limitation to UK-based data, it is, unfortunately, no option for researchers outside the United Kingdom. The project's homepage has very good points regarding caveats that can arise when using data from its databases. The first point they made is, that the project's data sets should be seen as "hypothesis generating" with the need to explore discovered associations further. Additionally, the project specifies that data has to be linked appropriately in both space and time to make sure that the researcher knows what is measured. In addition, lag periods should be considered (i.e., the time between exposure and effect). They also list several interesting statistical issues that should be accounted for, such as seasonality, multiple drivers, data linkage, random noise and more [17].

3 Analysis

One of the big problems of finding relevant links between our environment and diseases is the ability to easily obtain and store data from various environmental data sources such as weather stations or satellites. Naturally, this leads to the idea of providing an environmental API that can be used in addition to already existing solutions. This chapter will first provide information about possible scenarios, followed by use-cases, the proposed design and finally a short section about collected data, data sources and possible privacy concerns.

3.1 Scenario

Currently, lots of experiments and studies in the healthcare field involve new technology to provide high-dimensional data to researchers. However, most of these studies choose their own back-end technology stack to collect and aggregate such data. Consequently, it will be important to allow future projects to either directly integrate the planned API into their back-end or use the environmental data API as a stand-alone service. An example for one of the aforementioned studies is *Track your Tinnitus*[1]. It consists of mobile applications for the participant's smartphone that provides the questionnaires and a back-end that stores the collected data.

Track your Tinnitus launched with the goal of providing more data on a previously detected link between emotional dynamics and psychopathology [18]. This has been achieved by allowing users to track their own tinnitus perception and emotional state on

[1] `https://www.trackyourtinnitus.org/`, accessed: 2017-11-28

a daily basis using their own smart mobile devices. To achieve this, the users can create a new entry in the mobile application by answering questionnaires about their current mood and tinnitus perception. In turn, these entries provide a personal tinnitus diary for the user to adapt their behaviour. In the future, it may also help their doctors to adjust the tinnitus treatment. Users do need to register but do not have to enter any personal data. In turn, the collected data is available to the participant and to the researchers at Tinnitus Research[2]. The collected data, however, does not contain any personal information and can be used for further research and publications.

Although *Track your Tinnitus* does not keep track of environmental factors, it might be worthwhile to also store the data to be able to analyse whether a link between various environmental factors and tinnitus perception might exist or not. As such, an optional service that tracks the current location in addition to the already collected data could be deployed to obtain various environmental parameters.

Another similar application is *Manage My Pain*[3]. The application allows participants to track their pain. All participants can fill out a daily survey to keep track of their day in terms of their perceived pain. The application allows the patients to find patterns and have a history that might help with their pain management.

In addition, various studies attempted to incorporate environmental data into their design, however, they were unable to find significant links between environmental factors and specific diseases. Zebenholzer et al. performed a diary based study on 238 patients around one specific meteorological station in Vienna. It evaluated the effect of 11 meteorological parameters on migraines and headaches [19]. While the data did show several trends, the conclusion of the authors was that 'the influence of weather factors on migraine and headache is small and questionable'. As a result, Becker wonders, whether a larger study might have shown statistical significance [16]. Furthermore, the author elaborates, that putting the focus on specific environmental parameters is difficult, due to the sheer number of possible parameters. In addition, the author also mentions, that timing might be an issue, as the lag time between a trigger and a migraine onset may be variable. As such, a link between environmental factors and diseases

[2] `http://www.tinnitusresearch.org/`, accessed: 2017-11-28
[3] `https://www.managinglife.com/`, accessed: 2017-10-11

can currently neither be confirmed nor denied scientifically. Surprisingly enough, many patients already believe that those factors play a role in the (perceived) severity of their diseases [16], and as such an additional ability to collect environmental data for every participant benefits both patients and researchers in the future.

Thus, an API that is able to provide a multitude of environmental data by just receiving the location and a date would be one of the first steps to allow researchers to conduct larger studies on the influence of weather on various diseases without limiting the area of the studies based on the data source.

3.2 Proposed API Design

Consequently, a design proposal can be derived from the description of the scenario. The user should not have to disclose personal data to this additional API. Participation should be handled as an *opt-in procedure* by asking for permission from the user. This permission should include sharing the timestamps and geolocation of those personal entries, but never personal information such as the name. This also requires a smart mobile device that can retrieve the geolocation by any means and an existing mobile application. In addition, the existing study platform will then forward the collected geolocation and the date of the journal entry to the environmental data API. The latter stores the data and will query the available data sources to retrieve environmental data for each entry. The whole procedure is depicted in Fig. 3.1.

In contrast, a researcher should be able to query data from this API for one or more users in a specific timeframe to retrieve environmental data. As for the timeline (cf. Fig. 3.2), the API will return all available environmental data per *geolocation* and *date* combination per-user. Comfort options, like filtering the data by participants, specific parameters or time frames, should also be available to filter the available data. Additionally, a system that converts from one to another unit (e.g., from $^{\circ}C$ to $^{\circ}K$) should be also offered. This will, in turn, allow researchers to directly use the data without having to convert them manually after querying the API.

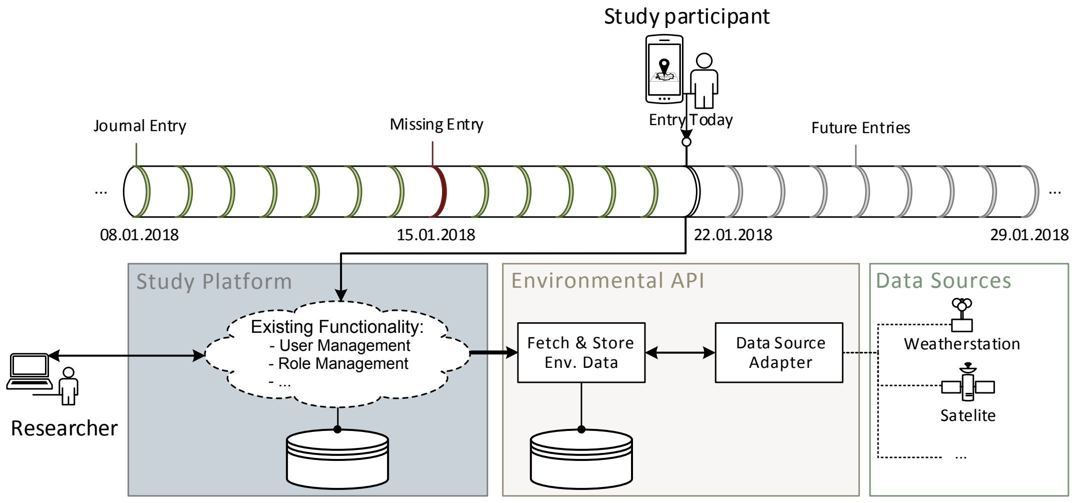


Figure 3.1: Proposed procedure for the participants of the study by visualizing the (daily) journal entries that consist of a geolocation and a date on a timeline. Those entries, in turn, allow the API to provide environmental data to the user.

Pre-Conditions for Existing Study Platforms

The API will not store complete user data due to privacy concerns and to avoid storing data redundantly. Thus, the study platform must provide its own user and role management module, as the API itself will only store the participant's id. This id can either be identical to the one used in the existing study platform or a hashed version of it. It can then be used to query the stored data on a per user basis.

Additionally, the existing study platform needs to provide additional routes that redirect calls to the API, in order to act as an intermediary service. This has the benefit of reducing the number of calls between the user and multiple services and can also be used to keep the environmental data API on the local network, instead of opening it to the public. The API should, in turn, prevent normal participants from calling the query routes and restrict their access to the appropriate routes.

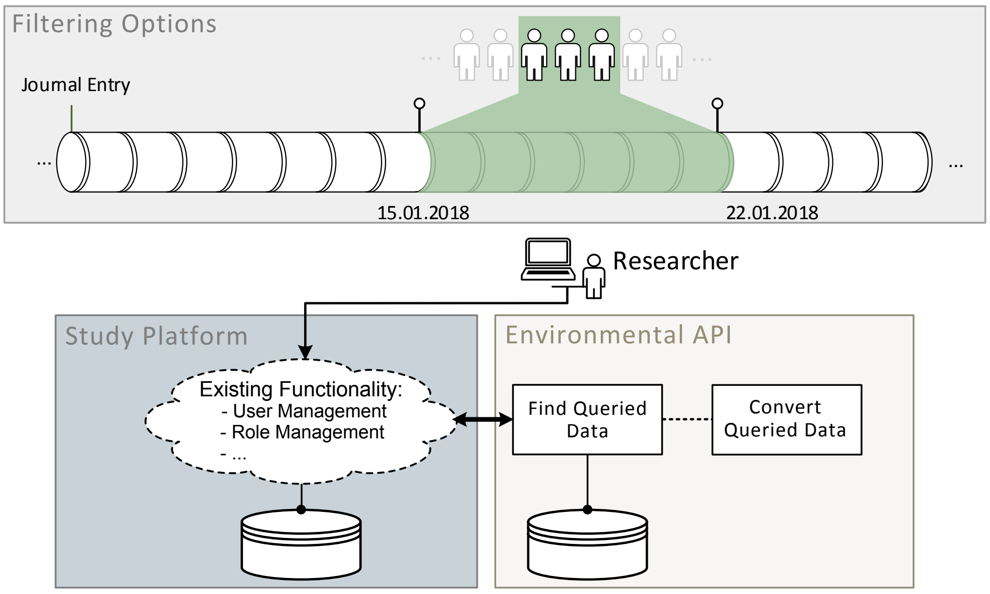


Figure 3.2: Proposed procedure for a researcher that includes querying the environmental database to retrieve data for one or more participants and optionally converting from one unit (e.g., °C) to another one (e.g., °K).

3.3 Use-Cases

This section will provide various use-cases that can be deducted from the described scenario above. Each use-case will pertain one specific actor (i.e., participants, researchers or administrators) and will consist of a description, preconditions, workflow and a result.

3.3.1 Participants

The API will need a way to identify participants in a study. Either by providing duplicate data that already exists or by storing an existing participant identifier that has been assigned in the original study platform. In turn, participants need to be able to store their geolocation which includes their current position (latitude and longitude) and a timestamp per entry. Such an entry can then be used to query environmental data.

Store Geolocation

Description: The participant of a survey needs to be able to store location data and a timestamp for each produced journal entry. This data, in turn, is used by the API to retrieve environmental data for this specific participant from all available data sources.

Preconditions: User data has to be available on the existing study platform, including a unique identifier for one user. In addition to that, the user needs to supply their geolocation to the API. The timestamp of the journal entry also has to be shared with the API.

Basic Flow: Create an entry in the API database to store the provided information.

3.3.2 Researchers

In comparison to the participants, the researchers will be able to only retrieve stored data from the API. This involves several options to pre-filter data and convert the queried data to other units. All filtering options that are specified in the following sections must work in combination with each other.

Query all Environmental Data

Description: The researcher needs to be able to retrieve all stored environmental data for all participants.

Preconditions: Queried environmental data exists.

Basic Flow: Researcher retrieves all existing environmental data for all participants.

Query Environmental Data for specific Participants

Description: The researcher can retrieve all stored environmental data for specific participants.

Preconditions: Queried environmental data exists. Additionally, participant ids are specified.

Basic Flow: Researcher retrieves all existing environmental data for the specified participants.

Exception Flow: When no participants with the given ids exist, this results in an empty response.

Query Specific Parameters in the Stored Environmental Data

Description: The researcher needs to be able to retrieve specific stored environmental data by specifying names of the required parameters.

Preconditions: Queried environmental data exists. Additionally, parameters that need to be filtered have been specified.

Basic Flow: Researcher retrieves the requested parameters from the existing environmental data in a universally known format.

Exception Flow: When no valid parameters are specified, this results in an empty response.

Query all Data for a Specific Time Frame

Description: The researcher needs to be able to specify two markers that symbolize a specific time frame to filter the stored environmental data.

Preconditions: Queried environmental data exists. Additionally, the user specifies a time frame by providing dates for **from** or **to**.

Basic Flow: Researcher retrieves the requested parameters from the existing environmental data in a universally known format, who happened to have their journal entry date between the specified *from* or *to* dates. When one of the dates is missing, it is supplemented with the lowest date or the current date, depending on which parameter was omitted.

Convert Existing Environmental Data from one Unit to Another Unit

Description: The researcher needs to be able to convert queried environmental data on the fly from one unit to another one.

Preconditions: Queried environmental data exists. Additionally, conversions have been supplied by the researcher.

Basic Flow: Researcher retrieves the requested parameters from the existing environmental data in a universally known format. All retrieved files have been converted from their unit to another one if the conversion is applicable.

Exception Flow: When no conversion is valid, the data is returned without converting it.

3.3.3 Administrator

In the context of this thesis, administrators are persons that can directly modify specific parts of their existing study platform and the proposed API. They should be supported in adding new environmental data sources to the API, change the way output is created and adjust various settings for the data retrieval process.

Extend the API to Support Other Data Sources

Description: An administrator should have a clear way of adding a new environmental data source to the API.

Preconditions: A new data source has been found. Additionally, the administrator is able to extend a module that acts as an adapter for the new data source to the API.

Basic Flow: Use the provided templates to adopt new data sources to the API.

Adapt the API Output Format

Description: An administrator should have a clear way of modifying the output of the API without having to know the code.

Preconditions: The administrator has a basic understanding of modifying files in the used language to adapt it to their needs.

Basic Flow: Modify the corresponding classes that shape the output to the desired format.

Adapt Settings of the API

Description: An administrator should have a way to adjust various API parameters, such as polling rate, URLs to internal service and other aspects.

Preconditions: The administrator can edit specific configuration files in the project.

Basic Flow: Change values in the settings file of the API to adjust the values.

3.4 Principles

First and foremost, the API needs to be flexible and extensible. It should be possible to adapt the output format to the need of users by providing a means to quickly and easily change the output, without having to change anything in the business logic. Additionally, it should be possible for other developers to quickly realize a small adapter between the original environmental data source and the API to store the data. This process involves two steps. The first one is retrieving the data, while the second one involves transforming from the source format into the expected format. Which, in turn, allows decoupling of both parts and allows asynchronous retrieval of the data.

In addition to the flexibility, privacy is another big concern. Instead of directly storing and managing the users, the API should facilitate existing study platforms from those projects in terms of user management and authorization. Those two topics are mostly custom to the study platform and adapting this API to the needs of various projects would mean more work instead of being ready to use out of the box. On the one hand, this approach reduces data duplication and the amount of HTTP calls the clients have to make. On the other hand, it also means changes to the existing study platform have to be made, by redirecting calls to the API routes.

Finally, when multiple users provide the same geolocation and almost the same time-frame, it would be possible, to retrieve the environmental data object once and assign it to the previously mentioned tuple (i.e., a combination of geolocation and timestamp). Due to different temporal resolutions that various data sources might offer, this approach might not work inside of the planned API. In turn, this might lead to a small amount of duplicate data but simplifies storage procedures. Depending on the needed data sources and their temporal resolutions, this might be one aspect that should be revisited.

4 Data Sources

The first step in providing various parameters of environmental data is to find suitable sources of environmental data. First and foremost, the sources should provide the data free of charge and available for everyone. This enables reuse and sharing of the application including the data access methods for a multitude of different application scenarios without adding constraints due to a difference in licensing models. Another important aspect is the way the data is collected, as this may have a huge influence on their availability and resolution. Which in turn leads to differences in the retrieval, transformation and storage of the weather data.

This chapter will provide insight on which data sources are available and what differentiates them from other available sources of environmental data. It will also include a detailed look at the selected sources including topics such as available parameters, restrictions and resolution of the data sets and a small outlook on available methods to retrieve the data. Followed by a short summary of what needs to be done to integrate the data into an application. In addition, other sources will be introduced briefly, which could be integrated in the future as well.

4.1 Deutscher Wetterdienst (DWD)

The German Weather Service (DWD) is responsible for a multitude of topics, such as providing meteorological services, safeguarding aviation and shipping and issuing official warnings about dangerous weather phenomena [20]. Additionally, the DWD has public weather data available as well on a publicly accessible server. As of July 2017 the

DWDG law [21][s. 4, par. 1] came into effect which commissioned the DWD to provide climate and weather data largely free of charge to the public. This led to more data being accessible to the public. As result, the data was placed under specific terms of use which can be found in the *GeoNutzV*-act [22][s. 3, par. 1,2]. The latter basically require that firstly the source must be included when using the data and secondly that modifications of the data also need to be marked with the origin of the data. In some cases, the source of the data may even require you to remove this reference in case of modification of the data.

4.1.1 Available Data Sets

DWD data can be accessed on its new Open-Data Server free of charge. This server, in turn, is split into two sections: *climate* and *weather*. The *climate* section is called Climate Data Center (CDC) and contains raw data in multiple resolutions and formats such as observed parameters from DWD weather stations, derived parameters at local stations and much more. In comparison, the weather section contains alerts, charts, forecasts, radar data and reports. According to the DWD data set introduction [23], the observed parameters at the DWD stations are grouped into eight categories. Each of those categories may contain one or more available parameters. Data is available in multiple temporal resolutions, ranging from multi-annual values, monthly, daily up to an hourly resolution. Currently, approximately 400 climate stations are active and provide environmental data across Germany. Table 4.1 lists all available hourly categories and provides a summary of contained parameters. Information about each category was extracted from the included data descriptions (for example the description of the air temperature data [24]) and an extensive list is provided in section B.1

List of DWD hourly Parameters In addition, each category also contains information about the quality of the measured data at the time for each data point. The `QN`-parameter defines the type of quality measurement e.g. `QN8`. One example for this is `QN_8` for the hourly cloudiness. This quality level, in turn, has a specific numeric value that encodes

Category	**Content**
Air Temperature	Contains two measured values: 2m air temperature and 2m relative humidity.
Cloudiness	Contains two measured values: index indicating whether the measurement was done by a human or instrument and total cloud cover in one eights.
Precipitation	Contains three measured values: hourly precipitation, an index to indicate whether there was precipitation and which form of precipitation.
Pressure	Contains two measured values: atmospheric pressure at sea and station level.
Soil temperature	Contains six measured values: soil temperature at 2cm, 5cm, 10cm, 20cm, 50cm, 100cm.
Solar	Contains four values but data is about one month old at the time of writing this thesis. Available data includes hourly sums of long-wave downward radiation, diffuse solar radiation, incoming solar radiation and sunshine duration per hour.
Sun	Contains the duration of sunshine per hour.
Wind	Contains two measured values: mean wind velocity in metres per second and wind direction given in degrees.

