# Aviso

*Release 0.11.1*

**ECMWF**

**Nov 08, 2022**

# USER GUIDE:

Aviso is a software developed by ECMWF that allows to notify **time-critical events** across HPC and Cloud systems in order to enable workflows among multiple domains.

It allows users to:

- Define events that require notification
- Define triggers to be executed once a notification is received
- Send and receive notifications

This enables the creation of automatic workflows, timely triggered as events are notified.

See *Overview* for more information.

CHAPTER

ONE

# OVERVIEW

Aviso is a scalable notification system designed for high-throughput. It is developed by ECMWF - European Centre for Medium-Range Weather Forecasts - with the aim of:

- Notifying **events**
- Triggering users' **workflows**
- Supporting a semantic **When** <this> … **Do** <that> …
- Persistent **history** and **replay** ability
- **Independent** of HPC or Cloud environments
- **Protocol agnostic**
- Highly reliable - built for **time-critical** applications.

Aviso is not designed to deliver data with the notifications. The payload is designed to provide lightweight message. If the intent is to provide notifications about data availability for example, then we suggest using the payload to inform about the data location (e.g. URL).

Aviso is a client-server application. We refer to the notification server as Aviso Server while to the client application as Aviso Client or just Aviso. This user guide and the reference are focused on Aviso client. More info on its architecture is available in *Aviso Client Architecture*.

The server system is based on a persistent key-value store where the events are stored, the key represents the event's metadata while the value, the event's payload. More info on the server architecture and its components is available in *Aviso Server Architecture*.

## 1.1 What could I use Aviso for?

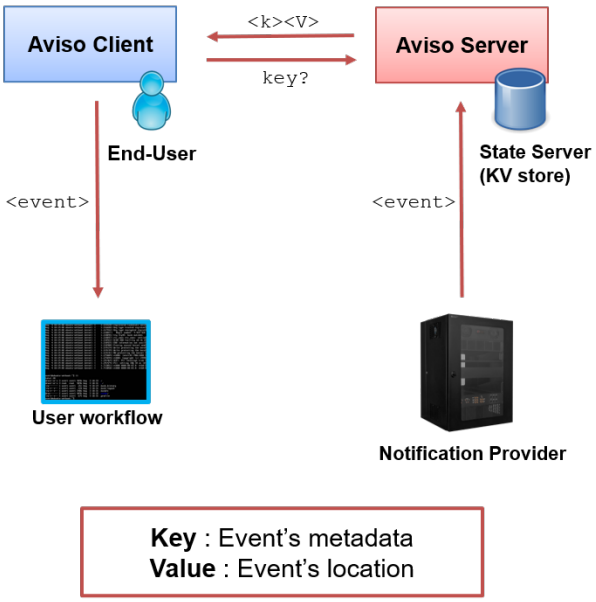
Aviso is developed with the intention of being generic and applicable to various domains and architectures, also independently of ECMWF software systems. Aviso can be used for:

- Automating users' workflows requiring notifications based on user-defined events.
- Automating users' workflows requiring ECMWF notifications on data availability. See *How Aviso Works at ECMWF* for more details on this service
- Automating multi-domain workflows across different Clouds and HPC centres. Aviso client can be extended to connect to various general purpose notification systems; similarly Aviso server can store generic events and be extended to integrate with legacy architectures
- Configuration Management. This functionality goes beyond Aviso's main aim but it is part of the notification workflow and can also be used independently. See *Configuration Management* for more info

## 1.2 Aviso general workflow

Figure below represents the general workflow of the Aviso system:

1. Aviso client allows an End-User to subscribe to an event and to program a trigger
2. Aviso client polls Aviso server for changes to the defined event
3. A notification provider submits a notification to Aviso server
4. The subscriber is notified with a new event
5. The event triggers the user's workflow



CHAPTER

TWO

# GETTING STARTED

Aviso Client can be used as a Python API or as Command-Line Interface (CLI) application. Below find a few steps to quickly get a working configuration on Linux.

**Note:** Currently Aviso supports only etcd as key-value store for the server side. The following quick start shows how to connect to a local basic installation of etcd. See *Configuration* on how to connect to a remote cluster.

## 2.1 Installing

**Warning:** Aviso requires Python 3.6+ and etcd 3.4+

1. Install Aviso Client, simply run the following command:

```
pip install pyaviso
```

2. Install etcd, below the basic steps to install a local server:

```
ETCD_VER=v3.4.14
DOWNLOAD_URL=https://github.com/etcd-io/etcd/releases/download

curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz -o /tmp/etcd-$
↪{ETCD_VER}-linux-amd64.tar.gz
mkdir /tmp/etcd-download-test
tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C /tmp/etcd-download-test --strip-
↪components=1

# start a local etcd server
/tmp/etcd-download-test/etcd
```

For more advanced configuration or installation on different platforms please refer to the official documentation on the release page. Note that the etcd version mentioned in the script above is the latest available at the time of writing this documentation. Use any compatible version.

## 2.2 Configuring

All the examples of this guide are based on a representative use case, the broadcast of flight events, such as landing or take-off, that could trigger workflows for flight trackers and related applications. This use case is available by default. The following steps show how to try it out. See *Make Your Own Event* to customise it to your application.

Create a configuration file in the default location *~/.aviso/config.yaml* with the following settings:

```
listeners:
- event: flight
  request:
    country: Italy
  triggers:
    - type: echo
```

This file defines how to run Aviso, the event to listen to and the triggers to execute in case of notifications. This is a basic example of a generic listener to events of type `flight`. `request` describes for which events the user wants to execute the triggers. It is made by a list of keys. The users have to specify only the keys that they want to use to identify the events they are interested into. Only the notifications complying with all the keys defined will execute the triggers. In this example the trigger will be executed only when flight events for Italy will be received. These keys are defined by the listener schema file, see *Make Your Own Event* for more info.

The trigger in this example is `echo`. This will simply print out the notification to the console output.

Check *Define my Listener* to create a more complex listener.

## 2.3 Launching

1. Launch Aviso application by running the following:

   ```
   aviso listen
   ```

   Once in execution this command will create a process waiting for notifications compliant with the listener defined above.

   The user can terminate the application by pressing the key combination `CTRL + C`

   **Note:** The configuration file is only read at start time, therefore every time users make changes to it they need to restart the listening process.

2. Submit a example notification, from another terminal:

   ```
   aviso notify event=flight,country=Italy,airport=fco,date=20210101,number=AZ203,
   ↪payload=Landed
   ```

This example represents the landing event for the flight AZ203 in Fiumicino(FCO) Airport in Rome on 01-01-2021.

3. After a few seconds, the trigger defined should be executed. The terminal where the listening process is running should display the following:

   ```
   "event": "flight",
   "payload": "Landed",
   "request": {
   ```

(continues on next page)

(continued from previous page)

```
   "country": "italy",
   "date": "20210101",
   "airport": "FCO",
   "number": "AZ203"
}
```

**Note:** `payload` is used to assign a value to the specific event notified. It is, however, optional. If not given the payload will be *None*. This last case is used when only an acknowledgement that something happened is needed.

The complete list of available commands can be found in *Notification CLI*

CHAPTER

THREE

# DEFINE MY LISTENER

Aviso configuration file allows the definition of multiple listeners. Alternatively, the listener configuration can be indicated as an independent file or multiple files. Regardless of where is defined, each listener is composed of:

- `event` - the kind of event to listen to
- `request` - a dictionary of keys identifying the specific events to be notified
- `triggers` - sequence of processes to be executed once a notification is received

All aspects of what kind of event can be used and which keys to use in a `request` are defined by the event lister schema. Each key is used to identify the events the user wants to be notified. The more keys are used the narrower the selection would be. When Aviso reads a listener `request` each key value is validated and formatted accordingly with the schema. Each key is associated to a type that provides a number of properties used during its validation. See *Make Your Own Event* for more info on how to edit the schema and create your own event.

The listener below uses all the keys available for the flight events. In this case the trigger will be executed only for the events regarding flights AZ203 on 01-01-2021 at the Fiumicino(FCO) and Ciampino(CIA) airport in Rome. Note that each key accepts single or multiple values.

```
listeners:
   - event: flight
   request:
      country: italy
      airport: [FCO, CIA]
      date: 20210101
      number: AZ203
   triggers:
      - type: echo
```

## 3.1 Triggers

The `triggers` block accepts a sequence of triggers. Each trigger will result in an independent process executed every time a notification is received. These are the triggers currently available:

- **echo** is the simplest trigger as it prints the notification to the console output. It is used for testing
- **log** is useful for recording the received event to a log file
- **command** allows the user to define a shell command to work with the notification
- **post** allows the user to send the notification received as HTTP POST message formatted accordingly to the CloudEvents specification

More information are available in *Triggers*.

