DWD SWIM Service API Description

Version v1.1.11

Deutscher Wetterdienst

Deutscher Wetterdienst

Table of contents

1. Home 3
1.1 DWD SWIM Services 3
1.2 Getting Started 4
2. Tutorial 5
2.1 Connecting to AMQP Services 5
2.2 Using AMQP Offset Tracking to Retrieve Past Messages 8
3. How-to 10
3.1 Consumer Loop Script 10
3.2 Creating and Using a Custom Subscription 12
3.3 Offset Tracking: Avoiding Message Loss 14
4. Explanation 18
4.1 Service Operations & Lifecycle 18
4.2 Subscription Management 20
4.3 Semantics of Subscriptions vs Queues 21
5. Reference 22
5.1 List of Services 22
5.2 IWXXM AMQP Headers 23
5.3 Business message format 26
5.4 Technical message format 27
5.5 SWIM Services Transition Overview 34
5.6 Glossary 36
5.7 Changelog 39

1. Home

1.1 DWD SWIM Services

Welcome to the documentation for the new generation of SWIM services offered by Deutscher Wetterdienst (DWD).

SWIM (**S**ystem **W**ide **I**nformation **M**anagement) Services represent a modern, standardized, and interoperable approach for accessing aviation meteorological information. DWD's service offer is compliant with the Eurocontrol SWIM Technical Infrastructure Yellow Profile[1]. A description of services currently offered by DWD can be found in the SWIM service registry[2].

In the registry you'll find services provided by the system described in this documentation, as well as other services provided by DWD. This new offering from DWD will replace or complement existing mechanisms that users may currently use, such as our WFS-based data delivery. While those traditional offerings remain available for the time being, users are strongly encouraged to transition to SWIM Services to benefit from improved reliability, scalability, security, and compliance with international standards such as the ICAO SWIM concept.

The page SWIM Services Transition Overview lists the services, which are already available via the new SWIM Services interface and which services are currently in transition or available via the legacy interfaces.

This documentation is targeted to software engineers who want to consume the service or implement a client application that interacts with the service. Additionally, it contains the concepts of the service, e.g., the service lifecycle and the service operations management, which are relevant for all users of the service.

A pdf version of this documentation is available here.

Version: `v1.1.11`

Tutorial

The perfect place to get started with the SWIM services!

You may want to know how to connect to the AMQP services and receive your first messages. Additionally, you can learn how to retrieve past messages using AMQP offset tracking.

How-to guides

Our How-to guides provide instructions for specific tasks, such as starter consumer loop script.

Explanation

In this section, underlying concepts of the service are explained.

The service lifecycle section contains details on the operations of the service (e.g., release management, operation details).

Reference

The AMQP headers section is a good starting point as it gives an overview of the specific AMQP headers sent by the service.

The message formats sections explain the data content of business (data) and technical messages in detail.

If you need help with technical or domain-specific terms please refer to the Glossary.

1. Specification for SWIM Technical Infrastructure (TI) Yellow Profile (https://www.eurocontrol.int/publication/eurocontrol-spec-170-eurocontrol-specification-swim-technical-infrastructure-ti-yellow) ↩
2. SWIM service registry (https://eur-registry.swim.aero/services) ↩

1.2 Getting Started

DWD offers different options to access meteorological data via SWIM Services. Currently, we mainly offer AMQP 1.0 based services, specifically providing IWXXM data. An account is necessary to access the services. Please reach out to vl.kundenservice@dwd.de to start the registration process.

For connecting to the AMQP services two options are offered: using a standard *"public"* subscription or creating a custom subscription. Please let us know during the registration process which option you would like to use. We recommend using the standard subscription but please find the feature comparison below to help you decide which option fits your use case best.

1.2.1 Feature Matrix

Feature	Option 1: Public Subscription	Option 2: Custom Subscription
Header based server-side filtering	✔	✘
Retrieve older Messages	✔	✘
Retrieve only new Messages after connection	✔	✘
Messages with higher priority are delivered earlier	✘	✔
Messages are removed when the TTL has expired	✘	✔
Multiple Services, Technical Messages and Heartbeats in one Subscription	✘	✔
Notifications about missed messages	✘	✔

1.2.2 Option 1: Using the Standard *"public"* Subscription

The standard *"public"* subscription (*public* in this case means that every registered customer can access these subscriptions) allows you to connect to predefined queues for each data type. You can find a list of available services and their corresponding queue names in the List of Services.

You can directly connect to these queues without any prior setup. All the messages are uniquely pushed to each connected receiver. However, note that only messages published *after* you created your AMQP receiver will be received. If you disconnect from the service and reconnect later, you will only receive messages published after your reconnection, but there is a way to retrieve past messages using offset tracking.

Additionally, this is the only option if you want to use header-based server-side filtering to reduce the number of messages sent to your receiver. Please refer to Rabbitmq's AMQP 1.0 Filter Expressions or SQL Filtering documentation for more information on how to use filtering.

To get started with this option, please refer to the connecting to AMQP services tutorial which guides you through connecting to the AMQP service and receiving your first messages.

1.2.3 Option 2: Using a Custom Subscription

This type of subscription is dedicated for customers wanting to use additional features for their subscriptions. The custom subscription is unique per user and keeps all messages sent to those subscriptions until they are consumed or the message TTL expires. *If* the message TTL expires, messages are removed from the subscription and an expired message may be sent to notify the user (this feature is optional and can be discussed during the subscription request).

This type of subscription also allows prioritization of messages, meaning that messages with a higher priority are delivered earlier.

Additionally, it's possible to combine multiple services, technical messages and heartbeats in a single subscription. This reduces the number of connections needed to receive all relevant data.

Please refer to the custom subscription how-to guide for more information on creating and using a custom subscription. The subscription management section provides insights into the lifecycle and operations of custom subscriptions.

2. Tutorial

2.1 Connecting to AMQP Services

Welcome to this tutorial on using the SWIM service from DWD! In this guide, we will walk you through setting up a connection and consuming messages. By the end of this tutorial, you will be able to access the service and start receiving messages using Python and the Apache Qpid Proton library.

This tutorial tries to describe everything in the abstract concept of services and subscriptions, without going into the details of the implementation. However, as queues are actually created on a technical level you might want to refer to the Semantics of Subscriptions vs. Queues document for more details on their relationship.

Prerequisites

Before starting, ensure you meet the following requirements:

- A basic understanding of the AMQP 1.0 protocol.
- Registration for the SWIM service at DWD. If you are not registered, please contact vl.kundenservice@dwd.de.

2.1.1 Step 1: Requesting a Subscription

You don't need a specific subscription for the services offered by DWD. Simply connect to one of the addresses of the services listed in the List of Services. You will only receive messages published *after* you created your AMQP receiver. Hence, if you disconnect from the service and reconnect later, you will only receive messages published after your reconnection.

However, if you need a dedicated subscription keeping track of your specific messages there exists the option of creating a custom subscription. If you want to receive messages from before you created your receiver you can use offset tracking to retrieve past messages.

2.1.2 Step 2: Receiving Your First Message

We will guide you through consuming your first message using Python. Although this example is Python-specific, the concepts apply to other AMQP 1.0 compatible libraries.

Setting Up Your Environment

Please refer to the official docs to setup python. We recommend setting up a virtual environment for this example, too.

For this tutorial we only need the qpid-proton library, which we install with

```
pip install python-qpid-proton==0.40.0
```

> **Note**: While version `0.40.0` was used for this tutorial, newer versions will likely work as well. We just pin it here to ensure compatibility. Feel free to try out newer versions!