Table 4.1: An overview of the available categories and parameters.

meaning. To stay with this example, Table 4.2 displays which information can be deduced from the `QN_8`-value of a specific line.

QN8 Code	**Description**
1	Formal examination.
2	Examined following specific criteria.
3	Old automatic examination and rectification.
5	Historic and subjective procedure.
7	Second examination done, pre-rectification.
8	Quality assurance outside of the routine.
9	Not all parameters have been rectified.
10	Quality assurance and rectification finished.

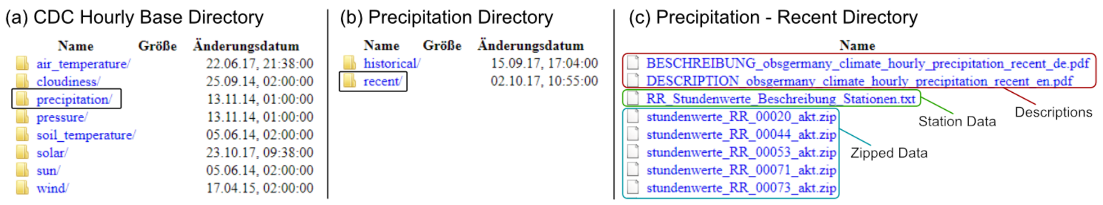
Table 4.2: `QN8` quality index explained in the data set description [25]

The specific information can also be obtained in the data set description. For this example, it can be found in the description pertaining precipitation [25]. As explained before, each parameter may use different quality measurement methods - where precipitation uses

QN_8, the air temperature specifies the quality in QN_9 - which might lead to differences in the interpretation of the data.

4.1.2 Accessing Hourly Data

CDC data is available on an open File Transfer Protocol (FTP) server provided by the DWD and can be used with most modern browsers without using specific software. Observed data can be retrieved in various time resolutions, which are stored in different sub-folders with varying amounts of available parameters. The hourly directory contains the previously mentioned eight parameter groups as single directories. This can be seen on in Fig. 4.1 a. Each of those directories, in turn, is split into two subdirectories - historical and recent data (Fig. 4.1 b). Those folders contain the environmental data, a list of stations that produced the data and a description of the possible parameters and other more specific details about the included data (Fig. 4.1 c). Finally, zip files can be found that contain the measurements done by a specific station. One file, in turn, contains various metadata in HTML or text format and one file that contains the actual data which can be found in Fig. 4.2. This example also shows that the unpacked data for this specific parameter total to about 630KB of data. Depending on the number of active stations that are required this can lead to a large amount of data that needs to be accessed daily.



(a) CDC Hourly Base Directory

Name	Größe	Änderungsdatum
air_temperature/		22.06.17, 21:38:00
cloudiness/		25.09.14, 02:00:00
precipitation/		13.11.14, 01:00:00
pressure/		13.11.14, 01:00:00
soil_temperature/		05.06.14, 02:00:00
solar/		23.10.17, 09:38:00
sun/		05.06.14, 02:00:00
wind/		17.04.15, 02:00:00

(b) Precipitation Directory

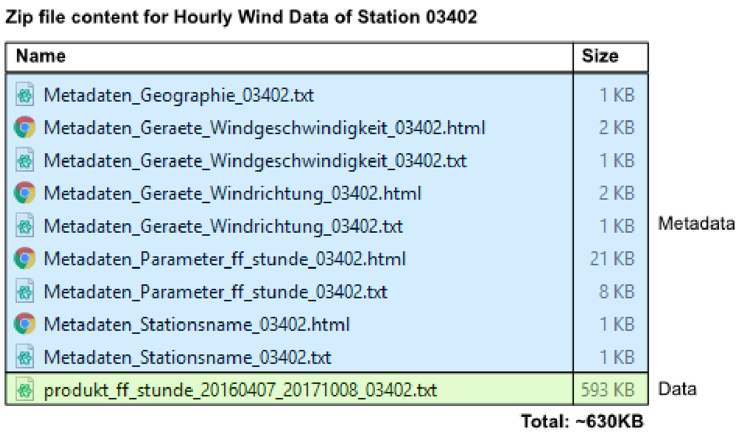
Name	Größe	Änderungsdatum
historical/		15.09.17, 17:04:00
recent/		02.10.17, 10:55:00

(c) Precipitation - Recent Directory

Name
BESCHREIBUNG_obsgermany_climate_hourly_precipitation_recent_de.pdf
DESCRIPTION_obsgermany_climate_hourly_precipitation_recent_en.pdf
RR_Stundenwerte_Beschreibung_Stationen.txt
stundenwerte_RR_00020_akt.zip
stundenwerte_RR_00044_akt.zip
stundenwerte_RR_00053_akt.zip
stundenwerte_RR_00071_akt.zip
stundenwerte_RR_00073_akt.zip

Figure 4.1: Navigating the CDC public FTP Server to find hourly precipitation data.

The typical workflow to obtain data for a specific time and location can be split into several single actions. First, obtain all stations that have been active in the respective time-frame. Secondly, filter all stations by the distance to the given latitude and longitude and find the nearest station. Third, download the zip archive, read the document that contains the values and filter the data by retrieval date. Finally, transform the retrieved data into the expected format. This simplified flow is also depicted in Fig. 4.3.



Zip file content for Hourly Wind Data of Station 03402

Name	Size	
Metadaten_Geographie_03402.txt	1 KB	
Metadaten_Geraete_Windgeschwindigkeit_03402.html	2 KB	
Metadaten_Geraete_Windgeschwindigkeit_03402.txt	1 KB	
Metadaten_Geraete_Windrichtung_03402.html	2 KB	
Metadaten_Geraete_Windrichtung_03402.txt	1 KB	Metadata
Metadaten_Parameter_ff_stunde_03402.html	21 KB	
Metadaten_Parameter_ff_stunde_03402.txt	8 KB	
Metadaten_Stationsname_03402.html	1 KB	
Metadaten_Stationsname_03402.txt	1 KB	
produkt_ff_stunde_20160407_20171008_03402.txt	593 KB	Data

Total: ~630KB

Figure 4.2: Content of the zip file that contains wind data for the station with the index 03402. Where the blue coloured part provides metadata and the green file contains the measured environment data.

Additionally, there are already several unofficial libraries to handle this process for different languages. With the caveat that none of them are official and can be outdated as soon as the location of the file changes even a bit.

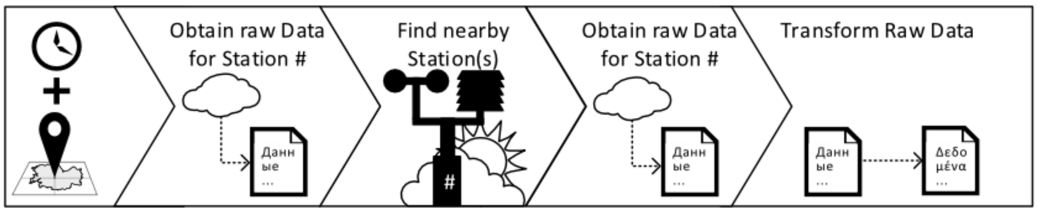


Figure 4.3: Simplified workflow to access the DWD Data. For a specific date and time, load the list of all stations. Then filter the list to get the nearest active station(s). Then download the data for this station and finally transform it into the target data model.

4.2 European Centre for Medium-Range Weather Forecasts (ECMWF)

ECMWF [26] is an independent intergovernmental organisation which is supported by most states in the European Union that provides a multitude of different data sets which are available to users under Regulation (EU) No 1159/2013 [27][p. 1-2]. As a result, access to the data is available after free registration at the ECMWF. Logged in users have access to all public datasets in two ways: access via the web interface or programmatically, which includes using a specific library, provided by the ECMWF.

Earlier this year another satellite for the Copernicus project was launched into space. Its task is to observe the earth and provide additional data about our environment in several data sets. Provided data is clustered into various service-groups by the ECMWF. One of those being the Copernicus Atmospheric Monitoring Service (CAMS) [28], which has been set up to supply everyone with various atmospheric environmental data.The collected data can, in turn, be used to determine the quality of air, formation of clouds, rainfall and various other parameters that might influence life on earth. Due to the scope of this application, the focus in this chapter lies on the obtainable data from CAMS, but other available options will also be introduced in the following subsections.

4.2.1 Available Data Sets

In this section, a subset of three possible data sets will be introduced in detail including limitations and resolutions. This also includes reasoning which one had the best fit for the scope of this thesis. Additional regional data sets are available as well, but they might provide fewer parameters than the CAMS near-real-time service and one other source of local German weather data were chosen already.

CAMS Near-Realtime

This data set contains daily near-real-time analyses and forecasts of global atmospheric composition [28]. It provides daily information on the global atmospheric composition by monitoring and forecasting various parameters [29]. Data is available from 2012-07-05 and is extended forward to real-time. Data is available in a 40km spatial (depiction of spatial resolution in Fig. 4.4) resolution which, at the time of the project, was the finest available resolution of the data sets available with a small delay of only five days. More information can be found on the appropriate website [30].

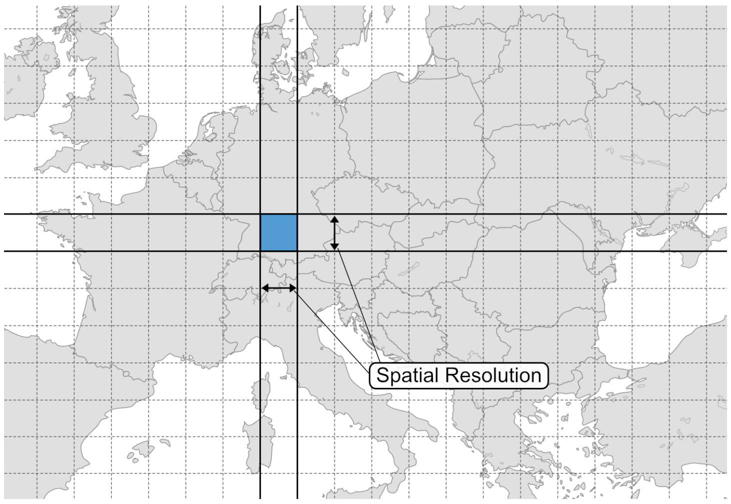


Figure 4.4: Earth's surface is divided into grids with variable cell counts that are determined by the spatial resolution of a data set. This, in turn, either enlarges or shrinks the given cells in a grid. Based on *Blank map of Europe*[1].

Additionally, data can be queried either as *analysis* or *forecast*. Analysis data is available at four points during each day: 00:00, 06:00, 12:00 and 18:00 respectively, whereas the forecast is using two base times to query data from either 00:00 or 12:00. These base times can, in turn, be modified by specifying steps. Those steps can be seen as a modifier for the base time. When choosing 00:00 as base time and three as a single step, the queried data will contain measurements at the times 00:00 and 03:00 respectively. An illustration of the difference between both analysis and forecast can be found in Fig. 4.5, where the forecast is selected with four steps 3, 6, 9, 12 to retrieve data from eight points during the day. Steps can reach up to 120h into the future in steps of 3h which can also be seen in Fig. 4.6 under the *Select Step* category.

[1]`File:Blank map of Europe (with disputed regions).svg` by *maix* Available: `https://commons.wikimedia.org/wiki/File:Blank_map_of_Europe_cropped.svg`, accessed: 2017-11-01

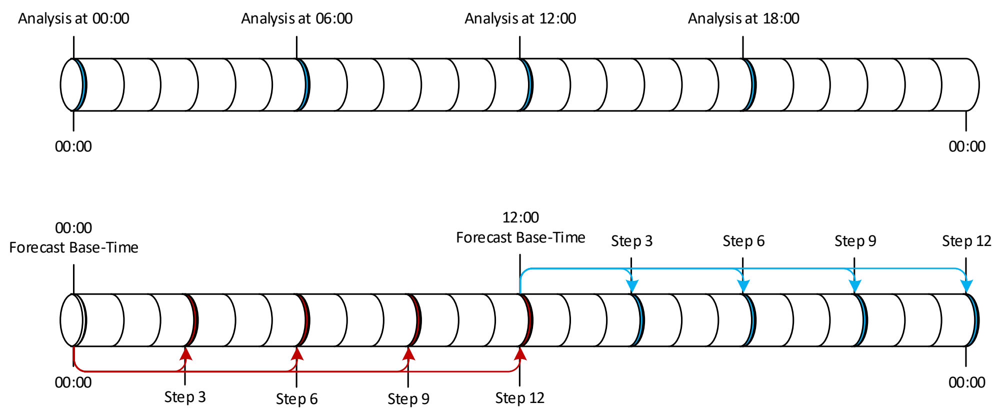


Figure 4.5: Available data types with the analysis being on top and the forecast below. Analysis is footnote 4 points during the day, whereas forecasts can be obtained in intervals of three hours.

European Reanalysis (ERA) Interim

This data set provides an atmospheric reanalysis. A reanalysis can span a long-time period of multiple decades or more and often time provides huge data sets [31]. One of the side effects of this type of data set is the low update rate compared to other data sets. ERA-Interim is updated once every month and has a delay of two months to allow for quality assurance. The spatial resolution of this data set is approximately 80km [32]. Temporal resolution is equal to the CAMS Near-real-time data set including the possibility to query both, analyses and forecasts. Additionally, the licence of this data set is restrictive in terms of forwarding the results of the analysis which also might prove a problem in the long run and may need a special permit from the ECMWF [33][s. 2]. In comparison to the CAMS Near-real-time data set, ERA-Interim provides more environmental parameters at the cost of availability.

ERA5

This data set is currently under construction and will cover the period from the 1950s to the present. As of writing, the most recent data available is from December 2016, which will be extended to be near-real-time as well. Production of this data set started in 2016 and it contains hourly analyses and forecasts with a spatial resolution of 31km. Access to the set was opened recently in mid-2017 [34]. Compared to the CAMS Near-real-time data set, it also contains more parameters and might be a suitable replacement as soon as it hits the near-real-time status due to the higher spatial and temporal resolution. In future, it might be necessary to re-evaluate the given terms of service to check if the data set allows usage as intended by this thesis.

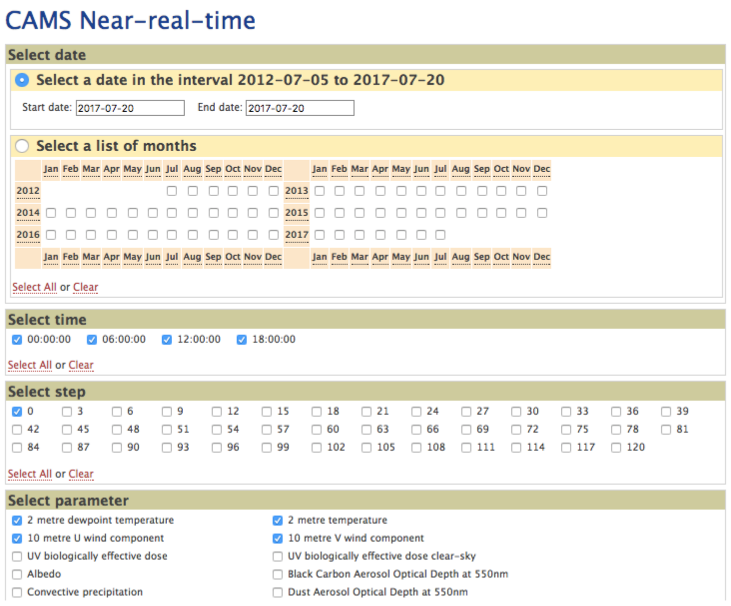
Conclusion

At the time of working on the thesis, the CAMS near-real-time data set seemed to be the most suitable for the given premise. The five-day delay is bearable for this use-case, and the resolution was the finest available with up to date data. Additionally, the licence of the data set does not restrict reuse and modification of the data. This allows others to use the application without having to worry about licensing by just registering at the ECMWF to obtain access to the data set.

4.2.2 Accessing the CAMS near-real-time Data

Retrieving data from CAMS near-real-time service is possible in two ways. A user can either get the data via the web interface (which is depicted in Fig. 4.6) or by using the provided libraries to automate the retrieval data from the appropriate ECMWF servers. As of the time of writing, only the python library (called `ecmwfapi`) is actively supported, while the other options for different languages are marked as discontinued on the support website [35]. The python library itself offers a simple way to retrieve weather data in a special format called *Gridded Binary* or *General Regularly-distributed Information in Binary form* (GRIB) designed by the World Meteorological Organization (WMO) [36]. In addition, the ECMWF also released a library called `ECCodes` for Unix platforms and three different programming languages: C, Fortran 90 and Python [37]. Which will provide a means to access and manipulate the downloaded data files.