The example below shows how to configure multiple listeners executing scripts for different set of notifications, all flights going to or from italy, all flights AZ203, or all flights concerning Fiumicino(FCO) airport.

```
listeners:
   - event: flight
   request:
      country: italy
   triggers:
      - type: command
        command: ./my_script_per_country.sh
   - event: flight
   request:
      number: AZ203
   triggers:
      - type: command
        command: ./my_script_per_flight.sh
   - event: flight
   request:
      airport: FCO
   triggers:
      - type: command
        command: ./my_script_per_airport.sh
```

More examples are available in *Listener Examples*

CHAPTER

FOUR

# AVISO AS A PYTHON API

Aviso can be used as a Python API. This is intended for users that want to integrate Aviso in a bigger workflow written in Python or that simply have their trigger defined as a Python function. Below find an example of a Python script that defines a function to be executed once a notification is received, creates a listener that references this function trigger and finally passes it to aviso to execute.

```
from pyaviso import NotificationManager

# define function to be called
def do_something(notification):
   print(f"Notification for step {notification['request']['step']} received")
   # now do something useful with it ...

# define the trigger
trigger = {"type": "function", "function": do_something}

# create a event listener request that uses that trigger
request = {"country": "italy"}
listeners = {"listeners": [{"event": "flight", "request": request, "triggers": [trigger]}
↪]}

# run it
aviso = NotificationManager()
aviso.listen(listeners=listeners)
```

**Note:** This example is using the default configuration file in *~/aviso/config.yaml* and the generic listener schema presented in *Getting Started*. Alternatively, a configuration object can be passed to the *NotificationManager*.

See *Python API* for more info.

CHAPTER

FIVE

# LISTENER EXAMPLES

## 5.1 Command

Below find an example of an event listener for `flight` events that will execute a `command` trigger in case of notifications. Note the parameter substitution mechanism for the command and the environment variables defined.

```
listeners:
  - event: flight
    request:
      country: [Italy, Germany]
    triggers:
      - type: command
        working_dir: examples
        command: ./my_script.sh --date ${request.date} --number ${request.number}
        environment:
          AIRPORT: ${request.airport}
```

Below find a similar example of a `command` trigger. This time the parameter substitution is passing the entire notification as json.

```
listeners:
  - event: flight
    request:
      country: italy
      airport: [FCO, CIA]
      date: 20210101
      number: AZ203
    triggers:
      - type: command
        command: ./my_script.sh --json ${json}
```

Below find a similar example of a `command` trigger. This time the parameter substitution is passing the file path to a json file containing the notification.

```
listeners:
  - event: flight
    request:
      number: AZ203
    triggers:
      - type: command
        command: ./my_script.sh --jsonpath ${jsonpath}
```

Finally, find below the example shell script executed by the triggers above. Note how the parameters are passed from the triggers to the script.

```
#!/bin/bash
# (C) Copyright 1996- ECMWF.
#
# This software is licensed under the terms of the Apache Licence Version 2.0
# which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
# In applying this licence, ECMWF does not waive the privileges and immunities
# granted to it by virtue of its status as an intergovernmental organisation
# nor does it submit to any jurisdiction.

# Example of a Command trigger

echo "Test demonstrating a Command trigger"
POSITIONAL=()
while [[ $# -gt 0 ]]
do
key="$1"

case $key in
    --date)
    DATE="$2"
    shift # past argument
    shift # past value
    ;;
    --number)
    NUMBER="$2"
    shift # past argument
    shift # past value
    ;;
    -j|--json)
    JSON="$2"
    shift # past argument
    shift # past value
    ;;
    -p|--jsonpath)
    JSONPATH="$2"
    shift # past argument
    shift # past value
    ;;
esac
done

echo Notification received for number $NUMBER on date: $DATE
echo airport $AIRPORT
echo json: $JSON
echo jsonpath: $JSONPATH
echo "Script executed successfully"
```

## 5.2 Echo

Below find an example of an event listener for `flight` events that will execute a `echo` trigger in case of notifications.

```
listeners:
  - event: flight
    request:
      country: [italy, france, Germany]
      number: [AZ203, AZ303]
    triggers:
      - type: echo
```

## 5.3 Log

Below find an example of an event listener for `flight` events that will execute a `log` trigger in case of notifications.

```
listeners:
  - event: flight
    request:
      date: 20210101
    triggers:
      - type: log
        path: log/testLog.log
```

## 5.4 Accessing to ECMWF archive

This section shows some real-life examples of how to use event listeners to be notified of ECMWF real-time data availability, `mars` events, and to promptly retrieve this data. The retrieval is performed using the MARS API, that allows to access to ECMWF archive.

Below find an example of a listener triggering the script mars_script.sh.

```
listeners:
  - event: mars
    request:
      class: od
      expver: 1
      domain: g
      stream: enfo
      step: [0, 1]
    triggers:
      - type: command
        working_dir: examples
        command: ./mars_script.sh --stream ${request.stream} --date ${request.date} --
↪time ${request.time} --step ${request.step}
```

Here the shell script executed by the trigger above. Note how the parameters are passed from the trigger to the script.

```
#!/bin/bash
# (C) Copyright 1996- ECMWF.
#
# This software is licensed under the terms of the Apache Licence Version 2.0
# which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
# In applying this licence, ECMWF does not waive the privileges and immunities
# granted to it by virtue of its status as an intergovernmental organisation
# nor does it submit to any jurisdiction.

# Example of a Command trigger

echo "Test demonstrating a command trigger executing MARS request"
POSITIONAL=()
while [[ $# -gt 0 ]]
do
key="$1"

case $key in
    --date)
    DATE="$2"
    shift # past argument
    shift # past value
    ;;
    --stream)
    STREAM="$2"
    shift # past argument
    shift # past value
    ;;
    --time)
    TIME="$2"
    shift # past argument
    shift # past value
    ;;
    --step)
    STEP="$2"
    shift # past argument
    shift # past value
    ;;
esac
done

echo Notification received for stream $STREAM, date $DATE, time $TIME, step $STEP
echo Building MARS request

REQUEST="
retrieve,
class=od,
date="$DATE",
expver=1,
levtype=sfc,
param=167.128,
stream="$STREAM",
time="$TIME",
```

(continues on next page)

(continued from previous page)

```
step="$STEP",
type=an,
area=75/-20/10/60,
target="my_data.grib"
"
echo Request built, sending it...
echo $REQUEST | mars
echo Script executed successfully
```

Equivalent operation can be done using the Aviso and MARS Python API. Note how easy is to construct the *MARS* request from the notification, they both speak the *MARS* language thanks to the MARS keys used in the listener schema. See *How Aviso Works at ECMWF* for more info.

```
# (C) Copyright 1996- ECMWF.
#
# This software is licensed under the terms of the Apache Licence Version 2.0
# which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
# In applying this licence, ECMWF does not waive the privileges and immunities
# granted to it by virtue of its status as an intergovernmental organisation
# nor does it submit to any jurisdiction.

from ecmwfapi import ECMWFService

from pyaviso.notification_manager import NotificationManager


# define function to be called
def retrieve_from_mars(notification):
    print(f"Notification for step {notification['request']['step']} received")
    # now do a MARS request with this notification...
    mars_server = ECMWFService("mars")
    request = notification["request"]
    # extend the notification with the attributes needed
    request.update({"type": "fc", "levtype": "sfc", "param": 167.128, "area": "75/-20/10/
↪60"})
    mars_server.execute(request, "my_data.grib")


# define the trigger
trigger = {"type": "function", "function": retrieve_from_mars}

# create a event listener request that uses that trigger
request = {"class": "od", "stream": "oper", "expver": 1, "domain": "g", "step": 1}
listener = {"event": "mars", "request": request, "triggers": [trigger]}
listeners = {"listeners": [listener]}

# run it
aviso = NotificationManager()
aviso.listen(listeners=listeners)
```

## 5.5 Multiple

Below find an example of an event listener for `flight` events that in case of notifications will execute a `echo`, a `log` and a `command` trigger. They will be executed in parallel.

```
listeners:
  - event: flight
    request:
      country: [italy, france, Germany]
    triggers:
      - type: echo
      - type: log
        path: log/testLog.log
      - type: command
        command: ./my_script.sh --date ${request.date} --number ${request.number}
```

Below find an example of three event listeners for different `flight` events.

```
listeners:
  - event: flight
    request:
      country: italy
    triggers:
      - type: command
        command: ./my_script_ita.sh
  - event: flight
    request:
      country: germany
    triggers:
      - type: command
        command: ./my_script_de.sh
  - event: flight
    request:
      country: france
    triggers:
      - type: command
        command: ./my_script_per_fr.sh
```

## 5.6 Post

Below find an example of an event listener for `flight` events that will execute a `post` trigger in case of notifications. Specifically, this trigger will format the notification according to the CloudEvents specification and will send it to either a endpoint as HTTP POST request or to a AWS Simple Notification Service (SNS) topic. The following example shows how to send it to a HTTP endpoint defined by the user. The type is `cloudevents_http` and `url` is the only mandatory parameter.

```
listeners:
  - event: flight
    request:
      country: italy
    triggers:
```

(continues on next page)

(continued from previous page)

```
    - type: post
      protocol:
        type: cloudevents_http
        url: http://my.endpoint.com/
```

Below find a similar example showing how to customise the CloudEvents fields as well as the HTTP headers using optional parameters.

```
listeners:
  - event: flight
    request:
      country: italy
    triggers:
      - type: post
        protocol:
          type: cloudevents_http
          url: http://my.endpoint.com/
          headers:
            Content-type: "application/json"
          timeout: 30
          cloudevents:
            type: aviso_cloudevents
            source: my_test
```

In the case of a notification to a AWS SNS topic defined by the user, the structure of the trigger is similar; the type has to be `cloudevents_aws` and `arn` and `region_name` are the only mandatory parameters.

The optional parameters are: `MessageAttributes`, `aws_access_key_id`, `aws_secret_access_key` for the AWS topic fields and `cloudevents` for the CloudEvents fields. Note that if `aws_access_key_id` and `aws_secret_access_key` are not specified the AWS credentials are taken from *~/.aws/credentials* if available.