Connecting to the Broker

Replace the placeholders with your connection details:

```python
from proton import ProtonException
from proton.utils import BlockingConnection

conn = BlockingConnection(
    "amqps://swimt.dwd.de",
    user="swim_user",
    password="your_password"
)
```

Creating a Receiver and Fetching Messages

You will connect a receiver to the `de-dwd.ops.heartbeat` queue to receive heartbeat messages. The systems sends a heartbeat every 60 seconds so we don't have to wait long for our first message.

```
receiver = conn.create_receiver(
    "/queues/de-dwd.ops.heartbeat"
)
```

> **Note**: The `/queues/` prefix is required when talking to RabbitMQ. It cleanly separates names in different endpoints, e.g., from an `exchange` of the same name.

To actually receive a message we need to call the `receive` method on the receiver. We use a timeout of 70 seconds to ensure we receive at least one heartbeat message. The receive call will block until a message is received or the timeout is reached.

```
msg = receiver.receive(timeout=70)
```

Displaying the Message Content

Now that we have received a message, let's display its content. The body of the message contains the actual payload. We do a type check to ensure we handle the body correctly. Please be aware that the body can contain different types of data depending on the service and message type. Here, we expect a binary encoded JSON payload. Qpid proton represents this as a `memoryview`, so we need to convert it to bytes before decoding it.

```
import json

if isinstance(msg.body, memoryview):
    print(json.loads(msg.body.tobytes().decode("utf-8")))
```

For the heartbeat message, you should see output similar to this template:

```
{
    "technical_message_type": "heartbeat",
    "timestamp": "YYYY-MM-DDTHH:MM:SS.ssssss"
}
```

Additionally, you can inspect the application properties to see metadata about the message. Qpid proton uses the `properties` attribute for this purpose.

```
print(msg.properties)
```

Which will generate output similar to:

```
{
    "properties.message-type": "technical",
    "properties.technical-message-type": "heartbeat"
}
```

Optional: Acknowledging the Message

To acknowledge that the message has been successfully processed and to remove it from the queue, use the following command:

```
receiver.accept()
```

Closing the Connection

After processing, close the connection to the broker:

```
conn.close()
```

2.1.3 Congratulations!

You have successfully received your first SWIM message!

Next Steps

- Receive past messages using offset tracking.
- Refer to the consumer loop script for a starter script to continuously consume messages.

For further assistance, contact swim.service@dwd.de.

2.2 Using AMQP Offset Tracking to Retrieve Past Messages

This guide explains how to use AMQP offset tracking to retrieve past messages from the SWIM service. It basically follows the same steps as the connection to AMQP services tutorial but adds the necessary filter to retrieve past messages.

2.2.1 Install Prerequisites

For this tutorial we need to install the qpid proton library and rich for pretty printing. Using rich is optional but recommended, because it makes the xml output much more readable.

```
pip install python-qpid-proton==0.40.0 rich
```

2.2.2 Setup the Offset Tracking Filter

The most important part of this tutorial is the setup of the offset tracking filter. We use qpid proton's `Filter` class to create a filter with the appropriate offset tracking specification.

```
import json

from proton import Described, symbol, ProtonException
from proton.reactor import Filter
from proton.utils import BlockingConnection
from rich import print # (1)!

offset_spec = "30m"

OFFSET_FILTER_SPEC = "rabbitmq:stream-offset-spec"
offset_filter = Filter(
    {
        symbol(OFFSET_FILTER_SPEC): Described(
            symbol(OFFSET_FILTER_SPEC), offset_spec
        )
    }
)
```

1. We use rich to print messages in a pretty way. This is optional but recommended for better readability.

2.2.3 Different Offset Tracking Options

The offset filter value (i.e., the variable `offset_spec` from above) defines from which point in time we want to receive messages. There are several options available:

- **`next`** : Start from the message after the last one received on this subscription, i.e., all new messages after connecting. **Default** if no filter is provided.
- **`first`** : Start from the first message in the stream.
- **`last`** : Start from the last message in the stream.
- **Offset**: A numerical value specifying the exact offset to start from. The offset of a message can be found in the message annotations under the key `x-stream-offset` , i.e., in qpid proton you can access it with `msg.annotations['x-stream-offset']` .
- **Timestamp**: A numerical value representing the UNIX time (milliseconds since epoch) to start from.
- **Duration**: A string value specifying the time interval relative to current time to receive messages from, e.g., `30m` for the last 30 minutes, `2h` for the last 2 hours, `1D` for the last day (valid units `Y` , `M` , `D` , `h` , `m` , `s`).

This is adapted from the RabbitMQ documentation, which also contains a more specific view on the behaviour of each option.

2.2.4 Connect and Receive Messages

With the filter set up, we can now connect to the broker and receive messages from our desired offset. The code below is similar to the one in the connecting to AMQP services tutorial but includes the offset filter (the `options` parameter in `create_receiver`) when creating the receiver. If you have trouble understanding the code, please refer to the mentioned tutorial for a detailed explanation.

```
conn = BlockingConnection(
    "amqps://swimt.dwd.de",
    user="swim_user",
    password="your_password"
)

receiver = conn.create_receiver(
    "/queues/de-dwd.weather.aviation.metar",
    options=offset_filter
)
msg = receiver.receive(timeout=10)

if isinstance(msg.body, str):
    print(msg.body)
elif isinstance(msg.body, memoryview):
    print(json.loads(msg.body.tobytes().decode("utf-8")))

receiver.accept()
conn.close()
```

As an output you should receive a bunch of METAR's (~40) from the last 30 minutes (or whatever offset you specified).

2.2.5 Further Steps

- The consumer loop script contains a fully featured consumer implementation including offset tracking and SQL filtering.

For further assistance or questions, contact swim.service@dwd.de.

3. How-to

3.1 Consumer Loop Script

This guide provides a starter script for continuously consuming messages from the SWIM service using Python and the Apache Qpid Proton library. The script contains SQL filtering and offset tracking capabilities and can be run directly from your terminal as is. It will connect to the METAR service and print all incoming messages and optionally application properties to the console. The script should give you a good starting point for building your own consumer applications.

If you need support or technical guidance contact us at swim.service@dwd.de.

3.1.1 Prerequisites

Ensure you have completed the following setup before proceeding:

1. A user account and password to access the SWIM service at DWD. If you are not registered, please contact vl.kundenservice@dwd.de and refer to getting started.
2. You have installed the necessary python libraries (ideally in a dedicated virtual environment):

```
pip install python-qpid-proton==0.40.0 python-dotenv==1.1.0 typer==0.20.0 rich==14.0.0
```

3.1.2 Code Example

The following example is a ready script which can be copy and pasted and directly used to connect to the SWIM service and consume messages in a loop.