Access to both libraries leads to a workflow to obtain specific values for a given time and latitude-longitude tuple. First, obtain the file containing the data either by using the web API or via the `ecmwfapi` library. Second, parse the retrieved file by using the `ECCodes` or other programs and find the data point whose location is closest to the



CAMS Near-real-time

Select date

(●) Select a date in the interval 2012-07-05 to 2017-07-20

Start date: 2017-07-20 End date: 2017-07-20

( ) Select a list of months

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2012							☐	☐	☐	☐	☐	☐	2013	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
2014	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2015	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
2016	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2017	☐	☐	☐	☐	☐	☐	☐					
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec

Select All or Clear

Select time

✓ 00:00:00 ✓ 06:00:00 ✓ 12:00:00 ✓ 18:00:00

Select All or Clear

Select step

✓ 0	☐ 3	☐ 6	☐ 9	☐ 12	☐ 15	☐ 18	☐ 21	☐ 24	☐ 27	☐ 30	☐ 33	☐ 36	☐ 39
☐ 42	☐ 45	☐ 48	☐ 51	☐ 54	☐ 57	☐ 60	☐ 63	☐ 66	☐ 69	☐ 72	☐ 75	☐ 78	☐ 81
☐ 84	☐ 87	☐ 90	☐ 93	☐ 96	☐ 99	☐ 102	☐ 105	☐ 108	☐ 111	☐ 114	☐ 117	☐ 120	

Select All or Clear

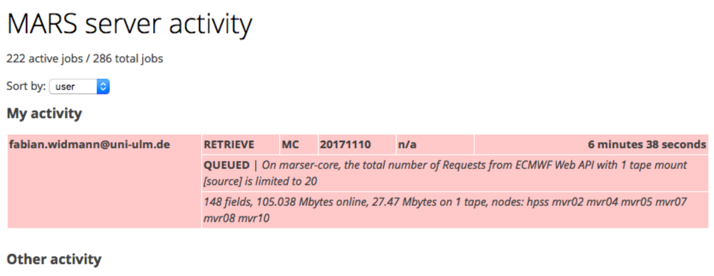
Select parameter

✓ 2 metre dewpoint temperature	✓ 2 metre temperature
✓ 10 metre U wind component	✓ 10 metre V wind component
☐ UV biologically effective dose	☐ UV biologically effective dose clear-sky
☐ Albedo	☐ Black Carbon Aerosol Optical Depth at 550nm
☐ Convective precipitation	☐ Dust Aerosol Optical Depth at 550nm

Figure 4.6: Catalogue of the CAMS near-real-time dataset, displaying the latest currently retrievable date, the times for the specific subset and available parameters.

queried position (and time). When querying the data set to only obtain data inside of Europe and analysis data only the typical GRIB file is about 135MB per day. One week of data would, in turn, sum up to about 1GB. In turn, the forecast data would likely result in even bigger space requirements, due to both the additional available parameters and the finer temporal resolution of 3h instead of 6h.

After making a request to the ECMWF servers, it is possible to track the status of the request, as seen in Fig. 4.7. This might be helpful if a request takes longer than anticipated due to high load.



MARS server activity

222 active jobs / 286 total jobs

Sort by: user

My activity

fabian.widmann@uni-ulm.de	**RETRIEVE**	**MC**	**20171110**	**n/a**	**6 minutes 38 seconds**
	QUEUED | *On marser-core, the total number of Requests from ECMWF Web API with 1 tape mount [source] is limited to 20*				
	148 fields, 105.038 Mbytes online, 27.47 Mbytes on 1 tape, nodes: hpss mvr02 mvr04 mvr05 mvr07 mvr08 mvr10				

Other activity

Figure 4.7: The ECMWF offers tracking for open requests on a separate website[2]

4.3 Additional Data Sources

In addition to the sources mentioned before several other options exist that provide environmental data for end-users. One of those being the Yahoo Weather API [38], which provides data free of charge for use by individuals or non-profit organizations or personal, non-commercial uses. There is no specific rate limit but an example of up to 2000 signed calls per day was given to retrieve data. When using data from the Yahoo Weather Service an attribution is expected to fulfil the terms of service. Data can be retrieved via their provided RESTful-API.

Another option would be the service provided by OpenWeatherMap [39], which provides data under the Open Data Commons Open Database Licence (ODbL) [40] that allows sharing, adapting and producing works from the database as long as the original is attributed and the product is shared under the same licence. Several account types are available, where the free membership has access to the current weather API as well as several other services. A free account may only call the API 60 times per minute. When this limit is reached, the user needs to go with one of the paid account types, which provide more benefits but require monthly payments.

4.4 Challenges

When combining different data sources, some problems may occur. First of all, the spatial and temporal resolution may vary greatly. One data source might be available hourly, others may only offer one set of data every few days. When combining the data for researcher those differences must be made visible by providing additional information about the retrieval date and distance between the queried point and the point of measurement.

In addition to that, different units might be problematic as well. When one of the weather sources provides all temperatures in degrees Celsius and another one uses degrees Kelvin, comparing or plotting values is taking more effort. Thus, the application would

[2] `http://apps.ecmwf.int/mars-activity/`, accessed: 2017-10-12

need to offer support to convert between different units to smoothen the problems with embedded data.

Naming conventions are another problem which might make comparisons harder. Each data source can provide different naming schemes for their provided data which might make querying data harder. The application should provide at least basic support to ease this problem.

Finally, integrating the data might be problematic as well as every data source can provide data in different formats. Thus, the transformation between those different formats needs to be implemented to allow usage of common interfaces between the application and different weather sources. Storage in the application, in turn, can be done either in separate models or in one common model. As such, the application should be easily extensible to integrate more weather sources and provide an interface which defines required and optional methods that might be needed for the integration of other sources.

5 Architecture

This chapter provides insights into the design and implementation phase of the application. While the first section provides reasoning on various design decisions, the second part describes the various mechanics that are built into the API.

5.1 Design Phase

One of the first parts of any projects is finding suitable tools and solutions to previously identified problems. The following section will introduce the paradigm and one architectural pattern that is used in the project afterwards it provides information about used programming languages, tools and frameworks.

5.1.1 RESTful API

The application is planned to be implemented as an API that uses the representational state transfer (REST) paradigm. This paradigm is an abstraction that builds onto the structure of the world wide web. Thus, REST is stateless, uses standard operations (`GET`, `PUT`, `PATCH`, `UPDATE` and `DELETE`). It aims for performance, reliability and scalability and was designed by Roy Thomas Fielding [41][p. 76ff] as a part of his doctoral thesis.

There are several architectural constraints that define a RESTful system:

- **Client-Server**: Separating the user interface from data storage concerns improves portability of the UI and scalability by removing components on the server.

- **Statelessness**: Each request from a client must contain all information necessary to perform the action without using any stored context, removing the need to store context.
- **Cacheability**: Resources should be labelled as cacheable or non-cacheable - this, in turn, might eliminate some interactions between client and server with the trade-off of having possible stale data stored.
- **Uniform Interface**: Decouples the server from the client implementation by using the URI to identify a resource, manipulate data via the HTTP standard, providing self-descriptive messages by using MIME Types and potentially provide hyperlinks and URI templates to further decouple the client from a specific URI structure. This is summarized under the term hypermedia as the engine of application state (HATEOAS) [42][p. 142ff].
- **Layered Systems**: A system can consist of multiple layers with a single access point for the end-user.
- **Code on demand**: The last and optional constraint consists of the possibility of transmitting the code to the client for execution e.g. JavaScript code within an HTML representation.

Those constraints provide an outline on how to apply the pattern to real-world applications without going into the details on how to implement those constraints. In turn, it also provides hints on how access to the implemented API could be structured.

5.1.2 PORTO Software Architectural Pattern

In addition to the RESTful approach, an architectural design pattern can be used to facilitate loose coupling and re-use of code. Porto [43] is a pattern that describes an alternative to the standard model-view-controller model widely used in applications. This pattern can be applied to RESTful APIs as well as normal web applications.

At the core, it describes two layers — the *Ship* and *Containers* layer. Those layers can be enhanced with an additional, optional set of components with predefined responsibilities, which both sit above the base framework as depicted in Fig. 5.1.

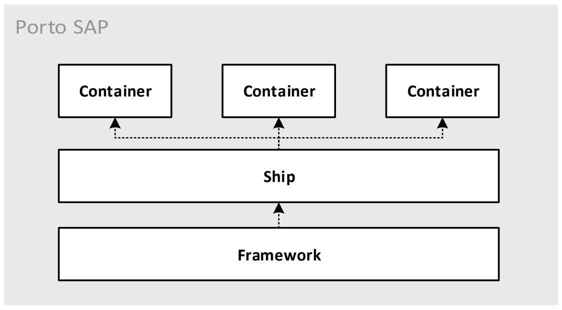


Figure 5.1: Overview about the different layers in the PORTO SAP[1].

The *Ship* layer contains mid-level code which can be used by all containers but should be kept thin. Commonly used business logic should reside encapsulated inside of single containers. Thus, this layer should only contain, either, shared code or the *core* code, which loads the containers and provides helper functions.

In contrast, the *Container* layer contains all business logic, through components, for a specific use-case. Therefore, this layer can be seen as a means to bundle up functionality. For example, everything that pertains users, like registering, viewing the user's information, listing users and of course the user model itself should be implemented in the `User` container. Whereas `Authentication`, in turn, would be a separate container to check tokens or other forms of authentication and provide login and logout features which depends on the `User` container. Using this approach leads to loosely coupled packages which adhere to the single responsibility principle.

Additionally, *Components* can be used in either layer to provide functionality. They are split into two categories *main* and *optional* components. Main components are essential for web applications and contain elements such as routes, controllers, requests, tasks, models, views, and transformers. In contrast to that, optional components are not explicitly, needed but can provide further functionality. Some of those components are repositories, exceptions, criteria or tests.

[1]Based on the original `Layers Diagram` by *M. Zalt* Available: `https://github.com/Mahmoudz/Porto#Layer-Diagram`, accessed: 2017-11-20

Lifecycle of a Request

One complete lifecycle of a request, be it from a web view, an API call or the command line interface can be traced in Fig. 5.2. It begins when a call hits an endpoint that is defined in a `Route`, as it will call a `Middleware` to handle the `Authentication` and the corresponding `Controller` function. After this, the `Request` is injected into the `Controller` and automatically applies all validation and authorization rules. Afterwards, an action is called including any data from the `Request` that was expected. In turn, the action might either handle the `Request` or call one or more `Task` to do that. With a Task doing only a single portion of the main Action in itself or any `Models`. Afterwards the `Action` prepares data the resulting data from processing the request back to the `Controller`. Finally, the Controller builds the Response by either using a `View` for the web or `Transformer` to return serialized information.

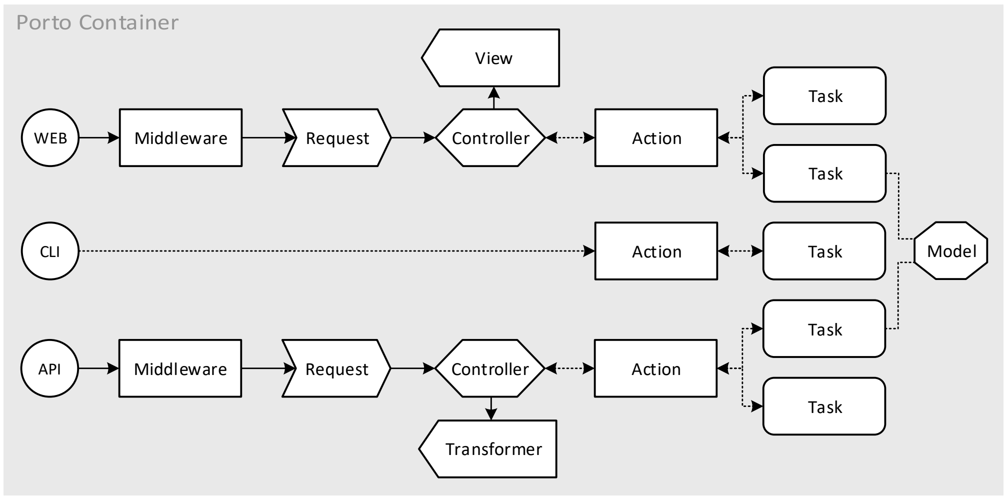


Figure 5.2: Interactions between the Components in Porto SAP[2].

Benefits

Consequently, the pattern provides tools to facilitate reuse of code and decouples the business logic from the framework. In addition to that, the possible user interfaces are also separated from the business logic, which makes them pluggable.

[2]Based on the original `Main Components Interaction Diagram` by *M. Zalt* Available: `https://github.com/Mahmoudz/Porto#Components-Interaction-Diagram`, accessed: 2017-11-20

Summarizing, this pattern facilitates decoupling, as the business logic is completely separated from the user interfaces and segregated in containers. This makes it easy to extend the scope of an application from being an API to also offer a web-application or command line interface by reusing already defined components. It also promotes single responsibility of components when the pattern is used as described. Another point is the ability to quickly find code when the pattern is taken to heart, as every component has a pre-defined location in a project. Currently, there is one implementation of this pattern available that is built on top of the Laravel[3] PHP framework.

Implementation of the Pattern

Apiato [44] implements all aforementioned components and offers various, optional, pre-made containers such as a generic user, authentication and authorization container. This, in turn, allows rapidly developing new ideas by just providing business logic instead of having to rewrite or reimplement reoccurring tasks. Despite being implemented with the architecture pattern in mind, Apiato does not force developers to strictly follow the Porto pattern. As such, developers can choose any way to implement their containers.

5.2 Tools and Frameworks

The API itself is developed in PHP, by utilizing the Apiato framework, and developed as a RESTful application. This approach allows the project to be consumed by various clients as all output will be in the form of JSON Responses utilizing the JSON API[4] specification. This allows the users to build their own clients on top of the provided project to further process and aggregate the collected data. JSON API provides a common structure for representing objects in JSON. Which can be used with a variety of ready to use libraries.

Additionally, the API also minimizes duplicate data, as information about users of specific studies will not be collected. All calls to the application for a specific study should be done via one registered user that is used as an intermediary between this application

[3] `https://laravel.com/`, accessed: 2017-11-20
[4] `http://jsonapi.org/`, accessed: 2017-11-20

and existing study platforms to improve re-usability for clients. In turn, the existing study platform needs to handle user management. This is needed to distinguish between researchers that can obtain stored data or normal users that can only produce new entries in this application. As a result, this also minimizes any privacy concerns as this application will only know about the participant identifier that is explicitly shared without storing additional information about participants.

Additionally, all developed containers should have minimal dependencies to other containers, so that other projects are able to plug the containers into study platforms to collect environmental data without having to include a multitude of different containers to get started. In addition, the provided containers might find use in other projects as well.

In addition to developing the RESTful application, it was also necessary to develop libraries that would retrieve the data from the chosen data sources. Language and Frameworks used differed based on the existing tools and frameworks.

MySQL 5.7.18 is used as the database for the environmental API. This specific version was chosen due to the added JSON support for columns. Which allows easier storage of data that will not be searchable or modifiable. The main use-case in this API is storing additional information about the DWD weather stations that does not change much.

5.3 Implementation

At the start of the project, the first step consisted of exploring which data was available for re-use, how to access the data, which constraints the data sources had and finally how to access the data. Afterwards, the work on the API began when the aforementioned parts were working as intended. This had the primary benefit of being sure, that retrieval with the given sources is possible while also minimizing the need to switch between the used programming languages. In contrast to that, this chapter will first introduce the work done on the RESTful application and then provide details on the data retrieval helpers.

5.3.1 Components of the API

All components of the API will be briefly introduced here, then in further detail in their own section. Additionally, the Apiato project provides several inbuilt mechanisms that can be plugged in as needed, which will be introduced as well. This application uses the provided `User`, `Authorization` and `Authentication` containers to handle external APIs as users. The API consists of two custom containers that contain the business logic. They can be either used stand-alone or plugged into existing Apiato projects to provide environmental data. Those containers are called `GeoLocation` and `WeatherData`. Retrieval of data is handled via various `Gateways` that provide a loose coupling between the API and the data source. This is done by scheduling jobs that retrieve data for one specific `GeoLocation` inside of a queue. In addition, the API also supports on the fly conversion between units, in case the user wants to convert either specific or all values to specific units when applicable.

5.3.2 Authentication and Authorization

Both the authentication and authorization are inbuilt into Apiato and are shipped with the framework including a container for users. This application utilizes those containers.

The `Authentication` container provides a ready to use authentication middleware that is based on Laravel Passport[5] and implements OAuth. This application uses the `password grant tokens` to authenticate third-party applications. As such, a `User` object is currently used to log into one survey - multiple surveys can be served with one instance of this API. In turn, the resource owner credentials grant also requires the admin of the API to register new application to the app manually by creating a new client. After the registration, a user (in this case, the external API) is able to login with the credentials and access the API.

In addition to that, the API also offers support for role-based access control (RBAC) inside the `Authorization` container. This system provides measures to protect specific routes by checking the roles associated with a user account. In addition to roles, the

[5] `https://laravel.com/docs/5.5/passport`, accessed: 2017-11-20

container also supports specific `permissions`. Both, `roles` and `permissions` can then be specified in the `$access` field of a `Request`. New users of the platform currently need to be manually set to have the `researcher` role. Which, in turn, allows the external API to also query the recorded data via the provided routes.

5.3.3 GeoLocation Container

Primarily, the `GeoLocation` container provides functionality to store, retrieve and delete existing `GeoLocations`. This container also provides a `GeoLocation` model that keeps track of user location (`latitude` and `longitude`), as well as a corresponding `timestamp`. Additionally, it also contains the `id` of the user who created the entry and a `participant_id`. Lastly, the model contains a flag which indicates whether the `GeoLocation` has been used to obtain the environmental data.

A distinction between user and participant was introduced to minimize privacy concerns and to reduce duplicate storage of user data. Registered users to this API are the external applications that want to integrate this project into their work. It also provides a means to limit access from one survey to others.

5.3.4 WeatherData Container

The main business logic of the project is implemented in this container. First of all, it only provides routes to query the collected environmental data. This container provides two models to store environmental data and additional information. One `WeatherData` will store exactly one environmental data parameter that consists of the date of the data point, distance to the queried point, latitude, longitude, unit and value. In addition to that, a `WeatherSource` model is only used to provide additional metadata and consists of the original (assigned by the data source), a source string and a data JSON column. It is used to dump the retrieved additional data. It was designed this way, as currently there was no need to ever update specific content inside the data column, either the field content is rewritten completely or never changed. Currently, it is used to provide information about the DWD Station that provided the retrieved data (cf. Section

4.1 for more specifics regarding stations). Fig. 5.3 shows the relationships between `GeoLocations`, `WeatherData` and `WeatherSource`.

