Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

DANIEL VAHLE
MASTER THESIS

DEVELOPMENT OF A MICROSERVICE FOR OPEN WEATHER DATA

Submitted on 10 July 2019

Supervisors:
Prof. Dr. Dirk Riehle, M.B.A.
Andreas Bauer, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 10 July 2019

License

Erlangen, 10 July 2019

Abstract

Since mid 2017 the Deutscher Wetterdienst (DWD) has been obligated to provide its weather data to the public for free. However, anyone who wants to access it needs to click through a file system hoping to still be on the right path to the desired weather data. Due to confusing folder names this turns into a frustrating experience very quickly. The data then comes in various formats, making it hard to automatically process it.

In order to stop this hassle, we present a microservice that adapts weather data from the DWD server to a more user friendly REST interface. This thesis describes the architecture and implementation of the microservice. As a result, users can fetch historical, current and forecast weather data of twenty different weather parameters in an easy to process JSON format.

Contents

Acronyms

DWD Deutscher Wetterdienst

HTTP HyperText Transfer Protocol

FTP File Transfer Protocol

CSV Comma-separated values

API Application Programming Interface

ODS Open Data Service

REST Representational State Transfer

JSON JavaScript Object Notation

GPS Global Positioning System

MOSMIX Model Output Statistics-MIX

CDC Climate Data Center

KISS Keep It Simple Stupid

LRU Least Recently Used

KML Keyhole Markup Language

KMZ Keyhole Markup Language Zipped

XML Extensible Markup Language

URL Uniform Resource Locator

TDD Test-Driven Development

JDK Java Development Kit

1 Introduction

In Germany weather data is collected by the DWD. It is part of the Federal Ministry of Transport and Digital Infrastructure. Its main task is warning against dangers from extreme weather conditions such as storms or black ice. On 25th of july 2017 a change to the Deutsche Wetterdienst law came into effect extending the tasks of the DWD. These include changes to the Geodatenzugangsgesetz obligating the DWD to provide most of its data for free to the public as stated in their press release [DWD17]. This allows anyone to use the data for own purposes. For instance this may be a hobby meteorologist who can now access old weather data and analyse it. Another example may also be an entrepreneur who uses solar data of previous years to convince customers to invest in a solar roof.

While there are many ways to use the data, actually getting it is not as easy as it should be. The DWD endeavors to give easy access to their data but none of their access points is suitable for automated data processing. There are two access points. One is the database reachable at `https://opendata.dwd.de/`. It can be accessed via HTTP and also links to the old FTP server. Both servers deliver equal data. Data is organized in several directories and stored in various file formats. Most of them follow standardized data schemes, while others contain tables in CSV format. Some of the files are compressed and multiple compression methods are used throughout the data sets. The content and format of each file is documented. Accessing a specific weather information from this database is everything but intuitive. Therefore, the DWD has started developing the Climate Data Center portal[1]. It allows the user to easily access and visualize parts of the data through an interactive website. However, at time of this writing not all data is accessible through this interface. For example there is a directory with solar data in the database that contains information like the sunshine duration in a ten minutes measuring interval of today. The portal does not offer any way to access data that was measured in the ten minutes interval at all. Even worse the only accessible sunshine duration data has been measured between 1961 and 1990, although there even is data from today.

[1] `https://cdc.dwd.de/portal/`

So even though the data is provided for free, interested people have the choice between finding, downloading, unpacking and interpreting the data from the data base and a graphical interface that only provides a fraction of the data. This may be good enough for some hobby meteorologist but certainly frustrating to anyone who wants to make a business from it like the solar panel selling entrepreneur. Since there is no Application Programming Interface (API), anyone who wants to do data processing on the data needs to go through the same steps to get the data off the server into a usable format.

This master thesis aims to solve this hassle. A new microservice is developed that offers a simple to use API to the open weather data, that the DWD is lacking of. Additionally, an adapter for the Open Data Service (ODS)[2] is developed, thereby extending it by a new source of data. The ODS is an open source software project developed here at the Professorship for Open-Source-Software. It collects heterogeneous data from various sources and makes this data easy to consume through a unified programmatic interface. Moreover, the ODS can improve the quality and availability of the data, and apply different operations to enhance the data. Users will have the choice between directly accessing the microservice API and using the ODS.

Similar efforts have been made by Borges et al. [BPP19] during time of this writing. Their OpenSense.network aims to provide environmental sensor data via a graphical interface and an API for developers. The weather data provided by the DWD serves as one of their first data sources. However, their focus is on providing raw sensor data for further processing. In consequence they only provide actually measured data up until yesterday. We on the other hand want to provide the weather information for a location to the user. This more specific goal allows a simpler output format, which in turn is also easier to process for the user. Furthermore we aim to provide forecast data as well and seamlessly connect it with the weather of today and the past.

Another similar project is the OpenWeatherMap[3]. They do offer free weather data for current weather and forecasts of the next four days. Access to historical data and longer term forecasts need to be paid though. This pretty much reflects the current situation for German weather data. But it also contradicts the idea of open data. The DWD provides all necessary data for free, so why should anyone have to pay for it just to be able to access the data through a proper API? Many other countries, especially America, are way ahead of Germany in this territory. Their data has been easily accessible from the National Weather Service[4] through a simple REST API since years.

[2] `https://github.com/jvalue/open-data-service`

[3] `https://openweathermap.org`

[4] `https://www.weather.gov/documentation/services-web-api`

The next chapters describe the development of the microservice in detail. It starts with a requirements specification that defines all functional and non functional requirements to the artifact. Next the software architecture is designed and different design choices are discussed. The Implementation chapter explains the algorithms and data structures that are used. It also discusses pros and cons compared to alternative solutions. Last the resulting artifact is evaluated with the evaluation scheme that was defined during the requirements specification. Additionally, we give an idea of how well the service performs based on four typical use cases.

2 Definition of requirements

2.1 Functional requirements

F1: Fetch DWD weather data

The microservice needs to be able to fetch current temperature and solar radiation data provided by DWD on their server[1] for a given location. Optionally forecast and historical data for a given location as well as other weather parameters may be fetched, too.

Additionally, a request that requires only a certain part of the data should not require downloading all the available data but only those files that contain related data. In order to evaluate this requirement, the microservice is supposed to log every file name that it downloads. This way one can check whether a request leads to a download of required files only.

F2: Transform DWD data to own weather model

The DWD data is organized in a way that measured data can be uploaded to the server very easily and is human readable. It is not a suitable data format to query weather information efficiently. To perform operations on the weather data with low complexity and decent performance, the data needs to be abstracted to a more suitable and consistent data format. Every measurement needs to contain all relevant information about when and where it was measured, what was measured and the measured value including a unit.

F3: Reasonable response time for weather requests

Users should not have to wait too long for requests of the current weather. The response time should be reasonable for this standard request. To evaluate this, one should be able to request the current weather 25 times per minute at one location.

[1] `https://opendata.dwd.de/`

Often requests to the microservice rely on the same data. One way to improve response times of the system is to cache that data.

F4: Location specification

Weather is always related to a location. In order to specify a location in a request the user can choose between GPS coordinates compound by latitude and longitude, city name and zip code. The system needs to be able to translate the given location representation to a uniform format that it can handle efficiently. It is sufficient to resolve German city names and zip codes. Other countries don't need to be supported since the DWD provides German weather data only.

The functionality of the feature is tested by three requests of the same data using the GPS coordinates, the city name and the cities zip code to specify the location. If the requirement is met, the microservice delivers an equivalent response for all three requests. Since there are changes to city names and zip codes constantly, one should only test locations that have not been changed in the past 5 years.

F5: Provide a user interface

To make use of the microservice an easy to use REST interface needs to be provided to the user. It needs to offer access to at least the following data for a given location:

- Current temperature
- Current radiation
- Forecasts (Optional)
- Historical weather records (Optional)

The response should have a human readable but yet easy to process format. The most wide spread way of achieving this is using a JavaScript Object Notation (JSON) response with meaningful attribute names.

F6: Adapter for the ODS

The Open Data Service uses different adapters to access different types of data sources. To ensure that the ODS can use all provided functionalities and data, an adapter should be developed. It is evaluated by a test request for every one of the interfaces that is offered by the microservice. All responses need to contain equivalent data, where the output formats may differ.

2.2 Non-functional requirements

Q1: Test coverage

Thorough testing of the developed software is an important part of software development to gain trust in the reliability of the artifact. First, all software components need to be structurally covered by unit tests. The focus here should be more on proper edge case testing than on covering every single line. This is because one could test every method with simple tests solely aiming for running every line, without covering any edge case. This would fail the intend of building trust, and hence is discouraged.

Second, integration testing needs to cover all system functionalities in its test scenarios. To make the outcome of integration tests deterministic the required DWD server with all its files needs to be mocked.

Third, interface testing shall ensure the interface of the DWD server hasn't changed. This can be used to trace down unexpected behavior to either a bug in the microservice or a change on DWD server side. Changes on the DWD interface can also be tracked via the provided change log file[2].

Q2: Accessibility

All available system functions need to be documented and shall be easily accessible by the user interface. The user neither needs to be a computer scientist nor needs to provide any meteorological knowledge. The user interface shall be documented on Swagger[3]. The documentation should be sufficient to access the desired data without any further help.

Q3: Deployment

The implemented microservice shall be easy to deploy in a controlled environment. This should eliminate inconsistent behavior on different platforms. Docker[4] is one popular way to do this.

[2] `https://opendata.dwd.de/climate_environment/CDC/Change_log_CDC_ftp.txt`
[3] `https://swagger.io/`
[4] `https://www.docker.com/`

3 Architecture and design

This chapter explains what technologies are used to meet the requirements and discusses alternatives to those, too. In addition to that, an overview of all created software components and their relation to each other is explained. For orientation figure 3.1 visualizes the role of the DWD adapter microservice that is developed in this thesis. On one end it needs to access the DWD server to fetch all necessary data. On the other end it provides a REST interface to it's users, as it is required by requirement F5, defined in section 2.1. The user can either use the REST API directly or access it indirectly via the ODS, optionally enhancing the data with functions of the ODS.

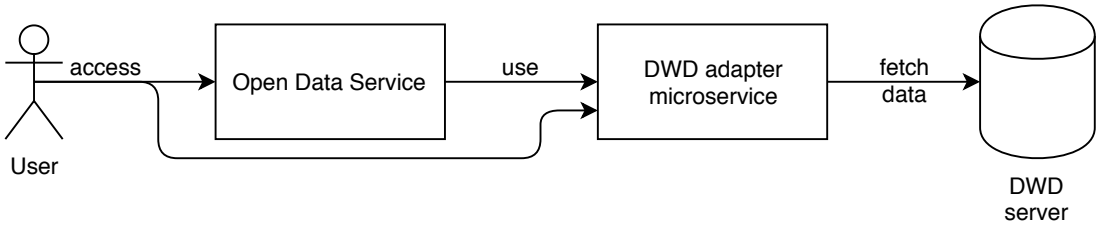


Figure 3.1: Overview of all involved services.

3.1 Definition of microservices

Before we start working on our microservice, we first need to know what actually is a microservice? The term itself is not standardized yet, so everyone has an own definition of it. One way to define it is this:

> "A microservice [...] is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility. It is a single responsibility in the original sense that it's got a single reason to change and/or a single reason to be replaced. But the other axis is a single responsibility in the sense that it does only one thing and one thing alone and can be easily understood." [Thö15]

This very briefly enumerates the biggest advantages of microservices compared to the conventional monolithic architecture. The only disadvantage is that microservices need to send messages over a network in order to communicate. This adds overhead every time two parts of the application need to exchange data. In a good design, however, most communication takes place within a microservice and communication with other microservices is reduced to a bare minimum.

3.2 Base technology for the microservice

When it comes to choosing technologies for creating a new microservice, one is spoiled for choice. Since it is a self contained system that uses the platform independent REST standard for all its interfaces, it may be written in pretty much any programming language, using any frameworks and libraries one could possibly want to use. But instead of choosing an exotic language and some barely documented frameworks and libraries, it is more reasonable to choose conventional technologies. Because no matter how perfectly fine the system works today, one day someone has to maintain it or wants to extend it.

In a multi case study from 2005 researchers analyzed the cost distribution over the life cycle of 30 IT application systems.

> "For a total production time of 5 years, the percentage of non-recurring costs amounts on average to 21% of the total life cycle costs. Therefore, 79% of the total costs are recurring costs, i.e. are further development and production costs. For a projected production time of 8 years, the ratio changes to 15% vs. 85%. The ratio differs significantly among application systems." [ZB05]

While this study certainly is a bit outdated and is based on imprecise data, it gives an idea of how important it is to minimize recurring cost. This denotes all cost that occurs after the initial development covered by this thesis.

This project is going to be maintained by the same developers that also maintain and improve the ODS. Hence, choosing the same technologies used by the ODS reduces maintenance cost in the future. For that reason, we choose Gradle[1] as build tool, Java[2] as programming language and Spring Boot[3] as framework for the microservice. All three are very well documented and established technologies with huge communities, so there is no real argument to justify changing one of them, anyway.

One must note that in parallel to this thesis the ODS is undergoing some struc-

[1] `https://gradle.org/`
[2] `https://www.java.com`
[3] `https://spring.io/projects/spring-boot`

tural changes. It's current monolithic structure will be split into microservices in the future [Sch19]. At the time of this writing the ODS still uses Dropwizard[4], an alternative choice to Spring Boot. But it is going to move to Spring Boot for the migration to microservices in the long term.

3.3 Resolving locations

To meet requirement F4, defined in section 2.1, one has to choose a uniform representation of a location throughout the microservice first. Locations are used in two places only. The first one being in the REST interface, where the user should be able to specify a location either by its geographic coordinates, city name or zip code. This is where the user input needs to be transformed to the uniform representation of a location. The other one occurs when choosing the weather stations that measure the requested data.

This anticipates chapter 4.5, which explains all the details on weather stations and how they are selected. At this point it is sufficient to know that each weather station has its location given by geographic coordinates. The algorithm that chooses the right weather stations needs to calculate the distance to the requested location for each station.

Therefore, choosing geographic coordinates, consisting of latitude and longitude, as uniform representation of a location is an easy decision. Given that, in contrast to zip codes and city names, there is a formula that calculates the distance between two coordinates. Any other representation requires an additional transformation back to geographic coordinates.

This brings us back to the actual challenge of requirement F4. If a user requests the weather using coordinates, the microservice does not need to do any more processing on that input. However, German city names and zip codes need to be mapped to their respective coordinates first. This process is called geocoding, transforming coordinates back to a city name and zip code is called reverse geocoding. There are two ways to support geocoding.

First, one can utilize an existing data set that contains a mapping of city name and zip code to its geographic coordinates. All of that data just needs to be imported to a database that is then accessed by the microservice to find the correct mapping for each request. While there are free data sets available, they are not complete by any means. Resolving big cities is no problem with those, but when it comes to tiny villages no free data set includes all of them. In addition to that, one must consider that this data is constantly changing. For example new villages may be founded or small villages merge into nearby cities. This indicates

[4] `https://www.dropwizard.io/1.3.9/docs/`

someone would have to find the latest data and import all changes to the data base on a regular basis, to keep the service up to date. In conclusion this solution is far from optimal.

Resolving locations is not part of this microservices responsibilities anyway, which is why creating a separate microservice for it is almost obligatory. This has been done by others before, most prominently by the Google Maps Platform[5]. For companies this is probably the best way to go, but it requires registering an account and depositing a credit card because the service is not for free. Therefore, this is not a viable option for this project.

Luckily there is the OpenStreetMap[6] project which also provides all kinds of geodata but does it for free. It includes the Nominatim[7] service, which does geocoding and offers a REST API. Test villages that revealed lacks in other free data sets all worked with Nominatim. The only downside of it is its restrictive usage policy, limiting the amount of requests to a maximum of one request per second. On the upside, one always has the opportunity to install an own instance of Nominatim on a local machine. This should only be required if this microservice becomes so popular that it violates the usage policy even when caching all results. Thus, Nominatim is the best option for the microservice at time of this writing.

[5] `https://cloud.google.com/maps-platform/`
[6] `https://wiki.openstreetmap.org/wiki/Main_Page`
[7] `https://wiki.openstreetmap.org/wiki/Nominatim`

3.4 Caching frequently used data

Using Nominatim as external geocoding service requires caching location information due to their restrictive usage policy. A sequence diagram of such a request to the cache is given in figure 3.2. Since there are too many locations to cache all of them in memory, a simple lightweight Least Recently Used (LRU) cache with a configurable size is used for this job. This caching strategy dumps the element from the cache that has not been used for the longest time and thus seems like a reasonable choice in this application. Not only is the cache part of the implementation that prevents violating Nominatims usage policy because it reduces the amount of necessary requests to the Nominatim service. But it also improves response times of the microservice because the delay for the request to the Nominatim service is replaced with a simple cache lookup.