consumer_loop.py

```python
import typer
from dotenv import load_dotenv
from proton import Described, symbol
from proton.handlers import MessagingHandler
from proton.reactor import Filter
from proton.utils import BlockingConnection
from rich import print
from typing_extensions import Annotated

load_dotenv()


class PrintMessageHandler(MessagingHandler): # (5)!
    def __init__(
        self,
        print_app_properties: bool = True,
        print_message: bool = False
    ):
        super().__init__()
        self.print_app_properties = print_app_properties
        self.print_message = print_message

    def on_message(self, event):
        msg = event.message

        if self.print_app_properties:
            print(msg.properties)

        if not self.print_message:
            return

        if isinstance(msg.body, memoryview):
            print(msg.body.tobytes().decode("utf-8"))
        elif isinstance(msg.body, str):
            print(msg.body)


def consume_message(
    user: Annotated[str, typer.Argument(envvar="SWIM_USER")],
    password: Annotated[str, typer.Argument(envvar="SWIM_PASSWORD")],
):
```

```
    conn = BlockingConnection(
        "amqps://swimt.dwd.de",
        user=user,
        password=password
    )

    receiver = conn.create_receiver(
        "/queues/de-dwd.weather.aviation.metar",
        handler=PrintMessageHandler( # (4)!
            print_app_properties=False,
            print_message=True
        ),
        options=Filter( # (1)!
            {
                symbol("rabbitmq:stream-offset-spec"): Described(
                    symbol("rabbitmq:stream-offset-spec"), "30m" # (2)!
                ),
                symbol("sql-filter"): Described(
                    symbol("amqp:sql-filter"),
                    "[properties.location-indicator] = 'EDDF'" # (3)!
                ),
            },
        ),
    )

    if receiver.container is None:
        conn.close()
        raise Exception("Failure in setting up the receiver")

    receiver.container.run()
    conn.close()


if __name__ == "__main__":
    typer.run(consume_message)
```

1. This filter enables retrieval of past messages based on an offset specification and SQL filtering. It is entirely optional. If you don't want to use this feature, you can remove the `options` parameter from the `create_receiver` call.
2. This specifies the offset from which to start receiving messages. For details on tracking the offset, refer to the offset tracking tutorial. Here, we use it to retrieve messages from the last 30 minutes (`30m`).
3. This specifies an SQL filter on the data. Here, we only receive messages where the `location-indicator` property equals `EDDF` (Frankfurt am Main airport).
4. We us the `handler` parameter to specify a custom message handler which processes incoming messages.
5. In case you want to adapt this example, e.g., writing messages to disk, this is the class where incoming messages are handled (specifically in the `on_message` method). You can modify or substitute this class as needed.

Handling Credentials

Additionally, create a `.env` file in the same directory as the script to store your SWIM credentials:

.env

```
SWIM_USER='your_swim_username'
SWIM_PASSWORD='your_swim_password'
```

We use `python-dotenv` to load these environment variables automatically.

Running the Script

Now you can run the script using:

```
python consumer_loop.py
```

You should see the latest METAR from `EDDF` which was sent in the last 30 minutes (all METARs are sent every 30 minutes) printed to the console and the loop will continue running, receiving new messages as they arrive.

3.2 Creating and Using a Custom Subscription

> **Advanced Feature**
>
> Custom subscriptions are intended for advanced users with specific requirements. They do not support header-based server-side filtering or retrieving past messages. Please read the getting started guide for more information on choosing the right subscription type.

In this guide, we will walk through setting using a dedicated subscription and consuming messages from it.

Prerequisites

Before starting, ensure you meet the following requirements:

- A basic understanding of the AMQP 1.0 protocol.
- Registration for the SWIM service at DWD. If you are not registered, please contact vl.kundenservice@dwd.de.

3.2.1 Step 1: Requesting a Subscription

Since there is no online subscription management interface available, you will need to contact vl.kundenservice@dwd.de to request a subscription. After processing your request, you will receive the subscription and connection details from DWD. The subscription is unique and will only be readable by the user it was created for.

Subscription States

- **PAUSED**: The default state for a new data queue. In this state, you will only receive technical and heartbeat messages.
- **ACTIVE**: Once activated by contacting DWD, data messages will be delivered.

You can also pause data delivery at any time by emailing DWD to set the queue state back to `PAUSED`.

Example Subscription Name

Your subscription name will have the following format:

```
<user>.<service>.<uuid>
```

For example: `swim_user.weather.aviation.metar.4b9506ea-2f7c-4958-a01c-d85b13ac5f2e`

3.2.2 Step 2: Receiving Your First Message

We will guide you through consuming your first message using Python. Although this example is Python-specific, the concepts apply to other AMQP 1.0 compatible libraries.

Setting Up Your Environment

1. **Setup Python** (if not already done):

   Please refer to the official docs to setup python. We recommend setting up a virtual environment for this example, too.

2. **Install the Qpid Proton Library**:

```
pip install python-qpid-proton==0.40.0
```

> **Note**: Version `0.40.0` was used for this tutorial. While newer versions may work, using this version ensures compatibility with this how-to.

Connecting to the Broker

Replace the placeholders with your connection details:

```
from proton import ProtonException
from proton.utils import BlockingConnection

conn = BlockingConnection(
    "amqps://swimt.dwd.de",
    user="swim_user",
    password="your_password"
)
```

Creating a Receiver and Fetching Messages

1. **Create a receiver using your subscription name**:

   ```
   receiver = conn.create_receiver(
       "/queues/swim_user.weather.aviation.metar."
       "4b9506ea-2f7c-4958-a01c-d85b13ac5f2e"
   )
   ```

   > **Note**: The `/queues/` prefix is required when talking to RabbitMQ. It cleanly separates names in different endpoints, e.g., from an `exchange` of the same name.

2. **Receive the first message**:

   ```
   msg = receiver.receive(timeout=10)
   ```

Displaying the Message Content

Decode and print the message body using `json`:

```
import json

print(json.loads(msg.body.tobytes().decode("utf-8")))
```

Optional: Acknowledging the Message

To acknowledge and remove the message from the queue:

```
receiver.accept()
```

> Skip this step if you want to keep the message in the queue.

Closing the Connection

After processing, close the connection to the broker:

```
conn.close()
```

3.2.3 Congratulations!

You have successfully received your first SWIM message!

Next Steps

- Implement a consumer loop.

For further assistance, contact swim.service@dwd.de.

3.3 Offset Tracking: Avoiding Message Loss

In this how-to we're going to expand the consumer loop script with the capability to keep track of the current stream position. This allows clients to resume messages consumption starting from their last received message across application restarts, system failures or network outages. Hence, this mechanism ensures no data is lost on the client side.

3.3.1 Prerequisites

You should have a working copy of a python consumer application that connects to and receives data from our SWIM service. While the example here is provided in Python it should be easy to translate to other programming languages. Please note that the underlying concepts are the same, since we only used Qpid Proton as a standard AMQP 1.0 client.

The Consumer Loop Script How-to is a good start for this. Make sure you have worked through it, before proceeding here.

3.3.2 Understanding Stream Position Indicators

The broker adds the key `x-stream-offset` to the AMQP message annotations. The value represents the current offset position and increases for each message. This allows us to use it as an indicator of the current position of the client we're using and re-attach to this position when re-connecting to the subscription.

The goal of this how-to is to extend the consumer loop to keep track of the offset position of the previously consumed messages and storing it persistently. Further, it reads the offset position on application restart and uses it as a starting offset when reconnecting to the broker in order to ensure that no data is lost during disconnects.

3.3.3 Step 1: Keep Track of the Current Stream Position

The `on_message` method of the `PrintMessageHandler` in the Consumer Loop Script can be extended in order to keep track of the current stream position.

The following code-example is examining the current message annotations for the stream position indicator and calls `write_pos_file` in order to store it to a cache file.

```
def on_message(self, event):
    msg = event.message
    pos = None
    if (msg.annotations is not None and
        "x-stream-offset" is in msg.annotations):
        pos = msg.annotations["x-stream-offset"]
        write_pos_file(
          "/queues/de-dwd.weather.aviation.metar", pos)
```

3.3.4 Step 2: Store the Stream Position to a Persistent Cache

A stream position cache is used to store the positions for each consumed stream. This could be a file-based, database or memory-cached store in order to keep this information available, especially between application restarts.