AWS SNS protocol does not enforce any specification on the message payload. Aviso uses the CloudEvents specification also in this case for consistency.

```
listeners:
  - event: flight
    request:
      country: italy
    triggers:
      - type: post
        protocol:
          type: cloudevents_aws
          arn: arn:aws:sns:us-east-2:848972885776:aviso
          region_name: us-east-2
          MessageAttributes:
            attribute1:
              DataType: String
              StringValue: valueAttribute1
            attribute2:
              DataType: String
              StringValue: valueAttribute2
          cloudevents:
            type: aviso_topic
            source: my_test
```

In case of a AWS FIFO topic `MessageGroupId` is required.

```
listeners:
  - event: flight
    request:
      country: italy
    triggers:
      - type: post
        protocol:
          type: cloudevents_aws
          arn: arn:aws:sns:us-east-2:848972885776:aviso.fifo
          region_name: us-east-2
          MessageGroupId: aviso
```

## 5.7 Python API

Below find a Python example of a basic event listener for `mars` events that will execute a `function` trigger in case of notifications.

```
# (C) Copyright 1996- ECMWF.
#
# This software is licensed under the terms of the Apache Licence Version 2.0
# which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
# In applying this licence, ECMWF does not waive the privileges and immunities
# granted to it by virtue of its status as an intergovernmental organisation
# nor does it submit to any jurisdiction.

from pyaviso.notification_manager import NotificationManager


# define function to be called
def do_something(notification):
    print(f"Notification for step {notification['request']['number']} received")
    # now do something useful with it ...


# define the trigger
trigger = {"type": "function", "function": do_something}

# create a event listener request that uses that trigger
request = {"country": "italy", "date": 20210101}
listeners = {"listeners": [{"event": "flight", "request": request, "triggers": [trigger]}
↪]}

# run it
aviso = NotificationManager()
aviso.listen(listeners=listeners)
```

CHAPTER

SIX

# TESTING MY LISTENER

Aviso provides the capability of submitting test notifications to a local server. This functionality can be used to test the listener configuration without any impact to the operational server.

1. Launch the aviso application in test mode. This allows connection to a local file-based notification server, part of the aviso application, that is able to simulate the notification server behaviour.

```
aviso listen --test
```

   The console should display a Test Mode message.

   **Note:** We are assuming the listener is defined in the default configuration file as shown in *Getting Started*.

2. Send a test notification. From another terminal, run the notify command. Here is an example for a `flight` event notification.

```
aviso notify event=flight,country=Italy,airport=fco,date=20210101,number=AZ203,
↪payload=Landed --test
```

3. After a few seconds, the trigger defined should be executed.

Test mode can be activated at global level by setting the notification engine type to `file_based` In this way the `--test` option is not needed. See *Configuration* for more info.

**Note:** The catch_up functionality is not available in Test Mode.

CHAPTER

SEVEN

# NOTIFICATION CATCH-UP

Before listening to new notifications, Aviso by default checks what was the last notification received and it will then return all the notifications that have been missed since. It will then carry on by listening to new ones. The first ever time the application runs however no previous notification will be returned. This behaviour allows users not to miss any notifications in case of machine reboots.

To override this behaviour by ignoring the missed notifications while listening only to the new ones, run the following:

```
aviso listen --now
```

This command will also reset the notification history.

Users can also explicitly replay past notifications until available. This can also be used to test the listener configuration with real notifications. Here is an example, launch Aviso with the following options:

```
aviso listen --from 2020-01-20T00:00:00.0Z --to 2020-01-21T00:00:00.0Z
```

It will replay all the notifications sent from 20 January to 21 January and the ones complying with the listener request will execute the triggers.

---

**Note:** Dates must be in the past and `--to` can only be defined together with `--from`. Dates are defined in ISO format and they are in UTC.

---

In absence of `--to`, the system, after having retrieved the past notifications, will continue listening to future notifications. If `--to` is defined Aviso will terminate once it has retrieved all the past notifications.

CHAPTER

# EIGHT

# MAKE YOUR OWN EVENT

The events accepted by Aviso are defined in the event listener schema; all the examples presented in this guide are based on an example schema that is loaded by Aviso as default. Here it is:

```
{
   "version": 0.1,
   "flight": {
      "endpoint": [{
         "engine": ["etcd_rest", "etcd_grpc", "file_based"],
         "base": "/tmp/aviso/flight/",
         "stem": "{date}/{country}/{airport}/{number}"
      }],
      "request": {
         "date": [{"type": "DateHandler", "canonic": "%Y%m%d"}],
         "country": [{"type": "StringHandler", "canonic": "lower"}],
         "airport": [{"type": "StringHandler", "canonic": "upper"}],
         "number": [{"type": "StringHandler"}]
      }
   }
}
```

The schema is valid JSON file. Below each part of the schema is briefly explained.

## 8.1 Event type

`flight` is an event type. More event types can be defined in sequence in the same schema file.

## 8.2 Endpoint

On the server side events are stored in a key-value store. This means that each event is associated to a unique key. `endpoint` defines how to map the event to a unique key. This unique key is the result of `base/stem` The parameters in {} will be substituted at runtime for the specific event. Using the example schema above, the following event:

```
aviso notify event=flight,country=Italy,airport=fco,date=20210101,number=AZ203,
↪payload=Landed
```

would be associated to the key `/tmp/aviso/flight/20210101/italy/FCO/AZ203` in the store. What to put in `base` and what in `stem` is a design choice as each of them plays a different role as explained below:

- `engine` defines for which engine adapter the configuration applies. Different engine adapter may require different key representation to match the specific store technology. In the example above the configuration chosen is applied to all the engine adapter available
- `base` is used during the listening process to define to which set of events to listen, i.e. which key prefix to query. In the examples above, aviso would listen to `/tmp/aviso/flight/`
- `stem` is used during the listening process to further select which events, among the ones received, will execute the triggers

## 8.3 Request

When Aviso reads a listener request each key value is validated and formatted accordingly with the schema. Each key is associated to a type that provides a number of properties used during its validation. Multiple types can be defined for the same key. In this case the validation process will consider one type at a time and it will fail only if the value is not valid for any of the types listed. The table below provides the full list of types and the corresponding properties that can be used.

| type | required | canonic | values | default | range | regex |
|---|---|---|---|---|---|---|
| StringHandler | ✓ | [lower, upper] | | | | |
| EnumHandler | ✓ | | ✓ | ✓ | | |
| DateHandler | ✓ | ✓ | | | | |
| TimeHandler | ✓ | ✓ | ✓ | | | |
| IntHandler | ✓ | ✓ | | | ✓ | |
| FloatHandler | ✓ | ✓ | | | | |
| RegexHandler | ✓ | | | | | ✓ |

- `required`- if specified the key would become mandatory. Note that keys used in `base` are mandatory by defaults
- `canonic` - Format to apply to the key value after validation
- `values` - List of valid values accepted
- `default` - Value given to the key if not specified
- `range` - Validity interval
- `regex` - Regex pattern to use during validation

## 8.4 How to customise the schema

Users can create their own schema following the syntax shown above. The new schema should be place in the default location *~/.aviso/service_configuration/event_listener_schema.json* . By doing so Aviso will only read this file ignoring the example provided above.

Alternatively, schema can be retrieved dynamically from a remote location. This can be activated using the `remote_schema` flag. See *Configuration Management* for more info.

Finally, a different schema parser can be indicated using the settings `schema_parser`, see *Configuration*. This can be used to extend the creation and loading of the schema according to users needs. An example of this is the `EcmwfSchemaParser` part of the `listener_schema_parser` module. See *How Aviso Works at ECMWF* for more info.

CHAPTER

NINE

# RUNNING AS A SERVICE

Aviso can be executed as a system service. This helps automating its restart in case of machine reboots. The following steps help to configure Aviso to run as a service that automatically restarts:

1. Identify the location of Aviso executable:

```
whereis aviso
```

2. Create a system service unit, by creating the following file in */etc/systemd/system/aviso.service*:

```
[Unit]
Description=Aviso

[Service]
User=<username> (if omitted it will run as root)
Group=<groupname> (optional)
WorkingDirectory= <home_directory> (optional)
ExecStart=<aviso_location> listen
Restart=always

[Install]
WantedBy=multi-user.target
```

3. Enable the aviso service:

```
sudo systemctl enable aviso.service
```

4. Reload systemd:

```
sudo systemctl daemon-reload
```

5. Start the service:

```
sudo systemctl start aviso.service
```

**Note:** If users change the Aviso configuration, Aviso service must be restarted otherwise the change will be ineffective.

CHAPTER

TEN

# HOW AVISO WORKS AT ECMWF

This section presents how Aviso has been configured and deployed at ECMWF. This is a real-life example of usage of Aviso as well as a service users can request to access.