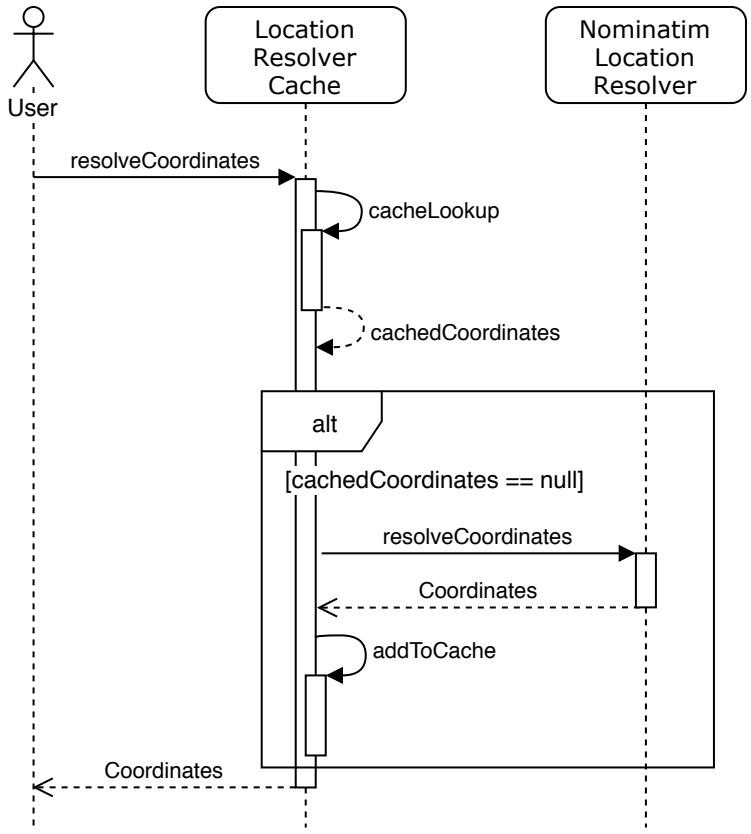


Figure 3.2: Sequence diagram of a User requesting a location. Only if the location is not contained in the cache, the request is forwarded to the Nominatim service and its result is cached for future requests.

It is not the only used cache, though. As explained in chapter 3.3 finding the right weather station is the first processing step of every request. Because a list of all weather stations takes longer to fetch than the current weather data, it

only makes sense to cache it as well. Since there are only a few thousand weather stations, it is no problem to keep the whole cache in memory all the time.

The cache is initialized on demand, which is why the very first request of a weather parameter always takes a little bit longer after a restart. However, the data changes from time to time whenever a new weather station is added or one of them is closed. But it is not necessary to permanently check for the latest data. As it turns out during implementation, the time a weather station is supposedly active does not exactly match the time where it actually produces data. In fact the first available data is usually measured a few days later. Therefore, it is sufficient to refresh the weather station data once a day.

Both the cache for Nominatim and the weather stations are implemented using the proxy design pattern. It has the intend to "provide a surrogate or placeholder for another object to control access to it" [Gam+95]. The pattern makes use of a common interface between the actual object and a proxy object. Figure 3.3 shows how we use it to cache data that is fetched from an external provider. This class diagram is just an abstracted example so it matches both our applications. The original pattern does not use the interface to depend on the underlying service but instead depends on the *ServiceImpl* directly. We modified this to be able to switch the *ServiceImpl* without having to adapt the cache implementation to the new one.

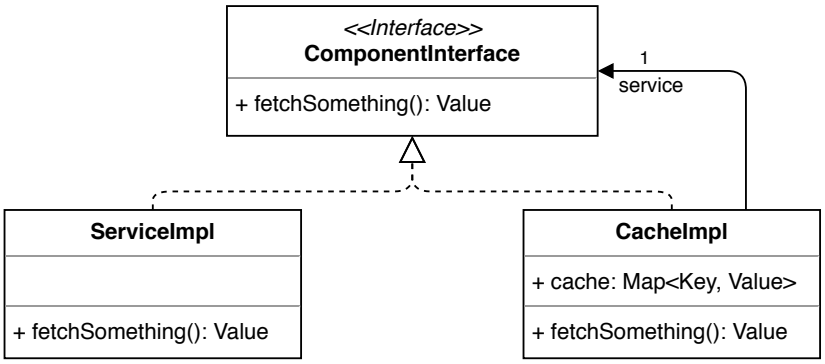


Figure 3.3: Class diagram of a cached service.

Now one may be wondering about caching weather data, too. In order to cache weather data one not only has to add a database to the microservice but also add a routine that ensures the data is not outdated. Since the data does not update within a fixed schedule, this requires constantly fetching the latest data and replacing the old one in some way. Depending on the actual usage of the microservice this introduces a constant load even if there is not a single request actually fetching the data. To reduce the constant load one can think about accepting outdated data up to a certain threshold, effectively trading data quality for better response times.

As it turns out during implementation, thanks to the small file size of current and forecast weather data, requests for that data have response times of less than 200 ms even without caching applied on weather data. According to [Nie93] this is fast enough for the user to only barely notice any delay at all.

So the bottom line is that, since caching data that constantly changes on an irregular basis is not trivial, and hence adds a lot of complexity to the microservice, we decided to not cache any weather data at all. The improvement in response times is not worth the introduced complexity and constant load of the microservice. Even without this optimization, response times exceed our expectations already.

3.5 Software components

With all technologies setup we can start thinking about the structure of the microservice. After figuring out what exactly the DWD server side offers, we collect all components that are required to resemble and abstract all aspects of it. Please note that in this thesis we only focus on a small fraction of the provided data. The DWD provides way more data in various formats created by many different measuring systems for different purposes. So keep in mind this is only a small piece of the whole picture.

Following observations of the data provided by the DWD can be made:

- Forecast weather data is given in a different format and structure than all current and historical records
- All measurements are tied to a weather station. A list of all weather stations is given for forecast data as well as a separate list for each weather parameter
- From time to time weather stations are modified. For instance the measuring instrument moves, or a new measurement method is used All changes are logged in meta data files and may be important to some users
- Some of the files are compressed and the compression method varies

Adding dependencies to the resulting components leads to the component diagram displayed in figure 3.4. Additionally to all DWD specific components, it also includes the necessary *SpringController* that provides the REST API of the new microservice as well as the *LocationResolver* discussed in chapter 3.3.

The *SpringController* is the interface between the microservice and the user. Therefore, it needs access to all components that provide it with all the different data requested by the user. This includes the *WeatherStationListProvider*, which is responsible for providing all weather station lists, and the *MetaDataProvider*.

Even though it is not specified by any requirement it seems quite important

to provide meta data to interested users. This kind of data may explain some unexpected surprises in the data and hopefully helps understanding the data while analyzing it. Meta data is basically just text explaining changes to the measuring process. It is distributed over multiple files. The microservice simply maps each file name to its plain text content. This way the user receives a response that perfectly resembles all available meta data of a weather station. All of this is handled by the *MetaDataProvider* component.

Of course the *SpringController* also needs access to the *WeatherDataRequester*, which is the interface to fetching, processing and concatenating the actual weather data. It hides all the complexity introduced by the variances in data and structure on the DWD server. One can access it through a uniform interface delivering uniform data no matter what weather parameter is requested during any time span. Doing so, it seamlessly connects measured observation data with predicted forecast data.

Since this is way to much complexity to handle in a single component, the *WeatherDataRequester* focuses on splitting the request into multiple smaller requests that are then delegated to the *ObservationDataProvider* and the *ForecastDataProvider*, depending on the requested time span. These in turn take care about fetching the data from the right path and converting it into a usable data format.

As one can see in figure 3.4 all data providing components delegate the actual fetching task to the *Downloader* component. This component does not just download the data provided on a given path but also handles decompressing it automatically depending on the file type. Files provided by the DWD are sometimes not compressed at all and other times compressed using different compression methods for different files. The *Downloader* component simplifies all other components by encapsulating the complexity of handling all these different file types seamlessly.

In addition to that it also handles different protocols, so files can be fetched by either HTTP, FTP or even a local path. This enables fetching the data either from the DWDs HTTP server or their FTP server, which hosts the same data but is not as responsive and reliable. Alternatively, one can test new features locally by downloading a snapshot of all necessary files and redirecting the path to the DWD server to point to the snapshot. Spring boot offers great support for configuration files where this can be done without touching the source code of the microservice.

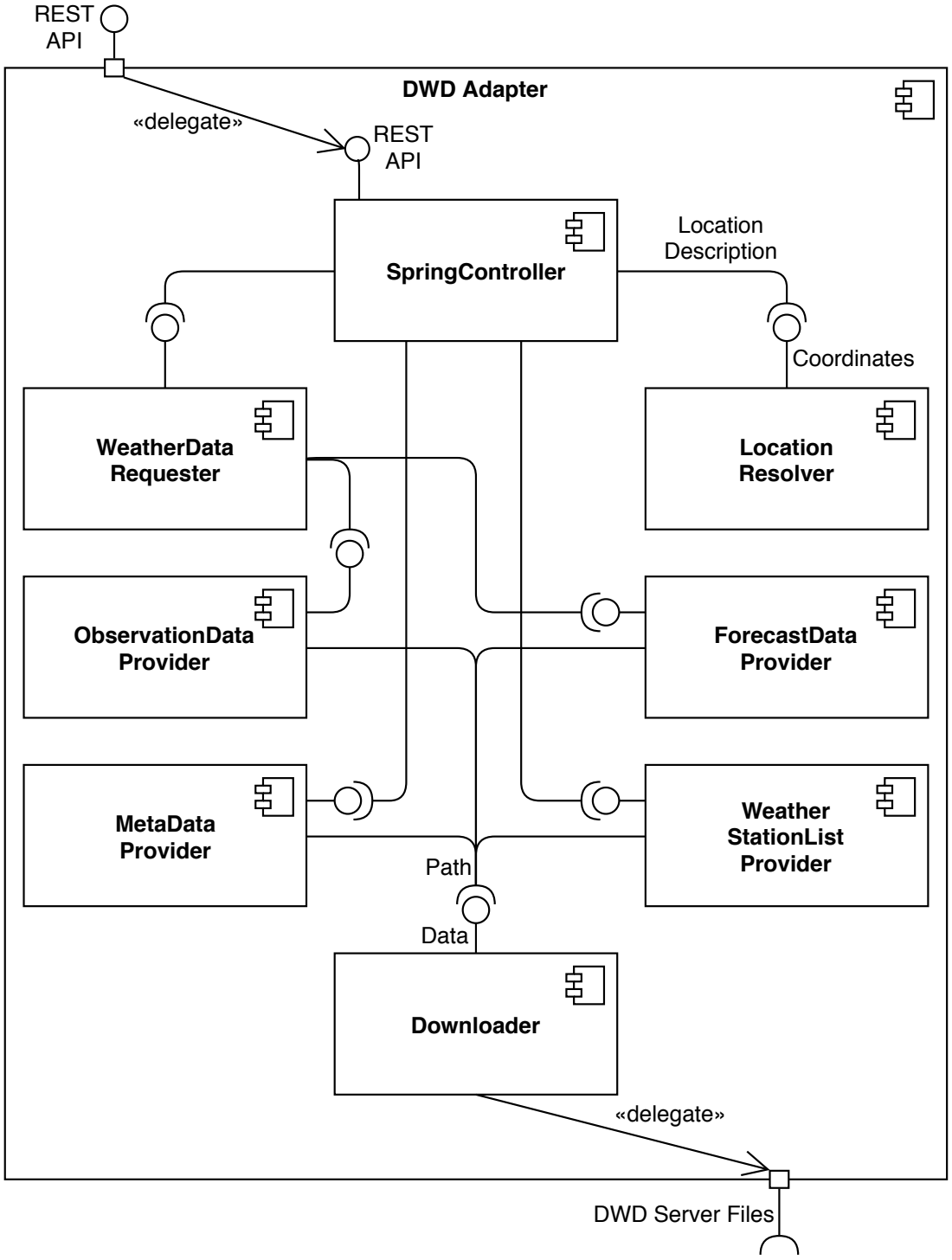


Figure 3.4: Component diagram of the DWD adapter microservice.

4 Implementation

In order to fulfill all requirements defined in chapter 2 there are a bunch of challenges to face. Fortunately the previous section 3.5 divides the responsibility to solve them into smaller components already. This chapter presents the solution to each individual problem. At the same time this is also supposed to serve as part of the artifacts documentation. Hence, created data structures and developed algorithms are explained in such a detail that one can not just comprehend how everything works but also how it emerged from the requirements to it.

4.1 Finding the right files

The first problem that needs to be solved is actually not directly mentioned in the requirements specification, although none of the previously defined components could do anything without it. In order to be able to download, extract, parse and further process data one first has to find out from where to download it. Data on the DWD server is structured in a file system alike manner. Originally files were only provided by a FTP server, the HTTP server was added later on, which is probably responsible for this structuring.

After getting an overlook of what data is actually provided by the DWD, the files contained in the Climate Data Center (CDC) directory[1] are exactly what we are looking for in this project. Unfortunately forecast data is not provided there but is collected in a separate forecast directory[2].

Finding the right forecast file is very simple because there is exactly one file provided for each weather station. Each file contains all available weather parameters. With observation data, however, things get a little bit more difficult. There are four dimensions in which the data is structured. First, one selects the time interval of the measurements, ten minutes, hourly, daily, monthly and so on. Next the parameter needs to be chosen. Unfortunately there is not simply one directory for each weather parameter. Instead most directories, and in con-

[1] `https://opendata.dwd.de/climate_environment/CDC/`
[2] `https://opendata.dwd.de/weather/local_forecasts/`

sequence the actual data file, accumulate a couple of parameters together to what we call a combined parameter. At first glance this way of structuring seems reasonable but there is a catch. There are different weather parameters combined together, depending on the time interval one chose in the previous step. This randomness prevents any general approach to resolve the correct directory from working. One has to create different mappings for each combination of weather parameter and time interval.

The third decision depends on the measurement time of the data. Files are sorted into historical, recent and now time spans. Now denotes data measured today and is only available if one chose the ten minutes interval in the first step. Recent is a sliding window of the last 500 days and historical collects everything ever measured until the end of last year.

The last step is to choose the right file using the station id of the measuring weather station, since it defines the file name. At least that is how it works for now and recent files. Historical files sometimes are split into multiple smaller files. Hence, their names are extended with a start and end date of the contained data. Therefore, the filename can not be predicted solely by the station id. To find the correct file names one has to first download all names and then filter them for the ones of the required weather station.

To resemble this structure we create a *DataSet* for each combined parameter in ten minutes and in hourly time interval. Each *DataSet* contains information on how to find all files of a weather station and how to parse them. Finally, the combination of a weather parameter and each of the three time spans, historical, recent and now are mapped to the respective *DataSet*. This map enables us to get all required information about where to find the right files and how to parse them just by knowing the requested weather parameter and time span. We statically create and insert all *DataSet* objects into the map upon startup.

Alternatively, one may also resemble the structure using configuration files. All file paths and regular expressions could be defined in it, as well as a configuration for a more generic parsing approach. As a result, all available weather parameters and their mappings to the right configuration also need to be defined in a configuration file, effectively making the whole accessible data set fully configurable.

The static insertion method is definitely easier to implement and hence less error-prone, whereas a configuration file offers the flexibility to be changed later on without having to touch the code.

Table 4.1 gives a list of all weather parameters that are available in the CDC database. It ticks the box if the parameter is available in ten minutes and hourly measuring interval during historical, recent, now and forecast time span. Only if there is no hourly data available we fall back on using the ten minute data, due to our targeted hourly data output format.