This container also provides several utility options. First of all, it contains a command that retrieves `GeoLocations`, which were not `executed` yet and for which the application determined, that data can be retrieved for, from all sources. In that context, `executed` means, that no environmental data has been received for the given `GeoLocation`. It then schedules a job to retrieve environmental data for this specific `GeoLocation` to the worker queue. Additionally, there is also a command to trigger the pre-fetching of Copernicus data, as the ECMWF service might take a longer time to respond. Both commands are scheduled to run at different intervals. Data retrieval will run every five minutes and schedule a flexible amount of jobs and pre-fetching of Copernicus data happens once daily at 00:00.

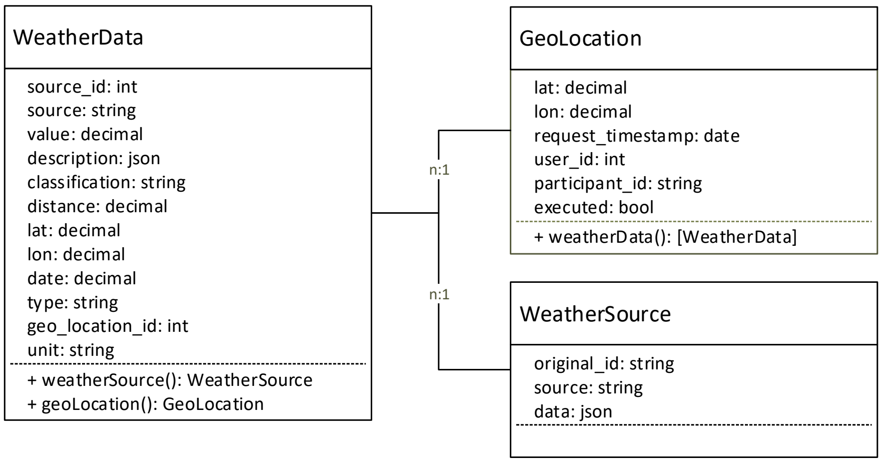


Figure 5.3: Class Diagram that shows how `GeoLocations`, `WeatherData` and `WeatherSource` are connected. All public functions provide a means to retrieve linked models.

5.3.5 Gateways to Weather Sources

In the developed application, a `Gateway` is introduced, to keep an abstraction between the library that will retrieve environmental data, and the API. It provides a common set of functions that are expected to be implemented to ensure re-usability. An outline of this adapter is given in `AbstractDataRetrievalGateway` and enforces subclasses to implement various methods to standardise retrieval. In turn, it provides one publicly available template function `getData($lat, $lon, Carbon $date, $geoLocationId)`. It will return an array that consists of two items. First, it provides the environmental data that has already been transformed into the expected `WeatherData` format and the second one carries the additional information in the format of a `WeatherSource`. Summarizing, the method encapsulates the retrieval and transformation steps. Subclasses will only be able to provide sub-functions of this function without being able to override the main functionality.

First of all each `Gateway` needs to provide a `retrieve($lat,$lon,Carbon $date)` method. This method is expected to return an array that contains two items, with the first being the raw weather data and the second providing raw source data. The latter can also be an empty array.

After this, the user needs to override the `parseToWeatherData($obj,$geoLocId)` function. This function takes one of the retrieved weather data objects and in turn will transform it into a valid array representing a `WeatherData` model. It can either be done manually or with the help of a `Transformer`. Those `Transformers` take one object and transform the content into other representations [42][p. 62ff]. The default `parseToWeatherSource($obj)` method will provide an empty array when called and can be overwritten in case the data source provides additional metadata which should be parsed into a `WeatherSource`.

Finally, each `Gateway` needs to provide the delay in days, which each data set inherently carries in the `getTimeDelayInDays()` function. For the *DWD hourly data set* this value would be *1 day*, while the *Copernicus (CAMS)* set has *5 days* of delay. Those values are used later on, to determine the which `GeoLocations` can be scheduled for data retrieval.

A class diagram of the existing `Gateways` can be seen in Fig. 5.4. Each child class implements only the functions needed to provide the expected data to the template method of the parent class. Depending on the weather source each subclass can decide to provide other means to pre-filter data. This can be seen in the `DWDRetrievalGateway` where the `variables` determine which kind of data should be queued with the default of retrieving all available environmental parameters.

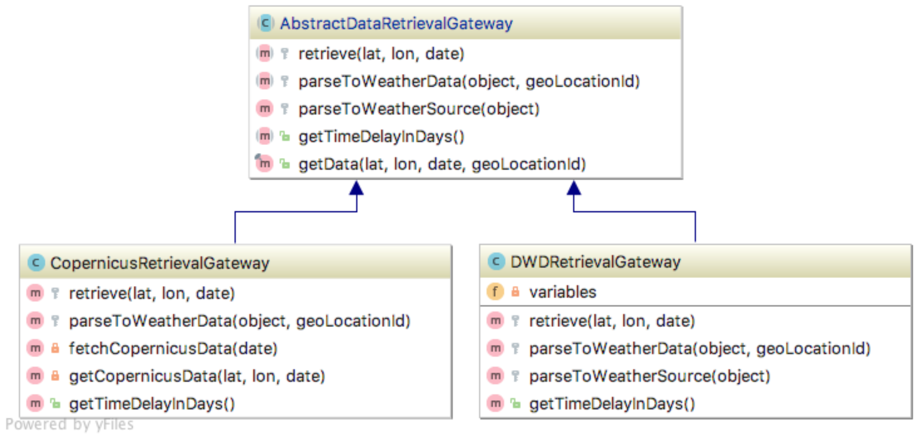

AbstractDataRetrievalGateway
retrieve(lat, lon, date)
parseToWeatherData(object, geoLocationId)
parseToWeatherSource(object)
getTimeDelayInDays()
getData(lat, lon, date, geoLocationId)
CopernicusRetrievalGateway
retrieve(lat, lon, date)
parseToWeatherData(object, geoLocationId)
fetchCopernicusData(date)
getCopernicusData(lat, lon, date)
getTimeDelayInDays()
DWDRetrievalGateway
variables
retrieve(lat, lon, date)
parseToWeatherData(object, geoLocationId)
parseToWeatherSource(object)
getTimeDelayInDays()
Powered by yFiles

Figure 5.4: Class Diagram showing the abstract parental class and the two subclasses for DWD and Copernicus data.

To summarize, the `AbstractDataRetrievalGateway` provides a template method that has to be reused when new data sources need to be introduced into the API. A new `Gateway` needs to fill in the abstract methods for retrieving and transforming raw environmental data into the expected `WeatherData`-form. All the active `Gateways` need to be added to the configuration file inside of `Containers/WeatherData/Configs/weather.php`.

5.3.6 Retrieving Environmental Data with Queued Jobs

The gateways mentioned above are used inside of a queued `Job`. In turn, Laravel will run those jobs in separate worker queues that are supported by multiple queue backends either by using the database to enqueue jobs or using specialized queue backends [45]. Connections to the different backends provide the option to run multiple queues in them. Queues also provide a means to prioritize jobs and group them by specifying a name. It is also possible to specify the number of retries per job by adding an optional flag to the queue startup like this: `php artisan queue:work --tries=1`. This enables finely granulated execution of jobs.

In turn, a `Job` has only one `handle(...)` function it has to implement, which contains the complete business logic of this one job. Additionally, jobs can be set to expire after running longer than anticipated by overwriting the `retryUntil()` method to return the date at which it should expire (e.g. `return now()->addSeconds(5);`). This will set the expiration date to job start plus five seconds. When a job hits this time limit it will fail and it will be noted in the back-end and retried later on. In those cases it is also possible to clean up after a job fails by overwriting the `failed()` method.

The API uses the concept of queued jobs to retrieve data from the implemented `Gateways` inside of its `DataRetrievalJob`. One such job retrieves all active and available `Gateways` from the aforementioned configuration file to retrieve the environmental data for one `GeoLocation`. Afterwards, it opens a database transaction to first store the additional metadata and then the environmental data. Should an error occur then no data will be stored for the `GeoLocation`. It will also not be marked as `executed`, which means that it will be re-queued during the next retrieval command. Alternatively, when no errors occur, the environmental data for the `GeoLocation` is stored inside of the database. Dependencies between all the models can be seen in Fig. 5.5.

5.3.7 Complete Lifecycle of Retrieving Data and Task Scheduling

The complete cycle of retrieving data can be seen in Fig. 5.6. First of all, Laravel is able to schedule commands and jobs [46]. So instead of configuring cronjobs that fire the specific modules it is possible to point only one cronjob ① to the inbuilt scheduler. Afterwards, scheduling commands is possible inside the `ConsoleKernel` ② in the ship section (e.g., `$schedule->command(CopernicusRetrieval::class)->daily()`, to fetch the newest Copernicus data daily). In the depicted scenario, the timed trigger calls the `RetrieveWeatherDataCommand` which will obtain the first n `GeoLocations` which are not `executed` and with their timestamp being older than the current maximum

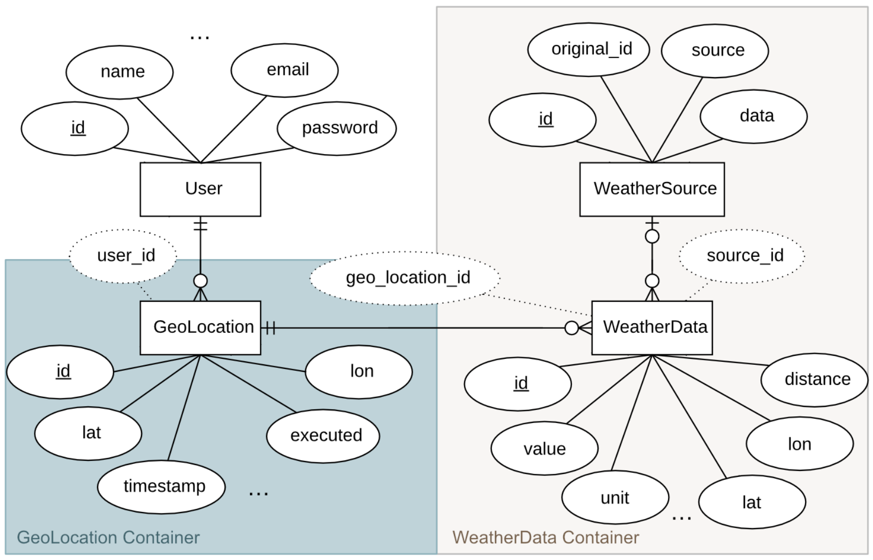


Figure 5.5: ER Diagram of all models introduced by the two implemented containers.

delay of all known data sources ③. Afterwards it will create a new job for each retrieved `GeoLocation` and add it to the queue ④.

Depending on the status of the queue, the job might reside in the queue for a while before actually being processed. When one of those jobs becomes active ⑤, it will retrieve the environmental data from all active sources (e.g., DWD, ...) ⑥ and store the data in a single database transaction ⑦. When this process is finished, the `GeoLocation` is marked as `executed`.

5.3.8 Querying Environmental Data

In addition to the detailed look at the application's mechanisms to retrieve and store data, the actual endpoint for researchers to obtain data is the other big component of this container. The retrieval options for researchers implement all possibilities mentioned in Chapter 3. The complete flow of one of those queries can be followed in Fig. 5.7. When receiving a query to the `WeatherData` endpoint, the controller calls the appropriate `Action` ①.

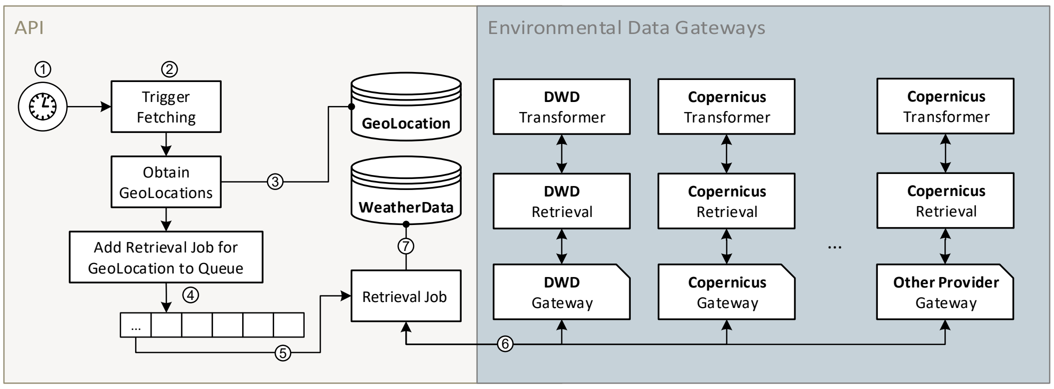


Figure 5.6: Shows the complete cycle between a time triggered fetching of `GeoLocations` up to the retrieval and storage of data inside one common `WeatherData` table.

First, it will check the request for provided parameters and will add that to the payload that is forwarded to the called `Task`. This can be done by using the so called *magical call* [47], which allows execution of `run(...)` methods from anywhere. In addition to that it also supports calling other methods beforehand, which can be used to provide additional filtering options in a structured way. In this example, the corresponding `Action` will add functions to run in the `Task` depending on the found url parameters and then call `run(...)` ②.

Afterwards, inside of the `Task`, each of those methods will add a new `Criteria` to the repository responsible for retrieving the data. `Criteria` can be seen as an abstraction for re-used queries that might occur more than once on a specific repository. Using `Critera` has several benefits. First of all one specific `Critera` can be re-used for several components inside of the application. In addition to that, it facilitates sharing common query conditions like finding objects older than a specific date or querying a specific row and comparing the value with a given value ③. This has the benefit of having to change the code for a specific database constraint only in a `Critera` instead of explicitly writing the query directly inside of an `Action` or a `Task`.

Finally, the `run(...)` method inside of a `Task` is called and retrieves the queried data ④. Afterwards, if the user added the `convert` url parameter, a second `Task` is called. This conversion task will first parse the given conversion string and then find matches inside of the previously retrieved data ⑤. A detailed description of the conversion process can be found in Section 5.3.9. The resulting data is then returned by putting it through a `Transformer` to transform it into the expected JSON-API compliant format ⑥. Then, the data is sent back.

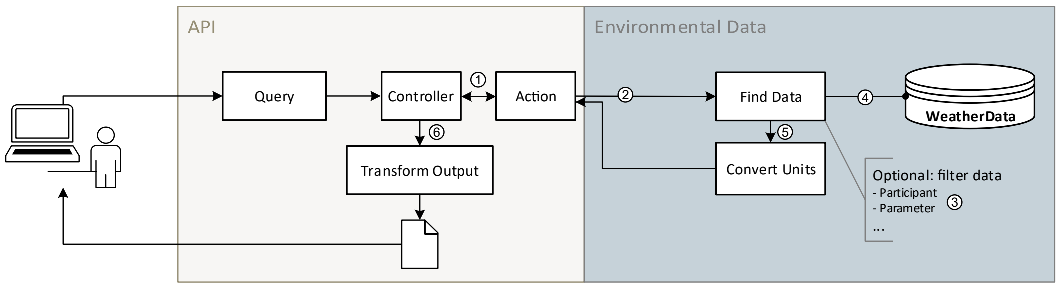


Figure 5.7: Researcher querying the API for environmental data via the provided endpoints.

Available Routes and Url Parameters

All parameters can be combined together to precisely extract only the required data. Table 5.1 depicts all different options a researcher can specify to shape the output to their specific needs. Two full-blown examples can be found in Section A.4, they contain the query including different parameters and the shortened sample output.

Table 5.1: All currently available parameters for obtaining data from the API. Base URL: `GET host/weatherData HTTP/1.1`

Option	**Explanation**
`?type=:type`	Filters data by the data type. Uses a `like` query with `*` as wildcard. Multiple values must be separated by a semicolon. An example would be the need to only obtain data pertaining the wind by adding `?type=*wind*`.

Option	Explanation
`?participantId=:ids`	Filters data by pre-filtering the data to only retrieve data from the given participant. The `ids` field can consist multiple valid ids separated by semicolons.
`?convert=:convert`	Tries to convert the filtered data. Either use `*` as wildcard to match all data or specify explicit types as csv. Example: `a,b,c:C` converts specific values a,b,c to °C whereas `*:C` converts all applicable values to ℃. It is possible to provide multiple conversions by separating them with a semicolon `*:C;*:KM`. If multiple compatible units are specified only the first conversion is applied so if the user would specify `*:C;*:K` all values would be only converted to °C and the other conversion is skipped. In comparison, it is still possible to convert specific values to different units of the same type: `a:C;b:K` will convert a to °C and b to °K. In addition, it is possible to match parameters by specifying wild cards inside of the specified parameter. An example would be finding all parameters that contain wind somewhere in their type: `*wind*`.
`?from=:date` & **`?to=:date`**	Allows the user to filter data inside of a specific date-range. If only one of both options is applied, the other one is replaced with either today in the case of `to`, or the lowest possible date in the case of a missing `from`. In addition, multiple date-formats are supported, ranging from year only up to ISO8601 timestamps. Example: `?from=2016&to=2017` will obtain data all data between the current day in 2016 and current day in 2017.

Option	Explanation
`?source=:source`	Filters the retrieved data to be from a specific source, which is determined in the `Gateways` - e.g., `?source=DWD` to only retrieve DWD parameters.
`?filter=:filter`	Provides an option to only display the values given in the `filter` field. Example: `?filter=id,value,unit` to only obtain the three mentioned values from the parameter, sample output can be found in listing A.1.

5.3.9 Conversions between Units

The conversion `Task` converts between different units and consists of two steps. First, the received `conversionString` is parsed. This string can contain multiple conversions which are separated by semicolons. Fig. 5.8 depicts one possible conversion string.

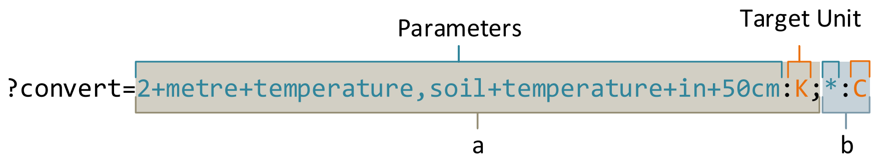


Figure 5.8: Explanation of a sample conversion string that contains two separate conversions, one using specific paramters and one that uses the wild card notation.