Here, we utilise a json-File based cache. For this we define the variable `pos_file` containing the file name of our cache.

```
pos_file = Path("stream_position.json")
```

The exemplary `write_pos_file` looks as follows:

```
def write_pos_file(stream_name: str, pos: int):
    data: dict[str, int] = {}

    if pos_file.exists():
      with pos_file.open("r", encoding="utf-8") as f:
        data = json.load(f)

    data[stream_name] = pos
```

```
    with pos_file.open("w", encoding="utf-8") as f:
        json.dump(data, f, indent=2)
```

3.3.5 Step 3: Reloading Stream Positions on Application Restart

On a restart of the consumer app, we read the last stored stream position as an initial step.

```
def read_pos_file():
    if pos_file.exists():
      with pos_file.open("r", encoding="utf-8") as f:
        data = json.load(f)
        return data

    return None
```

3.3.6 Step 4: Reconnecting at the Last Stream Offset

Having retrieved the last consumed message position from our cache file, we can increment this position counter in order to continue consuming at the first not-consumed message, e.g. we use it as a filter on the `create_receiver` invocation.

```
offset_map = read_pos_file()
pos = "next"
if offset_map is not None:
    pos = offset_map.get("/queues/de-dwd.weather.aviation.metar", -1) + 1

receiver = conn.create_receiver(
    "/queues/de-dwd.weather.aviation.metar",
    handler=PrintMessageHandler(print_app_properties=False, print_message=True),
    options=Filter(
        {
            symbol("rabbitmq:stream-offset-spec"): Described(
                symbol("rabbitmq:stream-offset-spec"), pos
            ),
        },
    ),
)
```

3.3.7 Additional Steps

In a production environment, you might want to consider using a more elaborate caching backend, such as key/value stores (e.g. MinIO, Redis) or a database.

Further, You might want to use a different offset tracking method as detailed in our Offset Tracking Tutorial, such as absolute unix timestamps or using a duration to retrieve all messages sent since the specified period.

3.3.8 Appendix - complete script

Here is the complete listing of our working example from this lesson.

consumer_example.py

```
import json
from pathlib import Path

import typer
from dotenv import load_dotenv
from proton import Described, symbol
from proton.handlers import MessagingHandler
from proton.reactor import Filter
from proton.utils import BlockingConnection
from rich import print
from typing_extensions import Annotated

load_dotenv()
pos_file = Path("stream_position.json")


class PrintMessageHandler(MessagingHandler):
    def __init__(
        self,
        print_app_properties: bool = True,
        print_message: bool = False,
    ):
        super().__init__()
```

```
        self.print_app_properties = print_app_properties
        self.print_message = print_message

    def on_message(self, event):
        pos = None

        msg = event.message
        if msg.annotations is not None and "x-stream-offset" in msg.annotations:
            pos = msg.annotations["x-stream-offset"]
            print("stream-offset is", pos)  # (1)!
            write_pos_file("/queues/de-dwd.weather.aviation.metar", pos)  # (2)!

        if self.print_app_properties:
            print(msg.properties)

        if not self.print_message:
            return

        if isinstance(msg.body, memoryview):
            print(msg.body.tobytes().decode("utf-8"))
        elif isinstance(msg.body, str):
            print(msg.body)


def write_pos_file(stream_name: str, pos: int):
    data: dict[str, int] = {}

    if pos_file.exists():
        with pos_file.open("r", encoding="utf-8") as f:
            data = json.load(f)

    data[stream_name] = pos

    with pos_file.open("w", encoding="utf-8") as f:
        json.dump(data, f, indent=2)


def read_pos_file():
    if pos_file.exists():
        with pos_file.open("r", encoding="utf-8") as f:
            data = json.load(f)
            return data

    return None


def consume_message(
    user: Annotated[str, typer.Argument(envvar="SWIM_USER")],
    password: Annotated[str, typer.Argument(envvar="SWIM_PASSWORD")],
):
    conn = BlockingConnection(
        "amqps://swimt.dwd.de",
        user=user,
        password=password,
    )

    offset_map = read_pos_file()  # (3)!
    pos = "next"
    if offset_map is not None:
        pos = offset_map.get("/queues/de-dwd.weather.aviation.metar", -1) + 1  # (4)!
    print(f"starting to read from {pos}")

    receiver = conn.create_receiver(
        "/queues/de-dwd.weather.aviation.metar",
        handler=PrintMessageHandler(
            print_app_properties=False,
            print_message=True,
        ),
        options=Filter(
            {
                symbol("rabbitmq:stream-offset-spec"): Described(
                    symbol("rabbitmq:stream-offset-spec"), pos  # (5)!
                ),
            },
        ),
    )

    if receiver.container is None:
        conn.close()
        raise Exception("Failure in setting up the receiver")

    receiver.container.run()
    conn.close()
```

```
if __name__ == "__main__":
    typer.run(consume_message)
```

1. Here we examine the current stream position and print it to `stdout`.
2. Write the current position to a file. In this example, we do this on every received message. If I/O becomes a bottleneck, one could consider writing this on a timely basis, e.g. every N seconds, every N messages or a combination thereof.
3. On application restart, we read the last stored position, or choose "next" if no position is found in the stored position file.
4. Increment the position counter in order to consume the first non-consumed message.
5. Contrary to the `consume_loop.py` example, we now use the filter every time we create a receiver using either the last-but-one stored stream position or `next`.

4. Explanation

4.1 Service Operations & Lifecycle

4.1.1 Release Management

The API uses semantic versioning to indicate backwards incompatibility or compatible changes. See semver.org for a comprehensive guide on semantic versioning. In short it follows a `MAJOR.MINOR.PATCH` style of versioning which increment if:

1. `MAJOR` when incompatible API changes are made
2. `MINOR` when functionality is added in a backward compatible manner
3. `PATCH` when backward compatible bug fixes are made

A pre-release version may also be indicated by appending a pre-release tag, e.g., `-alpha` to the version (see semver #9 for more details).

The registered customers will be notified in due course about upcoming releases.

Principles

DWD offers data services in a pre-operational (`swimt.dwd.de`) and operational phase (`swim.dwd.de`), which follow the following principles:

- DWD will announce draft implementations of the services on the pre-operational platform some time before the services become operational.
- DWD will inform about new service releases and discontinuation of service releases in due course.
- Customers will have access to the test services on the pre-operational platform to test their client applications before accessing the operational services.
- Customers can consume all public subscriptions (i.e, with prefix `de-dwd.`) with their accounts on both platforms.
- Customers may be set up with a subscription for each of their requested services on request. Multiple services may also be bundled in a single subscription. This is handled separately for the pre-operational and operational platforms.

4.1.2 Service Downtime

DWD maintains and monitors the service in order to ensure its availability and performance 24/7. However, there will be scheduled downtime of the service mainly due to maintenance reasons, as well as unintentional downtime due to errors or incidents. DWD will inform the service consumer about downtime in due time upfront to the occurrence, if possible. The targeted lead time for downtime maintenance is a week.

Scheduled downtime

In the case of intentional downtime, the service provider will inform the service consumer in due time upfront to the occurrence via AMQP. Additionally, a notification will be sent via AMQP to the service consumer. See the Scheduled Maintenance Message for details about the contents of such messages.

Non-intentional downtime and errors

For non-intentional downtime or others, the systems are setup to automatically inform the service consumer about the occurrence of such downtime or errors via AMQP. However, it might not always be possible to inform the service consumer in due time upfront to the occurrence of such downtime or errors due to the nature of the incident.