For every used cell in this table one would have to define the regular expression to the respective files, a configuration for the parser of those files and a mapping, so one can find the right configuration by a weather parameter and a time span. Besides the required implementation to interpret all these configurations, one would create giant configuration files that no one would ever want to touch again. Even worse, a structural change that can not be tolerated by the whole set of configurations anymore, requires a complete rebuild of the configuration jungle. Therefore, we decide to apply the Keep It Simple Stupid (KISS) principle and simply hardcode the mappings using a class hierarchy to omit code duplications. This yields a comprehensive definition of all weather parameters from table 4.1.

	ten minutes			hourly		
name	historical	recent	now	historical	recent	forecast
air_pressure	-	-	✓	✓	✓	✓
temperature200	-	-	✓	✓	✓	✓
temperature5	✓	✓	✓	✗	✗	✓
humidity	-	-	✓	✓	✓	✗
dew_point	✓	✓	✓	✗	✗	✓
precipitation_duration	✓	✓	✓	✗	✗	✓
precipitation_height	-	-	✓	✓	✓	✓
sunshine_duration	✓	✓	✓	✗	✗	✓
diffuse_solar_radiation	✓	✓	✓	✗	✗	✗
total_solar_radiation	✓	✓	✓	✗	✗	✓
longwave_downward_ radiation	✓	✓	✓	✗	✗	✗
wind_speed	-	-	✓	✓	✓	✓
wind_direction	-	-	✓	✓	✓	✓
visibility	✗	✗	✗	✓	✓	✓
cloudiness	✗	✗	✗	✓	✓	✓
max_temperature200	✓	✓	✓	✗	✗	✗
min_temperature200	✓	✓	✓	✗	✗	✗
min_temperature5	✓	✓	✓	✗	✗	✗
max_wind_speed	✓	✓	✓	✗	✗	✓
max_wind_ speed_direction	✓	✓	✓	✗	✗	✗

Table 4.1: Available weather parameters provided in ten minutes and hourly time interval.
✓: value is used
✗: value is not provided
- : value is provided but not used because it is available in hourly

4.2 Abstracting the weather model

As requirement F2 explains, the DWD focuses more on human readability than on a weather model that is easy to process. They provide forecast data in a standardized file format, which contains predictions for all weather parameters. Observation data on the other hand is provided in table form, combining a few weather parameters together in one CSV file. In order to be able to process the data it needs to be abstracted to one uniform representation of each measurement respectively each forecast prediction.

Therefore, we abstract the provided data to *DataPoint*s. Each of these value objects represents one measurement of a weather parameter at a certain time from one weather station.

Additionally, it contains information from what time interval it was created. This is the time that is covered by one measurement. There are various time intervals used by the DWD. The microservice primarily deals with the ten minutes, hourly and three hourly time intervals.

In order to estimate the precision of a measurement an *origin* field is carried as well. The DWD uses a scale to define the quality of a measurement in their observation data. Every index on the scale represents a processing step that controls or corrects the accuracy of a measurement. This is simplified for users of the microservice, since the index generally correlates with the time section the measurement is located on the DWD server. Remember the time sections are:

- **historical**: earliest record until end of last year
- **recent**: last 500 days until yesterday
- **now**: today starting at 00:00 until now
- **forecast**: now until approximately 240 hours in the future

Measurements become more accurate the earlier the time section ends. Since the time interval is not always constant for a parameter over several time sections, multiple ten minutes *DataPoint*s have to be accumulated to one hourly *DataPoint* to match the standard hourly output of this microservice. This accumulation process is covered in more detail by chapter 4.6. The accumulated values decrease the precision, hence this needs to be noted. Together with the four time sections this results in eight possible origins that are listed in table 4.2.

DataPoint origin	Quality level
HISTORICAL	high
RECENT	medium
NOW	low
FORECAST	very low / prediction
HISTORICAL_ACCUMULATED	accumulation of HISTORICAL values
RECENT_ACCUMULATED	accumulation of RECENT values
NOW_ACCUMULATED	accumulation of NOW values
FORECAST_ACCUMULATED	accumulation of FORECAST values

Table 4.2: Possible *DataPoint* origins and their rough quality level.

4.3 Parsing DWD data files

Data on the DWD server is organized in files, often compressed in archive files. The data within those needs to be parsed and mapped to *DataPoint*s, our abstracted measurement representation. One has to differentiate between forecast and observation data since they are provided in very different ways.

Parsing forecast data files

Every weather measurement created after the time of request on the microservice is considered a forecast. The DWD provides multiple ways to deliver forecast data. For instance one can access textual forecasts[3]. But this is not very useful for our purposes. Fortunately they also provide Model Output Statistics-MIX (MOSMIX) data. This is statistically optimized point forecast data available at roughly 5400 locations worldwide[4]. Over 3000 of them within or very close to Germany.

Users of the DWD server can choose between two sets of data. The MOSMIX_S dataset is updated hourly, covers 40 parameters and summarizes the forecast of all locations in a single file. The MOSMIX_L dataset is only updated four times a day, theoretically covers about 115 parameters and is split into one file for each location. Both datasets cover 240 hours of forecast data. Every update not only appends more values but also increases accuracy of values that have already been available in previous versions. The microservice has to download, unzip and parse a 38 MB file for the MOSMIX_S dataset but only 16 KB for each MOSMIX_L dataset. Since the expected use case is a weather request for a specific location, the response time of the microservice is significantly lower compared to using the MOSMIX_L over the MOSMIX_S dataset.

[3] `https://opendata.dwd.de/weather/text_forecasts/`

[4] `https://www.dwd.de/DE/leistungen/met_verfahren_mosmix/met_verfahren_mosmix.html`

A MOSMIX file is provided in the Keyhole Markup Language (KML) file format compressed with zip to a Keyhole Markup Language Zipped (KMZ) file. One can parse this file format very easily with the help of an XML parser. There are two important elements in each file.

The first one is called *dwd:ForecastTimeSteps*. It lists all time stamps that this file contains data for. Each time stamp is represented by an ISO 8601 [org04] conform UTC time.

The second one, *kml:ExtendedData*, lists all parameters. Each parameter contains a list of forecast values for it. The index of each value determines the index of the time stamp it belongs to. In order to map it to *DataPoint*s, each value is combined with its time stamp. Further information like the used abbreviations for the parameter names and the measuring unit of each parameter are located in a separate file. We assume these will not change over time and forgo relying on it. Parsing forecast data is handled by the *MosmixDataParser* class.

Parsing observation data files

Unfortunately parsing observation data is not so easy. Besides the higher complexity behind finding the correct file, which is explained in chapter 4.1, the file content doesn't follow a predefined standard. It appears to be primarily made with focus on human readability.

In general files contain a table with a header row in the first line. Columns are separated with semicolons. Rows are separated by a newline character. Each file usually combines multiple weather parameters to a combined parameter. The first three columns contain the unique id of the weather station, the time stamp of the measurement and the quality level. The remaining columns are specific to the combined parameter.

In contrast to the time stamps defined in MOSMIX files, the ones defined in observation data files follow the short version of the ISO 8601 standard. It matches the general pattern yyyyMMddHHmm of Java's *SimpleDateFormat*[5] class. The letter y relates to the year, M to the month, d to the day, H to the hour and m to the minute. For example November the ninth, 1989 at 5:57 pm is represented as 198911091757. If the exact minute is not required because the measurements are taken hourly -always on a full hour-, the pattern will translate to yyyyMMddHH. If it is measured on a daily basis, the hourly letters will be dropped and so on.

Parsing of these time stamp formats is required in multiple places but not both are natively supported by Java. Thus the complexity is moved into an own *TimeStampParser* class. This also allows the user to seamlessly choose between ISO 8601 and its short version to define time stamps in requests.

[5] `https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html`

Besides the different time stamp formats there are more difficulties hidden behind the table format. It took three approaches to find a parsing method that tolerates all minor differences that may occur within each file and between files of different combined parameters reliably. Since there are many different combined parameters, only parsing of a single data row should be handled separately for each combined parameter. The remaining complexity is bundled in the *ObservationDataParser* class. This avoids code duplication for things like exception handling and result collection.

The parser needs to be able to deal with an interruption in the table. This appears very rarely and was revealed by a lucky coincidence only. In a few files there is plain text explaining something in the middle of the table, where the parser expects another data row. Hence, the first approach for the parser crashes because the row doesn't contain as many columns as expected.

Another small detail is an additional *eor* column. It marks the end of row additionally to the newline. The problem is that it is not used for a combined parameter throughout in all files but only occurs in some files. The most intuitive way to handle this case is to extract the columns by searching for the next semicolon. With this second approach one could just skip everything after the last expected column including the 'eor' column. While this solution certainly solves the issue, it introduces quiet a bit of overhead. With this implementation, every single row needs to be iterated character by character to look out for the next semicolon, before its content can be interpreted. This applies to all other parser implementations that look for patterns, too. Since there are over a million rows in some of the files, this is not an acceptable solution.

This is one of the most important performance critical parts of the whole microservice. Therefore, the ideal solution uses the fact that the width of each column seems to be constant in every row. The third approach uses that knowledge to split the row into columns. By using the *subString* method of Java's *String*[6] class all characters of each column are visited only once to copy them into a separate *String* and one more time to parse the value from that copy. One could eliminate the copy step with a custom implementation of *subString* but we don't want to trade too much code readability for performance. Tests have shown that with this improved approach the limiting factor of the microservice changed to the download speed of the file, anyway.

As the attentive reader may have noted the width of each column only seems to be constant in every row. That is until the implementation is tested on some example files just to notice that again a few files contain random shifts after hundreds of thousands of perfectly aligned rows. Therefore, approach number three is modified to support this case as well.

[6] `https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`

In order to use the *subString* method one has to know the position of all semicolons. Instead of finding those positions before parsing the first row only, it also updates the positions whenever a shift is detected. This leads to the control flow shown in figure 4.1.

The only downside of approach three is that it can not tolerate inconsistencies with the 'eor' column within a file. One must assume it is used in every row if it is used in the first one. This did hold true for all tested files and should be a valid assumption because the 'eor' column is also declared in the header line whenever it is used.

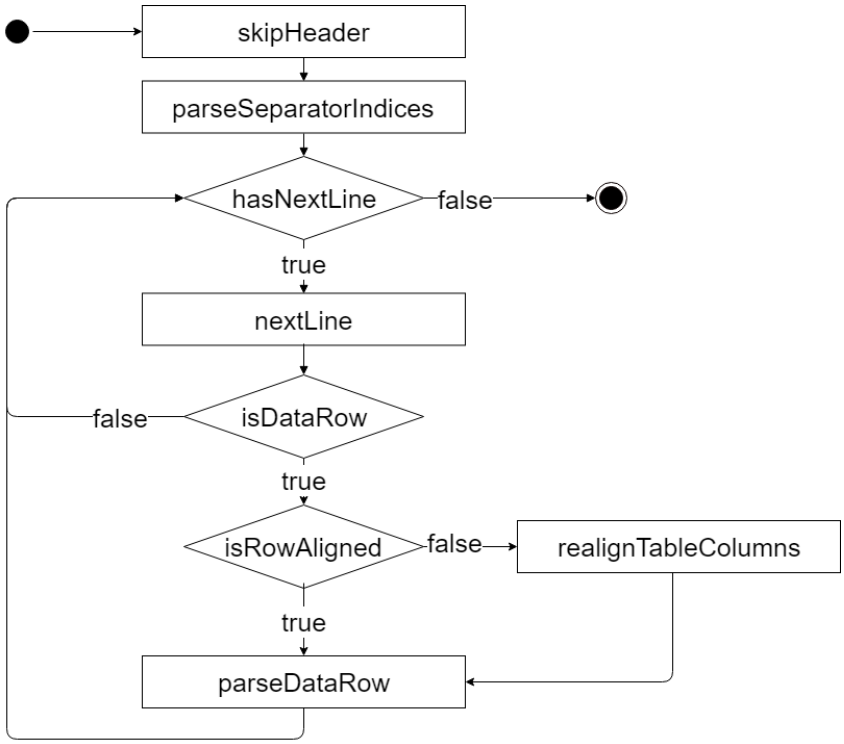


Figure 4.1: Control flow diagram of the *ObservationDataParser* parsing an observation data file.

Once all values are parsed from a row they are stored in an intermediate value object. The next processing step transforms it to a *DataPoint* as section 4.6 explains in more detail.

Parsing lists of weather stations

In addition to the weather data files, one also needs some lists of all weather stations, both forecast stations and observation stations. Section 4.5 explains the usage of these lists in more detail. Fortunately the DWD provides a file of all

MOSMIX stations as well as a file for each combined parameter of all observation stations.

Parsing those files is very similar to parsing observation data. Each file contains a table with a header row in the first line. Columns are separated with semicolons. Rows are separated by a newline character. In contrast to parsing observation data there is no 'eor' column that randomly appears in some files. But there are interruptions in the table that both the *MosmixStationParser* and *ObservationStationParser* have to tolerate.

The provided information about each weather station is equal for MOSMIX and observation stations except the active time that is missing for MOSMIX weather stations. It defines the start and end date of when a weather station supposedly measured data. The observation station list contains all stations that have ever been active. The MOSMIX station list on the other hand only contains those that are currently active. For later processing this information is supplemented.

4.4 Time data structures

The microservice needs to deal with time in various places. Internally, time is represented by an object of the *Instant*[7] class. In this application it is always interpreted as a UTC time time stamp.

4.4.1 Definition of time spans

A lot of the microservice‘s internal logic requires to composite two times to create a time span. A time span is defined by its start and end time. This is represented by an instance of the *TimeSpan* class. There are two main use cases for it. First and foremost it is part of any request for a specific set of weather data. Users define a *TimeSpan* to filter the queried data by its time of measurement. This can be done explicitly by specifying a start date and time and an end date and time. Alternatively, users can choose from a set of predefined *TimeSpan*s. Table 4.3 lists all *PredefinedTimeSpan*s and their explanation.

PredefinedTimeSpan	**Explanation**
NOW	today starting at 00:00 until now
RECENT	last 500 days until yesterday
HISTORICAL	earliest record until end of last year
TODAY	today between 00:00 and 23:59
TOMORROW	next day between 00:00 and 23:59
NEXT_WEEK	next seven days starting with tomorrow
FORECAST	time of all available forecast (ca. 10 days)

Table 4.3: List of all *PredefinedTimeSpan*s and their explanation, where now is equivalent to the time of request.

In addition to that, *TimeSpan*s are required due to the structure of the data on the DWD server. It is split into three folders, each one contains data for a certain timespan. Those are equivalent with the NOW, RECENT and HISTORICAL *PredefinedTimeSpan*s. The microservice figures out which of these do overlap with the queried *TimeSpan* to decide what files are required to collect the requested data. This excludes unneeded files and therefore is part of the solution to fulfill requirement F1 defined in section 2.1.

Methods

Besides the method *overlaps* that checks for an overlap between two *TimeSpan*s, the class also offers a method *covers* which can be used to determine whether one *TimeSpan* completely covers another one. That is exactly when the start is equal

[7] `https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html`

or before the start of the other *TimeSpan* and the end is equal or after the other *TimeSpan*. Additionally, the *includes* method is implemented so one can test whether an *Instant* is between start and end of a *TimeSpan*. This is required to filter the downloaded data for measurements taken during the queried *TimeSpan*.

4.4.2 Adding gaps to time spans

While a continuous time span is ideal for the user to request data, there are use cases that need to extend the simple representation of a *TimeSpan* by gaps. This extension is implemented by the *CuttableTimeSpan* subclass. It starts out as an ordinary continuous time span but provides the functionality to cut out time spans from it in order to create gaps.

Use cases

Weather stations measure the data that can be queried by this microservice. But they are not a constant set of institutions that have always been there. The set of weather stations constantly changes over time. Therefore, a weather station has an active time. That is the *TimeSpan* starting with the date of when the weather station started measuring data and ending with the date it was closed.

The active time is an important criteria when selecting the weather station for a query. If a weather station has not been active during the requested time, it obviously would not have measured any data that could be used to answer the request. However, just because the requested time is covered by the active time span, does not mean there is any data either. There are many reasons for gaps in the measured data. For example there may have been maintenance work on the measuring instrument or even a defect on it. These random events can cause several hours or even days of missing data. This is quite common but occasionally gaps are as big as multiple years.

The goal is to minimize gaps in the response data. To achieve this, the active time needs to be represented by a *CuttableTimeSpan* that is not continuous but includes gaps. Whenever the application detects a gap in the data it cuts the missing *TimeSpan* from the active time.

In order to achieve an active time that perfectly resembles the continuity of the measured data, one would have to iterate through all measurements. Since this is too resource expensive, a *TimeSpan* is only cut from the active time if no data could be fetched from the station during a requested *TimeSpan*. This way a user never gets an empty response without a lot of extra effort for the microservice. The downside is that if the requested *TimeSpan* covers more than the gap, the active time is not cut and the response lacks the data during the gap. An explicit request for the data during the gap yields a cut of the active time. As a result all future requests query the missing data from an other weather station.

CuttableTimeSpan is also used in case the active time of a single weather station does not cover the requested *TimeSpan*. This scenario requires data from multiple weather stations. The list of all required weather stations is created by cutting the active time of a weather station from the requested *TimeSpan* until it is empty. Empty means the entire *CuttableTimeSpan* has been cut away. Section 4.5 explains the solution to this scenario in more detail.