It can be split into two separate conversions which are applied in the given order. In turn, each conversion consists of two parts, one or more parameters that should be converted including the option to use $\star$ as the wildcard for all applicable data separated by a colon followed by the target unit. For the example above this will result in the following array. The units are being used as key and the values that need to be converted are inside of an array.

a) K: [`2 metre temperature`, `soil temperature in 50cm`]

b) C: [*]

Afterwards, the algorithm loops through each of the conversions and tries to apply the conversion from the existing unit to the target unit to every `WeatherData` model (more information in Listing A.2). The conversion itself is handled in a separate PHP library called *Convertor* [48] and adapted to the needs of this application, cf. Listing A.3. In summary, while iterating through the *WeatherData* models, the application checks whether a conversion is needed.

This check first makes sure, that the current item's unit is not already equal to the target unit and that it has not yet been converted. Additionally, it will check if the current item is meant to be converted (either because it was explicitly meant to be converted or due to the wildcard character). When all checks are passed, the item may be converted. Should the check fail, the current item is skipped and the loop continues. Refer to Table B.1 to see which of the available units are currently supported, valid target units can be found on the documentation of the Convertor library [48], added target units are listed in section B.2. New conversions can be added easily by writing your own conversion file (cf. Section 5.6 for more information).

Finally, the converted data is sent back to the correct `Action` that called the method. In turn, the data can then be transformed into the expected output format and returned to the caller.

5.4 Copernicus Retrieval Wrapper & Microservice

Directly after finding out about the Copernicus environmental data and the way to access it, wrapping the existing libraries in one new library that would abstract away some parts seemed to be the way to go. This would facilitate the integration into other applications.

Thus, the new library is wrapping both, the `ecmwfapi` library to fetch the raw data in the GRIB format and `ECCodes`, to parse the files for specific data in one wrapper package.

In turn, the `CopernicusRetrieval` wrapper covers both tasks. The big benefit of the wrapper is, that most of the things one would have to encode by hand after reading the existing documentation are encoded into enums to be used directly. Those enums include information about available retrieval times, steps, data sets and the parameters per set.

Furthermore, a microservice which uses the wrapper had to be created, as it was problematic to call a python script directly from PHP on the test platform. This has several benefits. First of all, this microservice can run on another computer and does not have to be handled by the same machine as the main API. Furthermore, it provides a certain layer of abstraction as all communication is now based on the same principles. Finally, it also allows interested users to use the microservice stand-alone instead of having to create it themselves. In turn, the following sections will first provide specific details about the wrapper library and afterwards about the microservice.

5.4.1 Copernicus Retrieval Wrapper

As already described, one of the main benefits of using the `CopernicusRetrieval`[6] wrapper instead of utilizing the provided libraries is the ease of use it provides. It contains several abstractions which facilitate ease of use. This wrapper was first intended to be used by using PHPs option to do system calls which can be seen in the overview in Fig. 5.9.

Overview

The following sequence diagram shows the complete cycle of first retrieving files and the retrieving specific data from the file and can be seen in Fig. 5.10. Both the `ECMWFDataServer` and the `ECCodes` depict the interaction of this wrapper with both ECMWF libraries. The `get_nearest_value(...)` method has been shortened, it does

[6]available on Github at `https://github.com/FWidm/CopernicusRetrieval`

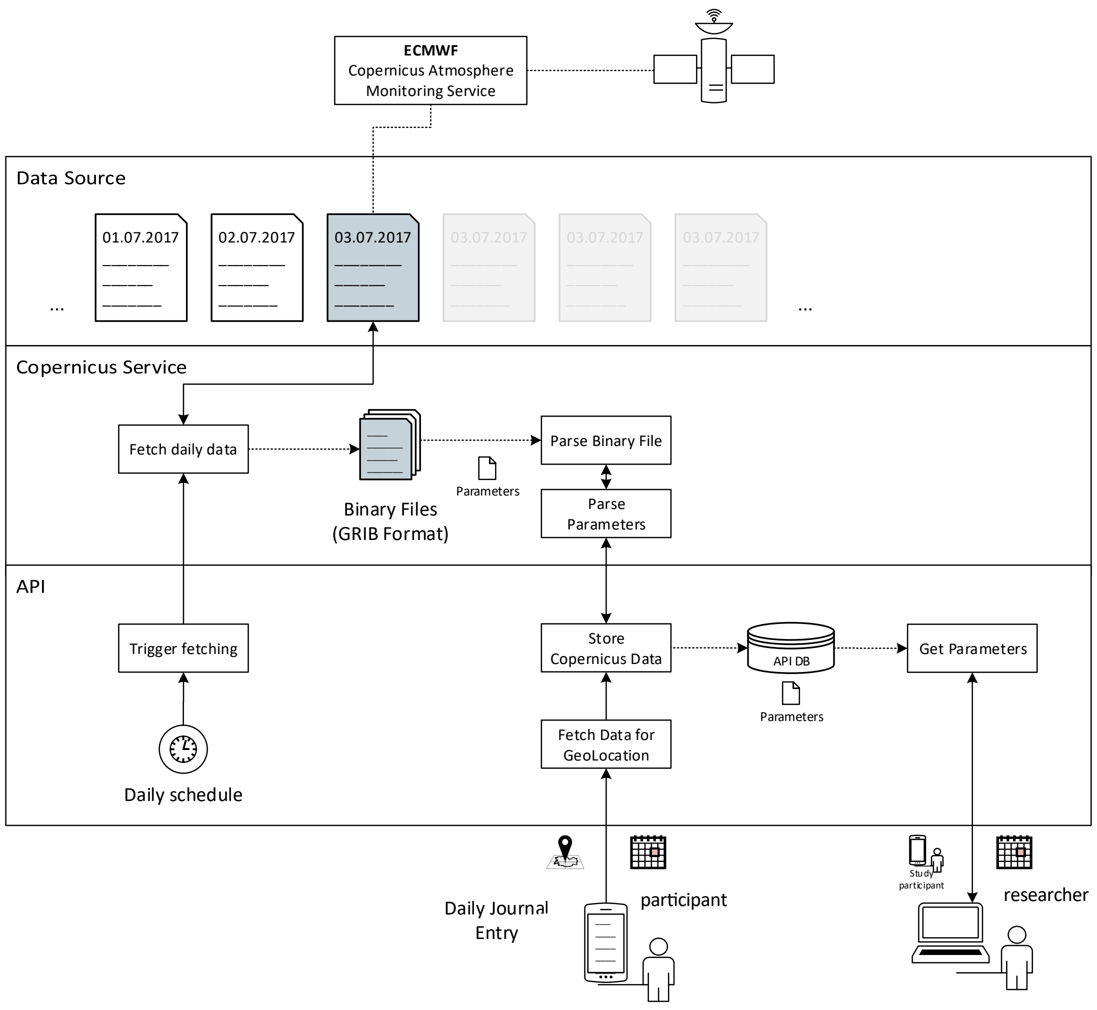


Figure 5.9: Planned Copernicus Wrapper including planned API access.

not only find the nearest data in the grib file, but also retrieves meta information such as the used unit and type of parameters by calling `retrieve_metadata(grib)`.

Retrieval

The retrieval process is available in two ways. One user of the wrapper can choose to use the traditional way to configure the request by creating a dictionary with all expected data and the wrapper will send this exact request to retrieve the required file. Conversely, the other option is using the new method which contains several optional parameters

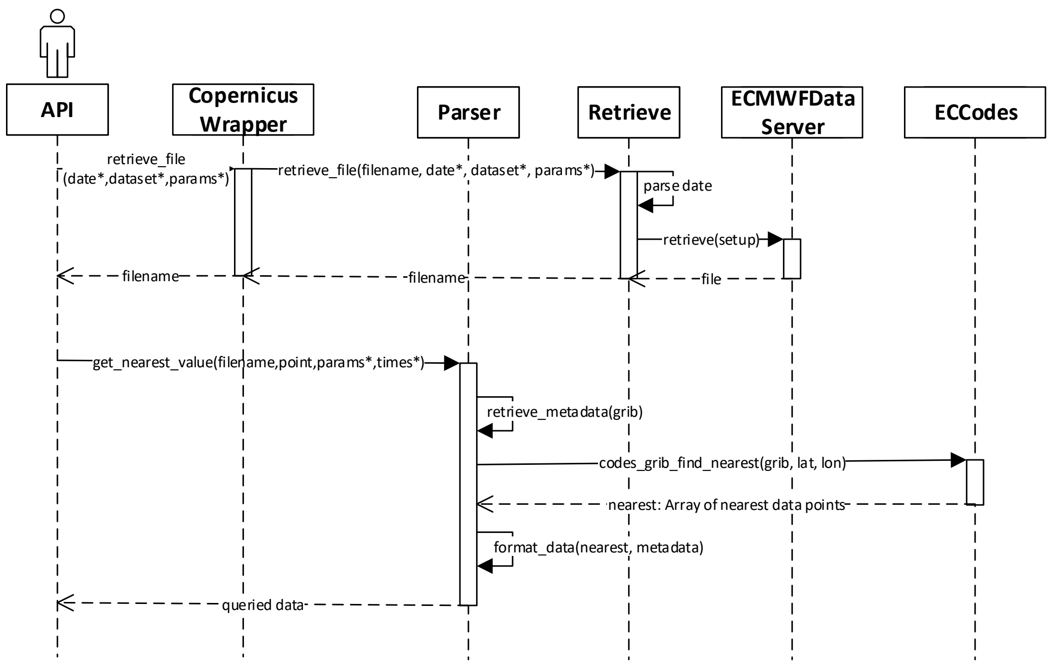

API
Copernicus Wrapper
Parser
Retrieve
ECMWFData Server
ECCodes
retrieve_file (date*,dataset*,params*)
retrieve_file(filename, date*, dataset*, params*)
parse date
retrieve(setup)
file
filename
filename
get_nearest_value(filename,point,params*,times*)
retrieve_metadata(grib)
codes_grib_find_nearest(grib, lat, lon)
nearest: Array of nearest data points
format_data(nearest, metadata)
queried data

Figure 5.10: Sequence diagram depicting the API first retrieving a file with specific date, dataset and parameters and then retrieving nearest data for a specific point. All parameters that are denoted with an asterisk should be seen as the user choosing the values for them.

to customize the retrieval without having to know a thing about the expected request format. This function has only one non-optional parameter which is the file name. In addition, it also takes a date (the default is `today`), the data set (as an `enum`, default is the `CAMS` set), times (an enum representing the four available times: 00:00, 06:00 12:00, 18:00), the data type (analysis or forecast, default is analysis), the steps (zero, per default) and finally a boolean flag which restricts the retrieved data to Europe (cf. Section 4.2.2). Afterwards, the method constructs the request dictionary by using the provided information and will retrieve the file for the user.

To be more specific about the ease of use this new function provides, think of the following scenario. Instead of specifying the specific code for a parameter like this:

```
"param": "151.128/167.128"
```

to retrieve the temperature and mean sea level pressure as a new file, the user can use the wrapper method which per default retrieves all available parameters, while also being able to provide specific required parameters in an array as an optional parameter when it is specified in the function call:

```
params=[Enums.ParameterCAMS.TWO_METRE_TEMPERATURE,...]
retrieve.retrieve_file(..., parameters=params)
```

This is possible, as all `CAMS` parameters have been parsed into a dictionary form and inserted as an entry to `ParameterCAMS`. This enum provides information about each available parameter. In addition, various other important aspects of the retrieval and parsing process have been encoded into similar enums such as the type of data (analysis or forecast), available times (00:00, 06:00 ...), an enum that contains a function to retrieve a classification for a specific parameter by its name and the available data sets including their names, delay and used date-format.

Parse

After downloading one GRIB file, the user can parse it for the data by using a `Parser` object. It provides a method to retrieve the nearest values by specifying the file, latitude and longitude as tuple and optional parameters such as `n` to specify the number of points to retrieve (either the nearest point or the four nearest points on the grid), which parameter to parse with the default being all parameters, the requested time (per default it includes all times) and whether the resulting dictionary should be grouped by parameters or not (default being true).

The next step is parsing each item inside of the grib file by extracting the available metadata if it matches the parameter the user wants to retrieve. This includes a `timestamp`, `unit`, `parameter id`, `name` and various other variables. Afterwards, both the data and metadata of the current item are combined in one dictionary, which in turn is parsed into an instance of a `CopernicusData` object and is stored for later retrieval. All parameters that are not required (not in the method arguments) are skipped. The result of this function is a dictionary that has all available parameters as keys with an array of the retrieved data as value. This can be seen in the provided example output in Listing A.9.

Additionally, the `Parser` objects also provide a method that will parse all parameters inside of a GRIB file and produces a list of parameters equal to the one seen in Listing 5.1. This list can then be used to produce new enums for other data sets in the future.

Enums

Enumerations were chosen as a substitute for having to memorize or look up various variables used for parsing and receiving data from various ECMWF data sets. All enums are currently placed in one single file. Currently, there are five different enums which are also depicted in the class diagram shown in Fig. B.1. `ParameterERA5` is currently not developed but still included as a placeholder.

Classification: Provides the utility to apply a custom classification based on the parameter name

DataType: The type of data, currently either forecast or analysis are supported.

Time: An enum that contains the available times for retrieval, contains a method to parse a `datetime` or `timestamp` into an object of this enum.

ParameterCAMS: All available `CAMS` parameters for both analysis and forecast an example can be found in Listing 5.1.

ParameterERA5: Currently unimplemented placeholder for ERA5 parameter information.

DataSets: Information about the supported data sets resides in this enum. Entries are dictionaries that contain the name of the set, the class as specified by the ECMWF, the delay of the data set and if necessary the date format used.

Some of the listed enums also provide a method to retrieve a list of all entries inside of the enum by calling the `all()` function if available.

```
MEAN_SEA_LEVEL_PRESSURE = {'eraId': '151.128', 'shortName': 'MSL', 'id': 151, 'unit':
    ↪  'Pa', 'description': 'Mean sea-level pressure'}
```

Listing 5.1: Example enum entry including the available fields for the mean sea level pressure of the CAMS data set.

5.4.2 Implementation of the Microservice

The microservice `CopernicusAPI`[7], that facilitates the features provided in the wrapper mentioned above is also written in Python and uses the Flask framework. Flask is a lightweight web service framework that has the benefit of being easy to use, simple to set up, but also scalable to fit future requirements. It has built-in support for logging, caching and modular applications and in turn, provides a solid foundation for a RESTful microservice.

The built microservice contains two routes, which are used to retrieve files from the ECMWF servers by using the retrieval part of the `CopernicusRetrieval` library and to parse files by wrapping the parser part of the previously mentioned library. In addition, there are two utility routes. They provide information about locally stored files and a means to check if the service is running at all.

Design Decisions

The application is using flasks blueprint system. A blueprint will extend the base application and allows plugging sets of operations into the core application. In this case, the blueprint system is used to decouple the routes to specific resources from the main application. This means that all routes that exist for a specific resource share one common blueprint. A blueprint acts like a `Controller` in the environmental data API and will parse the request parameters and check necessary conditions before calling the `Actions` that contain business logic to retrieve or parse a file. This structure was chosen to partly emulate the Porto pattern described in Section 5.1.2 (cf. Fig. 5.11). In addition, the action itself was separated from the route definition.

[7]available on Github at `https://github.com/FWidm/CopernicusAPI`

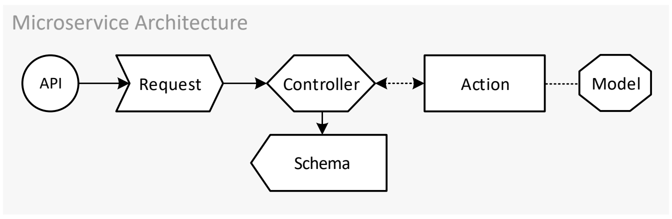


Figure 5.11: Adapted life-cycle similar to a minimalistic Porto implementation.

Available Routes

`/` : this route will display a string and can be used to check whether the api is up and running.

`/files` : lists all locally available files. The JSON response looks like this: `"files": ["an-2017-09-18.grib"]`

`/retrieve` : triggers the retrieval of files and expects the user to provide a valid timestamp in the url.

- Example: `GET host/retrieve?timestamp=ts HTTP/1.1` with `ts=2017-09-18T15:21:20%2B00:00`.
- Output 1: Response code 202 (File is not available/still downloading):

```
{
"message": "Download is currently in progress. Retry this operation to
  ↪  retrieve the filename in a minute."
}
```

- Output 2: Response code 200 (File is available, message contains additional info e.g. cache hit):

```
{
"data": {
        "fileName": "an-2017-10-25.grib"
},
"message": "Cache hit."
}
```

- Output 3: Response code 404 (Given date is not available currently, returns the latest retrieval date)

```
{
"data": {
        "latest_retrieval_date": "2017-09-13"
},
"message": "Cannot retrieve files for this date."
}
```

/parse parses an available file. Expects various parameters that have to be provided by the user.

- Example: `GET host/parse?timestamp=ts&lat=la&lon=lo HTTP/1.1` with `ts=2017-09-18T15:21:20%2B00:00`, `la=48.4` and `lo=9.6`.
- Output 1: Response code 404 (File is not available/still downloading):

```
{
        "data": {
                "files": [
                "an-2017-09-25.grib",
                ...
                ]
        },
        "message": "Given filename=an-2017-12-25.grib could not be found in
          ↪  the available files are attached."
}
```

- Output 2: Response code 200 (File is available): (cf. Listing A.8). Currently the output is identical to the one from the wrapper.

File Status

When starting the microservice, it needs to keep track of locally available files. This could have been done by using a database, but due to the nature of the stored data, a simple class holding a dictionary was sufficient. Consequently, the `file_status` class contains one dictionary, which has the names of the available files as keys with their boolean availability as value. When starting the microservice, a singleton of this class is instantiated to keep track of all files in a pre-defined directory. Furthermore, it provides methods to track new files, mark them as either available or not and remove files. In turn, the only problem that could arise would be a user removing files from the directory while the API is running. This issue, however, would also not be solved by deploying a database for file tracking.