Additionally, DWD will inform the service consumer via automatic AMQP messages about the occurrence of internal data errors (e.g., data format errors, delays, etc.).

To receive technical AMQP messages, the service consumer needs to setup an AMQP receiver for the system messages (please refer to the List of Services for the AMQP address). The Data Error Message Schema contains details about the contents of such messages.

To ensure the availability of the service, DWD provides heartbeat messages to the service consumer every minute and will inform about the absence of data delivery in case of errors in the internal data delivery chain. To receive heartbeat messages refer to the List of Services for the address of the heartbeat messages.

In case of wider problems or incidents which affect the service availability, DWD proceeds along the lines of DWD's business continuity management and will inform the service consumer via E-Mail about the occurrence of such incidents.

4.1.3 Expiration of AMQP Messages

> **Info**
>
> This only applies to custom subscriptions, i.e., a subscription created specifically for a user. The subscription is then prefixed with the user name, e.g., `swim_user.weather.aviation.metar.<uuid>`. This does **not** apply to public subscriptions, i.e., subscriptions starting with `de-dwd.`.

The AMQP messages sent by the service have a time-to-live (TTL) appropriate for the type of message. In case the TTL is exceeded, the message will be discarded by the AMQP broker. The consumer will be informed about the expiration of a message via an Expired Message.

4.2 Subscription Management

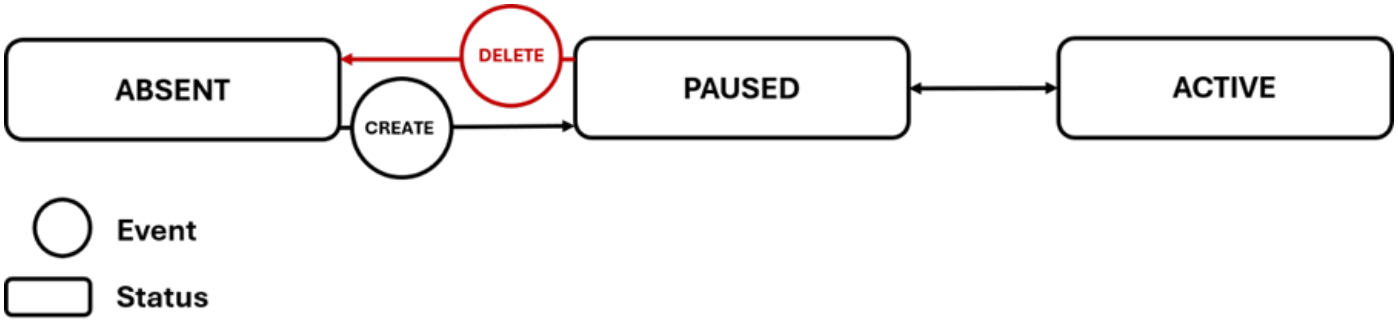
> **Info**
>
> This only applies to custom subscriptions, i.e., a subscription created specifically for a user. The subscription is then prefixed with the user name, e.g., `swim_user.weather.aviation.metar.<uuid>`. This does **not** apply to public subscriptions, i.e., subscriptions starting with `de-dwd.`.

4.2.1 Subscription Creation

To learn about how to create a subscription, refer to the Getting Started guide.

4.2.2 Subscription Lifecycle

Subscriptions can be in one of three states: `ABSENT`, `ACTIVE`, and `PAUSED`. These states determine the technical availability of the data queue. All state changes are communicated through Subscription Change Messages.



Lifecycle of SWIM subscriptions.

State	Description
ACTIVE	The subscription is operational and delivers data.
PAUSED	The subscription can be connected to but does not receive any business data. It still receives technical or heartbeat messages.
ABSENT	The subscription is not available for connection. This may be the case before creation, after deletion or during a maintenance of the services.

A change of subscription state may be requested by the consumer or initiated by the provider. The provider can change the state of a subscription at any time, but must inform the consumer about the change.

The typical lifecycle of a subscription consists of the following steps:

1. **Creation**: the subscription is in the state `PAUSED`. Consumers are informed via offline communication (email, phone, etc.) about the connection details. A Subscription Change Message informs about the creation of the subscription.
2. **Data delivery**: When ready to consume data, the consumer requests activation via an offline channel. The provider then sets the queue to `ACTIVE` and confirms via a Subscription Change Message.
3. **Pausing**: The provider may transition the queue from `ACTIVE` to `PAUSED` due to maintenance, technical issues, or consumer requests. Planned maintenance is announced via a Scheduled Maintenance Message, with `maintenance-start` and `maintenance-end` events sent at respective times.
4. **Resuming**: The provider may re-activate a `PAUSED` queue, notifying the consumer via a Subscription Change Message.
5. **Deletion**: At the request of the customer the subscription may be deleted, which means to transition it from `PAUSED` to `ABSENT`.

4.3 Semantics of Subscriptions vs Queues

The naming of different contexts in the SWIM services using AMQP can be confusing, especially when it comes to the terms `subscriptions` and `queues`, as well as `exchanges` or `topics` vs. `services`. This document clarifies the semantics of these terms.

4.3.1 What are Services and Subscriptions?

The term `service` and `subscription` are used in the SWIM context to denote an offer of data that can be consumed by a customer. A service is identified by a unique name, which is used to create a subscription. A list of services can be found in the SWIM Service Registry. A service may contain multiple datasets with multiple endpoints. The details and the actual mapping to the technical endpoints are explained in the SWIM Service Registry.

When a customer wants to consume data from a service, they have to create a subscription. This subscription is a unique identifier that allows the customer to connect to the service and receive data. It follows the Subscription Lifecycle and is managed by the service provider.

4.3.2 What are Queues and Exchanges?

While there is no notion of `exchanges` or `queues` in the AMQP 1.0 protocol, DWD's SWIM service uses the RabbitMQ implementation of AMQP 1.0, which introduces the concept of `exchanges` and `queues`. The `exchange` concept is used to route messages to the appropriate queues based on the routing key. The concept of `queues` is used to denote a storage mechanism for messages that are sent by the service and to offer a private endpoint for the customer to connect to and consume the messages. If the user disconnects the messages are stored in the queue until the user reconnects and consumes them.

These two concepts directly relate to the subscription and service concepts:

- **Subscription**: A subscription is a unique identifier that allows the customer to connect to a service and receive data. It is represented by a queue in RabbitMQ.
- **Service**: A service is a unique name that identifies the data being offered. It is represented by an exchange in RabbitMQ.

When a subscription is created, a queue is created in RabbitMQ that is bound to the exchange of the service. So for the AMQP service offer the concepts of `services` / `subscriptions` and `exchanges` / `queues` are closely related. However, this documentation uses the abstract concept of `service` and `subscription` to avoid confusion with the RabbitMQ implementation details. The documentation may only refer to the technical terms whenever necessary, e.g., when talking about the connection details or the AMQP protocol itself.

In AMQP 1.0 as well as in RabbitMQ there is no concept of a `topic` as it might be known from other messaging systems used, e.g. in the WMO WIS2 system. There are, however, `topic exchanges` in RabbitMQ, which are used to route messages to queues based on the routing key. This is an implementation detail of RabbitMQ and is of no concern for the user of the SWIM service.

5. Reference

5.1 List of Services

This is a list of service names provided by the new service infrastructure on `swimt.dwd.de`.

5.1.1 AMQP 1.0 Services

Please note that for connection to the services you need to prefix the address with `/queues/` when creating your receiver.