Requirements

CuttableTimeSpan needs to be a specialization of *TimeSpan* so they can be used interchangeably. It needs to offer an additional *cut* method to introduce gaps and empty it. A *CuttableTimeSpan* starts as a *TimeSpan*. They should behave equal until cut is used. If the *CuttableTimeSpan* does not cover the *TimeSpan* that shall be cut out or if part of it has been cut before, it will be useful to return the *TimeSpan* that has actually been cut. The remaining *TimeSpan*s should be retrievable by the *getTimeSpanFragments* method. An *isEmpty* method is required for the second use case to know whether a *CuttableTimeSpan* has been cut entirely.

Data structure

There are multiple ways to build a data structure that can fulfill above requirements. We decide to use a tree structure for two reasons. First, it is fairly simple to implement the non trivial cut operation on a recursive data structure. A recursive algorithm is a lot less complex and therefore less error-prone than an iterative solution. Secondly, it offers logarithmic run time in the average case of cut operations. This is better than the linear run time of most list based solutions. However, performance in this case is less important than readability because not too many cut operations on a *CuttableTimeSpan* are to be expected. The implementation is thread safe, since it is used by multiple threads.

The tree specifically is a binary search tree made from *TimeSpanNode*s. A node consists of a *TimeSpan* and two fields for its children *beforeChild* and *afterChild*. An inner node bridges the gap between exactly two *TimeSpan*s. The node covering the *TimeSpan* before the gap is assigned to *beforeChild*. The node covering the *TimeSpan* after the gap is assigned to *afterChild*. The *TimeSpan*s covered by the two child nodes must not overlap. Leaf nodes represent the actual *TimeSpan*s that have not been cut yet. In case a *TimeSpan* has been cut entirely, a new empty node is inserted. It covers a *TimeSpan* where start and end are equal. Once created, only leaf nodes are modified to add new children.

When applying these rules, the data structure looks like demonstrated in figure 4.2. The example shows an inner node covering the two *TimeSpan*s (A - B) and (C - D). It has a *TimeSpan* of (A - D) bridging the gap between B and C.

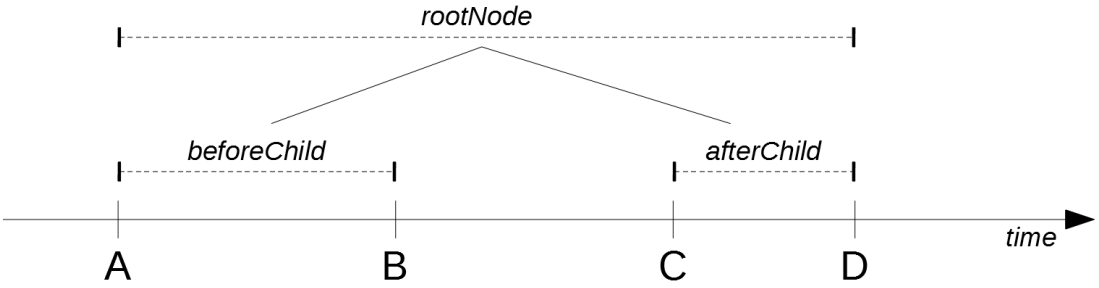


Figure 4.2: Simple example of a *CuttableTimeSpan* initialized with a *TimeSpan* from A to D and an added gap between B and C.

Its *beforeChild* has a *TimeSpanNode* assigned whose *TimeSpan* is (A - B) and *afterChild* has a *TimeSpanNode* assigned covering the *TimeSpan* (C - D).

The recursive cut algorithm

Initially the root node is created with the *TimeSpan* of the *CuttableTimeSpan*.

There are four cases that can occur when cutting a *TimeSpan* from a leaf *TimeSpanNode*:

1. The *TimeSpan* overlaps the start but does not cover the end:
   - Add an empty *beforeChild*
   - Add an *afterChild* node covering the rest of the *TimeSpan*
2. The *TimeSpan* overlaps the end but does not cover the start:
   - Add a *beforeChild* node covering the rest of *TimeSpan*
   - Add an empty *afterChild*
3. The *TimeSpan* is between start and end similar to the example shown in figure 4.2:
   - Add a *beforeChild* node covering the *TimeSpan* between start of this node and start of the *TimeSpan* that is cut out.
   - Add an *afterChild* node covering the *TimeSpan* between end of the *TimeSpan* that is cut out and end of this node.
4. The *TimeSpan* covers the entire *TimeSpan* of this node:
   - Add an empty *beforeChild*
   - Add an empty *afterChild*

Note that the *TimeSpan* of any *TimeSpanNode* is not modified ever. Inner nodes take advantage of the recursive data structure and the binary search tree property. They direct any cut method call to their children provided the *TimeSpan* of the child overlaps with the *TimeSpan* that shall be cut.

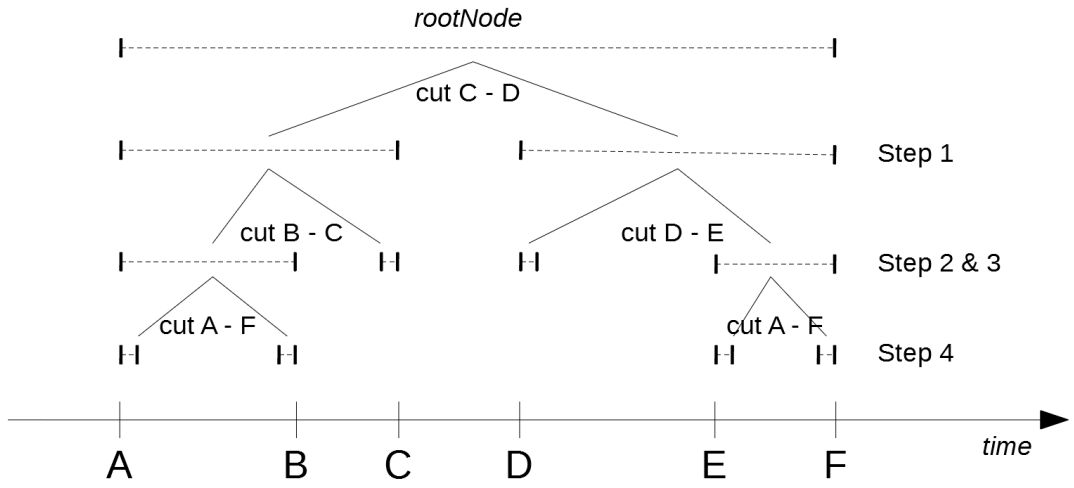


Figure 4.3: Example of a *CuttableTimeSpan* initialized with a *TimeSpan* from A to F that is cut in 4 steps until it is empty.

Figure 4.3 displays another example *CuttableTimeSpan*. It is the result of four cut operations. Each cut operation is considered a step. The *CuttableTimeSpan* is initialized with the *TimeSpan* between A and F. Where A, F and all the other following literals can be any point in time. No two literals may be the exact same time and they need to be ordered according to the time beam.

The first step cuts the *TimeSpan* between C and D from the *CuttableTimeSpan*. This adds two child nodes according to case three from above. The *TimeSpan* is between start and end but includes neither of them. Therefore, the *beforeChild* of the *rootNode* covers the *TimeSpan* between A and C where C is excluded. And the *afterChild* of the *rootNode* covers the *TimeSpan* between D and F where D is excluded. To exclude a point in time, a nanosecond is added to the start or subtracted from the end of the *TimeSpan*. A nanosecond is the smallest unit of time when represented by Java's *Instant* class.

Step two and three target the cases one and two from above. One cuts a *TimeSpan* from B to any point in time between C and D. The other one cuts a *TimeSpan* that reaches from any point in time between C and D to E. This yields two child nodes for each step. Both times one of them is empty, the other one covers what is left from the parents *TimeSpan*.

The final fourth step cuts the entire *TimeSpan* that is covered by the *rootNode*.

According to case four from above, this adds empty *TimeSpanNode*s to all non empty nodes. One can create the exact same tree by cutting the remaining *TimeSpan*s separately. Once all leaf nodes are empty, the *CuttableTimeSpan* is considered empty as well. Any additional cut operations on an empty *CuttableTimeSpan* have no effect and return an empty *TimeSpan*.

In the entire application *CuttableTimeSpan* is the only data structure that may be modified in parallel by multiple threads. This happens if two requests are worked on simultaneously and both rely on the same weather station. Therefore, *TimeSpanNode*s are implemented to be thread safe.

4.5 Managing weather stations

For every request one first has to find the correct weather station to fetch the data from. That is because data on the DWD server is structured in a way that there is a separate file for each weather station. A weather station is represented by the equally named *WeatherStation* class. It combines all relevant information such as its unique id, name and position. As explained in section 4.4.2 each *WeatherStation* contains an active time during which it measured data. That active time is modified to compensate gaps in the measured data.

The DWD only provides one list of stations for forecast data and one list for each combined parameter. But because a gap in the data of one weather parameter does not automatically indicate a gap in the data of an other parameter that is part of the same combined parameter, a separate list of *WeatherStation*s is required for every single weather parameter. Lists of *WeatherStation*s are managed by the class *WeatherStationList*. It provides all necessary functionality, whereas *WeatherStation*s are just value objects. In order to provide them as quickly as possible, all weather stations are kept in memory all the time.

Use cases

One use case of *WeatherStationList*s is finding the *WeatherStation* object that is referenced by a certain unique station id. This for example is used in requests for meta data of a certain weather station.

Most importantly, it is a key element when selecting the data files that contain the requested data. As explained in section 3.3 a request is always linked to a location represented by coordinates. The user wants the weather data for the requested location to be as accurate as possible. Hence, the data should have been measured as close to the requested coordinate as possible. This translates to the data of the closest *WeatherStation*. The *WeatherStationList* finds the *WeatherStation*s that will be used to fetch data from.

Selecting the forecast weather station

Finding the closest *WeatherStation* for a request on forecast data is pretty simple. One has to compute the distance between the stations coordinates and the requested coordinates and select the one with minimum distance. This is done by the *findClosestStation* method. Any station that is contained in the forecast *WeatherStationList* should provide data suitable to the request. If it does not provide any data for the requested parameter, it is removed from the list, and the next closest station is used instead. This way a *WeatherStation* that does not actually measure a parameter is not even considered in future requests anymore.

Currently all this takes place on a simple list, using linear reduction to find the minimum. This can be done in one statement using Java Streams. The performance is sufficient for current needs but not optimal though. It could be improved with a pretty cumbersome solution. One could store all *WeatherStation*s in a two dimensional grid. One dimension is sorted by latitude and the other one by longitude. Performing binary search over both dimensions on that grid narrows down the closest latitude and longitude in logarithmic run time. This alternative solution has two obvious downsides. It requires more memory and it is way more complex to implement.

Selecting observation weather stations

With a request for observation data the algorithm becomes a bit more difficult. The catch is that one can not assume that the closest *WeatherStation* has been active during the requested time span as it is with forecast data. *WeatherStationList*s for observation data contain any weather station that has ever measured any parameter of the combined weather parameter. Therefore, the requested *TimeSpan* is an additional criteria next to the requested coordinate.

Filtering for all *WeatherStation*s that have been active during the requested *TimeSpan* does not work, since the requested *TimeSpan* may be longer than any *WeatherStation* has ever been active. Even if there is one, the solution would not be optimal because in the meantime there may have been a *WeatherStation* that is closer to the requested coordinate, and hence could provide more accurate data.

Required is an algorithm that finds all *WeatherStation*s that are as close as possible to the requested coordinate and together cover the requested *TimeSpan* with their active times. At any point in time there should not be any *WeatherStation* that is closer to the requested coordinate than the found one. The results are collected in *TimeStationAssignment*s. Each one assigns a *WeatherStation* to a list of *TimeSpan*s during which it is supposed to provide data.

To solve this problem we come up with a three step algorithm that utilizes a

CuttableTimeSpan to cover the requested *TimeSpan*. In order to demonstrate how it works, let us suppose we need to solve the problem for an example request, which is to fetch the temperature in Erlangen (49.5981187° : 11.003645°) between the years of 1950 to 2000.

The first step is to sort all *WeatherStation*s of the temperatures observation *WeatherStationList* by their distance to Erlangen. The result is visualized in table 4.4. One might have the idea to speed up sorting by filtering the list of stations for those who are within a certain radius. This does not work that easily because the density of stations has quickly been rising since the late 20th century. Before that, the amount of stations in the early 20th century or even late 19th century is too small to determine a radius that does not filter all stations that were active during the requested time at all. At the time of this writing there are about 650 *WeatherStation*s, so sorting those is not too expensive yet.

WeatherStation	**Distance [km]**	**Active time**
Erlangen-Frauenaurach	4.27	1994-11-01 - 2001-05-01
Möhrendorf-Kleinseebach	5.74	1986-11-01 - now
Nürnberg	11.20	1951-01-01 - now
Gräfenberg-Kasberg	17.47	2006-10-01 - now
Nürnberg-Netzstall	26.33	2005-03-01 - now
Markt Erlbach-Mosbach	27.13	2004-11-01 - now
Bamberg	31.27	1961-01-01 - now
Bamberg (Sternwarte)	32.58	1947-01-01 - 1955-01-01
Pommelsbrunn-Mittelburg	40.66	2005-03-01 - now
Roth	43.06	2002-01-01 - now
...	...	...

Table 4.4: List of all *WeatherStation*s sorted ascending by distance to Erlangen.

The second step is to create a *CuttableTimeSpan* from the requested *TimeSpan*. This is easy using the constructor of the class.

The third and final step is where it gets interesting. The sorted list of all *WeatherStation*s is iterated, starting with the closest station. Each one of them cuts its active time from the *CuttableTimeSpan*. We iterate the list until the *CuttableTimeSpan* is empty, which is exactly when it is covered entirely by all stations. Since an active time also is a *CuttableTimeSpan*, each fragment of it needs to be cut separately. Remember from section 4.4.2 that the *cut* method returns a list of *TimeSpan*s that were actually cut from the *CuttableTimeSpan*. The results are accumulated in a list.

If the active time does cut something from the *CuttableTimeSpan*, a new *TimeStationAssignment* is created from the current station and the *TimeSpan*s that it cut

away. It is then added to the resulting list of *TimeStationAssignment*s and the next iteration can begin until either all *WeatherStation*s are iterated or the *CuttableTimeSpan* is empty. The result of this final step executed on the example request is listed in table 4.5. Note that multiple *TimeSpans* may be assigned to a *WeatherStation*, even though this does not happen in the example.

WeatherStation	**Assigned time spans**
Erlangen-Frauenaurach	[1994-11-01 - 2000-01-01]
Möhrendorf-Kleinseebach	[1986-11-01 - 1994-10-31]
Nürnberg	[1951-01-01 - 1986-10-31]
Bamberg (Sternwarte)	[1950-01-01 - 1950-12-31]

Table 4.5: List of all *TimeStationAssignment*s in order of creation.

This greedy algorithm guarantees to choose the closest active *WeatherStation* for any point in time. Moreover, it covers the entire requested TimeSpan without any gaps provided there was an active *WeatherStation* at that time.

4.6 Processing all fetched weather data

The previous chapters cover the most interesting steps on how to choose required weather stations, find the right files containing their data and parsing them.

To give a good overview of the complete data flow that is triggered by a typical request for one weather parameter, figure 4.4 visualizes it using a Data Flow Diagram. One can comprehend all processing steps required to transform data files provided by the Deutscher Wetterdienst server into the JSON response that is sent to the user.

The goal is to create a list of *DataPoint*s for the requested parameter, one point per hour. The list must be sorted ascending by time and should not contain duplicate time stamps. As illustrated in the diagram, there are different processing steps required for forecast data and observation data. Thus, the next sections explain them separately until they are concatenated.

Processing of forecast data

The content of forecast files, or more precise MOSMIX files, satisfies almost all requirements already, hence only a few processing steps are required. The first two processing steps, unzipping the downloaded file and parsing it, are covered in chapter 4.3 in more detail. The short version is the MOSMIX file is provided in KMZ file format which needs to be unzipped to the KML format. That is a Extensible Markup Language (XML) file with standardized elements, so it can be parsed using any XML Parser. One file covers the prediction of the next 240 hours for all parameters at one MOSMIX station. This means after parsing, one has a mapping of each parameter to 240 *DataPoint*s.

The next step is to select the requested parameter from the map and use the *DataPoint*s for further processing. After that, they are filtered by the requested *TimeSpan*. Fortunately all forecast data is sorted ascending by time and measured in an hourly or longer time interval, so there is no further processing required to achieve the desired output format.

In case the selected MOSMIX station does not deliver any data, the application detects that and repeats everything with another station. This is illustrated by figure 6.3 in appendix A. Note though that this case should theoretically never occur, since forecast data is not measured but computed. Therefore, there is no such reason as a defect in the measuring instrument that could cause gaps or missing data. Nevertheless, the case is covered should it ever appear in the future.