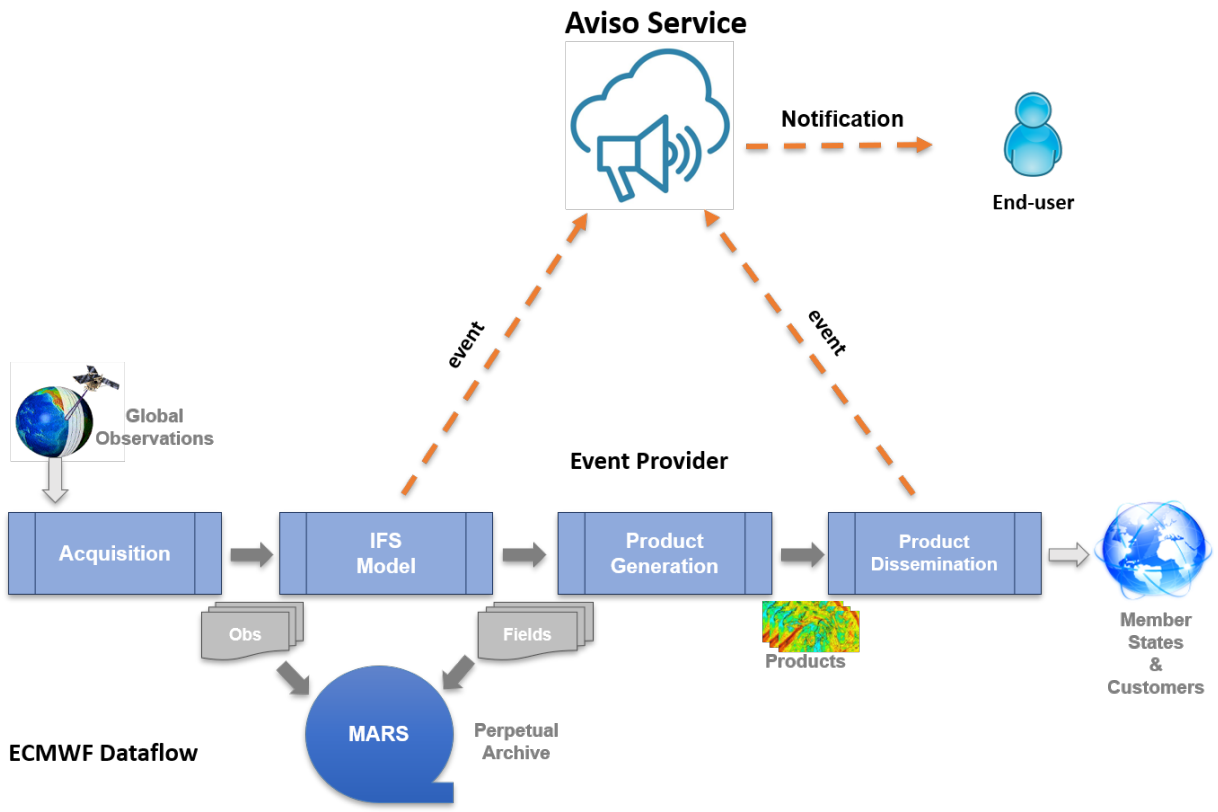
**Warning:** ECMWF Aviso Data Notification service is currently limited to registered users only. Please contact ECMWF Service Desk for more details and for configuration instructions.

## 10.1 ECMWF Aviso service

ECMWF has deployed Aviso as a notification service for the data availability of:

- Real-Time Model Output Data
- Products delivered via ECMWF dissemination system

Figure below shows ECMWF data flow; it starts from the data assimilation of observations, it then follows to the generation of the model output, the real-time global forecast. This is a time critical step for users' workflows and therefore its completion is notified by Aviso. The data flow continues with the generation of derived products that are then disseminated via ECMWF dissemination system. The delivery of these products is also notified by Aviso as users depend on custom products for their downstream applications.




This service is based on the Aviso server solution presented in *Aviso Server Architecture*. This server is currently receiving over 300k notifications a day.

## 10.2 ECMWF event listeners

The yaml below shows an example of a listener configuration for ECMWF events.

```
listeners:
   - event: mars
   request:
      class: od
      expver: 1
      domain: g
      stream: enfo
      step: [1,2,3]
   triggers:
      - type: echo
```

This is a basic example of a listener to real-time forecast events, this is identified by the keyword `mars`. `request` contains specific keys that are a subset of the ECMWF MARS language.

### 10.2.1 Events

Aviso is currently offering notifications for the following types of events:

- **dissemination** event is submitted by the ECMWF product dissemination system. The related listener configuration must define the `destination` key. A notification related to a dissemination event contains the field `location` for the URL to access to the product notified
- **mars** event is designed for real-time data from the ECMWF model output. The related listener configuration does not have any mandatory keys. Moreover the related notification does not contain the location field because users will have to access to it by the conventional ECMWF MARS API

### 10.2.2 Request

The table below shows the full list of keys accepted in `request`. These keys represent a subset of the ECMWF MARS language.

| Key | Type | Event | Optional/Mandatory |
|---|---|---|---|
| destination | String, uppercase | dissemination | Mandatory |
| target | String | dissemination | Optional |
| date | Date (e.g.20190810) | All | Optional |
| time | Values: [0,6,12,18] | All | Optional |
| class | Enum | All | Optional |
| stream | Enum | All | Optional |
| domain | Enum | All | Optional |
| expver | Integer | All | Optional |
| step | Integer | All | Optional |

### 10.2.3 Listener schema

All aspects regarding the keys above are defined by the ECMWF schema that is retrieved remotely as explained in *Configuration Management* . This section shows a part of the ECMWF schema as a real life example of a schema configuration.

```
{
   "version": 0.1,
   "payload": "location",
   "dissemination": {
      "endpoint": [
         {
            "engine": ["etcd_rest", "etcd_grpc"],
            "admin": "/ec/admin/{date}/{destination}",
            "base": "/ec/diss/{destination}",
            "stem": "date={date},target={target},class={class},expver={expver},domain=
↪{domain},time={time},stream={stream},step={step}"
         },
         {
            "engine": ["file_based"],
            "base": "/tmp/aviso/diss/{destination}",
            "stem": "{target}/{class}/{expver}/{domain}/{date}/{time}/{stream}/{step}"
         }
      ],
```

(continues on next page)

(continued from previous page)

```
      "request":
      {
         "domain": [{"type": "EnumHandler", "default": "g"}],
         "target": [{"type": "StringHandler"}],
         "stream": [{"type": "EnumHandler"}],
         "destination": [{"type": "StringHandler", "required": true}],
         "expver": [{"type": "IntHandler", "canonic": "{0:0>4}"}],
         "step": [{"type": "IntHandler", "range": [0, 100000]}],
         "time": [{"type": "TimeHandler", "canonic": "{0:0>2}", "values": [0, 6, 12, 18]}
↪],
         "date": [{"type": "DateHandler", "canonic": "%Y%m%d"}],
         "class": [{"type": "EnumHandler"}]
      }
   },
   "mars": {"..."}
}
```

The schema above regards to the `dissemination` event; the `mars` event definition would just follow. `endpoint` shows a different key construction depending on the engine adapter to use. The one reserved for etcd allows the key to be human-readable while the one for `file_based` to be compatible with a file system. `admin` key is used by the Aviso-admin component of Aviso Server to carry out maintenance on the store.

`request` contains a number of keys some of which are of type `EnumHandler`. Note that no values are provided. This would normally raise an error at runtime. However, this schema would be parsed by the ECMWF parser implemented by `EcmwfSchemaParser` class. This loads the enum values directly from the ECMWF MARS language definition.

Finally `"payload": "location"` is used to substitute the word *payload* with the word *location* in the notifications. This helps to customise the notifications to its domain; in the case of ECMWF data availability, location indicates where to access to the data.

CHAPTER

# ELEVEN

# CONFIGURATION

A number of settings can be edited. For each of them users can override the defaults by means of one or a combination of mechanisms. The final configuration used by the application is the result of the following sequence where each step merges on the previous one:

1. Loading defaults
2. Loading system config file */etc/aviso/config.yaml*
3. Loading Home config file *~/.aviso/config.yaml*
4. Loading config defined by environment variables, AVISO_CONFIG
5. Loading config defined by command line option, `-c`, `--config`
6. Loading all environment variables
7. Loading all command line options

System and Home config files are optional.

The rest of this chapter presents all the settings available. For each of them, we present the type, the default value, how to change them using command line options, environment variables or the configuration file. Not all these mechanisms are available for all settings.

## 11.1 Application

These settings are applied at application level.