Asynchronous Retrieval

In its current state, the asynchronous retrieval is using a `ThreadPoolExecutor`. Afterwards, the instantiated executor is imported into the `retrieve_action`. Should a requested file not be available, the future filename is added to the `file_status` (though not marked as available) and a `download_action` is submitted to the executor. In turn, the application checks whether the download is finished or the file is marked as available. In this case, the filename is returned, to indicate that the file exists. Contrarily, it will return a message that will notify the user about the file still downloading.

The `download_action` itself is straight-forward and uses the `Retrieve` functionality provided by the `Copernicus Wrapper` to download the file. When the download finishes, the file is marked as available inside the `file_status`.

Experimental Caching

Additionally, the microservice facilitates the inbuilt `SimpleCache`, which is a memory cache for single process environments as it is not fully thread-safe. When using it in production, it would be necessary to either remove the implemented caching or switching to another cache system. Currently, both the retrieval and parsing calls are cached by storing the result of a call with the `request.uri` as key. In turn, within each route definition, the microservice checks the cache for a hit before calling the action to speed up the response time. Should this API be used in a production environment it might be necessary to either remove caching completely or swapping the caching system by simply instantiating another system instead of the used one in the `cache.py` script.

Transformation of Data

Experimental support for the transformation of output data is also available. It is possible to pre-filter the output data in the wanted formats. This is possible by applying Marshmallow[8] `Schemas` to the resources. To only output specific fields when querying the parse endpoint, it is possible to add the following to the `only` parameter of the `Schema` constructor:

```
schema = CopernicusDataSchema(many=True,only=('type','value'))
```

Currently, this microservice has two pre-defined `Schemas`. One for the generic messages and one for the `CopernicusData` output. They define a specific format that can be applied to existing data. Refer to Listing A.11 to view the `CopernicusDataSchema`.

5.5 DWD Hourly Crawler

In addition to the `Copernicus Wrapper`, it was also necessary to create a similar tool to retrieve the DWD data from their servers. There were many solutions available which were either available for different languages or no longer maintained, as the paths to the files may have changed. This led to the creation of the `DWD Hourly Crawler` library[9]. It is written in plain PHP. An overview of the core functionality can be seen in Fig. 5.12.

The library is highly configurable to be easily adapted in case any of the paths on the DWD servers change. A more detailed, technical view of the library is depicted in Fig. 5.13.

Overview

First, the user needs to specify, which of the available services he wants to use to query the DWD data, by creating a `DWDHourlyParameters` object and adding the needed parameters (l. 2; cf. Listing 5.2 for a real-world example). In addition, the user specifies the retrieval data (l. 3-4). Afterwards, the user uses an instance of the `DWDLib` to retrieve data, either in an interval, or the complete data for one day by calling the corresponding function (l. 5). This object will then create the services from the given parameters and pass them to an instance of `DWDHourlyCrawler`. The crawler will check each service and call the `parseHourlyData(...)` method.

[8] `https://marshmallow.readthedocs.io/en/latest/`, accessed: 2017-11-20
[9] avilable on Github at: `https://github.com/FWidm/dwd-hourly-crawler`

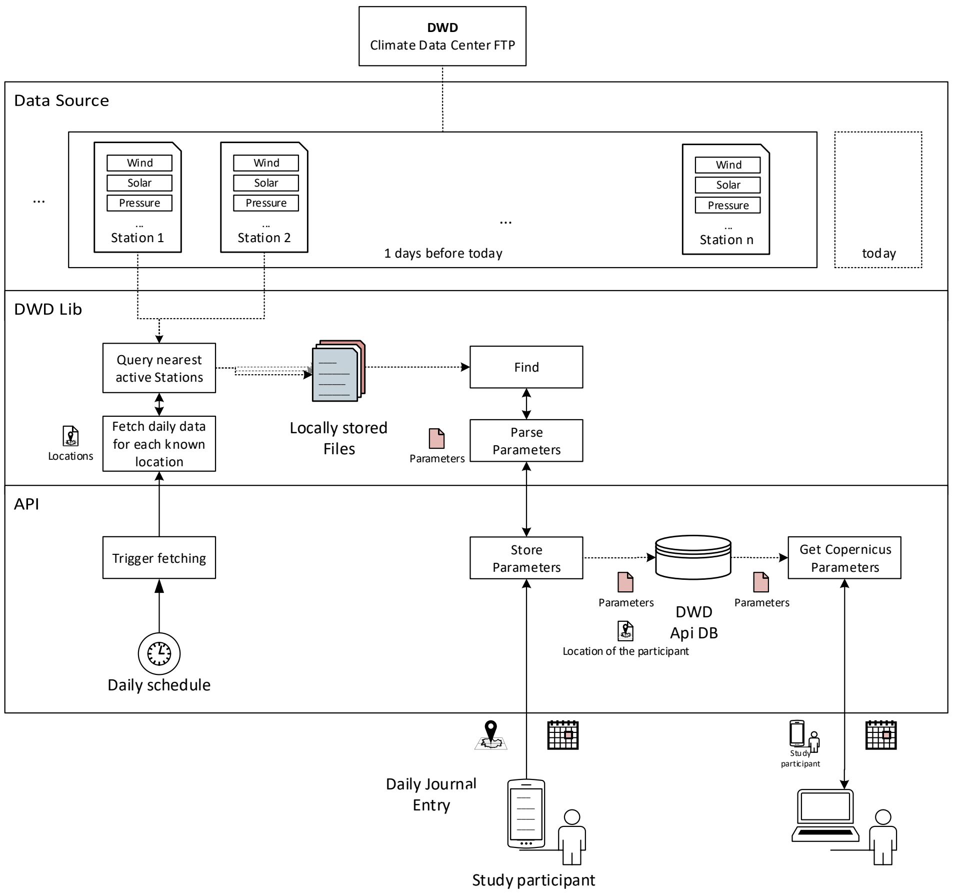


Figure 5.12: Planned integration of the library onto the RESTful environmental API.

Afterwards, the crawler will go through each service and retrieve data for the parameter. This is possible, as all services extend `AbstractService`, which, in turn, provides a template method to retrieve and parse data. Each subclass must implement the corresponding `createParameter(...)` method to parse the parameter and instantiate the corresponding model. An overview of the provided methods can be seen in Fig. 5.14. The resulting output data is put into an array (cf. Listing A.10). All retrieved parameters have their own classes to store the data in a structured format, including the weather stations themselves.

Retrieved files are currently stored on the file system and are not unpacked. Each access to specific data will operate on the zip files themselves to save disk space.

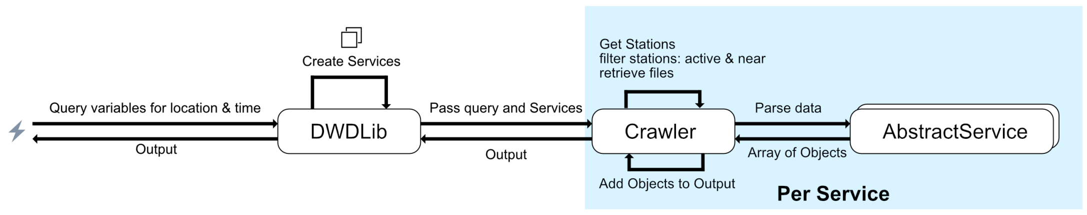


Figure 5.13: A short introduction to the inner workings of the library including core components.

```
$coordinates=new Coordinate(48.3751,8.9801);
$vars = new DWDHourlyParameters();
$vars->addAirTemperature()->addWind()/*->add...*/;
$date=new Carbon()->modify("-4 days");
$out=$dwdLib->getHourlyByInterval($vars,$date,
 ↪  $coordinates->getLat(),$coordinates->getLng());
```

Listing 5.2: Usage of the `DWD Hourly Crawler` library. Retrieves data for one specific point and date to retrieve temperature and wind data.

Functionality

The library queries all available hourly parameters on the DWD's Climate Data Center server (cf. Chapter 4.1.2). It provides ready to use models for each parameter and a model for weather stations. In addition, each of the parameter models also contains a method to split all contained variables into single objects. This is useful when one wants to store single weather data entries without having to adhere to the given structure by the DWD.

Finally, the library contains various safety mechanisms to make sure that the requested data will be retrieved, even if the DWD data contradicts itself. Imagine querying data for a specific location: It may happen, that the closest station to that point is marked as active, but no data exists for the requested parameter. In such cases, the library automatically tries to use the next available stations. This can lead to one query that will provide data for all parameters but from various weather stations.

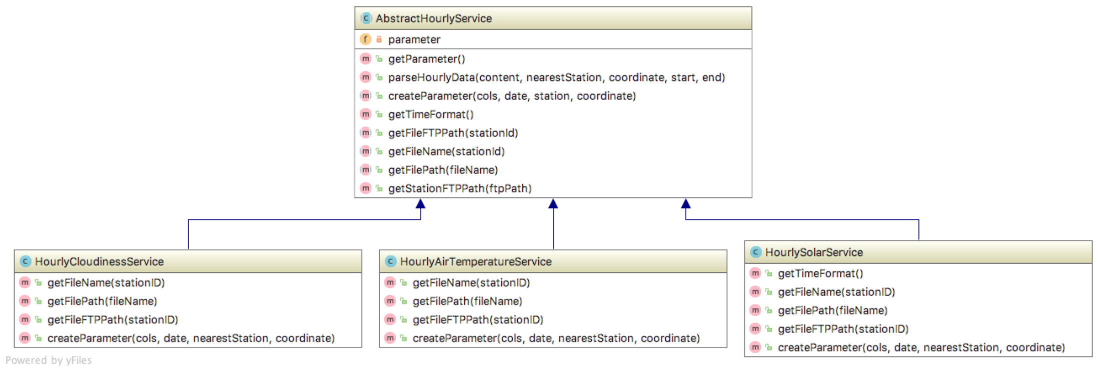

AbstractHourlyService
parameter
getParameter()
parseHourlyData(content, nearestStation, coordinate, start, end)
createParameter(cols, date, station, coordinate)
getTimeFormat()
getFileFTPPath(stationId)
getFileName(stationId)
getFilePath(fileName)
getStationFTPPath(ftpPath)
HourlyCloudinessService
getFileName(stationID)
getFilePath(fileName)
getFileFTPPath(stationID)
createParameter(cols, date, nearestStation, coordinate)
HourlyAirTemperatureService
getFileName(stationID)
getFilePath(fileName)
getFileFTPPath(stationID)
createParameter(cols, date, nearestStation, coordinate)
HourlySolarService
getTimeFormat()
getFileName(stationID)
getFilePath(fileName)
getFileFTPPath(stationID)
createParameter(cols, date, nearestStation, coordinate)
Powered by yFiles

Figure 5.14: Class hierarchy of the service classes, with a selection of three out of eight subclasses.

5.6 Converting Units

Finally, to add the ability to convert between different units, Convertor[10] was chosen, because of its easy to use approach. To convert a value from a specific unit to another unit, the user only has to instantiate a new `Convertor` object that takes the value and the unit as a string. After that, the user can either call the `to($unit)` method or the `toAll()` function to convert from the base unit to either a specific other one or to all other available units.

Adaptions

Unfortunately, the library at first did not support composer to be able to easily re-use the package in other projects. This is the first minor adjustment to the library.

In addition, the library did not have custom `Exceptions` but just threw the one generic `\Exception` provided by PHP for all possible causes. This was changed to be able to differentiate between different types of errors that can occur. In turn, the library now has different exceptions for various scenarios (e.g., when trying to convert from meters to hours, a `ConvertorDifferentTypeException` is thrown).

Additionally, the unit conversions were hard-coded into the single library file. This was fixed by allowing users to choose between different inbuilt conversions or to let

[10]Available at github: `https://github.com/olifolkerd/convertor`, accessed: 2017-11-20

them provide their own conversions in an array. This was necessary due to different naming conventions of units, as the provided conversions used `km2` for square kilometres, whereas the ECMWF used the `km**2` notation.

Finally, the newly added data set also provides conversions regarding area density. This allows the user to convert from kilograms per square meter to various other formats.

6 Summary

This chapter will first provide insights on several statistics regarding the API, afterwards, it will provide an overview of available features and finally the current limitations.

6.1 The API in Numbers

Taking a look back at the scenario described in Section 3.1, it is possible to determine the amount of data that needs to be stored, as well as the number of requests the API should be able to handle.

Imagine a new study branch for *Track your Tinnitus* that runs over a timespan of 6 months (183 days) for all participants, with N=300 participants. In addition to that, each participant will most likely provide data once per day. Consequently, this results in about 300 requests per day. Each of those will, in turn, be used to retrieve environmental data after waiting for the maximum required retrieval delay (cf. Section 5.3.7). Consequently, each of those requests will be used to fetch 56 single environmental parameters (19 from the DWD hourly and 37 from the CAMS set). Accordingly, the API has to store 16.800 parameters daily, resulting in 3.074.400 parameters for the entire duration of the study.

Looking at this scenario from a data storage point of view, the needed space differs depending on the data source. Currently, the daily retrieved CAMS data provides all weather data for Europe in the form of a ~134MB binary file. In comparison, the storage costs for the DWD data varies depending on the participant's locations. Should all participants be near a minimum amount of stations, the storage costs are significantly less compared to a more likely scenario where almost all the stations have to be queried. In addition, instead of querying all the data, the DWD provides parameter groups which have to be queried independently. In turn, one of those groups on average is about 65KB compressed (~650KB uncompressed). This results in about 520KB of compressed data per day for all parameter groups for one station. At the time of writing, about 506 stations were marked as active by the DWD. This results in a worst-case scenario of ~257MB of data that would need to be downloaded, should each station be queried once every day.

Consequently, over the duration of the study, this results in about ~24.5GB of ECMWF raw binary data. The required space for the DWD is significantly lower, as each station's data file contains data for multiple days. So, while the data has to be transferred several times, storage does not increase significantly over time and is mostly dependent on the number of queried stations.

In addition, all raw data is transformed and stored in a separate table in the database of the API. When looking at the current table design we can extract the size of the table at various amounts of stored data (cf. Listing A.12 for the query). Table 6.1 shows the status of the database table after retrieving data for 1, 21 and 42 different `GeoLocations`. Note, this test was done by retrieving data for the same latitude, longitude and timestamp, but without having compression enabled in the MySQL database.

GeoLocation	Parameters	Data Size	Index Size	Table Size
1	56	48.00 KB	32.00 KB	80.00 KB
21	1176	464.00 KB	32.00 KB	496.00 KB
42	2352	1.52 MB	128.00 KB	1.64 MB

Table 6.1: Size of the MySQL database table with various amounts of stored parameters.

6.2 Current Status

The developed environmental API allows anyone who is in the process of conducting a study to additionally collect environmental data. The data itself is retrieved with the help of additional libraries and is retrieved from two completely different data sets that are combined inside the API. One of those data sources is the DWD hourly data set that collects data from a multitude of weather stations, while the other one uses the Copernicus satellites to collect and aggregate data. Libraries for both aforementioned data sets have been implemented in the scope of this thesis in order to collect and parse the retrieved data.

Environmental API

In total, the API retrieves 56 environmental parameters 19 of them belong to the DWD hourly data set and the rest comes from the CAMS Near-real-time data set by processing stored `GeoLocation` objects. In addition, adding new data sources is easily manageable by extending the provided `AbstractDataRetrievalGateway`. In turn, the retrieval process is executed using a queue to allow asynchronous retrieval of the environmental data. The retrieved data is then transformed into the expected format and stored afterwards.

In turn, the API also supports having multiple studies on the platform at the same time by using the inbuilt user to manage projects. One project is mapped to exactly one user-entity and can, in turn, only access data from their own participants.

The querying process of the API offers various filtering options to the data and provides a way to convert between different units. Data can be filtered by *participant ids*, the *data source* string, the *parameter type* or a *date* range by specifying *from* and *until*.

Retrieval Libraries

Both libraries provide a simple way to retrieve data from their respective datasets and have been developed to be integrated into the API. The *DWD Hourly Crawler* will access the FTP server to retrieve the files from the hourly data set. The library is customizable and also provides a way to modify it should parameters or path to the files change. The output is configurable by the developer by either using the supplied transformers or creating their own transformers.

In comparison, the *Copernicus Retrieval* wrapper wraps the supplied python libraries to provide an easy-to-use alternative to using a website to get a valid download request.

In addition, it allows the extraction of specific parameters from the retrieved binary file. Consequently, as the environmental API is written in PHP, it was necessary to provide the wrapper functions by building a small microservice (*CopernicusAPI*; cf. Section 5.4.2) around it. This microservice allows the user to retrieve and parse files for a requested timestamp. In addition, the format of the returned data can be modified by changing the schemas in the code.

6.3 Future Work

Several things are still on the list to either improve the performance or provide new features for both the API and retrieval libraries.

Environmental API
First, instead of providing a dedicated user and role management itself, this API relies heavily on the abilities of the existing study platform (e.g. Track Your Tinnitus). In future, it should be an option to let this API handle user and role management. Due to the used framework, this should not prove too difficult but leads to duplicate data between this API and the existing study platform. Should this be implemented, it might also be necessary to add a *study group* entity to be able to re-use existing configurations of researchers and participants.

In the future, it might also be good to re-inspect the provided database templates. Currently, many of the `varchars` are initialized to the default values and not to an optimized length. As are most other fields, to be as non-restrictive as possible. In the same vein, it might be worth it to let the application also handle database backups for retrieved environmental data. The API could also schedule regular backups of the database and facilitate restoring data. This would help the migration from one to another platform and also help to keep the requests safe. Alternatively, the study platform that is in place could back up all requests to the environmental API, as retrieval of environmental data is possible multiple times.

In addition, Apiato is not the fastest framework. Currently, a request directed to an inbuilt route takes about 500-700ms. While this is fine for the use-case, it could be optimized in the future to enhance the user experience by a huge margin. For a comparison, the minimal microservice that was written in Flask, has response times between 50-400ms, depending on the requested data without using the inbuilt caching.

Additionally, new `Gateways` may be made available to retrieve data from other data sources. It might also be necessary to provide more documentation regarding the implementation of such gateways and the expected input and output formats in order to make sure they are as extendible as planned.