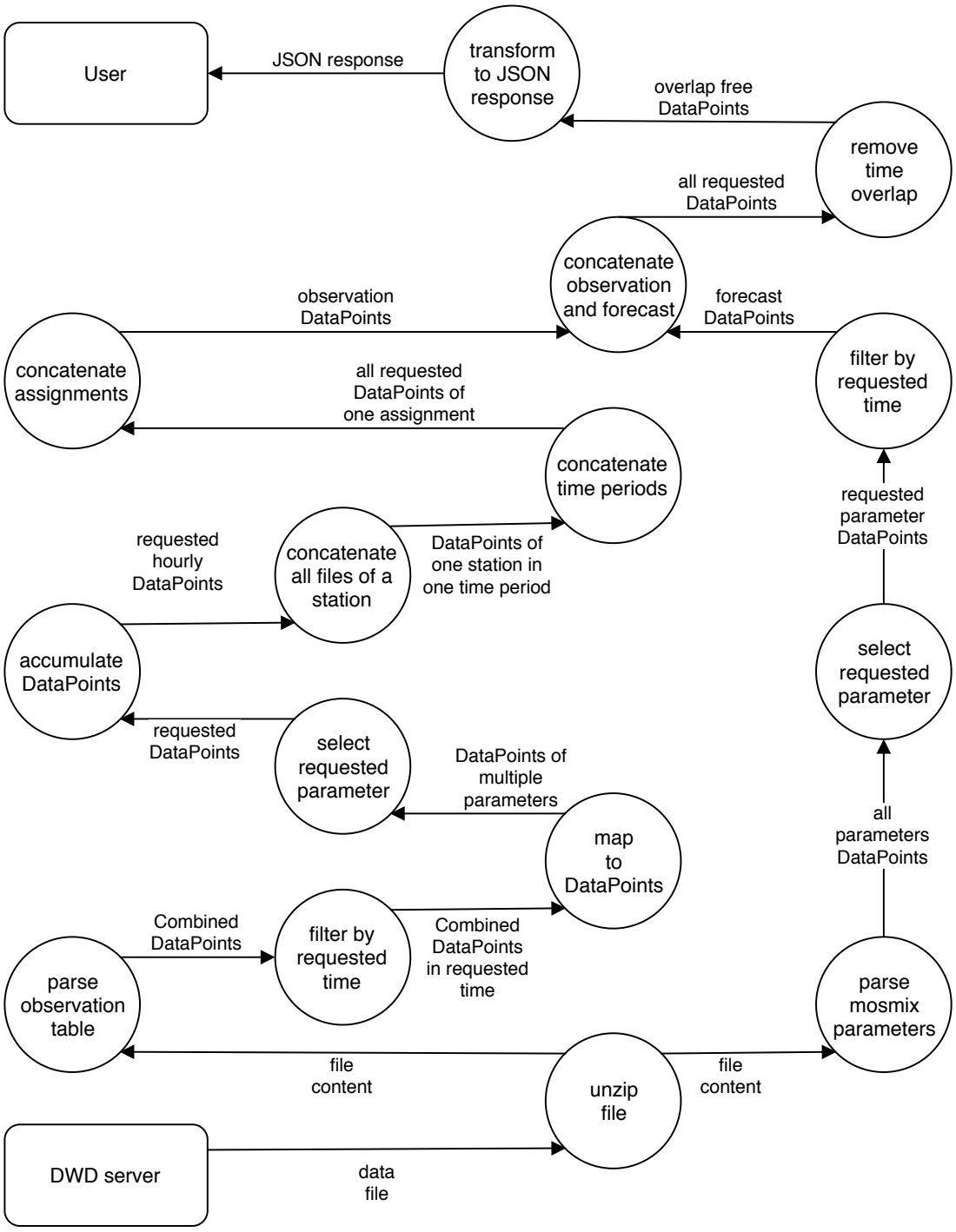


Figure 4.4: Data Flow Diagram of the microservice processing data from the DWD server to create the JSON response for the user.

Observation data processing

Processing forecast data is pretty much straightforward. Observation data on the other hand requires significantly more processing because of the wide variety of existing observation files. It starts similarly with downloading a data file and unzipping it. This is also covered by section 4.3 in more detail. In contrast to forecast, there are not all parameters contained in one file but only one combined parameter per row in the table. Remember, a combined parameter is a collection of a few parameters. A row is parsed into an intermediate *CombinedDataPoint*, which collects all parameters and their values at the time stamp of the row. Just like the forecast file, the table is sorted ascending already, so fortunately sorting of the *CombinedDataPoint*s is not necessary.

To minimize the amount of processed data as quickly as possible the *CombinedDataPoint*s are filtered by the requested *TimeSpan* next. This may be optimized even further if parsing of the rest of the row was skipped as soon as the parsed time stamp is not included in the requested *TimeSpan*. The implementation of this performance optimization messes up the simplicity of the parser too much to justify its use though. But if performance becomes a problem in the future, this is the best opportunity to gain some.

After filtering by time, the *CombinedDataPoint*s are mapped to their *DataPoint*s. There is one for each parameter that is included in the respective *CombinedDataPoint*. Only the *DataPoint* of the requested parameter is kept, all others are disregarded. Now having all *DataPoint*s sorted ascending during the requested *TimeSpan* there is just one little detail to be fixed. And that is the time interval of the *DataPoint*s.

Accumulating data

Observation data is provided in different time intervals by the DWD. Precipitation for instance is measured and uploaded once every minute. But it is also accumulated by the DWD to a measurement point every ten minutes and one every hour. Other parameters on the other hand are provided in a ten minute interval only and some only in an hourly interval. The goal is to deliver measurements in an hourly interval to the user. Therefore, every measurement that is not provided in an hourly or longer time interval, needs to be accumulated.

To be consistent with the DWD we use all *DataPoint*s measured in one hour to create the accumulated *DataPoint* of the next full hour. A *DataPoint* measured exactly on a full hour is used for the accumulated *DataPoint* of the respective hour. As a result all time stamps of accumulated *DataPoint*s are on a full hour, just like all forecast *DataPoint*s and all hourly measured observation *DataPoint*s. Therefore, any measurement delivered to the user is on a full hour, no matter where it originates from.

A different *AccumulationFunction* is required depending on the parameter. The function takes all *DataPoint*s of an hour and computes a new accumulated *DataPoint* from it. Our implementation supports:

- AVERAGE: used for air pressure, temperature and many more
- SUM: used for sunshine duration and precipitation height and duration
- MAX: used for maximum temperature and maximum wind speed
- MIN: used for minimum temperature

More functions can be added in the future by implementing the *AccumulationFunction* interface.

Collecting all data sources

After normalizing the observation data by previous processing steps, the data of all required sources needs to be collected. First up there can be multiple files on the DWD server for each weather station in the historical time period. They split the data by time, presumably to reduce file size. This is only done for historical data that is measured in a ten minute or shorter time interval because those usually contain the most measurements. Each file covers a different part of the weather stations active time and needs to be concatenated again. To make sure all *DataPoint*s are sorted ascending by time, some properties of the DWDs file structure are used. The files are sorted alphabetically. The name of each file consists of the unique id of the respective weather station as well as the start and end time of the measurements in the file. Therefore, the order of the listed files of a weather station is equivalent to the order in which the *DataPoint*s need to be accumulated.

For each station there may also be a data file in recent and now time period in addition to the historical file. The covered time of each period is listed in chapter 4.4.1. After concatenating all *DataPoint*s of these files with the concatenated ones from the historical files, one has all *DataPoint*s from one station during a requested *TimeSpan*. Since the historical and recent time period do overlap, the *DataPoint*s do as well. This is fixed later after another overlap is introduced. If one removes all overlaps at this point, the work will be done twice.

However, more than one station may be required to cover the entire requested *TimeSpan* due to the limited active time of weather stations. If a *TimeSpan* is requested that exceeds the active time of the closest weather station, the microservice finds the next closest weather station that can cover at least a part of the left *TimeSpan*. This is repeated until the whole *TimeSpan* is covered. The algorithm doing this, including a concrete example case, is explained in chapter 4.5. As a result it creates *TimeStationAssignment*s. Each of them assigns a

TimeSpan to a *WeatherStation*. In the final processing step of observation data, all assignments need to be concatenated as well. In order to ensure the concatenated result is sorted correctly, the assignments are sorted by their *TimeSpan*s. Since all assigned *TimeSpan*s are disjoint, one can use their start time to sort them. This way all *DataPoint*s of the assignments only need to be concatenated in the order of their assignments.

Finally, the observation *DataPoint*s and the forecast *DataPoint*s are concatenated, too. This introduces another overlap in the data due to the compensation that is done when fetching forecast data. To fill the gap to the latest observation data we fetch forecast data starting six hours before the time of request. At this point one has all requested *DataPoint*s sorted ascending by time including some overlaps that will be removed in the next step.

Removing time overlaps in data

The last processing step is to remove all overlaps in the data. Overlaps are introduced in two places. The first one is caused by the time overlap of the historical and the recent time period. Recent denotes the last 500 days until yesterday. Historical denotes everything ever measured until the end of last year. This theoretically means the longest possible distance between the last historical measurement and yesterday is one year. This implies an overlap of at least 136 days and up to 500 days if the historical data was updated yesterday. In practice though it is a few months shorter because the DWD does not move all the data from one year into their historical data over night on the last day of the year. For instance the move of data from 2018 has been completed around March 2019.

The second overlap is introduced between data from the now time period and the forecast. Now data is measured at the day of the request. It is not updated every time there is a new measurement but only every couple of hours. Forecast data is updated in a similar way and therefore usually contains the data that is missing between the last measurement in now and the actual time of the request. If it may be required we fetch all available data from the forecast file. This does not just close the gap but also leads to a possible overlap of a few hours.

Time overlaps contradict the goal to have no duplicate data. Since all *DataPoint*s are basically sorted ascending, one can use this property to remove all overlaps very easily. No duplicates and an ascending order mean that every valid *DataPoint* has a time stamp that is after the time stamp of the last valid *DataPoint*. One can assume the first one is valid. Therefore every following *DataPoint*, that does not meet the condition, is disregarded.

After this final processing step the *DataPoint*s are transformed to a JSON response, which is easy to parse for the user. This is explained in section 4.7.

Improving performance

Finding the right files is almost negligible compared to actually downloading and processing them when it comes to performance improvements. In the very early stage of implementation all processing steps described in the previous section were done one after another. The file was downloaded. Then it was unzipped and parsed to a list of *DataPoint*s. For every processing step the list was iterated and the result was added to a new list. Concatenation of partial results was done by adding them to a new bigger list. As list implementation we chose Java's *LinkedList*[8].

While this solution is easy to debug, it brings some significant performance drawbacks. On the one hand all created lists are in memory at the same time. Depending on the request there can be over 20 lists, if data from multiple stations and time periods is required. This adds up to over 1.5 GB of system memory usage for every request just for all the lists.

On top of that, concatenating partial results was computationally very expensive. As it turns out this is caused by the implementation of *LinkedList*. This list implementation was chosen because we assumed it could concatenate another *LinkedList* with constant effort by using its *addAll* method. Apparently it does not do so but rather just iterates the list and adds every single element one by one. Just because it could be done in theory, doesn't mean it is done in practice. However, this can be improved by using Java's *ArrayList*[9] implementation. This one at least uses *System.arraycopy* but requires transforming the list to an array first. This again creates a shallow copy of the underlying array of the *ArrayList*. Therefore, even this improved implementation is far from optimal.

There are two things that needed to be improved. First, ideally only the data that is currently processed shall be kept in memory. As soon as it is filtered out, mapped or accumulated it shouldn't be kept in an old list anymore. And second, concatenating data sources should be done with constant effort. The data size is too large to be iterated every time it needs to be concatenated.

What we came up with is a data pipeline. One might already have guessed that from the concept of processing steps, which are typical for pipelines. So instead of downloading the whole data file, then parsing it line by line and then processing the whole list of *DataPoint*s, only a single line is downloaded at a time. The line is parsed to a *DataPoint* which is then further processed before the next line of the file is even downloaded. This can not be done with the small forecast data files due to their format though.

[8] `https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html`
[9] `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

All processing steps are lazy evaluated. This is implemented using Java *Stream*s[10] for parsing, filtering and mapping steps. Concatenation and the more complex processing steps to accumulate *DataPoint*s and to remove time overlaps is done using *Iterator*s[11]. Concatenating could also be done in constant time with *Stream*s but since accumulation yields an *Iterator* it would be inefficient to transform it back to a *Stream*.

*Stream*s are great for stateless processing steps like filtering or mapping. But even though they do provide ways to do it, implementing processing steps that require a state contradicts the functional programming idea of no side effects. Therefore, we switch to traditional *Iterator*s for these processing steps. Java provides a method[12] to transform a *Stream* into a lazy evaluated *Iterator*. Similarly the accumulation of *DataPoint*s and the removal of overlaps is implemented to only process one *DataPoint* ahead.

Usually pipelines are constructed with each processing step running in parallel but we decided against that. Processing each step in parallel means overhead for every single step and every single *DataPoint*. The most obvious overhead is introduced by thread safe queues that are required to move the data between processing steps. Since each step does not require a lot of processing -and there are literally a million *DataPoint*s that need to go through a dozen of steps per request- the overhead may become quite significant.

The benefit of parallel processing is that one can process different steps for multiple *DataPoint*s and lines at the same time. But since a single request is limited by the Internet bandwidth and responsiveness of the DWD server already, faster processing at higher cost does not improve the microservices overall performance. Because of the introduced overhead it might actually make it worse.

We do use parallelism where it makes sense though. For instance all download requests of the required files are established in parallel, this way the delay between request and start of the actual download does not add up with the number of required files. This improves response times of the microservice by up to 200 ms at a ping to the DWD server of about 15 ms depending on the amount of required files.

[10] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

[11] `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`

[12] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#iterator--`

4.7 Definition of the user interface

As stated in requirement F5 from section 2.1 the microservice needs to provide a user interface. It should be realized by an easy to use REST interface with responses in the well known JSON format. Users of the microservice benefit from this technology stack because it sets the barrier to access the service to a very low level. So low that interested users may even just use a web browser, type in the address of the microservice, choose one of the routes below and try it out. One does not need to be a computer scientist to do this.

This simplicity is owed to Roy Fielding, who created the Representational State Transfer standard in the late 1990s [FT00]. Since then it has had a significant impact on standardizing the Internet and has emerged as a standard for web interfaces. For data serialization there are two commonly used standards in web services, XML and JSON. However, according to a case study [Nur+09] by Nurseitov et al. XML is "primarily used for remote procedure calls" whereas JSON "is designed to be a data exchange language which is human readable and easy for computers to parse and use". In addition to this, results of the case study "indicate that JSON is faster and uses fewer resources than its XML counterpart". Hence, JSON is the better choice for our needs.

Without further ado the rest of this section describes all endpoints provided by the microservice. Since the service is an adapter to the read only DWD server, HTTP GET is the only request method that is being used here.

This section is just a quick overview of all available routes and what they do. For an extensive documentation including the exact format of the responses with examples, check out the Swagger documentation provided by the microservice on the "/swagger-ui.html" path. An excerpt is given in appendix B.

Info routes

API Route	Description
/api/v1/info/timespans	Get a list of all predefined time spans
/api/v1/info/parameters	Get a list of all weather parameters
/api/v1/info/parameters/{parameter}	Get detailed information about a weather parameter
/api/v1/info/stations/{parameter}	Get a list of all weather stations
/api/v1/info/stations/{parameter}/{stationId}	Get meta data about a specific weather station

Table 4.6: List of all info routes.

As described in section 4.4 there is a set of predefined time spans (see table 4.3). This set is provided by the very first route of table 4.6. Users can use the name of a predefined time span instead of manually providing a start and end time in a query. This makes the usage of the service more comfortable and helps querying the latest data by using a static URL.

The next two routes help to acquire some extra information on weather parameters. Users can fetch the list of all supported parameters (also listed way back in table 4.1), which comes in handy because most other routes require to know the correct name of the desired parameter. And for each of those, a short description of what the parameter measures and the unit of its measurements can be queried, too.

A full list of all weather stations that measure a certain parameter can be fetched using the fourth route. Each of the listed weather stations is composed of the unique station id, the name of the weather station and its position. Where the position is defined by latitude and longitude as well as the station height in meters above sea level. This route combines forecast and observation weather stations. As a general rule of thumb most forecast weather stations contain a letter in their station id whereas all observation stations only use digits in their id. By adding the station id of an observation station to the route, one can fetch its meta data. There is no meta data for forecast weather stations available. Section 3.5 explains a little bit more about meta data.