### 11.1.1 Logging

The application takes advantage for the Python logging module. Users can define a custom file configuration and pass it using any of the following methods.

| | |
|---|---|
| Type | string, file path |
| Defaults | Info log on console output |
| Command Line options | `-l`, `--log`, |
| Environment variable | AVISO_LOG |
| Configuration file | `logging: <logging configuration>` |

**Note:** The configuration file method accepts directly the logging configuration rather than a file path to it.

### 11.1.2 Debug

If True the application will show the debug logs to the console output.

| | |
|---|---|
| Type | boolean |
| Defaults | False |
| Command Line options | `-d`, `--debug` |
| Environment variable | AVISO_DEBUG |
| Configuration file | `debug: False` |

### 11.1.3 Quiet

If True the application will not show any info logs to the console output. Only errors will be displayed.

| | |
|---|---|
| Type | boolean |
| Defaults | False |
| Command Line options | `-q`, `--quiet` |
| Environment variable | AVISO_QUIET |
| Configuration file | `quiet: False` |

### 11.1.4 No Fail

If True the application will always exit with error code 0, even in case of errors. This can be useful when used in a automated workflow that is required not to stop even if Aviso exits because of errors.

| | |
|---|---|
| Type | boolean |
| Defaults | False |
| Command Line options | `--no-fail` |
| Environment variable | AVISO_NO_FAIL |
| Configuration file | `no_fail: False` |

### 11.1.5 Authentication Type

Type of authentication to use when talking to the server. `ecmwf` is required if accessing to the ECMWF Aviso service. See *How Aviso Works at ECMWF* for more information. In case of talking directly to the store the other authentication methods may be used. If `none` is selected, settings as `username`, `username_file` or `key` will be ignored.

| | |
|---|---|
| Type | Enum [ecmwf, etcd, none] |
| Defaults | none |
| Command Line options | N/A |
| Environment variable | AVISO_AUTH_TYPE |
| Configuration file | `auth_type: none` |

### 11.1.6 Username

This is used to authenticate the requests to the server.

| | |
|---|---|
| Type | string |
| Defaults | None |
| Command Line options | `-u`, `--username` |
| Environment variable | AVISO_USERNAME |
| Configuration file | `username: xxxx` |

### 11.1.7 Username File

If set, the username will be read from the file defined. This takes priority over *username*.

| | |
|---|---|
| Type | string, file path |
| Defaults | None |
| Command Line options | N/A |
| Environment variable | AVISO_USERNAME_FILE |
| Configuration file | `username_file: xxxx` |

### 11.1.8 Key

File from where to read the password to use to authenticate the requests to the server.

| | |
|---|---|
| Type | string, file path |
| Defaults | /etc/aviso/key |
| Command Line options | `-k`, `--key` |
| Environment variable | AVISO_KEY_FILE |
| Configuration file | `key_file: /etc/aviso/key` |

### 11.1.9 Schema Parser

Type of parser to use to read the event listener schema. `ecmwf` is required if accessing to the ECMWF Aviso service.

| | |
|---|---|
| Type | Enum [generic, ecmwf] |
| Defaults | generic |
| Command Line options | N/A |
| Environment variable | AVISO_SCHEMA_PARSER |
| Configuration file | `schema_parser: generic` |

### 11.1.10 Remote Schema

If *False* the listener schema is read locally from the expected default location. In this case all the configuration engine settings are ignored. If *True* the listener schema is retrieved dynamically from the configuration server when the application starts. More info in *Configuration Management*

| | |
|---|---|
| Type | boolean |
| Defaults | False |
| Command Line options | N/A |
| Environment variable | AVISO_REMOTE_SCHEMA |
| Configuration file | `remote_schema: False` |

## 11.2 Notification Engine

This group of settings defines the connection to the notification server. The current defaults allow the connection to a default *etcd* local installation.

### 11.2.1 Host

| | |
|---|---|
| Type | string |
| Defaults | localhost |
| Command Line options | `-H`, `--host` |
| Environment variable | AVISO_NOTIFICATION_HOST |
| Configuration file | `notification_engine:`<br>`  host: localhost` |

### 11.2.2 Port

| | |
|---|---|
| Type | integer |
| Defaults | 2379 |
| Command Line options | `-P`, `--port` |
| Environment variable | AVISO_NOTIFICATION_PORT |
| Configuration file | `notification_engine:`<br>`  port: 2379` |

### 11.2.3 Type

This defines the protocol to use to connect to the server. In case of `file_based` Aviso will run in *TestMode* by connecting to a local store, part of Aviso itself. In this mode, users can execute any of the commands described in *Notification CLI*. The only restriction applies to retrieving past notifications that are not available. See *Testing my Listener* for more info. In case of `etcd_grpc` or``etcd_rest`` Aviso will connect to a etcd store either by its native gRPC API or by the RESTfull API implemented by the etcd gRPC gateway.

| | |
|---|---|
| Type | Enum: [ etcd_rest, etcd_grpc, file_based ] |
| Defaults | etcd_rest |
| Command Line options | N/A |
| Environment variable | AVISO_NOTIFICATION_ENGINE |
| Configuration file | `notification_engine:`<br>`  type: etcd_rest` |

### 11.2.4 Polling Interval

Number of seconds between successive requests of new notifications to the server .

| | |
|---|---|
| Type | integer, seconds |
| Defaults | 30 |
| Command Line options | N/A |
| Environment variable | AVISO_POLLING_INTERVAL |
| Configuration file | `notification_engine:`<br>`  polling_interval: 30` |

### 11.2.5 Timeout

Timeout for the requests to the notification sever

| | |
|---|---|
| Type | integer, seconds |
| Defaults | 60 |
| Command Line options | N/A |
| Environment variable | AVISO_TIMEOUT |
| Configuration file | `notification_engine:`<br>`  timeout: 60` |

### 11.2.6 HTTPS

| | |
|---|---|
| Type | boolean |
| Defaults | False |
| Command Line options | N/A |
| Environment variable | AVISO_NOTIFICATION_HTTPS |
| Configuration file | `notification_engine:`<br>`  https: False` |

### 11.2.7 Catchup

If True the application will start retrieving first the missed notifications and then listening to the new ones. See *Notification Catch-up* for more information.

| | |
|---|---|
| Type | boolean |
| Defaults | True |
| Command Line options | `--catchup` |
| Environment variable | AVISO_NOTIFICATION_CATCHUP |
| Configuration file | `notification_engine:`<br>`  catchup: True` |

### 11.2.8 Service

Key identifying Aviso application in the configuration management system. See *Configuration Management* for more information.

| | |
|---|---|
| Values | string |
| Defaults | aviso/v1 |
| Command Line options | N/A |
| Environment variable | AVISO_NOTIFICATION_SERVICE |
| Configuration file | `notification_engine:`<br>`  service: "aviso/v1"` |

### 11.2.9 AUTOMATIC RETRY DELAY

Number of seconds to wait before retrying to connect to the notification sever. This prevents the application to terminate in case of temporarily network issues for example.

| | |
|---|---|
| Type | integer, seconds |
| Defaults | 15 |
| Command Line options | N/A |
| Environment variable | AVISO_AUTOMATIC_RETRY_DELAY |
| Configuration file | `notification_engine:`<br>`  automatic_retry_delay: 15` |

## 11.3 Configuration Engine

This group of settings defines the connection to the configuration management server. The current defaults allows connecting to a default *etcd* local installation. This is however not a requirement and different servers can be used. See *Configuration Management* for more information.

### 11.3.1 Host

| | |
|---|---|
| Type | string |
| Defaults | localhost |
| Command Line options | `-H, --host` |
| Environment variable | AVISO_CONFIGURATION_HOST |
| Configuration file | `configuration_engine:`<br>`  host: localhost` |

### 11.3.2 Port

| | |
|---|---|
| Type | integer |
| Defaults | 2379 |
| Command Line options | `-P`, `--port` |
| Environment variable | AVISO_CONFIGURATION_PORT |
| Configuration file | `configuration_engine:`<br>`  port: 2379` |

### 11.3.3 Type

| | |
|---|---|
| Type | Enum: [ etcd_rest, etcd_grpc ] |
| Defaults | etcd_rest |
| Command Line options | N/A |
| Environment variable | AVISO_CONFIGURATION_ENGINE |
| Configuration file | `configuration_engine:`<br>`  type: etcd_rest` |

### 11.3.4 Timeout

Timeout for the requests to the notification sever

| | |
|---|---|
| Type | integer, seconds |
| Defaults | 60 |
| Command Line options | N/A |
| Environment variable | AVISO_TIMEOUT |
| Configuration file | `configuration_engine:`<br>`  timeout: 60` |

### 11.3.5 HTTPS

| | |
|---|---|
| Type | boolean |
| Defaults | False |
| Command Line options | N/A |
| Environment variable | AVISO_CONFIGURATION_HTTPS |
| Configuration file | `configuration_engine:`<br>`  https: False` |

### 11.3.6 Max File Size

This is the maximum file size allowed by during a push operation.

| | |
|---|---|
| Type | integer, KiB |
| Defaults | 500 |
| Command Line options | `--catchup` |
| Environment variable | AVISO_MAX_FILE_SIZE |
| Configuration file | `configuration_engine:`<br>`  max_file_size: 500` |

### 11.3.7 AUTOMATIC RETRY DELAY

Number of seconds to wait before retrying to connect to the configuration sever. This prevents the application to terminate in case of temporarily network issues for example.

| | |
|---|---|
| Type | integer, seconds |
| Defaults | 15 |
| Command Line options | N/A |
| Environment variable | AVISO_AUTOMATIC_RETRY_DELAY |
| Configuration file | `configuration_engine:`<br>`  automatic_retry_delay: 15` |

CHAPTER

TWELVE

# TRIGGERS

This section lists the various triggers currently available. Each trigger will result in an independent process executed every time a notification is received.

## 12.1 Echo

This is the simplest trigger as it prints the notification to the console output. It is used for testing and it does not accept any extra parameters.

```
triggers:
  - type: echo
```

## 12.2 Log

This trigger logs the event to a log file. It is useful for recording the received event.

```
triggers:
  - type: log
    path: testLog.log
```

**Note:** The trigger process will fail if the directory does not exist.

## 12.3 Command

This trigger allows the user to define a shell command to work with the notification.

```
triggers:
  - type: command
    working_dir: $HOME/aviso/examples
    command: ./script.sh --date ${request.date} --number ${request.number}
    environment:
      AIRPORT: ${request.airport}
      COUNTRY: "The country is ${request.country}"
```

- `command` is the command that will be executed for each notification received. This is a mandatory field.