Retrieval Libraries

In addition to the above optimizations, there are several improvements for both libraries used to retrieve data. The *DWD Hourly Crawler* does take about 4 seconds to parse already downloaded files. One improvement for this might be to extract the files locally, instead of accessing the content of the zip files in memory. In addition, it might be an option to store the retrieved data in a database instead of storing the files themselves to further optimize the retrieval process. This would probably reduce the processing time per parameter by a noticeable amount but would result in higher data-overhead by introducing another database.

Almost the same approach is applicable to the *Copernicus Retrieval* wrapper. However, instead of storing the files locally, it might be worth it to parse the binary file once and store the content in a database instead of having to access the file every time it is requested. In addition, the wrapper can be enhanced by adding more data sets and available parameters to query. Currently, it is also possible to request parameters that are only available in forecasts when querying analysis data. In future, it would be a preferable option to strictly divide the enumerations (enums) to avoid confusing developers that want to use the wrapper.

Finally, the `CopernicusAPI` can be optimized by modifying the implemented caching approach, adding authentication and allowing the developer to modify the requested data schema via supplied parameters instead of having to adapt the code of the microservice itself. Another addition would be supporting the JSON-API format to be consistent with the environmental API, instead of using its own format.

Additionally, providing more methods to request and retrieve data could be implemented. One of those options could be seen as a bulk retrieval mode that takes large amounts of latitude, longitude and timestamp tuples and returns the requested environmental data.

Currently, due to the scope of this thesis, this API also does not support switching between analysis and forecasts, which could be needed when having multiple surveys running that use different datasets.

6.4 Conclusion

The goal of building a ready to use API, that handles retrieval of environmental parameters, while being easy to embed into existing study platforms, has been met. To achieve this goal, two supporting stand-alone data retrieval libraries have been developed that handle collecting of environmental data from data different data sets. Currently, both the hourly data set provided by the DWD and the CAMS dataset from the ECMWF are being supported by the API. The collected data is then stored in one common database after being transformed into the required data format.

One of the main benefits of the current implementation is, that it has no pre-defined time-scale for user entries. The granularity of the provided data is determined by the chosen and available data sources. Currently, this is either hourly when looking at the DWD data or every six hours for the CAMS set. In the future, when more fine-grained data is available it is possible to add this new set to the API. Additionally, the current solution is not locked to the given data sources. Currently one of the sources provides data for Germany while the other has the option to retrieve data worldwide. As of now, data is retrieved from inside of Europe only, but sources for the US or other countries can be added easily by extending the provided template for new data sources.

In turn, researchers are then able to query, filter and convert this data to support them in finding links between environmental factors and diseases. This allows the researchers to obtain data for specific participants, periods of time and parameter types. Additionally, data can be normalized in a way, so that the resulting data is converted into one common unit instead of having to do this later on.

One of the next steps to advance the work started in this thesis would be to use the API for a trial run. This includes gathering feedback from both administrators and researchers on regarding the usability and provided functionality. Gathered feedback could then be used to adapt specific parts of the API that requires more fine-tuning or even remodelling. Most probably, this will also show bottlenecks and limitations of the current design when working with a bigger number of participants compared to the imagined scenario (cf. Section 6.1).

In addition, it might be helpful to provide a fork of this project that does not depend on functionality from the existing study platform. As such, the fork would have to be extended to provide user and role management but also a way to differentiate concurrently running studies on the API.

Another addition would be providing more gateways to different data sets from various sources. The used structure to integrate new gateways does facilitate this, but learning about the data sets access, possibilities and limitations will take some time.

In summary, there are many points that can be adjusted and enhanced in the resulting API. However, the current implementation allows studies to incorporate environmental data by integrating the provided API into their existing study platform. In turn, this allows researchers to possibly find links between specific environmental factors and diseases. Which, in turn, might lead to a deeper understanding of the factors that may increase or decrease the perceived severity of the disease. In the future, when links have been found, it may also be an option to include weather forecast data as weather sources. The data could then be used to provide users with predictions and helpful tips.

Bibliography

[1] M. S. Shutty, G. Cundiff, and D. E. DeGood, “Pain complaint and the weather: weather sensitivity and symptom complaints in chronic pain patients,” *Pain*, vol. 49, no. 2, pp. 199–204, May 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439599290143Y [Accessed: 2017-07-18]

[2] K. J. Hagglund, W. E. Deuser, S. P. Buckelew, J. Hewett, and D. R. Kay, “Weather, beliefs about weather, and disease severity among patients with fibromyalgia,” *Arthritis & Rheumatism*, vol. 7, no. 3, pp. 130–135, Sep. 1994. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/art.1790070306/abstract [Accessed: 2017-12-07]

[3] L. Robbins, “Precipitating Factors in Migraine: A Retrospective Review of 494 Patients,” *Headache: The Journal of Head and Face Pain*, vol. 34, no. 4, pp. 214–216, Apr. 1994. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1111/j.1526-4610.1994.hed3404214.x/abstract [Accessed: 2017-07-18]

[4] H. Chabriat, J. Danchot, P. Michel, J. E. Joire, and P. Henry, “Precipitating factors of headache. A prospective study in a national control-matched survey in migraineurs and nonmigraineurs,” *Headache: The journal of head and face pain*, vol. 39, no. 5, pp. 335–338, May 1999.

[5] P. Höppe, S. v. Mackensen, D. Nowak, and E. Piel, “Prävalenz von Wetterfühligkeit in Deutschland [Prevalence of weather sensitivity in Germany],” *DMW - Deutsche Medizinische Wochenschrift*, vol. 127, no. 01/02, pp. 15–20, Jan. 2002. [Online]. Available: http://www.thieme-connect.de/DOI/DOI?10.1055/s-2002-19429 [Accessed: 2017-12-08]

[6] D. Guedj and A. Weinberger, “Effect of weather conditions on rheumatic patients.” *Annals of the Rheumatic Diseases*, vol. 49, no. 3, pp. 158–159, Mar. 1990. [Online]. Available: http://ard.bmj.com/content/49/3/158 [Accessed: 2017-12-06]

[7] A. A. Gorin, J. M. Smyth, J. N. Weisberg, G. Affleck, H. Tennen, S. Urrows, and A. A. Stone, “Rheumatoid arthritis patients show weather sensitivity in daily life, but the relationship is not clinically significant,” *Pain*, vol. 81, no. 1, pp. 173–177, May 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030439599900010X [Accessed: 2017-07-20]

[8] H. Walach, A. Schweickhardt, and K. Bucher, “Hat das Wetter Einfluss auf Kopfschmerzen? Does weather modify headaches? An empirical evaluation of bio-weather categorization,” *Der Schmerz*, vol. 16, no. 1, pp. 1–8, Feb. 2002. [Online]. Available: https://link.springer.com/article/10.1007/s004820100066 [Accessed: 2017-12-06]

[9] J. Vergés, E. Montell, E. Tomàs, G. Cumelles, G. Castañeda, N. Marti, and I. Möller, “Weather conditions can influence rheumatic diseases,” *Proceedings of the Western Pharmacology Society*, vol. 47, pp. 134–136, 2004.

[10] P. J. Villeneuve, M. Szyszkowicz, D. Stieb, and D. A. Bourque, “Weather and Emergency Room Visits for Migraine Headaches in Ottawa, Canada,” *Headache: The Journal of Head and Face Pain*, vol. 46, no. 1, pp. 64–72, Jan. 2006. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1111/j.1526-4610.2006.00322.x/abstract [Accessed: 2017-07-18]

[11] M. J. H. Huibers, L. E. de Graaf, F. P. M. L. Peeters, and A. Arntz, “Does the weather make us sad? Meteorological determinants of mood and depression in the general population,” *Psychiatry Research*, vol. 180, no. 2, pp. 143–146, Dec. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165178109003618 [Accessed: 2017-07-18]

[12] E. H. Bos, R. Hoenders, and P. de Jonge, “Wind direction and mental health: a time-series analysis of weather influences in a patient with anxiety disorder,” *BMJ Case Reports*, vol. 2012, Jun. 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4543179/ [Accessed: 2017-12-07]

[13] K. D. Ricketts, A. Charlett, D. Gelb, C. Lane, J. V. Lee, and C. A. Joseph, “Weather patterns and Legionnaires’ disease: a meteorological study,” *Epidemiology & Infection*, vol. 137, no. 7, pp. 1003–1012, Jul. 2009. [Online]. Available: https://www.cambridge.org/core/journals/epidemiology-and-infection/article/weather-patterns-and-legionnaires-disease-a-meteorological-study/30898B91EAD54C578927BA0AA7082078 [Accessed: 2017-12-06]

[14] W. Schmidt, C. Sarran, N. Ronan, G. Barrett, D. J. Whinney, L. E. Fleming, N. J. Osborne, and J. Tyrrell, “The Weather and Ménière’s Disease: A Longitudinal Analysis in the UK,” *Otology & neurotology: official publication of the American Otological Society, American Neurotology Society [and] European Academy of Otology and Neurotology*, vol. 38, no. 2, pp. 225–233, Feb. 2017. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5224697/ [Accessed: 2017-11-29]

[15] L. H. Tecer, O. Alagha, F. Karaca, G. Tuncel, and N. Eldes, “Particulate Matter (PM2.5, PM10-2.5, and PM10) and Children’s Hospital Admissions for Asthma and Respiratory Diseases: A Bidirectional Case-Crossover Study,” *Journal of Toxicology and Environmental Health, Part A*, vol. 71, no. 8, pp. 512–520, Mar. 2008. [Online]. Available: https://doi.org/10.1080/15287390801907459 [Accessed: 2017-07-20]

[16] W. Becker, “Weather and migraine: Can so many patients be wrong?” *Cephalalgia*, vol. 31, no. 4, pp. 387–390, Mar. 2011. [Online]. Available: https://doi.org/10.1177/0333102410385583 [Accessed: 2017-11-29]

[17] European Centre for Environment and Human Health University of Exeter Medical School and Royal Cornwall Hospital Truro, “Caveats for interpreting data.” [Online]. Available: https://www.data-mashup.org.uk/data/interpretation-of-data/ [Accessed: 2017-12-07]

[18] B. Langguth, R. Pryss, T. Probst, and W. Schlee, “Emotion dynamics and tinnitus: Daily life data from the “TrackYourTinnitus” application,” *Scientific Reports*, vol. 6, p. srep31166, Aug. 2016. [Online]. Available: https://www.nature.com/articles/srep31166 [Accessed: 2017-11-22]

[19] K. Zebenholzer, E. Rudel, S. Frantal, W. Brannath, K. Schmidt, Ç. Wöber-Bingöl, and C. Wöber, "Migraine and weather: A prospective diary-based analysis," *Cephalalgia*, vol. 31, no. 4, pp. 391–400, Mar. 2011. [Online]. Available: https://doi.org/10.1177/0333102410385580 [Accessed: 2017-07-20]

[20] DWD, "Wetter und Klima - Deutscher Wetterdienst - Sphere of Tasks." [Online]. Available: https://www.dwd.de/EN/aboutus/tasks/task_node.html [Accessed: 2017-10-23]

[21] "Erstes Gesetz zur änderung des Gesetzes über den Deutschen Wetterdienst," *Bundesgesetzblatt Teil I*, no. 49, p. 2642, Jul. 2017. [Online]. Available: http://www.bgbl.de/xaver/bgbl/start.xav?startbk=Bundesanzeiger_BGBl&jumpTo=bgbl117s2642.pdf [Accessed: 2017-10-23]

[22] Bundestag, "Verordnung zur Festlegung der Nutzungsbestimmungen für die Bereitstellung von Geodaten des Bundes vom 19. März 2013 (BGBl. I S. 547)." Mar. 2013. [Online]. Available: http://www.gesetze-im-internet.de/geonutzv/BJNR054700013.html [Accessed: 2017-07-25]

[23] Deutscher Wetterdienst, "Datensatzbeschreibung - Historische stündliche Stationsmessungen der Wolkenabdeckung für Deutschland," Jun. 2017. [Online]. Available: ftp://ftp-cdc.dwd.de/pub/CDC/observations_germany/climate/hourly/cloudiness/historical/BESCHREIBUNG_test_obsgermany_climate_hourly_cloudiness_historical_de.pdf [Accessed: 2017-06-22]

[24] DWD, "Data Set Description Air Temperature." [Online]. Available: ftp://ftp-cdc.dwd.de/pub/CDC/observations_germany/climate/hourly/air_temperature/recent/DESCRIPTION_obsgermany_climate_hourly_tu_recent_en.pdf [Accessed: 2017-10-16]

[25] ——, "Data Set Description Precipiation." [Online]. Available: ftp://ftp-cdc.dwd.de/pub/CDC/observations_germany/climate/hourly/precipitation/recent/DESCRIPTION_obsgermany_climate_hourly_precipitation_recent_en.pdf [Accessed: 2017-10-16]

[26] ECMWF, “Who we are | ECMWF.” [Online]. Available: https://www.ecmwf.int/en/about/who-we-are [Accessed: 2017-10-21]

[27] Council of European Union, “Council regulation ({EU}) no 1159/2013,” 2013. [Online]. Available: http://www.copernicus.eu/sites/default/files/library/Commission_Delegated_Regulation_1159_2013.pdf [Accessed: 2017-10-20]

[28] ECMWF, “Copernicus Atmosphere Monitoring Service | ECMWF.” [Online]. Available: https://www.ecmwf.int/en/about/what-we-do/copernicus/copernicus-atmosphere-monitoring-service [Accessed: 2017-07-25]

[29] Copernicus Website, “Copernicus Atmosphere Monitoring Service,” Dec. 2014. [Online]. Available: http://copernicus.eu/main/atmosphere-monitoring [Accessed: 2017-10-21]

[30] ECMWF, “What data is available through CAMS (Copernicus Atmosphere Monitoring Service)? - Copernicus Knowledge Base - ECMWF Confluence Wiki.” [Online]. Available: https://software.ecmwf.int/wiki/pages/viewpage.action?pageId=56659592 [Accessed: 2017-10-21]

[31] ——, “Climate reanalysis | ECMWF.” [Online]. Available: https://www.ecmwf.int/en/research/climate-reanalysis [Accessed: 2017-10-22]

[32] ——, “ERA-Interim | ECMWF.” [Online]. Available: https://www.ecmwf.int/en/research/climate-reanalysis/era-interim [Accessed: 2017-10-22]

[33] ——, “Use of data from this server - License Agreement.” [Online]. Available: http://apps.ecmwf.int/datasets/licences/copernicus/ [Accessed: 2017-10-20]

[34] ——, “ERA5 data documentation.” [Online]. Available: https://software.ecmwf.int/wiki/display/CKB/ERA5+data+documentation [Accessed: 2017-07-25]

[35] ——, “Web-API Downloads - ECMWF Web API.” [Online]. Available: https://software.ecmwf.int/wiki/display/WEBAPI/Web-API+Downloads [Accessed: 2017-10-21]

[36] W. World Meteorological Organization, *Manual on Codes - International Codes, Volume I.1, Annex II to the WMO Technical Regulations: part A- Alphanumeric Codes*, 2011th ed., ser. WMO. WMO, 2016.

[37] ECMWF, “ecCodes Home.” [Online]. Available: https://software.ecmwf.int/wiki/display/ECC/ecCodes+Home [Accessed: 2017-10-23]

[38] Yahoo, “Yahoo Weather API.” [Online]. Available: https://developer.yahoo.com/weather/ [Accessed: 2017-10-23]

[39] OpenWeatherMap, “Weather API - OpenWeatherMap.” [Online]. Available: https://openweathermap.org/api [Accessed: 2017-10-23]

[40] “Open Data Commons Open Database License (ODbL),” Feb. 2009. [Online]. Available: https://opendatacommons.org/licenses/odbl/ [Accessed: 2017-10-23]

[41] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf [Accessed: 2017-10-30]

[42] P. Sturgeon, *Build APIs You Won’t Hate: Everyone and their dog wants an API, so you should probably learn how to build them*, 1st ed. S.l.: Philip J. Sturgeon, Aug. 2015.

[43] M. Zalt, “Porto is a Modern Software Architectural Pattern,” Oct. 2017, original-date: 2016-02-29T04:31:00Z. [Online]. Available: https://github.com/Mahmoudz/Porto [Accessed: 2017-10-30]

[44] ——, “Welcome to Apiato.” [Online]. Available: https://github.com/apiato/apiato/ [Accessed: 2017-11-02]

[45] Laravel Documentation, “Queues - Laravel - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/5.5/queues#class-structure [Accessed: 2017-11-12]

[46] ——, “Task Scheduling - Laravel - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/5.5/scheduling [Accessed: 2017-11-12]

[47] M. Zalt, “Magical Call.” [Online]. Available: https://github.com/apiato/apiato/miscellaneous/magical-call/ [Accessed: 2017-11-13]

[48] O. Folkerd, “Convertor.” [Online]. Available: http://olifolkerd.github.io/convertor/ [Accessed: 2017-11-13]

A
Code

A.1 Filtered Query to the WeatherData Endpoint

```
{
    "data": [
        {
            "type": "WeatherData",
            "id": "4a670vyzzwm3wngd",
            "attributes": {
                "value": "9.700000",
                "type": "2 metre temperature",
                "unit": "C"
            },
            "relationships": {
                "source": {
                    "data": {
                        "type": "WeatherSource",
                        "id": "zwn0ydvezv4kg9jx"
                    }
                }
            }
        },
    ],
    "included": [
        {
            "type": "WeatherSource",
            "id": "zwn0ydvezv4kg9jx",
            "attributes": {
                "data": {
                    "id": "03402"
                }
            }
        }
    ]
}
```

Listing A.1: Response to a query pertaining the WeatherData filtered to only show the id, type, unit and value.