Standard routes

API Route	Description
/api/v1/current	Get the latest weather data of today
/api/v1/forecast	Get the forecast weather data

Table 4.7: List of standard parameters routes.

Two very useful routes are listed in table 4.7. Even though the microservice is focused on delivering data for one weather parameter at a time, the current and the forecast routes allow to fetch a whole set of standard parameters in one request. Those include the temperature, air pressure, precipitation, wind and the sunshine duration parameters. The current route yields the latest available measurement for each of these at a specific location. The forecast route on the other hand yields the whole ten day forecast at a location.

Locations can be specified by setting the *city* query parameter to either the locations name or its zip code. Alternatively, one can also use coordinates by setting *lat* to its latitude and *lon* to its longitude. So for instance users can fetch the current weather in Erlangen using the URL *.../api/v1/current?city=erlangen*.

This request is equal to *.../api/v1/current?city=91052* and may also be expressed by using coordinates: *.../api/v1/current?lat=49.5981187&lon=11.003645*.

Historical routes

API Route	Description
/api/v1/historical/{parameter}	Get data for one weather parameter of today
/api/v1/historical/{parameter}/{time}	Get data for one weather parameter at a certain time
/api/v1/historical/{parameter}/{start}/{end}	Get data for one weather parameter in a defined time span

Table 4.8: List of historical routes.

This location specification needs to be used in all following historical routes, too. They aim to provide weather data for a single weather parameter, during a specified time at a certain location. Users can omit a time specification to query data of today. This not only consists of measured data until now but is supplemented with forecast data, yielding 24 values in total.

This is equal to using the predefined time span *TODAY* in the second historical route. Of course the name of a predefined time span does not need to be in capital letters to be recognized by the service. As an alternative, users can also set the time parameter to a specific date. This is interpreted as a request for data starting at the specified time and covering everything until now.

There are two valid formats to define a time. One way is to use the standardized format defined by ISO 8601 [org04]. As an example February 3, 2001 at 4:56 am is denoted as 2001-02-03T04:56:00.00Z where the zeros for seconds and milliseconds may be omitted. The other way is the short version of the standard following the same pattern but without separators. The example date is denoted as 200102030456 in the short version. The specified time is always interpreted as a UTC time. Everything but the year may be omitted here and is replaced by the earliest possible time that fits the given specification. For instance 2019 is expanded to 201901010000, which represents January 1, 2019 at midnight.

The last route requires a start and an end time in addition to a weather parameter and a location. This route fetches all available data from all known sources and connects it to cover the complete defined time span. There are no limits introduced by the microservice with this route. All weather data that is supported by the microservice is accessible from this route.

4.8 Testing the microservice

Testing a software is at least as important as actually developing it. There are two typical work flows here. The first, and usually considered the better one, is to develop software using the principles of Test-Driven Development (TDD). Kent Beck, the inventor of TDD, defines two simple rules for it:

- Write new code only if an automated test has failed
- Eliminate duplication

Based on these, one can derive the whole idea of TDD. A new test is written before the tested feature is implemented, ensuring it behaves according to its specification. Then it is actually implemented until the test does not fail anymore. Finally, the code is cleaned up. From there one continues with writing the next test and so on [Bec03]. This development style shifts the main focus from the application implementation to its testing. Thus, it solves the problem of tests being written only to prove that a given peace of code works.

The other way to get a software tested is by doing it the other way around. A feature is implemented first and subsequently tested. This is what a programmer intuitively does if given the task to solve a problem. Programmers are problem solvers. Thus it is tempting to build a solution and then verify that it indeed works by adding biased tests that cover the inputs one had in mind during implementation. In this order it is difficult to keep the idea of testing in mind.

> "Testing is the process of executing a program with the intent of finding errors." [Mye+04]

Trying to break something, that one build just a few minutes ago, is simply counterintuitive. Hence, tests written after implementation easily tend to only prove that a given peace of code works.

Apart from that, there is one advantage of this method over the first one. If one does not exactly know how the solution to a problem looks like before starting the implementation, one often has to change the code multiple times before it is clean and works properly. This may be caused for instance by extracting code duplications into helper methods. If tests are written before the set of required methods and their specification is completely clear, tests will have to be rewritten, too. This can make the TDD method very time expensive. Sure, one can argue this is only due to a bad design, but in reality there often is not enough time to think about every little detail of the implementation before starting it.

This little introduction into the section only gives a brief summary of our mindset on the topic of testing. One can fill books discussing best practices for software testing, so theory aside, let us see how the microservice is tested.

The microservice and its components are tested in three levels. First, all classes and methods are covered by unit tests, as requested by requirement Q1 from section 2.2. Second, the combinations of components are tested using integration tests. Finally, interface tests ensure the compatibility of the microservice with the interface of the DWD server. Tests are written using the well proven JUnit[13] test framework, using its test suites to group related tests together. TDD was not used, but as described above, tests are written with the intention to find errors.

Unit testing all components

The idea of unit testing is to test each component of the program individually to ensure it works as specified. Unfortunately most components are tied into the rest of the program in a way that they depend on other components to function correctly. In order to properly unit test each component, one has to isolate it from its environment. This can be done by creating Mock Objects to replace the environment of the component under test.

> "A Mock Object is a substitute implementation to emulate or instrument other domain code." [MFC00]

In the application of the microservice, access to every component is established using Spring Beans. The Spring framework automatically injects these into all dependencies of the components. Now that we want to unit test each component, we theoretically only have to replace every Spring Bean with its Mock Object. Unfortunately though, dependency injection in Spring only works for objects that are provided as a Spring Bean themselves. After replacing all Spring Beans with their Mock Objects, none of the components is able to inject their dependencies automatically anymore. Therefore, it is necessary to manually set them when creating the test object.

Our created Mock Objects implement the interface of the component it is mocking in a very simple but highly configurable way. Before each test one can set it up to deliver a specific result, count method calls or even throw a specific exception. This allows testing scenarios that can usually not be created in a test environment and only very rarely occur in production.

For example when testing the cache for the location resolver, the underlying location resolver is mocked and set up to count method calls. Using this fully controllable environment, the test can now query a request on the cache multiple times and then check whether the cache actually works and only tries to fetch the result from the location resolver once. Without mocking the location resolver, one could not write such an important test unless the production code is extended with functionality that is only required by the test, which is highly discouraged.

[13] `https://junit.org/`

Using this technique our 365 unit tests can cover almost all lines and most of the branches as shown in table 5.1 located in section 5.2.

Integration testing the microservice

After having tested each component to work as specified, now the integration test shall ensure that all components work together as intended. This means one at least has to test whether every API route actually delivers valid data. The catch here is that all of this is supposed to happen without using any external services like the DWD server and the Nominatim geocoding service in every test run. This has several reasons. For one, every test run fetches several megabytes of data from the services. Second, the provided data changes over time. If a test fails and the data changes again, one may not be able to recreate the faulty test scenario to debug it. Third, we do not want to test functionality of a service that is not under our control here, since an integration test should not fail due to a change on an external service.

This indicates that both these services need to be mocked. Therefore, we create something similar to the Mock Objects used for unit testing, just as additional REST services instead of objects. To make the microservice use the mock services during testing, the service addresses are modified to point to localhost in the test configuration file.

Unfortunately it is not as easy to generate mock data as it is with the little Mock Objects during unit testing. Sure, one can set up the mocked Nominatim service to always deliver the response for one location. But the mocked DWD service has to provide dozens of proper files, each on a different route. Even if one invests all the work to create the files, what if the DWD service changes its interface? No one wants to manually adapt a route or a mock file in that case.

The easiest and least error-prone solution here is to do this automatically and create a snapshot. On the first run of the integration tests all requests to the mock services are forwarded to the actual services. The response is saved in a snapshot directory. From there the path to the respective file is equal to the path of the request. If one runs the integration tests again and the snapshot directory exists already, the mock services return the respective file content.

This solution requires fetching all the data once, but also has some advantages. First, the test data is real world data containing real world problems. Mocked data files may cover little anomalies that may result in bad behavior. Second, one can see what files are actually fetched from the server and what the content of these files is, making debugging a failing test and fixing it a lot easier.

However, this solution does also have a big drawback. Since the test does not know what the exact content of the files is, it can not check whether a specific

value appears in the result or not. The same applies to gaps in data. They may be valid because the microservice does not fill them with data from another weather station unless the gap covers the whole requested time span. This limits the tests in what they can assert. Nevertheless, they can test the functionality of each API route, ensure no unexpected errors are thrown, data is overlap free, sorted ascending by time, is not outside the requested time span and contains measurements from all requested origins. The biggest integration test queries data of the last 1000 days, today and three days of forecast for each of the twenty weather parameters.

Very attentive readers may wonder now, how can one fetch weather data of today and the forecast if all files are saved in a snapshot that may be days or maybe even weeks old? By the time one schedules the requests, all the forecast data has become recent data.

Fixing this problem is actually quite easy because the time of request has been very important throughout all data providing components. Getting the current time is simple but one must ensure different components do not mix up different request times for the same request. This may lead to inconsistent results. Thus, the time of request is determined only once. Subsequently it is distributed through all method calls. Additionally, wrapping the query for the current time in a Spring Bean allows mocking it in the test. This in turn allows setting the current time to any time one wants. In addition to the snapshot, the mocked service also remembers the time of the request, enabling us to successfully mock the time for the microservice. The dedicated Spring Bean to query the current time is the only required modification on the production code for all tests.

In conclusion we implement 42 integration tests that cover all provided functionality of the microservice as demanded by requirement Q1.

Interface testing third party services

In order to recognize and locate interface changes on either the DWD server or the Nominatim service, interface tests are added. These do not test our microservice but the compatibility with the services that it relies on. For example if the DWD decides to change the file path to one of its station list files, the microservice can not function correctly anymore. Therefore, a parameterized test is created that tries to fetch all station list files from the expected path for each known weather parameter. If one of them can not be fetched but all unit and integration tests do not detect any errors, one can determine the file path that must have changed very quickly.

This is not just done for station list files but for all other required files, too. The tests assert that there is at least one measurement for every parameter in *NOW*, *RECENT*, *HISTORICAL* and *FORECAST* time span, except we know

the parameter is not available in that time span. The Nominatim geocoding service is easier to test. We send a request to geocode Erlangen and expect the resulting *Coordinates* to be in a reasonable range around the city. Any compatibility issues will yield an exception and cause the test to fail.

Summing up, a total of 127 test cases ensure none of the used external interfaces changes. In case of a failing test this helps to quickly identify the changed resource.

4.9 Deploying the microservice

After implementing and testing all the microservices functions, it is time to think about deploying the application. Basically one can deploy it in any environment that has access to the Internet, Java 12[14] installed and about a gigabyte of system memory left. That is all it takes for the microservice to function.

At least that is the theory. Even though Java is platform independent and backwards compatible there are still many pitfalls that can cause malfunctions of the service. Some of them even appeared during development. For example the file path to the snapshot directory, used in the integration tests, initially could not be found on Linux systems but worked just fine on Microsoft Windows. Another problem occurred due to different Java compiler versions. A library, used for a practical *Pair* implementation, could not be resolved by the new compiler version because it apparently was excluded from Java's standard libraries in a recent update. All those problems are just too familiar to system integrators, who regularly experience effects like these after a software update or the migration of a system and its services.

Fortunately one can solve these problems using virtualization. There are two options, container virtualization using for instance Docker[15] or a hypervisor based virtualization technique like Xen[16].

Docker allows creating a container image of the application and its required environment.

> "A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings." [Inc19]

The container image can then be executed in virtualized containers on top of the host kernel, allowing the application to run in a different environment than the

[14] `https://www.java.com`
[15] `https://www.docker.com/`
[16] `https://xenproject.org/`

rest of the system. From the outside, a running container appears to be a normal process just like any other application.

Additionally, using a two-stage Dockerfile[17] allows to build the whole application without any build dependencies, like the Java Development Kit (JDK), installed.

Hypervisor based virtualization runs virtual machines, which do not rely on the host kernel but include a separate operating system with an own kernel. A hypervisor running on the host operating system manages the virtual machines with their guest operating systems. In general this approach introduces more overhead in form of system resources during execution as well as image size than container virtualization. While a Docker image only contains the application and the software for the environment, the image of a virtual machine also needs to include an entire operating system [Bui15].

Docker is available on Linux, Windows and macOS. Installation is not much more difficult than of any other program. It comes with all required tools and a great documentation to create and deploy containers very easily. In contrast, creating a virtual machine image and setting up an environment to deploy it is not that easy. Hence, it should be obvious now why there is a literal hype around Docker in the tech world. It serves the perfect platform for web services such as our microservice, which is why we also choose to use it.

[17] `https://docs.docker.com/develop/develop-images/multistage-build/`

4.10 Adding an adapter to the ODS

As part of this thesis the ODS shall be extended to use the new microservice as an additional data source. The ODS is an open source software project developed and maintained here at the Professorship for Open-Source-Software. It collects heterogeneous data from various sources and makes this data easy to consume through a unified programmatic interface.

In a typical use case one defines the name of a new data source first. Next a filter chain for the data source is added. It contains the name of the adapter to the data of interest and all required input parameters. Additional filters and an execution interval can be specified, too. This way a user can improve fetched data and tell the ODS to fetch new data once every hour. Finally, users can query the data, that is stored in a database once it is fetched from the data source.

Extending the ODS by an adapter to a new data source is very easy. After specifying the name of the input parameters and implementing an interface, the new adapter only needs to be registered, so the ODS can map the adapter name from the user input to the adapter class.

Due to the way the ODS works, it does not really make much sense to adapt to all functionalities of the microservice as requested by requirement F6 specified in section 2.1. Specific requests for one weather parameter at a location within a certain time span will usually not be queried more than once. However, the ODS would store the possibly huge result in its database waiting for someone to request it again. On the other hand, requests for current weather and forecasts are perfect for the ODS. For example users can setup a data source that yields the latest weather for Erlangen by adding a filter chain that schedules a new request for the current weather once every hour. The result is stored in the database and hence, can be queried by all citizens of Erlangen without creating a huge load on the microservice or the DWD. This way the ODS complements the microservice with a cache layer only for data that is actually worth caching it. Appendix C gives an example of how this can be done.

5 Evaluation

In chapter 2 we define the requirements to the microservice that is created as subject of this thesis. Here in this chapter we want to look back and evaluate our implementation of the microservice. If applicable we apply the evaluation scheme that is formulated with the requirement and check which of the requirements are fulfilled, exceeded and missed by our solution.

5.1 Functional requirements

Requirement F1: Fetch DWD weather data

This requirement set the expectations low because at the beginning of this thesis it was unclear what data is actually provided by the DWD. Therefore, only the current temperature and solar radiation must be available. All other data from the DWD server may also be supported as an option.

Fortunately we were able to find and support twenty different weather parameters. Most of them are available as historical, recent, now and forecast data. A complete list of the supported parameters is given in table 4.1 way back in section 4.1. So in this regard we were able to exceed the expectations. However, *total_solar_radiation* is the only solar radiation parameter that is available at the same day of measurement. At time of this writing the other two, *diffuse_solar_radiation* and *longwave_downward_radiation*, are not provided in now time by the DWD. Maybe this will change in the future. In that case it is very easy to add support for those in the microservice.

Additionally, the microservice should only download required files from the DWD server instead of crawling all files for the requested data. In order to be able to evaluate this, the URL to every downloaded file is logged.

The logs prove that the requirement is fulfilled for single weather parameter requests, using one of the historical routes. But they also reveal an inefficiency when it comes to requests of current and forecast data. As explained earlier, a single file usually contains measurements of more than one parameter. The mi-

croservice simply ignores this and downloads one file for each of the parameters, that are included in forecast and current requests. This possibly makes it download the same file more than once to get all data for one request. Especially in case of forecasts because every file contains all available parameters. This could be optimized in the future. Because all parameters are fetched in parallel and the files are very small, the optimization will not noticeably improve response times but only decrease the amount of used system resources. Even though this scenario has not been optimized, we still consider this second part of requirement F1 as fulfilled.

Requirement F2: Transform DWD data to own weather model

Requirement F2 is addressed in section 4.2. The implemented parsers extract all information from the data file and create a *DataPoint* object for each measurement. Each object combines:

- the time stamp of the measurement
- a reference to the weather station that measured it
- the measured weather parameter
- the time interval of measurements in order to accumulate it to hourly
- the origin of the measurement:
  FORECAST, NOW, RECENT or HISTORICAL. If the time interval of the *DataPoint* is modified, it changes to ..._ACCUMULATED
- the measured value

The unit of the measurement is included in the *WeatherParameter* class as each one defines the unit of all its measurements. Forecast and observational data sets may have different units for each parameter. This inconsistency is managed by the *WeatherParameter* class itself. It knows what unit is used in which *DataSet*. When parsing a value it is directly converted to the standard unit of the microservice for that weather parameter.