- `environment` is a user defined list of local variables that will be passed to the command shell. This is an optional field.
- `working_dir` defines the working directory that will be set before executing the command. This is an optional field.

Moreover, the system performs a parameter substitution in the command and environment fields, for every sequence of the pattern:

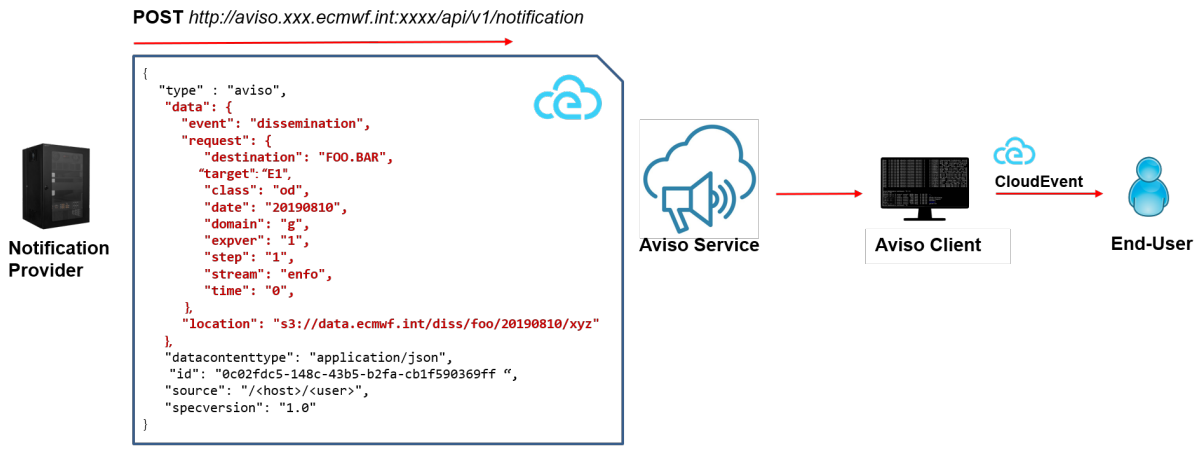
- `${name}`, it replaces it with the value associated to the corresponding key found in the notification received.
- `${json}`, it replaces it with the whole notification formatted as a JSON inline string.
- `${jsonpath}`, it replaces it with the file name of a JSON file containing the notification.

A notification is a dictionary whose keys can be used in the parameter substitution mechanism described above. Here is an example of a notification:

```
{
  "event": "flight",
  "payload": "Landed",
  "request": {
     "country": "italy",
     "date": "20210101",
     "airport": "FCO",
     "number": "AZ203"
  }
}
```

## 12.4 Post

This trigger will format the notification according to the CloudEvents specification and will send it to either a endpoint as HTTP POST request or to a AWS Simple Notification Service (SNS) topic. This trigger basically turns Aviso client in a proxy forwarding the notification to the user's notification system compatible with CloudEvents specification, as shown by the figure below:




Here is a basic example of a Post trigger sending the notification to a HTTP endpoint defined by the user. The type is `cloudevents_http` and `url` is the only mandatory parameter.

```
triggers:
  - type: post
    protocol:
      type: cloudevents_http
      url: http://my.endpoint.com/api
```

This is the basic configuration. More parameters can be specified to customise the CloudEvents message. More info in the reference documentation.

The CloudEvents message sent would look like the following:

```
{
  "type" : "aviso",                                 # this is customisable by the user
  "data": {                                         # this is aviso specific
    "event": "flight",
    "payload": "Landed",
    "request": {
       "country": "italy",
       "date": "20210101",
       "airport": "FCO",
       "number": "AZ203"
    }
  },
  "datacontenttype": "application/json",
  "id": "0c02fdc5-148c-43b5-b2fa-cb1f590369ff",  # UUID random generated by aviso
  "source": "https://aviso.int",           # this is customisable by the user
  "specversion": "1.0",
  "time": "2020-03-02T13:34:40.245Z",            # Timestamp of when this message is␣
↪created
}
```

Here is a complete example showing how to customise the CloudEvents fields as well as the HTTP headers using optional parameters:

```
triggers:
  - type: post
    protocol:
      type: cloudevents_http
      url: http://my.endpoint.com/api
      headers:
        HTTP_TEST: "test"
      timeout: 30
      cloudevents:
        type: test_cloudevent
        source: my_test
```

In the case of a notification to a AWS SNS topic defined by the user, the structure of the trigger is similar; the type has to be `cloudevents_aws` and `arn` and `region_name` are the only mandatory parameters.

The optional parameters are: `MessageAttributes`, `aws_access_key_id`, `aws_secret_access_key` for the AWS topic fields and `cloudevents` for the CloudEvents fields. Note that if `aws_access_key_id` and `aws_secret_access_key` are not specified the AWS credentials are taken from *~/.aws/credentials* if available.

AWS SNS protocol does not enforce any specification on the message payload. Aviso uses the CloudEvents specification also in this case for consistency.

```
triggers:
  - type: post
    protocol:
      type: cloudevents_aws
      arn: arn:aws:sns:us-east-2:848972885776:aviso
      region_name: us-east-2
      MessageAttributes:
        attribute1:
          DataType: String
          StringValue: valueAttribute1
        attribute2:
          DataType: String
          StringValue: valueAttribute2
      cloudevents:
        type: aviso_topic
        source: my_test
```

Finally, in case of a AWS FIFO topic `MessageGroupId` is required.

## 12.5 Function

Differently from the previous triggers, this trigger is not file based. It allows the user to define a Python function to be executed directly by Aviso. This is intended for users that want to integrate Aviso Python API into a workflow or application written in Python.

Below find an example of a python script that defines a function to be executed once a notification is received, creates a listener that references to this function trigger and finally passes it to aviso to execute.

```
from pyaviso import NotificationManager

# define function to be called
def do_something(notification):
   print(f"Notification for step {notification['request']['step']} received")
   # now do something useful with it ...

# define the trigger
trigger = {"type": "function", "function": do_something}

# create a event listener request that uses that trigger
request = {"country": "Italy"}
listeners = {"listeners": [{"event": "flight", "request": request, "triggers": [trigger]}
↪]}

# run it
aviso = NotificationManager()
aviso.listen(listeners=listeners)
```

See *Python API* for more info on how to use Aviso API.

CHAPTER

# THIRTEEN

# NOTIFICATION CLI

Aviso provides a Command Line Interface (CLI) for listening to notifications from the server system described in *Aviso Server Architecture*. This section describes in detail the various commands associated with this functionality.

```
% aviso -h
Options:
 --version   Show the version and exit.
 -h, --help  Show this message and exit.

Commands:
 key     Generate the key to send to the notification server according to...
 listen  This method allows the user to execute the listeners defined in...
 notify  Create a notification with the parameters passed and submit it to...
 value   Return the value on the server corresponding to the key which is...
```

## 13.1 Listen

This command allows to listen to notifications compliant with the listeners defined:

```
aviso listen -h
Usage: aviso listen [OPTIONS] [LISTENER_FILES]...

This method allows the user to execute the listeners defined in the YAML
listener file

:param listener_files: YAML file used to define the listeners

Options:
-c, --config TEXT               User configuration file path.
-l, --log TEXT                  Logging configuration file path.
-d, --debug                     Enable the debug log.
-q, --quiet                     Suppress non-error messages from the console output.
--no-fail                       Suppress any error exit code.
-u, --username TEXT             Username required to authenticate to the server.
-k, --key TEXT                  File path to the key required to authenticate to the␣
↪server.
-H, --host TEXT                 Notification server host.
-P, --port INTEGER              Notification server port.
--test                          Activate TestMode.
```

(continues on next page)

(continued from previous page)

```
--from [%Y-%m-%dT%H:%M:%S.%fZ]  Replay notification from this date.
--to [%Y-%m-%dT%H:%M:%S.%fZ]    Replay notification to this date.
--now                           Ignore missed notifications, only listen to new ones.
--catchup                       Retrieve first the missed notifications.
-h, --help                      Show this message and exit.
```

The parameter `listener_files` is used to define the event listeners and the triggers to execute in case of notifications. If not present the system will look for the default listeners which can be defined in the configuration files. Here is an example of invoking this command with one listener file:

```
aviso listen examples/echoListener.yaml
```

Once in execution this command will create a background process waiting for notifications and a foreground process in busy waiting mode. Multiple files can also be indicated as shown below:

```
aviso listen listener1.yaml listener2.yaml
```

Most of the options accepted by this command are used to change the application configuration. Below are presented only the options that are not covered by the *Configuration* section.

### 13.1.1 No fail

If the option `--no-fail` is present, the application will always exit with error code 0, even in case of errors. This can be useful when used in a automated workflow that is required not to stop even if Aviso exits because of errors.

### 13.1.2 Test

If the option `--test` is present, the application will run in *TestMode*. See *Testing my Listener* for more information.

### 13.1.3 Now

If the option `--now` is present, the application will start ignoring the missed notifications while listening only to the new ones. See *Notification Catch-up* for more information.

### 13.1.4 Catchup

If the option `--catchup` is present, the application will start retrieving first the missed notifications and then listening to the new ones. See *Notification Catch-up* for more information. This option is enabled by default. See *Configuration* for more information.