IWXXM

AMQP Address	AMQP Headers	Description / SWIM Registry Link
`/queues/de-dwd.weather.aviation.metar`	IWXXM AMQP Headers	METAR in IWXXM
`/queues/de-dwd.weather.aviation.taf`	IWXXM AMQP Headers	TAF in IWXXM
`/queues/de-dwd.weather.aviation.sigmet`	IWXXM AMQP Headers	SIGMET in IWXXM
`/queues/de-dwd.ops.system-messages`	IWXXM AMQP Headers	Technical Messages from the System
`/queues/de-dwd.ops.heartbeat`	IWXXM AMQP Headers	Heartbeat Messages from the System

5.2 IWXXM AMQP Headers

The use of AMQP Properties is useful for gathering message information without having to parse the full message payload. The AMQP 1.0 protocol[1] defines three types of properties: **header**, **properties** and **application properties**. While the **header** and **properties** are fixed keys standardized in the protocol, the **application properties** allow applications to define their own keys.

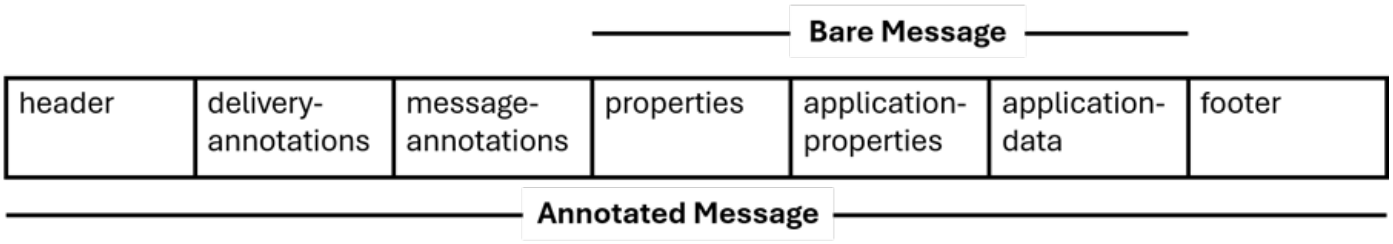


Diagram of a full AMQP message.

5.2.1 General headers

The lists below detail the usage of the AMQP headers and properties within the DWD SWIM Service.

Application Properties | Message Properties | Headers

Key	Description
message-type	Describes the type of the message content. Can be one of `business-notification-url`, `business-direct-data` or `technical`. For `technical` the type of technical message is further specified in `properties.technical-message-type`

Key	Description
message-id	The universally unique ID (RFC4122[3] uuid4) of the message.
absolute-expiry-time	UNIX-like UTC timestamp in milliseconds, indicating when the message should expire in the broker and from durable queues.
creation-time	Unix-like UTC timestamps in milliseconds, indicating when the AMQP message was initially created.
subject	Represent the type of data transmitted, e.g., `weather.aviation.metar` for `METAR`s.
content-type	MIME type of the message (e.g., `application/xml; charset=utf-8`)
content-encoding	Indicates whether the payload is gzip-compressed, in MIME type syntax

Key	Description
priority	`SIGMET`: 7 `SPECI`: 6 `TAF AMD`: 6 `TAF`: 5 `METAR`: 4 All other messages: 3 (7 = highest, 3 = lowest)
ttl	Duration in milliseconds (set by the DWD) for which the message is to be considered “alive”. **Business Message**: The DWD will calculate a TTL per message type (payload-subject). **Technical Message**: TTL is one week. **Heartbeat**: TTL is 70 seconds for a heartbeat frequency of 1 per 60 seconds.

Additional Headers

Please note that additional headers such as `user-id` and `to` may be present in the AMQP messages. However, these headers are automatically set by the technical infrastructure used for message delivery. Hence, the shall not be interpreted as part of the DWD SWIM service.

5.2.2 Application Properties for Business Messages

Key	Description
conformsTo	Link to the SWIM service definition the data conforms to.
properties.report_status	One of the following values: `NORMAL` : Regular report `AMENDMENT` : Amendment of the previous report `CORRECTION` : Correction of the previous report
properties.icao_location_identifier	`METAR` / `SPECI` , `TAF` : The ICAO 4-letter location indicator of the respective aerodrome. `SIGMET` : The respective Flight Information Region (`FIR`) or Upper Flight Information Region (`UIR`) for which the SIGMET has been created.
properties.icao_location_type	Type of location identifier. Can be optionally used for disambiguation when the location referred to by the properties.icao_location_identifier is not clear.
properties.issue_datetime	RFC3339[2] formatted publication/issue time extracted from `iwxxm:issueTime` .
properties.datetime	RFC3339[2] formatted observation time. Only for `METAR` / `SPECI` .
properties.start_datetime	Start of validity in RFC3339[2] format. Only for `TAF` / `SIGMET` .
properties.end_datetime	End of validity in RFC3339[2] format. Only for `TAF` / `SIGMET` .
payload-subject	DWD specific identifier of data delivery for debugging (Product identifier/Filename for filtering: Subset LA (METAR), LC (Short TAF), LT (Long TAF), LP (SPECI), LS (SIGMET)) // Example: A_**LA**DL31EDDT292250_C_EDZO_20170729225000). Please use the `subject` message property instead.
API-version	Indication of the actual API version.
Content-SHA256	`DEPRECATED` : This value will also be provided in the future but the header will be renamed. SHA256 hash of the payload.

5.2.3 Application Properties for Technical Messages

Key	Description
technical-message-type	The type of technical message. Must be one of `subscription-change` , `scheduled-maintenance` , `data-error` or `heartbeat` .
event	The event type of the technical message. This is only sent for `subscription-change` technical messages and contains the `event` field of the json payload.

1. https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-messaging-v1.0-os.html#type-header ↩
2. Standard date format of the form `YYYY-MM-DDThh:mmZ` https://datatracker.ietf.org/doc/html/rfc3339 ↩↩↩↩
3. https://datatracker.ietf.org/doc/html/rfc4122 ↩

5.3 Business message format

The headers, properties and application properties of business messages are specified in the general AMQP Headers part of this documentation. The format here specifies the actual message format. Currently, two formats are supported, which are indicated by the message type (indicated by the `message-type` key in the application properties) `business-direct-data` or `business-notification-url`.

5.3.1 Business direct data

This message type can be recognized by the application property `message-type: business-direct-data`. The payload of the data is the raw content of the meteorological information being sent, e.g., a raw IWXXM file xml file .

5.3.2 Business notification url

> **Currently these types of messages are not sent by the service.**
>
> There is still a standardisation process ongoing how to define the content of these messages.

This message type can be recognized by the application property `message-type: business-notification-url`.

5.4 Technical message format

Technical messages contain `application/json` data sent through the AMQP application data. They are described by json schema files for validation. The headers, properties and application properties of technical messages are specified in the general AMQP Headers part of this documentation.

5.4.1 Subscription Change Message

A message to notify about a changing subscription status.

Field	Type	Description
technical_message_type	string	This is a subscription status message to notify about a change in subscription status. **Must be** `subscription-change`.
reference	string	The related reference (e.g., queue or exchange name) to which the message has been sent.
event	string	The event which triggered the subscription change. **Must be one of** `create`, `delete`, `maintenance-start`, `maintenance-stop`, `resume`, `user-request`.
new_status	string	The status of the subscription after this message has been sent. **Must be one of** `active`, `paused`, `absent`.
reason	string | null	Business reason of the subscription status change.
requester	string	The requester of this subscription change. **Must be one of** `USER`, `DWD`.