This format has proven its ease of use throughout the implementation of the microservice. Responses send to the user only omit fields that are redundant with the request and optimize the output JSON format to serialize fields like the weather station only once. The conversion to the specific response format has a performance impact of linear time and space complexity. It is the only processing step that requires holding all *DataPoint*s, that are included in the response, in memory at the same time. Additionally, the constructed response object and the serialized JSON need to be in memory, too, causing a peek in memory usage for every big request. Fortunately though, memory usage during all other processing steps is negligible as every downloaded and parsed line is immediately processed

all the way until this very last step usually filtering out most of the data as soon as possible.

Since each *DataPoint* carries all information, is easy to use and performs well in the microservice, we consider requirement F2 as fulfilled.

Requirement F3: Reasonable response time for weather requests

At beginning of this project we could not really estimate how long it would actually take to fetch several files from the DWD server. There were many possible bottlenecks like the response time and bandwidth of the DWD server, the complexity of the required processing steps and so on. Requirement F3 was added in order to ensure response times for requests of current weather are reasonable. Users should be able to query the current weather for one location 25 times per minute.

In section 3.4 we explain how locations of the Nominatim service are cached and how the lists of weather stations are cached to improve response times. Additionally, we added a couple of performance optimizations such as the data pipeline, explained in section 4.6, and parallel fetching of files. It also turned out the DWD server performs surprisingly well. This way, we were able to improve response times of the microservice to be far better than expected at the beginning of this thesis. Section 5.3 gives an impression of response times for typical requests.

To evaluate requirement F3 we restart the microservice and send a request for the current weather of Erlangen so the location and all required weather station list caches are filled. Then we run a loop sending a request for the current weather of Erlangen and receiving its response in every iteration. The microservice is running on the same system as during the performance evaluation in section 5.3. The loop is interrupted after one minute. As a result we counted more than 400 iterations in each of the three test runs, thereby easily exceeding the goal of 25 requests per minute.

Requirement F4: Location specification

To make our microservice more user friendly, one should be able to specify a German location either by a city name, a zip code or directly using coordinates compound by latitude and longitude. Fortunately we did not have to reinvent the wheel and create our own geocoding service to meet this requirement. Instead we rely on the Nominatim service, which is explained in section 3.3. It is capable of translating a city name or a zip code to coordinates. To find the closest weather station one has to compute the distance between the specified location and the weather stations, which is only possible with coordinates. Therefore, they are the standard format for the microservice to deal with locations, all other inputs are directly converted to coordinates.

Requirement F4 is verified by three requests of the same location, using city name, zip code and coordinates. This indeed yields equal data for an example request of todays temperature in Erlangen, 91052, (49.596361°, 11.004311°). Erlangen, as many big cities, actually has multiple zip codes causing the geocoded coordinates of the city name to vary a little from the coordinates of the zip code. As long as the variance does not lead to different weather stations being selected, the data will always be equal. Therefore, we consider requirement F4 to be fulfilled.

Requirement F5: Provide a user interface

Requirement F5 is similar to F1 in regards of what data shall be available using the user interface. It at least should provide access to measurements of current temperature and current radiation. Optionally forecasts and historical weather records may be provided as well, depending on what is actually provided by the DWD.

As explained in more detail in section 4.7 the API supports all twenty weather parameters in all time spans where measurements are provided by the DWD. Additionally, to make it more user friendly, a list of all weather parameters, a list of all weather stations for a parameter and a list of all predefined time spans is provided by the info routes. And because it may be important to some users of the microservice, one can also fetch all meta data of a parameter for observation weather stations. In conclusion the API allows access to all supported data, thereby exceeding requirement F5.

Requirement F6: Adapter for the ODS

After the microservice is done, an adapter shall be added to the ODS so it can make use of the new data source. Requirement F6 wants to enable the ODS to access the entire functionality of the microservice.

At time of requirements specification it was not clear what functionality the new service can actually offer. Requirements F1 and F5 only specify it shall at least provide access to current temperature and current radiation data. Since we exceed both requirements there is a lot more functionality than what was expected when specifying F6.

When it was time to implement the adapter, we figured it does not make much sense to add certain functionalities to the ODS because of the way it works. It is not designed for one time requests of huge amounts of data but rather to query repetitive requests like the current weather or forecast for a city. As explained in section 4.10, this is why we decide to only support access to the current and to the forecast route of the API and not support any of the info or historical routes. It is not difficult to add support for those routes, too, but we accept to miss requirement F6 due to these circumstances.

5.2 Non-functional requirements evaluation

Requirement Q1: Test coverage

Of course the implemented software needs to be tested, too. Requirement Q1 was added, in order to define a guideline of how it should be tested. It asks for structural coverage by unit tests, integration tests that cover all system functionalities as well as interface tests that ensure compatibility with all third party services. Section 4.8 explains the implementation details of the tests.

Structural coverage comprises 100 % line and branch coverage. A line is covered if it is executed during at least one test. A branch is created by conditional statements like if conditions and loops. In order to cover a branch the condition has to be evaluated to both true and false by the tests. It is not covered if all tests cause the condition to be evaluated to only one of the two.

During development a total of 365 unit tests have been written achieving the coverage listed in table 5.1. Each line in the table contains the coverage of one of the top level packages, where total sums them all up.

Package	Method	Line	Branch
crawler	99 % (333/334)	99 % (1281/1293)	59 % (118/197)
model	99 % (188/189)	99 % (590/593)	77 % (104/134)
request	100 % (43/43)	100 % (197/197)	88 % (31/35)
rest	100 % (54/54)	100 % (195/195)	48 % (13/27)
spring	95 % (23/24)	97 % (42/43)	100 % (0/0)
util	100 % (10/10)	100 % (34/34)	72 % (8/11)
total	100 % (651/654)	99 % (2339/2355)	68 % (274/404)

Table 5.1: Structural test coverage from unit tests.

The unit tests can almost cover all lines but not all branches, thereby not quite reaching the full structural coverage we aimed for. Covering the missing branches just seems unreasonable for this project, since it is not a safety-critical system. We consider the software to be well unit tested, so we see the first criteria to be fulfilled.

In addition to the unit tests there is a total of 42 integration tests. These test all API routes querying every single weather parameter as well as all location input methods. The tests cover all the way from fetching the right file and parsing it to the assembly of the finished response. This fulfills the second criteria of requirement Q1.

Finally, 127 interface tests ensure none of the used DWD file paths or file formats has changed. Because we additionally rely on the Nominatim service, it is checked

to deliver responses in the expected format for our requests, too. These tests will notify us if there is a major change at one of the external services that might break the compatibility with them, requiring a change on the microservice. This fulfills the third criteria of requirement Q1.

In conclusion we fulfill Q1 with a total of 534 test cases.

Requirement Q2: Accessibility

The service shall be easy to use and accessible for as many people as possible. Thus, requirement Q2 asks for documentation of all system functions and an easy to use interface.

This thesis is part of the documentation. Especially section 4.7 explains all routes of the REST interface. Additionally, users can query the Swagger documentation provided by the service on the "/swagger-ui.html" path. It contains an explanation and an example response for each route. Every information one needs to send requests is documented there. Appendix B shows an example excerpt of the Swagger documentation.

The easiest way for users to access the microservice is using a simple web browser. One only has to type in the address of a running service instance together with a valid route and the response will show up as human readable JSON text. This can be done by anyone and does not require any special knowledge. Therefore, we consider requirement Q2 to be fulfilled.

Requirement Q3: Deployment

Section 4.10 covers more details on the deployment of the microservice. Requirement Q3 wants the deployment to be easy and makes the service run in a controlled environment. To address this requirement we add a Dockerfile to the project which can be used to create a Docker image that in turn can be run in a virtualized container. Docker is available on all common platforms and is very user friendly. The container guarantees the software environment of the running service to be the same no matter on what machine it is running. This satisfies requirement Q3.

5.3 Microservice performance evaluation

Finally, we want to evaluate the performance of the microservice. There is no requirement with an acceptance threshold because one really could not estimate what to expect at time of requirement specification. In order to evaluate the performance of the service there are two crucial metrics:

- The response time, measuring the time that passes between sending a request and starting to receive a response
- The size of the response, determining how long it will take to fully receive it depending on the bandwidth of the user and the service

We test these with four different routes to get an impression of what users can expect. Due to significant fluctuations in response times for different locations and parameters one would have to test far more scenarios to get a precise performance measure. For each route we send five subsequent requests. The service is restarted before the first request is send, so all caches are empty.

The first request queries Berlin as location, the other four query data for Erlangen. We do this because the very first request has to fetch all weather stations for every requested weather parameter. These are cached for all following requests. We then change the location to Erlangen, so the second request has to resolve the coordinates of the location. The final three requests can make full use of the cached data and optimizations like the modified active times for required weather stations. This prevents the algorithm from fetching data of a station that apparently has a gap covering the requested time span. In that scenario it reselects the weather stations and fetches all data again.

We choose four typical routes to be requested:

- The current weather
- A ten day forecast
- Air temperature between five years ago all the way until ten days in the future, covering historical, recent, now and forecast data (combined)
- Air temperature between the years 1980 and 2000 (historical)

For this evaluation the requests are sent from the system the service is running on, eliminating additional latency between service and user. The system features Windows 10 (version 1803), an Intel Core i7 8700K and an Internet Bandwidth of 35 MBit per second in download direction. The ping to the DWD server is 15 ms.

Figure 5.1 shows the results grouped by the requested route. The bars are sorted from first request to last.

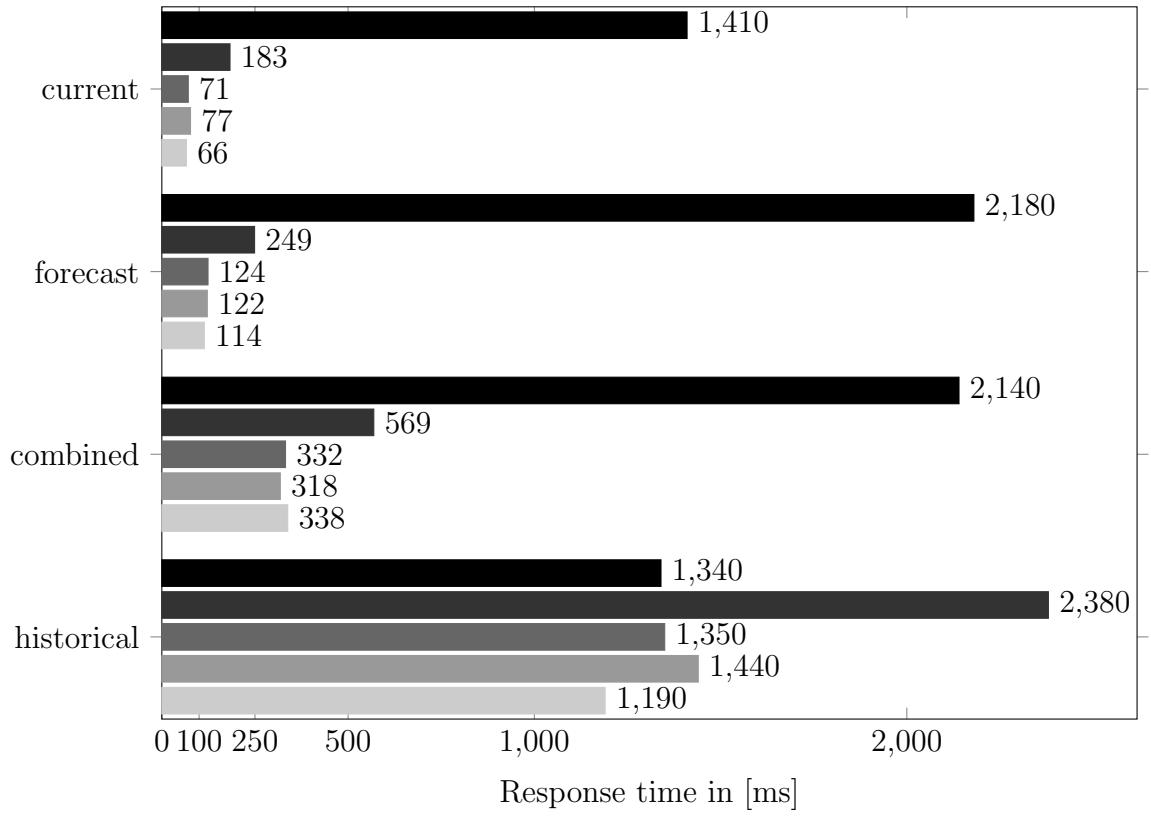


Figure 5.1: Response times of current, forecast, combined (about 5 years in all time sections) and historical (1980 - 2000) requests. One request for Berlin followed by four requests for Erlangen.

In general one can see that the very first request yields a significantly higher response time than its subsequent requests because the list of all weather stations needs to be fetched first. The second request is also a little bit slower than the final three because the location changes to Erlangen. This requires querying the Nominatim service and in case of the historical route a modification of a stations active time along a second fetching iteration. Once all that is done, the following three requests yield the best and pretty constant response times.

Especially current and forecast response times are very fast. Due to response sizes of 2.5 KB for current and 142 KB for the ten day forecast most users will have to wait less than 200 ms to finish receiving the results. According to [Nie93] this is fast enough to barely notice any lag at all.

The combined request queries temperature data of over five years. The response is about 3.5 MB in size. We consider downloading this plus half a second response time is still very well acceptable for the amount of data the user receives.

The historical request queries data of twenty years. The very first request for

data from Berlin is almost twice as fast as the second one from Erlangen. For Berlin a single weather station covers the entire twenty years, whereas Erlangen requires data of three stations. This is actually not the cause of the much higher response time because all files are fetched in parallel and the Internet bandwidth is not bottlenecking.

In fact one of the three selected stations does not deliver any temperature data during the time span it should have covered. Therefore, its active time is modified, the stations are reselected and fetched again. The second attempt relies on two weather stations, which both deliver data. So the second iteration of fetching is responsible for the almost doubled response time. The three following requests do not need to reselect any stations, so they are almost twice as fast again.

The request for Berlin yields a response size of 14.1 MB but the request of Erlangen only 6.4 MB. It turns out one of the two selected weather stations for Erlangen does not provide any measurements between the years of 1988 and 2006. One can find that out by querying the stations meta data as explained in section 4.7. This gap in data can be closed by sending a request for temperature data in Erlangen between 1989 and 2000. This way the microservice recognizes the station does not deliver any data and modifies its active time. After that a request for temperature between 1980 and 2000 in Erlangen also yields about 14 MB of data.

This example request shows that historical routes have a higher response time but the response size outweighs this because downloading the response takes a lot longer than that. This reveals the disadvantages of choosing a human readable response format like JSON. However, we consider the response size to be acceptable for these kind of requests. They are not meant to be queried on a regular basis but more likely be queried once and then analyzed.

Summing up one can say the microservice performs far better than we anticipated. Common requests for current and forecast data are responded almost free of noticeable delay and even requests for data from multiple years are responded within seconds instead of minutes.

6 Conclusion

We implemented a new microservice that adapts the complicated and inconsistent interface provided by the DWD to an easy to use REST interface. Furthermore, we presented all steps of the development process as part of this thesis.

First, we specified functional and non-functional requirements that the created software should fulfill at the end of the project. Chapter 3 then explains what technologies are used and how the systems architecture has been designed. Details about the implementation process are revealed in chapter 4. We thoroughly explained the problems that occurred during implementation and presented custom algorithms and data structures that solve them. After implementation was finished we evaluated the previously specified requirements.

Overall we met most requirements and even exceeded in some points. We only missed requirement F6. The developed ODS adapter does not provide access to all functions of the microservice as demanded in the requirement. We figured some of the functions do not fit the design of the ODS, and hence should not be added. Therefore, the ODS adapter is limited to requests of current and forecast weather, whereas the microservice additionally gives access to historical and meta data.

In section 5.3 we gave an impression on how well the service performs for some typical requests. We showed that users can request current and forecast weather for any location in Germany in a fraction of a second and only have to wait a few seconds for queries of data covering multiple years.

Users can query a total of twenty different weather parameters and use either the city name, zip code or coordinates to specify the location. Data is delivered in an hourly time interval starting from whenever the DWD provides data all the way to a ten day forecast. The provided JSON responses are human readable and easy to process for data analysis. All this makes open weather data easy to consume for both humans and systems like the ODS.