A.2 Conversion

```
private function convert(&$weatherData, &$converted, $targetUnit, $values = [])
{
    //Convert single value
    if (get_class($weatherData) == WeatherData::class) {
        $this->convertWeatherData($weatherData, $targetUnit);
        return;
    }
    if (is_iterable($weatherData)) {
        foreach ($weatherData as $item) {
            try {
                //figure out if we can skip the conversion
                if ((count($values) > 0 && !in_array($item->type, $values))
                 ↪  //searching for specific values: check if values are set,
                 ↪  then check if its not in the searched values
                    || in_array($item->id, $converted) || (strtoupper($item->unit)
                     ↪  == strtoupper($targetUnit))) // already converted OR unit
                     ↪  is already the same.
                    continue;
                $this->convertWeatherData($item, $targetUnit);
            } catch (ConvertorInvalidUnitException |
             ↪  ConvertorDifferentTypeException | ConvertorException $exception)
             ↪  {
                continue;
            }
            $converted[] = $item->id;
        }
        return;
    }
    //If the weatherData content is neither of type WeatherData::class nor an
     ↪  iterable, this conversion fails.
    throw new \InvalidArgumentException("...")); //shortened
}
```

Listing A.2: The function checks whether it gets a single value or an iterable. Depending on the received type it either applies the conversion directly or iterates through all items. Afterwards, the conversion is applied. Should the conversion fail for a single item it will continue with the next one.

A.3 Converting Weather Data

The function shown below takes one instance of a `WeatherData` model as a reference and the target unit and applies the conversion by using the Convertor library.

```
    public function convertWeatherData(&$item, $targetUnit)
    {
        $conv = new Convertor($item->value, strtolower($item->unit));
        $item->values = $conv->to(strtolower($targetUnit));
        $item->unit = strtoupper($targetUnit);
    }
```

Listing A.3: Response to a query pertaining the WeatherData filtered to only show the id, type, unit and value.

A.4 Customized Fetching of Environmental Data

Example 1 Obtain *wind* and *temperature* data from one specific participant. Then convert everything to ℃ and only output the *id*, *value* and *unit*.

```
GET api.apiato.dev/v1/weatherData?type=*wind*;*temperature*&participantId=test
&convert=*:C&filter=id;value;unit;type HTTP/1.1
```

```
{
    "data": [
        {
            "type": "weatherdatas",
            "id": "4a670vyzzwm3wngd",
            "attributes": {
                "value": "9.700000",
                "type": "2 metre temperature",
                "unit": "C"
            },
            "relationships": {
                "source": {
                    "data": {
                        "type": "weathersources",
                        "id": "zwn0ydvezv4kg9jx"
                    }
                }
            }
        },
        ...
    ],
    "included": [
        {
            "type": "weathersources",
            "id": "zwn0ydvezv4kg9jx",
            "attributes": {
                "data": {
                    "data": {
                        "id": "03402"
                    }
                }
            }
        }
    ],
    "meta": {
        "include": [
            "participant"
        ]
    }
}
```

Example 2 Obtain all values that contain *pressure* or end with *temperature* with a fixed participant and convert everything to *bar* if possible.

```
GET api.apiato.dev/v1/weatherData?type=*pressure*;*temperature&participantId=test
&convert=*:BAR HTTP/1.1
```

```
{
    "data": [
        {
            "type": "weatherdatas",
            "id": "qdgw4xv3gomr7a63",
            "attributes": {
                "object": "WeatherData",
                "created_at": {
                    "date": "2017-10-05 14:50:13.000000",
                    ...
                },
                "updated_at": { ... },
                "source": "DWD",
                "value": "1025.300000",
                "description": { ... },
                "classification": "Atmosphere",
                "distance": "24.901572",
                "lat": "48.441800",
                "lon": "9.921600",
                "date": { ... },
                "type": "mean sea level pressure",
                "geo_location_id": "ze6bqg8wlv3r7pan",
                "unit": "BAR",
            },
            "relationships": {
                "source": {
                    "data": {
                        "type": "weathersources",
                        "id": "zx6ow3v798qdg975"
                    }
                }
            }
        },
    ]
}
```

A.5 Copernicus Output Example

```
{
  "TWO_METRE_TEMPERATURE": [
    {
      "index": 45048,
      "description": {
        "dataTime": 0,
        "name": "2 metre temperature",
        "date": 20170816,
        "step": 0,
        "units": "K",
        "shortName": "2t",
        "paramId": 167,
        "convertedUnit": "C"
      },
      "classification": null,
      "distance": 16.327599462504207,
      "longitude": 10.0,
      "date": "2017-08-16T00:00:00+00:00",
      "value": 290.118896484375,
      "latitude": 48.29265233053008,
      "type": "2 metre temperature"
    }
  ]
}
```

Listing A.8: Example JSON output for parsing a grib file that contains the `2 metre temperature` parameter.

```
{
        'TWO_METRE_TEMPERATURE': [
        CopernicusData: {
                'index': 45048,
                        'description': {
                                'dataTime': 0,
                                 'name': '2 metre temperature',
                                 'date': 20170901,
                                 'step': 0,
                                 'units': 'K',
                                 'shortName': '2t',
                                 'paramId': 167,
                                 'convertedUnit': 'C'
                        },
                        'classification': 'Temperature',
                        'distance': 16.327599462504207,
                        'longitude': 10.0,
                        'date': datetime.datetime(2017, 9, 1, 0, 0, tzinfo=<UTC>),
                        'value': 285.4701843261719,
                        'latitude': 48.29265233053008,
                        'type': '2 metre temperature'
                }
        ]
}
```

Listing A.9: Example internal python representation after parsing a grib file that contains the `2 metre temperature` parameter.

A.6 DWD Hourly Crawler Output Example

```
array (size=2)
  'values' =>
    array (size=7)
      'air_temperature' =>
        array (size=1)
          0 =>
            object(FWidm\DWDHourlyCrawler\Model\DWDAirTemperature)[43]
              ...
  'stations' =>
    array (size=3)
      'station-02074' =>
        object(FWidm\DWDHourlyCrawler\Model\DWDStation)[625]
          private 'id' => string '02074' (length=5)
          private 'from' =>
            object(Carbon\Carbon)[624]
              ...
          private 'until' =>
            object(Carbon\Carbon)[623]
              ...
          private 'height' => string '522' (length=3)
          private 'latitude' => string '48.3751' (length=7)
          private 'longitude' => string '8.9801' (length=6)
          private 'name' => string 'Hechingen' (length=9)
          private 'state' => string 'Baden-Wuerttemberg' (length=18)
          private 'active' => boolean true
    ...
```

Listing A.10: Output of the crawler after finding data for a specific location and date.

A.7 Marshmallow Schema Definition

```python
from marshmallow import Schema, fields


class CopernicusDataSchema(Schema):
    index = fields.Number()
    type = fields.Str()
    latitude = fields.Number()
    longitude = fields.Number()
    date = fields.DateTime()
    description = fields.Dict()
    distance = fields.Number()
    value = fields.Number()
    classification = fields.Str()
```

Listing A.11: The Schema definition that is applied to all weather data returned by the microservice.

A.8 Querying the Table Size of the Environmental Data Table

```sql
SELECT
    CONCAT(FORMAT(DAT/POWER(1024,pw1),2),' ',SUBSTR(units,pw1*2+1,2)) DATSIZE,
    CONCAT(FORMAT(NDX/POWER(1024,pw2),2),' ',SUBSTR(units,pw2*2+1,2)) NDXSIZE,
    CONCAT(FORMAT(TBL/POWER(1024,pw3),2),' ',SUBSTR(units,pw3*2+1,2)) TBLSIZE
FROM
(
    SELECT DAT,NDX,TBL,IF(px>4,4,px) pw1,IF(py>4,4,py) pw2,IF(pz>4,4,pz) pw3
    FROM
    (
        SELECT data_length DAT,index_length NDX,data_length+index_length TBL,
        FLOOR(LOG(IF(data_length=0,1,data_length))/LOG(1024)) px,
        FLOOR(LOG(IF(index_length=0,1,index_length))/LOG(1024)) py,
        FLOOR(LOG(IF(data_length+index_length=0,1,data_length+index_length))
                /LOG(1024)) pz
        FROM information_schema.tables
        WHERE table_schema='environmental_api'
        AND table_name='environmental_data'
    ) AA
) A,(SELECT 'B KBMBGBTB' units) B;
```

Listing A.12: Query that retrieves the size of the data, index and the complete `environmental_data` table in the `environmental_api` DB. Source: `https://stackoverflow.com/q/6474591`, accessed: 2017-11-03

B Additional Information

B.1 Available Hourly Parameters from the DWD Server

- **Air temperature** - Contains two measured values:
  - **TT_TU**: air temperature 2m from the ground in degrees Celsius (°C).
  - **RF_TU**: relative humidity as percentage (%)
- **Cloudiness** - Contains two measured values:
  - **V_N_I**: index to differentiate measurements done by human observation or a device.
  - **V_N**: vloudiness differentiation is described by eights - and lies between 0 and 8. Which includes *-1* as the value if it is not recognisable.
- **Precipitation** - Contains three measured values:
  - **R1**: hourly precipitation in millimetres (mm).
  - **RS_IND**: index of precipitation states whether there was or was no precipitation (bool, 0 or 1).
  - **WRTR**: form of precipitation.
- **Pressure** - Contains two measured values:
  - **P**: atmospheric pressure at sea level in hector Pascal (hPA).
  - **P0**: atmospheric pressure at station level in hector Pascal (hPA).
- **Soil temperature** - Contains six measured values:

  - **V_TE002**: soil temperature depth at 2 centimetres (cm).
  - **V_TE005**: soil temperature depth at 5 centimetres (cm).
  - **V_TE010**: soil temperature depth at 10 centimetres (cm).
  - **V_TE020**: soil temperature depth at 20 centimetres (cm).
  - **V_TE050**: soil temperature depth at 50 centimetres (cm).
  - **V_TE100**: soil temperature depth at 100 centimetres (cm).
- **Solar** - Contains 4 values but data is about a month old at the time of writing (2017, July).
  - **ATMO_STRAHL**: hourly sum of long-wave downward radiation in J/cm^2.
  - **FD_STRAHL**: hourly sum of diffuse solar radiation in J/cm^2.
  - **FG_STRAHL**: hourly sum of incoming solar radiation in J/cm^2.
  - **SD_STRAHL**: hourly sum of Sunshine duration per hour in minutes.
- **Sun** - Contains one measured value:
  - **SD_SO**: sunshine minutes per hour (min).
- **Wind** - Contains two measured values:
  - **F**: mean wind velocity in metres per second (m/s).
  - **D**: mean wind direction measured in degrees (°).

B.2 Available Conversions in the adapted Convertor units

Table B.1: The following table contains all currently used units inside of the api and marks all units that can be converted using the adapted Convertor Library. More information about the implementation can be found in Section A.2.

Unit	**Supported**
(0 - 1)	
%	
% (n/8 where -1 means error)	
~	
bool (0 no precipitation, 1 precipitation)	
C	x
deg	
hPA	x
integer (0-9)	
K	x
kg m**-2	x
kg s**2 m**-5	
m s**-1	
m**2 s**-2	
min	x
mm	x
Pa	x

B.3 Custom Convertor Units

Table B.2: Additional area density units defined to allow conversion. To use the unit in query urls replace spaces in the unit name with +.

Unit Name	**Unit**
`kg m**-2` (base unit)	$\frac{kg}{m^2}$
`kg km**-2`	$\frac{kg}{km^2}$
`kg cm**-2`	$\frac{kg}{cm^2}$
`kg mm**-2`	$\frac{kg}{mm^2}$
`g m**-2`	$\frac{g}{m^2}$
`mg m**-2`	$\frac{mg}{m^2}$
`st m**-2`	$\frac{st}{m^2}$
`lb m**-2`	$\frac{lb}{m^2}$
`oz m**-2`	$\frac{oz}{m^2}$

B.4 Copernicus Enums

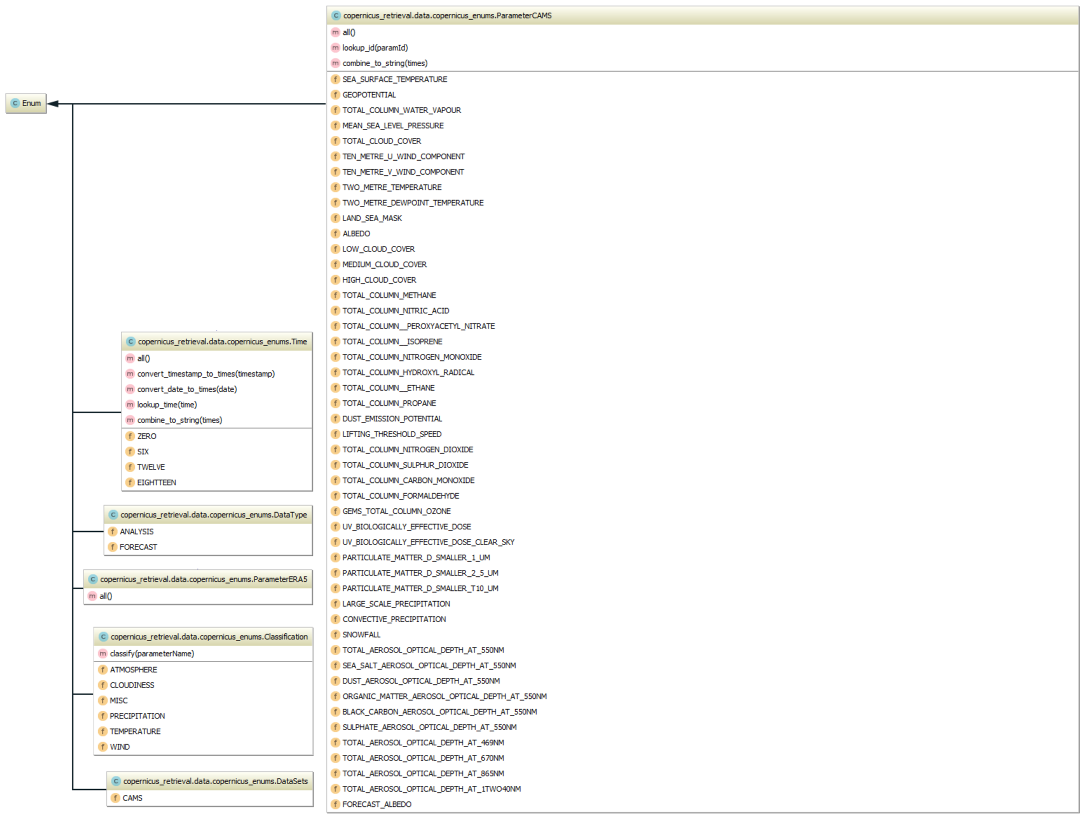

Enum
copernicus_retrieval.data.copernicus_enums.ParameterCAMS
all()
lookup_id(paramId)
combine_to_string(times)
SEA_SURFACE_TEMPERATURE
GEOPOTENTIAL
TOTAL_COLUMN_WATER_VAPOUR
MEAN_SEA_LEVEL_PRESSURE
TOTAL_CLOUD_COVER
TEN_METRE_U_WIND_COMPONENT
TEN_METRE_V_WIND_COMPONENT
TWO_METRE_TEMPERATURE
TWO_METRE_DEWPOINT_TEMPERATURE
LAND_SEA_MASK
ALBEDO
LOW_CLOUD_COVER
MEDIUM_CLOUD_COVER
HIGH_CLOUD_COVER
TOTAL_COLUMN_METHANE
TOTAL_COLUMN_NITRIC_ACID
TOTAL_COLUMN__PEROXYACETYL_NITRATE
TOTAL_COLUMN__ISOPRENE
TOTAL_COLUMN_NITROGEN_MONOXIDE
TOTAL_COLUMN_HYDROXYL_RADICAL
TOTAL_COLUMN__ETHANE
TOTAL_COLUMN_PROPANE
DUST_EMISSION_POTENTIAL
LIFTING_THRESHOLD_SPEED
TOTAL_COLUMN_NITROGEN_DIOXIDE
TOTAL_COLUMN_SULPHUR_DIOXIDE
TOTAL_COLUMN_CARBON_MONOXIDE
TOTAL_COLUMN_FORMALDEHYDE
GEMS_TOTAL_COLUMN_OZONE
UV_BIOLOGICALLY_EFFECTIVE_DOSE
UV_BIOLOGICALLY_EFFECTIVE_DOSE_CLEAR_SKY
PARTICULATE_MATTER_D_SMALLER_1_UM
PARTICULATE_MATTER_D_SMALLER_2_5_UM
PARTICULATE_MATTER_D_SMALLER_T10_UM
LARGE_SCALE_PRECIPITATION
CONVECTIVE_PRECIPITATION
SNOWFALL
TOTAL_AEROSOL_OPTICAL_DEPTH_AT_550NM
SEA_SALT_AEROSOL_OPTICAL_DEPTH_AT_550NM
DUST_AEROSOL_OPTICAL_DEPTH_AT_550NM
ORGANIC_MATTER_AEROSOL_OPTICAL_DEPTH_AT_550NM
BLACK_CARBON_AEROSOL_OPTICAL_DEPTH_AT_550NM
SULPHATE_AEROSOL_OPTICAL_DEPTH_AT_550NM
TOTAL_AEROSOL_OPTICAL_DEPTH_AT_469NM
TOTAL_AEROSOL_OPTICAL_DEPTH_AT_670NM
TOTAL_AEROSOL_OPTICAL_DEPTH_AT_865NM
TOTAL_AEROSOL_OPTICAL_DEPTH_AT_1TWO40NM
FORECAST_ALBEDO
copernicus_retrieval.data.copernicus_enums.Time
all()
convert_timestamp_to_times(timestamp)
convert_date_to_times(date)
lookup_time(time)
combine_to_string(times)
ZERO
SIX
TWELVE
EIGHTTEEN
copernicus_retrieval.data.copernicus_enums.DataType
ANALYSIS
FORECAST
copernicus_retrieval.data.copernicus_enums.ParameterERA5
all()
copernicus_retrieval.data.copernicus_enums.Classification
classify(parameterName)
ATMOSPHERE
CLOUDINESS
MISC
PRECIPITATION
TEMPERATURE
WIND
copernicus_retrieval.data.copernicus_enums.DataSets
CAMS

Figure B.1: All available enums inside of the wrapper including available methods and fields.

List of Figures

List of Tables

Name: Fabian Widmann Matriculation number: 750836

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources and tools used are correctly acknowledged.

Ulm, 30th January 2018: ....

Fabian Widmann