JSON schema

subscription_change_message.json


```
{
    "examples": [
        {
            "event": "create",
            "new_status": "paused",
            "reference": "my_queue",
            "requester": "DWD",
            "type": "Subscription-Status"
        }
    ],
    "properties": {
        "technical_message_type": {
            "const": "subscription-change",
            "default": "subscription-change",
            "description": "This is a subscription status message to notify about a change in subscription status.",
            "title": "Technical Message Type",
            "type": "string"
        },
        "reference": {
            "description": "The related reference (e.g., queue or exchange name) to which the message has been sent.",
            "title": "Reference",
            "type": "string"
        },
        "event": {
            "description": "The event which triggered the subscription change.",
            "enum": [
                "create",
                "delete",
                "maintenance-start",
                "maintenance-stop",
                "resume",
                "user-request"
            ],
            "title": "Event",
            "type": "string"
        },
        "new_status": {
            "description": "The status of the subscription after this message has been sent.",
            "enum": [
                "active",
                "paused",
                "absent"
            ],
            "title": "New Status",
            "type": "string"
        },
        "reason": {
            "anyOf": [
                {
                    "type": "string"
                },
                {
                    "type": "null"
                }
            ],
            "default": null,
            "description": "Business reason of the subscription status change.",
            "title": "Reason"
        },
        "requester": {
            "description": "The requester of this subscription change.",
            "enum": [
                "USER",
                "DWD"
            ],
            "title": "Requester",
            "type": "string"
        }
    },
    "required": [
        "reference",
        "event",
        "new_status",
        "requester"
    ],
    "title": "SubscriptionChangeMessage",
    "type": "object"
}
```

5.4.2 Scheduled Maintenance Message

A message which notifies about a scheduled maintenance in the future.

Field	Type	Description
technical_message_type	string	This is a scheduled maintenance message to notify about a scheduled maintenance. **Must be** `scheduled-maintenance`.
reference	string	The related reference (e.g., queue or exchange name) to which the message has been sent.
planned_status	string	The status of the subscription during the maintenance period. **Must be one of** `active`, `paused`, `absent`.
start	string	The start datetime in ISO8601 format of the maintenance. If no timezone information is provided the datetime is in UTC.
duration	string	The duration in ISO8601 format of the maintenance.

JSON schema

scheduled_maintenance_message.json


```
{
    "properties": {
        "technical_message_type": {
            "const": "scheduled-maintenance",
            "default": "scheduled-maintenance",
            "description": "This is a scheduled maintenance message to notify about a scheduled maintenance.",
            "title": "Technical Message Type",
            "type": "string"
        },
        "reference": {
            "description": "The related reference (e.g., queue or exchange name) to which the message has been sent.",
            "title": "Reference",
            "type": "string"
        },
        "planned_status": {
            "description": "The status of the subscription during the maintenance period.",
            "enum": [
                "active",
                "paused",
                "absent"
            ],
            "title": "Planned Status",
            "type": "string"
        },
        "start": {
            "description": "The start datetime in ISO8601 format of the maintenance. If no timezone information is provided the datetime is in UTC.",
            "example": "2025-01-01T12:00:00Z",
            "format": "date-time",
            "title": "Start",
            "type": "string"
        },
        "duration": {
            "description": "The duration in ISO8601 format of the maintenance.",
            "example": "P3D",
            "format": "duration",
            "title": "Duration",
            "type": "string"
        }
    },
    "required": [
        "reference",
        "planned_status",
        "start",
        "duration"
    ],
    "title": "ScheduledMaintenanceMessage",
    "type": "object"
}
```

5.4.3 Data Error Message

A message which notifies about a current error in the queue.

Field	Type	Description
technical_message_type	string	This is a data error message to notify the consumer that something went wrong during data processing. **Must be** `data-error`.
error_type	string	This is the machine readable type of the error. If the type is 'unknown' an unknown error occurred and additional details can be found in 'error_message'. **Must be one of** `validation-error`, `processing-error`, `system-delay`, `missing-data`, `message-expired`, `unknown`.
error_message	string | null	This is a human-readable error message with details about the error.

JSON schema

data_error_message.json

```
{
    "properties": {
        "technical_message_type": {
            "const": "data-error",
            "default": "data-error",
            "description": "This is a data error message to notify the consumer that something went wrong during data processing.",
            "title": "Technical Message Type",
            "type": "string"
        },
        "error_type": {
            "description": "This is the machine readable type of the error. If the type is 'unknown' an unknown error occurred and additional details can be found
in 'error_message'.",
            "enum": [
                "validation-error",
                "processing-error",
                "system-delay",
                "missing-data",
                "message-expired",
                "unknown"
            ],
            "title": "Error Type",
            "type": "string"
        },
        "error_message": {
            "anyOf": [
                {
                    "type": "string"
                },
                {
                    "type": "null"
                }
            ],
            "default": null,
            "description": "This is a human-readable error message with details about the error.",
            "title": "Error Message"
        }
    },
    "required": [
        "error_type"
    ],
    "title": "DataErrorMessage",
    "type": "object"
}
```

Expired Message

A message to notify about expired messages in the queue

Field	Type	Description
technical_message_type	string	This is a data error message to notify the consumer that something went wrong during data processing. **Must be** `data-error`.
error_type	string	This error notifies about a message which expired in this subscription. **Must be** `message-expired`.
error_message	string | null	This is a human-readable error message with details about the error.
original_subject	string	The original message subject of the expired message.
original_reference	string	The original queue/exchange/topic in which the message expired.
original_message_id	string	The id of the expired message.
expiration_time	string	The original timestamp of the expired message.

JSON schema

expired_message.json


```
{
    "properties": {
        "technical_message_type": {
            "const": "data-error",
            "default": "data-error",
            "description": "This is a data error message to notify the consumer that something went wrong during data processing.",
            "title": "Technical Message Type",
            "type": "string"
        },
        "error_type": {
            "const": "message-expired",
            "default": "message-expired",
            "description": "This error notifies about a message which expired in this subscription.",
            "title": "Error Type",
            "type": "string"
        },
        "error_message": {
            "anyOf": [
                {
                    "type": "string"
                },
                {
                    "type": "null"
                }
            ],
            "default": null,
            "description": "This is a human-readable error message with details about the error.",
            "title": "Error Message"
        },
        "original_subject": {
            "description": "The original message subject of the expired message.",
            "title": "Original Subject",
            "type": "string"
        },
        "original_reference": {
            "description": "The original queue/exchange/topic in which the message expired.",
            "title": "Original Reference",
            "type": "string"
        },
        "original_message_id": {
            "description": "The id of the expired message.",
            "title": "Original Message Id",
            "type": "string"
        },
        "expiration_time": {
            "description": "The original timestamp of the expired message.",
            "format": "date-time",
            "title": "Expiration Time",
            "type": "string"
        }
    },
    "required": [
        "original_subject",
        "original_reference",
        "original_message_id",
        "expiration_time"
    ],
    "title": "ExpiredMessage",
    "type": "object"
}
```


5.4.4 Heartbeat Messages

The heartbeat message shall be settled by the consumer.

Field	Type	Description
technical_message_type	string	This is a heartbeat message to notify the consumer that the connection is established. **Must be** `heartbeat` .
timestamp	string	Time in ISO8601 format at which this heartbeat was sent.