Appendix A Sequence diagrams of a request

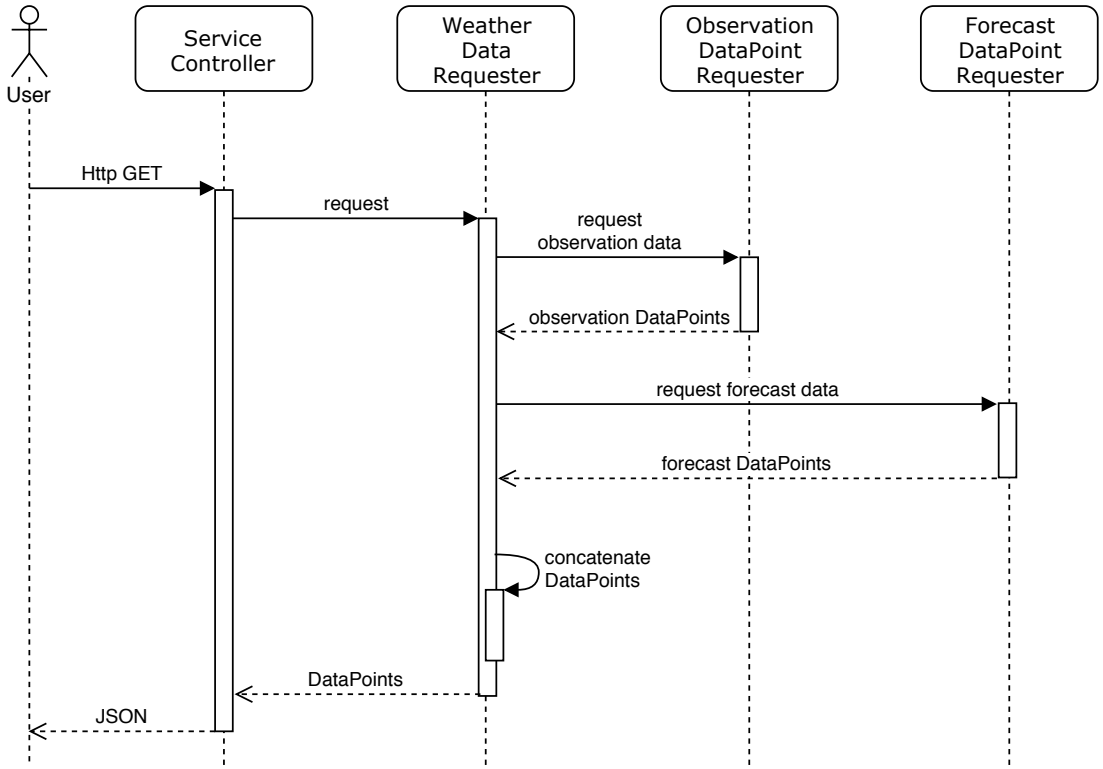


Figure 6.1: Sequence diagram of a user request being forwarded down to the *ObservationDataPointRequester* (illustrated in more detail by figure 6.2) and *ForecastDataPointRequester* (figure 6.3.)

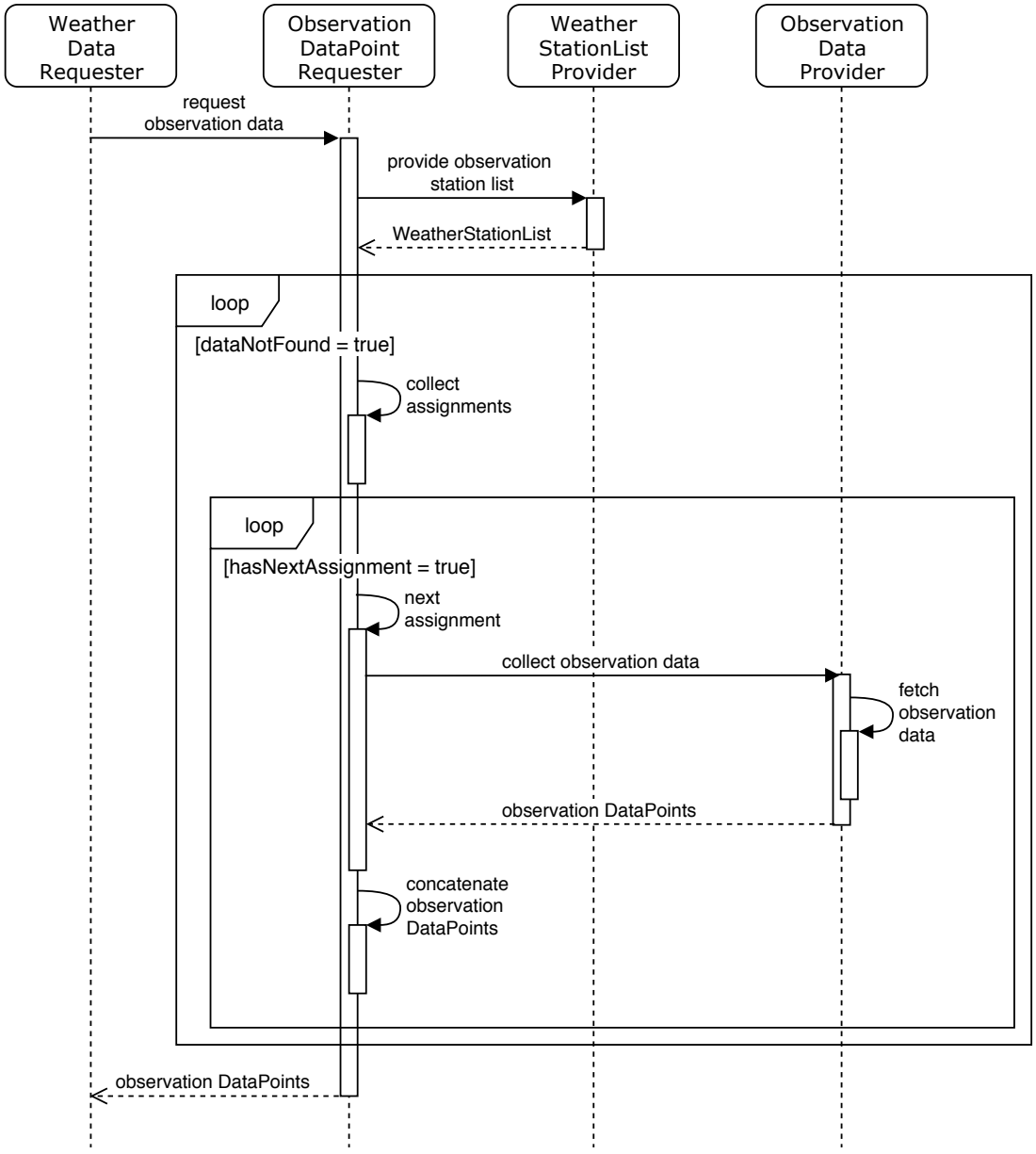


Figure 6.2: Sequence diagram of the *ObservationDataPointRequester* fetching assignments using the *ObservationDataProvider* until all assignments provide usable data.

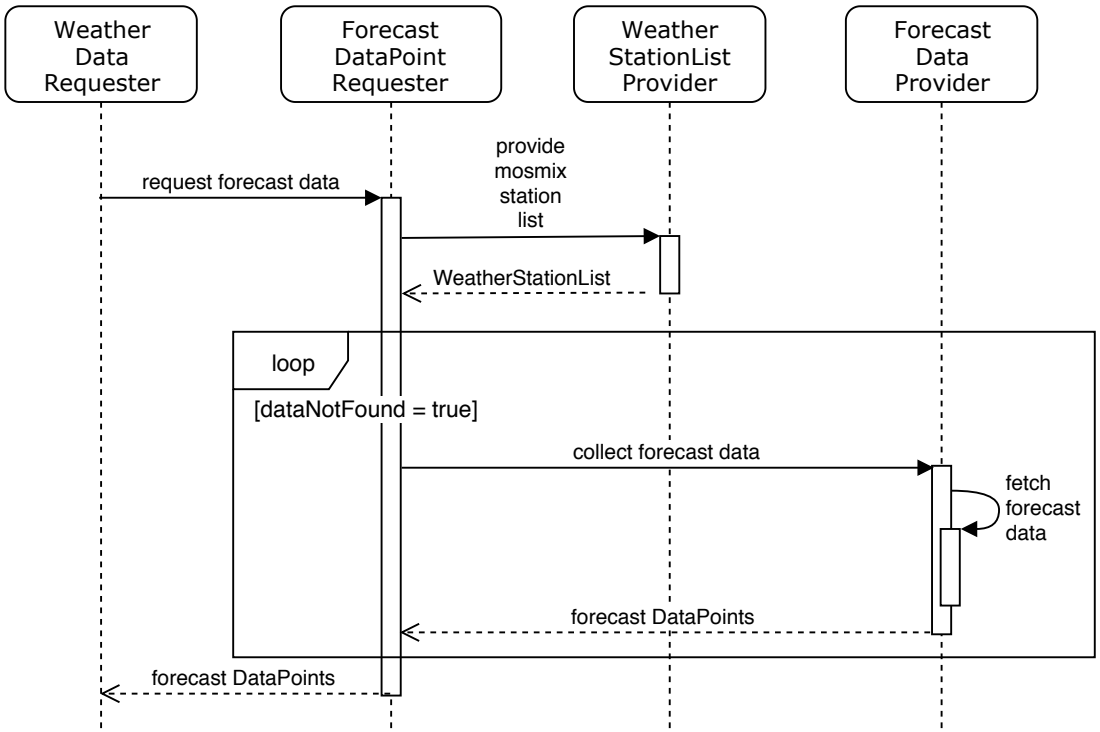
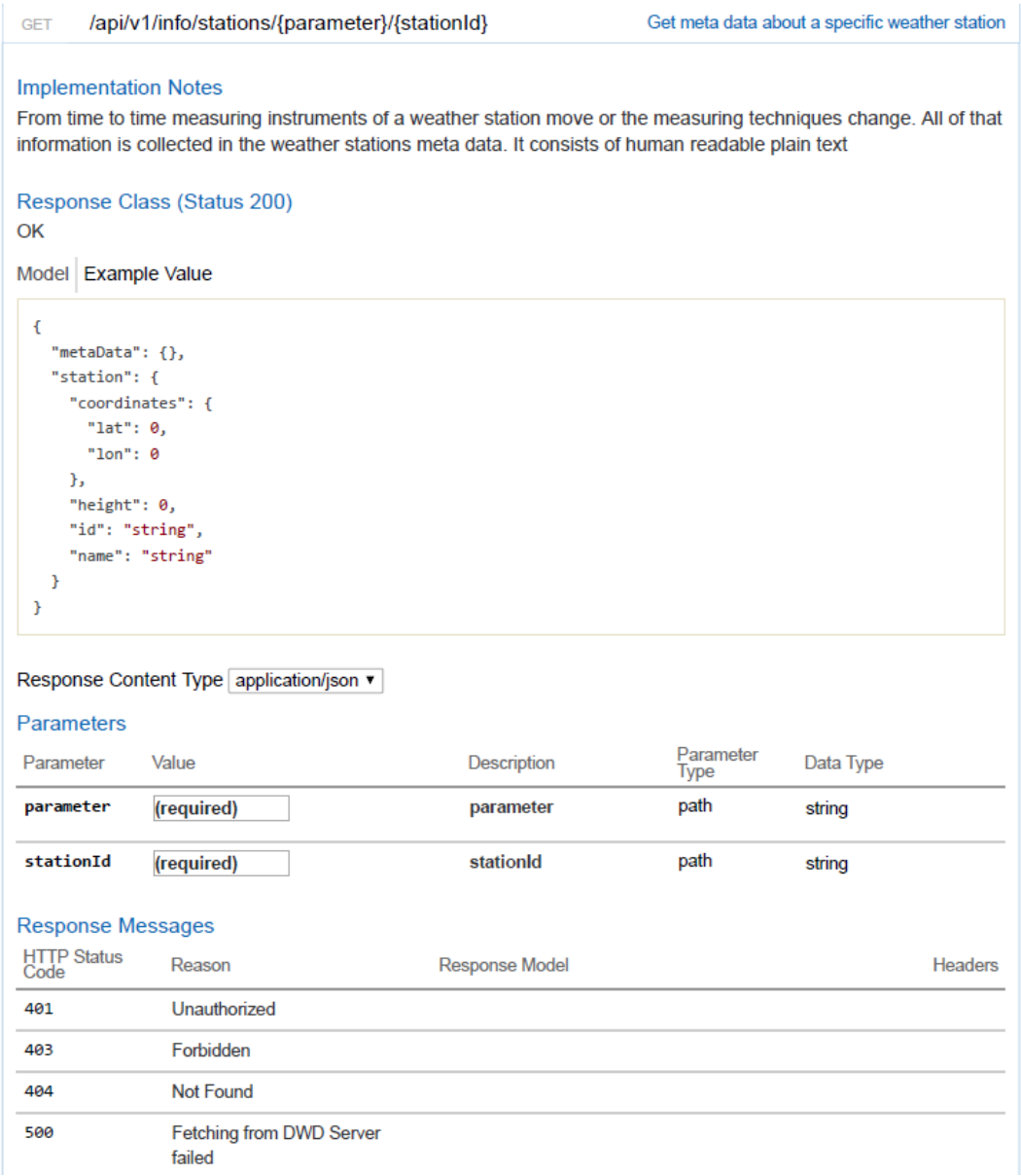


Figure 6.3: Sequence diagram of the *ForecastDataPointRequester* fetching the forecast using the *ForecastDataProvider* until usable data is found.

Appendix B Swagger documentation excerpt



GET /api/v1/info/stations/{parameter}/{stationId} — Get meta data about a specific weather station

Implementation Notes

From time to time measuring instruments of a weather station move or the measuring techniques change. All of that information is collected in the weather stations meta data. It consists of human readable plain text

Response Class (Status 200)

OK

Model | Example Value

```
{
  "metaData": {},
  "station": {
    "coordinates": {
      "lat": 0,
      "lon": 0
    },
    "height": 0,
    "id": "string",
    "name": "string"
  }
}
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
parameter	(required)	parameter	path	string
stationId	(required)	stationId	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		
500	Fetching from DWD Server failed		

Figure 6.4: Excerpt from the Swagger documentation of the meta data route.

Appendix C Get current weather using the ODS

1. HTTP PUT {{ods_url}}/datasources/currentErlangen

```
{
  "domainIdKey": "/location",
  "schema": {},
  "metaData": {
    "name": "ExampleName",
    "title": "ExampleTitle",
    "author": "",
    "authorEmail": "",
    "notes": "",
    "url": "",
    "termsOfUse": ""
  }
}
```

2. HTTP PUT {{ods_url}}/datasources/currentErlangen/filterChains/mainFilter

```
{
  "processors" : [
    {
    "name" : "DwdWeatherServiceSourceAdapter",
      "arguments" : {
        "location" : {
          "city": "Erlangen"
        },
        "time" : "current"
      }
    },
    {
      "name" : "DbInsertionFilter",
      "arguments" : {
        "updateData" : true
      }
    }
  ],
  "executionInterval" : {
  "period" : 60,
  "unit" : "MINUTES"
  }
}
```

3. HTTP GET {{ods_url}}/datasources/currentErlangen/data?count=100

References

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BPP19] Maria C. Borges, Frank Pallas and Marco Peise. "Providing Open Environmental Data - The Scalable and Web-Friendly Way". In: (2019).

[Bui15] Thanh Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[DWD17] DWD. *Deutscher Wetterdienst Pressemitteilung vom 25.07.2017*. 2017. URL: `https://www.dwd.de/DE/presse/pressemitteilungen/DE/2017/20170725_dwd-gesetz.pdf?__blob=publicationFile&v=6` (visited on 04/07/2019).

[FT00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000.

[Gam+95] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995, pp. 207–217.

[Inc19] Docker Inc. *What is a Container*. 2019. URL: `https://www.docker.com/resources/what-container` (visited on 04/07/2019).

[MFC00] Tim Mackinnon, Steve Freeman and Philip Craig. "Endo-testing: unit testing with mock objects". In: *Extreme programming examined* (2000), pp. 287–301.

[Mye+04] Glenford J Myers et al. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

[Nie93] J. Nielsen. *Usability engineering*. Academic Press, 1993.

[Nur+09] Nurzhan Nurseitov et al. "Comparison of JSON and XML data interchange formats: a case study." In: *Caine* 9 (2009), pp. 157–162.

[org04] International standardization organization. *ISO 8601: 2004 (E): Data Elements and Interchange Formats, Information Interchange, Representation of Dates and Times*. ISO, 2004.

[Sch19] Georg-Daniel Schwarz. "Migrating the JValue ODS to micorservices". Master's thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, Mar. 2019.

[Thö15] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[ZB05] Ruediger Zarnekow and Walter Brenner. "Distribution of Cost over the Application Lifecycle - a Multi-case Study". In: Jan. 2005, pp. 68–79.