## 13.2 Key

This command can be used to generate the key accepted by the notification server as part of the notification key-value pair. This command is mostly used for debugging.

```
% aviso key -h
Usage: aviso key [OPTIONS] PARAMETERS

Generate the key to send to the notification server according to the
current schema using the parameters defined

:param parameters: key1=value1,key2=value2,...

Options:
-c, --config TEXT    User configuration file path.
-l, --log TEXT       Logging configuration file path.
-d, --debug          Enable the debug log.
-q, --quiet          Suppress non-error messages from the console output.
--no-fail            Suppress any error exit code.
-u, --username TEXT  Username required to authenticate to the server.
-k, --key TEXT       File path to the key required to authenticate to the
                    server.
-H, --host TEXT      Notification server host.
-P, --port INTEGER   Notification server port.
--test               Activate TestMode.
-h, --help           Show this message and exit.
```

Here is an example of this command:

```
aviso key event=flight,country=Italy,airport=fco,date=20210101,number=AZ203
```

Note all the keys are required. The output from this command will be something like:

```
/tmp/aviso/flight/20210101/italy/FCO/AZ203
```

Note how the format and the order of the parameters have been adjusted to complying with the listener schema presented in *Getting Started*

All the options accepted by this command are covered in *Listen* and in *Configuration*.

## 13.3 Value

This command is used to retrieve from the store the value associated to a specific key using the same syntax of the command `key`.

```
% aviso value -h
Usage: aviso value [OPTIONS] PARAMETERS

Return the value on the server corresponding to the key which is generated
according to the current schema and the parameters defined

:param parameters: key1=value1,key2=value2,...
```

(continues on next page)

(continued from previous page)

```
Options:
-c, --config TEXT    User configuration file path.
-l, --log TEXT       Logging configuration file path.
-d, --debug          Enable the debug log.
-q, --quiet          Suppress non-error messages from the console output.
--no-fail            Suppress any error exit code.
-u, --username TEXT  Username required to authenticate to the server.
-k, --key TEXT       File path to the key required to authenticate to the
                    server.
-H, --host TEXT      Notification server host.
-P, --port INTEGER   Notification server port.
--test               Activate TestMode.
-h, --help           Show this message and exit.
```

Here is an example of this command:

```
aviso value event=flight,country=Italy,airport=fco,date=20210101,number=AZ203
```

Note the list of parameters required, this is the same list required by the `key` command. The output from this command will be something like:

```
Landed
```

Not all keys have corresponding values because it is optional. In this case the output would be `None`

All the options accepted are covered in *Listen* and in *Configuration*.

## 13.4 Notify

This command is used to directly send a notification to the server using the same syntax of the command `key`

```
% aviso notify -h
Usage: aviso notify [OPTIONS] PARAMETERS

Create a notification with the parameters passed and submit it to the
notification server :param parameters: key1=value1,key2=value2,...

Options:
-c, --config TEXT    User configuration file path.
-l, --log TEXT       Logging configuration file path.
-d, --debug          Enable the debug log.
-q, --quiet          Suppress non-error messages from the console output.
--no-fail            Suppress any error exit code.
-u, --username TEXT  Username required to authenticate to the server.
-k, --key TEXT       File path to the key required to authenticate to the
                    server.
-H, --host TEXT      Notification server host.
-P, --port INTEGER   Notification server port.
--test               Activate TestMode.
-h, --help           Show this message and exit.
```

Here is an example of this command:

```
aviso notify event=flight,country=Italy,airport=fco,date=20210101,number=AZ203,
↪payload=Landed
```

Note the list of parameters required, this is the same list required by the `key` command with the addition of the `payload` pair. This is needed to assign a value to the key that will be saved into the store. If not given the value will be `None`. This last case is used when only an acknowledgement that something happened is needed.

All the options accepted by this command are covered in *Listen* and in *Configuration*.

CHAPTER

# FOURTEEN

# PYTHON API

Aviso provides a Python API for the key operations that concern the notification workflow: `listen` and `notify`. This API has the same level of expressiveness as the CLI. Moreover users can create and customise a `user_config.UserConfig` object. This object allows to programmatically define any setting described in in *Configuration*.

This is intended for users that want to integrate Aviso in a workflow or application written in Python. An example of integration of Aviso in a external Python application is the server component **Aviso REST**, described in *Aviso Server Architecture*. This component internally relies on Aviso client to submit notifications to the store.

## 14.1 Listen

This method is used to start the polling for changes from Aviso client to Aviso server. This allows the user to retrieve new notifications as they are submitted to Aviso server.

Below is an example of a python script that defines a function to be executed once a notification is received, creates a listener that references to this function trigger and finally passes it to Aviso to execute.

```
from pyaviso import NotificationManager

# define function to be called
def do_something(notification):
   print(f"Notification for step {notification['request']['step']} received")
   # now do something useful with it ...

# define the trigger
trigger = {"type": "function", "function": do_something}

# create a event listener request that uses that trigger
request = {"country": "Italy"}
listeners = {"listeners": [{"event": "flight", "request": request, "triggers": [trigger]}
↪]}

# run it
aviso = NotificationManager()
aviso.listen(listeners=listeners)
```

This script will put the main process is busy waiting while polling at regular time the server. All the various types of triggers presented in *Triggers* can also be defined or manually loaded from file.

The object `NotificationManager` can take as parameter a `UserConfig` object that the user can create and customise. If not passed the manager object will instantiate a config object that follows the criteria explained in *Configuration*. This example shows the latter, moreover, it is using the default listener schema presented in *Make Your Own Event*.

## 14.2 Notify

This method is used to submit notification. The example belows shows how to send a generic notification compliant with the generic listener schema presented in *Make Your Own Event*

```
from pyaviso import NotificationManager

aviso = NotificationManager()

# define the parameters of the notification
notification = {
   "event":"flight",
   "country": "italy",
   "date": "20210101",
   "airport": "FCO",
   "number": "AZ203",
   "payload": "Landed"
}

# send the notification
aviso.notify(notification)
```

CHAPTER

FIFTEEN

# CONFIGURATION MANAGEMENT

Aviso can also be used to store and retrieve configuration files for external applications. In this case it acts as a configuration management system.

From the sever side storing configurations equates to a key-value pair where the key is the configuration file path and the value is the configuration content. This means that the two Aviso functionalities, notification and configuration, share the same server technology and architecture, and therefore most of user options presented in *Notification CLI*.

This functionality can be used as part of the Aviso notification workflow. Specifically, by enabling the `remote_schema` flag, Aviso will dynamically pull the event listener schema when Aviso client starts. This allows to share and update this schema with the notification providers. The notification provider is required to comply with the notification format otherwise the notification will be wrongly identified by the listeners. This solution exploits the scalability of the server architecture already in place. See *Configuration* on how to enable it.

The following section presents the commands available with the configuration CLI.

```
% aviso-config -h
Usage: aviso-config [OPTIONS] COMMAND [ARGS]...

Options:
--version   Show the version and exit.
-h, --help  Show this message and exit.

Commands:
pull    Pull all files associated with the service defined.
push    Push all files from the directory selected to the service...
remove  Remove all files associated with the service defined.
revert  Revert all files associated with the service defined to the...
status  Retrieve the status of the service defined.
```

**Note:** The commands above inherit the options and configuration described in *Notification CLI* and in *Configuration*. These options are then omitted from the descriptions that follow.

## 15.1 Pull

The pull operation is used to retrieve the configuration files of a specific service.

```
% aviso-config pull -h
Usage: aviso-config pull [OPTIONS] SERVICE

Pull all files associated with the service defined.

Options:
-H, --host TEXT      Configuration server host.
-P, --port INTEGER   Configuration server port.
-D, --dir TEXT       Destination directory to pull into.
--delete             Allows delete of local files if they do not exist on the server.
```

Below is an example of how to use it:

```
aviso-config pull aviso/v1 --dir config/event_listener/
```

In this case the configuration files associated to the service `aviso/v1` will be pulled and saved in the directory indicated. If any of these files is already present it will be overridden.

**Note:** Options `-H` and `-P` are used to set the configuration server as aviso-config does not use any notification server.

## 15.2 Push

The push operation is used to push configuration files of a specific service.

```
% aviso-config push -h
Push all files from the directory selected to the service defined,
respecting the subdirectory structure.

Options:
-H, --host TEXT      Configuration server host.
-P, --port INTEGER   Configuration server port.
-D, --dir TEXT       Directory to push.  [required]
-m, --message TEXT   Message to associate to the push.  [required]
--delete             Allows delete of files on server if they do not exist locally.
```

Below is an example of how to use it:

```
aviso-config push aviso/v1 --dir config/event_listener/ -m 'event listener schema update'
```

In this case the content of the directory *config/event_listener* is pushed under the service `aviso/v1` Note that every time something is pushed to a service location, the service status is updated with the message passed and the user information and the version are incremented.

## 15.3 Remove

The remove operation is used to remove all the configuration files of a specific service.

```
% aviso-config remove -h
Usage: aviso-config remove [OPTIONS] SERVICE

Remove all files associated with the service defined.

Options:
-H, --host TEXT      Configuration server host.
-P, --port INTEGER   Configuration server port.
-f, --doit           Remove without prompt.
```

Below is an example of how to use it:

```
aviso-config remove aviso/v1 -f
```

In this case the configuration files associated to the service passed will all be removed from the configuration server.

Without the option `-f` the application only lists the files associated to the service. It can therefore be used just to list the files associated with the service.