JSON schema

heartbeat_message.json


```
{
    "properties": {
        "technical_message_type": {
            "const": "heartbeat",
            "default": "heartbeat",
            "description": "This is a heartbeat message to notify the consumer that the connection is established.",
            "title": "Technical Message Type",
            "type": "string"
        },
        "timestamp": {
            "description": "Time in ISO8601 format at which this heartbeat was sent.",
            "example": "2025-01-01T12:00:00Z",
            "format": "date-time",
            "title": "Timestamp",
            "type": "string"
        }
    },
    "required": [
        "timestamp"
    ],
    "title": "HeartbeatMessage",
    "type": "object"
}
```

5.5 SWIM Services Transition Overview

The DWD SWIM service is transitioning to a new service model that will enhance the way data is provided to consumers. This document outlines the key changes and how they affect the current service offerings and where users can consume data from. Services in the two main categories **Publish-Subscribe Services** and **Request-Reply Services** are offered.

In the first phase of the transition, the publish-subscribe services will be offered.

5.5.1 Publish-Subscribe Services

The new system offers the IWXXM services via AMQP 1.0. Please refer to the Getting Started tutorial for more details on how to consume these services. The List of Services provides an overview of the currently available services.

NowCastMIX

- `/topic/convection.direct`

Format: `GML/XML`

Eucom Radar Data

In hdf5 format

- `/topic/eucom.direct`

Format: `HDF5`

NCM aviation

- `cloudtopheight.direct`

Format: `GML/XML bz2 compressed`

Crossborder Convection Forecast

- `crossborder.direct`

Format: `GML/XML`

5.5.2 Request-Reply Services

Currently, the request-reply services are all offered by layers in the DWD GeoServer. If you want to learn how to connect to these layers, please refer to the DWD GeoServer documentation or contact us at vl.kundenservice@dwd.de.

The current offered via the WFS layers are:

- `dwd:CROSSBORDER_FINAL`
- `dwd:CROSSBORDER_INFO_FINAL`
- `dwd:ICON_ADWICE_POLYGONE` ; Also part of the hazardous meteorological polygon service
- `dwd:ICON_ADWICE_POLYGONE_GLOBAL`
- `dwd:ICON-EU-Turbulence-Polygons` ; Also part of MET-CEUS
- `dwd:ICON-Turbulenz-Polygone`
- `dwd:ICON_EDPP_POLYGONS`
- `dwd:NCSA_V1_EU`
- `dwd:NCSA_V1_GL`
- `dwd:SWIM_METAR_IWXXM3`
- `dwd:SWIM_SIGMET_IWXXM3`
- `dwd:SWIM_TAF_IWXXM3`
- `dwd:NCM_aviation_swetter_polygone_cth` ; Also part of the hazardous meteorological polygon service
- `dwd:ICONV`

In the future, the request-reply services will be offered via an OGC Environmental Data Retrieval (EDR) API on the new platform as well. The EDR API is currently still under development and services will be offered in a phased approach.

5.6 Glossary

5.6.1 AMQP

Advanced **M**essage **Q**ueuing **P**rotocol

5.6.2 ATM

Air **T**raffic **M**anagement

5.6.3 Data Format

A structure of data elements, records and files arranged to meet standards, specifications or data quality requirements.

5.6.4 EU

European **U**nion

5.6.5 HTTP

Hyper**t**ext **T**ransfer **P**rotocol

5.6.6 ICAO

International **C**ivil **A**viation **O**rganization

5.6.7 Information Service

A type of service that provides an ATM information sharing capability.

5.6.8 Interface Binding

Specification of the protocols and data formats to be used in transmitting messages defined by the associated interface. Note: Two systems that implement the same interface binding are considered technically interoperable and are able to connect to each other and exchange information. There are two types of interface bindings to be distinguished based on their position in the TCP/IP protocol: service bindings and network bindings. Service bindings specify the service interface protocols (e.g. protocols to interface with the applications, such as HTTP and AMQP). Network bindings specify the transport and network related protocols that will be used to exchange data over the network (e.g. TCP, IP v4/v6).

5.6.9 ISO

International **S**tandards **O**rganization

5.6.10 ISO/IEC

International **S**tandards **O**rganization / **I**nternational **E**lectrotechnical **C**ommittee

5.6.11 IWXXM

ICAO **M**eteorological **I**nformation **E**xchange **M**odel

5.6.12 Message

A discrete unit of communication intended by the source for consumption by a given recipient or group of recipients. Note: The term message refers to a unit of information exchange between systems that communicate via information services. Although there are similarities, no direct comparison should be made with the term message used in other ICAO documents (e.g. Controller Pilot Data Link Communications (CPDLC) message).

5.6.13 Protocol

A set of semantic and syntactic rules for exchanging information.

5.6.14 Service

A mechanism to enable access to one or more capabilities using a prescribed interface. Note: In the context of system wide information management, the notion of service addresses machine-to-machine interaction based on service oriented architecture principles, and is not to be confused with the notion of service as used in ICAO provisions referring to business services such as AIS, ATS, etc.

5.6.15 Service Consumer

An entity which seeks to satisfy a particular need through the use of capabilities offered by means of a service.

5.6.16 Service Description

Information needed in order to use, or consider using, a service.

5.6.17 Service Instance

The service deployed into a running ICT system.

5.6.18 Service Interface

The means by which the underlying capabilities of a service are accessed.

5.6.19 Service Operation

Specification of a transformation or query that an object may be called to execute.

5.6.20 Service Provider

An entity (person or organization) that offers the use of capabilities by means of a service.

5.6.21 SWIM

System **W**ide **I**nformation **M**anagement

5.6.22 SWIM TI

A technical infrastructure conforming to one or more SWIM TI specifications (e.g. SWIM TI YP Specification).

5.6.23 SWIM TI Profile

Specification defining an implementation of the SWIM TI. Multiple SWIM TI Profiles can coexist, each of them focused on the implementation of technical infrastructure but with different scope and applicability.

5.6.24 TCP/IP

Transmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol

5.6.25 TLS

Transport **L**ayer **S**ecurity

5.6.26 TI

Technical **I**nfrastructure

5.6.27 YP

Yellow **P**rofile

5.6.28 WMO

World **M**eteorological **O**rganization

5.6.29 XML

Extensible **M**arkup **L**anguage

5.7 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

5.7.1 v1.1.11 - 2026-10-01

- Documentation updates: eur-registry links updated
- Keepalive for upstream proxy to server
- Dropped IPv6 support from proxy config
- Security updates

5.7.2 v1.1.10 - 2026-09-03

- Security updates

5.7.3 v1.1.9 - 2026-08-06

- Security updates
- Added a public registration feature
- Rate limiting and auth for EDR

5.7.4 v1.1.8 - 2026-07-02

- Security and minor updates

5.7.5 v1.1.7 - 2026-06-11

- Security and minor updates

5.7.6 v1.1.6 - 2026-05-07

- Rabbitmq update
- Security and minor updates

5.7.7 v1.1.5 - 2026-04-14

- Security and minor updates

5.7.8 v1.1.4 - 2026-03-10

- Security and minor updates
- Improved logging and error handling for reverse proxy
- Adapt alerting rules for regional airports

5.7.9 v1.1.3 - 2026-02-10

- Update to IWXXM Version 2025-2 (switch date was 2026-01-22)
- Security updates.

5.7.10 v1.1.2 - 2026-01-13

- Security updates.

5.7.11 v1.1.1 - 2025-11-28

- Removed self-subscription feature.
- Security updates.

5.7.12 v1.1.0 - 2025-11-27

- Adapted AMQP headers to follow the MET3SG CP1 Guidance.
- Introduced stream-based workflow.