## 15.4 Revert

The revert operation is used to restore the previous version of all the configuration files of a specific service.

```
% aviso-config revert -h
Usage: aviso-config revert [OPTIONS] SERVICE

Revert all files associated with the service defined to the previous
version.

Options:
-H, --host TEXT      Configuration server host.
-P, --port INTEGER   Configuration server port.
```

Below is an example of how to use it:

```
aviso-config revert aviso/v1
```

---

**Note:** If this command is run twice consecutively, this results in no changes to the files on the server but the version will be incremented.

---

## 15.5 Status

The status operation is used to retrieve the status of a specific service.

```
% aviso-config status -h
Usage: aviso-config status [OPTIONS] SERVICE

Retrieve the status of the service defined.

Options:
-H, --host TEXT      Configuration server host.
-P, --port INTEGER   Configuration server port.
```

Below is an example of how to use it:

```
aviso-config status aviso/v1
```

This would return something on these lines:


```
{
   "aviso_version": "0.3.0",
   "date_time": "2020-02-04T16:25:45.521Z",
   "engine": "ETCD_REST",
   "etcd_user": "root",
   "hostname": "viron",
   "message": "test",
   "prev_rev": 55054,
   "unix_user": "maci",
   "version": 23
}
```

CHAPTER

SIXTEEN

# WHAT'S NEW

## 16.1 v0.11.1 (02 February 2022)

HTTP 404 has been added among the exceptions handled by the automatic restart mechanism of aviso listeners. This is needed during maintenance sessions.

## 16.2 v0.11.0 (14 December 2021)

The main new feature of this release is the extension of the Post trigger to support AWS topics. More info available in the dedicated page *Post*.

### 16.2.1 Breaking changes

Post trigger of type `cloudevents` is now of type `cloudevents_http`. This to distinguish it from the Post trigger to AWS topic that is of type `cloudevents_aws`

## 16.3 v0.10.0 (26 April 2021)

The main new feature of this release is the implementation of an automatic retry mechanism for the listening process to reconnect in case of network issues or sever unavailability. Thanks to this the listening process should never require a manual restart.

## 16.4 v0.9.2 (4 February 2021)

First public release.

CHAPTER

SEVENTEEN

# HOW TO DEVELOP

Aviso source code is available on GitHub at https://github.com/ecmwf/aviso

Please report any issues on GitHub at https://github.com/ecmwf/aviso/issues

Below a few steps to guide the development:

- Clone Aviso repository:

```
git clone https://github.com/ecmwf/aviso.git
```

- Install pyaviso for development, from inside the main aviso folder:

```
pip install -e .
```

- Install development dependencies:

```
pip install -U -r tests/requirements-dev.txt
```

- Unit and system tests for pyaviso can be run with pytest with:

```
pytest tests -v --cov=pyaviso --cache-clear
```

- Ensure to comply with PEP8 code quality:

```
tox -e quality
```

**Note:** In order to run the tests, an instance of etcd has to run on `127.0.0.1/2379` with default configurations. Please check *Getting Started* for more info on how to install it.

To develop on the Aviso server components:

- Install the following module:

```
pip install -e aviso-server/monitoring
pip install -e aviso-server/rest
pip install -e aviso-server/auth
pip install -e aviso-server/admin
```

- Before submitting a pull request run all tests and code quality check:

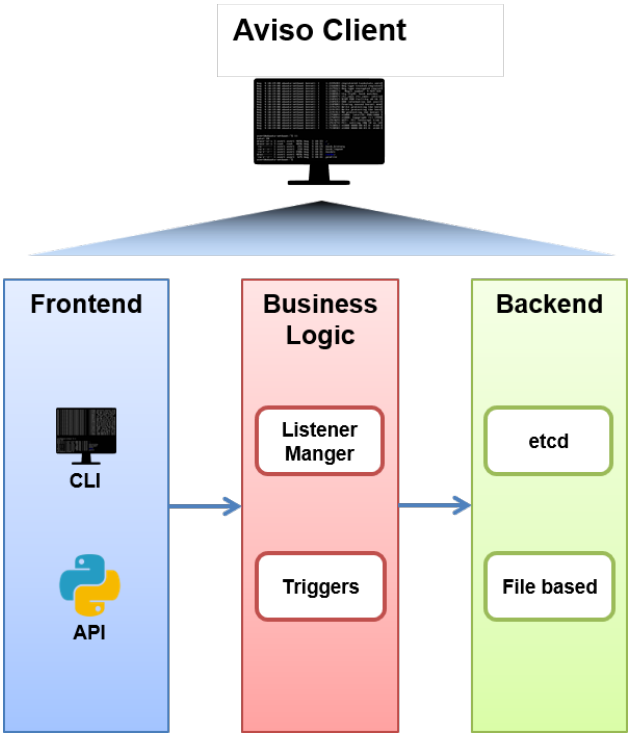
```
tox
```

CHAPTER

# EIGHTEEN

# AVISO CLIENT ARCHITECTURE

Figure below shows the high-level architecture of Aviso client.




The whole Aviso Client application is implemented by the project package `pyaviso`. It provides two kinds of interfaces to users, a Python API for integration in user's applications and a Command Line Interface (CLI) to use it as standalone application. The latter uses internally the Python API, implemented by the `notification_manager` module. This module relies on the `listener_manager` module to translate the user's listener configuration into requests for the store.

The Listener Manager encapsulates the domain-specific listener semantic and is therefore in charge of the listener validation and the creation of the various `EventListener`. These entities map users' requests and represent independent listening threads that execute the triggers as independent processes in case of a valid notification is received.

The backend of the application is implemented by the `engine` package. The `Engine` offers a common interface to the requests arriving from the business layer and directed to the key-value store. Different implementations are available depending on the protocol used by the Key-Value store. Currently the store considered are etcd and a file-based store used only in *TestMode*.

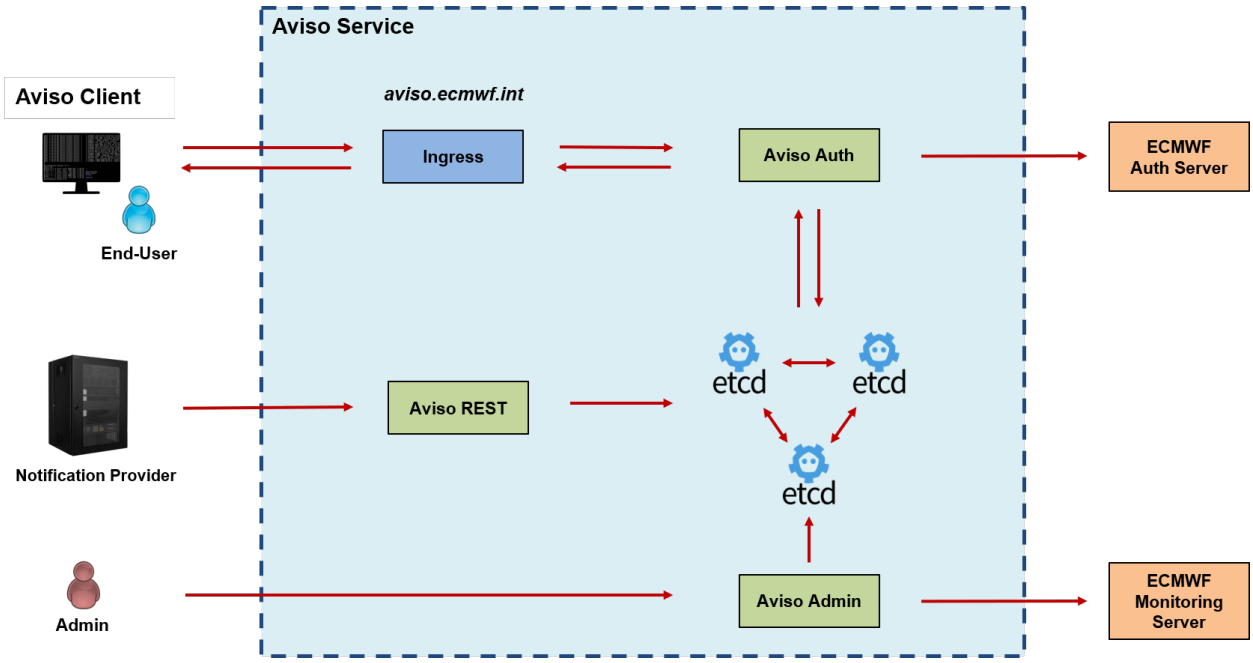
CHAPTER

NINETEEN

# AVISO SERVER ARCHITECTURE

This section presents the general architecture for Aviso Server. It shows components that are ECMWF specific such as Aviso Auth while others that are generic such as the key-value store. This architecture can therefore evolve and adapt to different infrastructures that are available to the users.

Figure below shows the current high-level architecture of Aviso server.




The source of the components presented here is available in the `aviso-server` folder of the project. The remaining part of this section, briefly introduces each component.

## 19.1 Key-Value store

The core component of Aviso Server is a Key-Value store. This is a critical component because it guarantees persistence and consistency for all the notifications processed. The current Key-Value store technology used is etcd. This is a strongly consistent, distributed key-value store able to reach consensus thanks to Raft algorithm. This allows it to gracefully tolerate machine failure and network partition. Moreover, it is designed for high-throughput. We are running it in is default configuration of a cluster of 3 components.

**Note:** All the other components of the Aviso Server are built independent of the technology used for the store. The

same applies for Aviso Client that completely hides to the user the *etcd* API.

## 19.2 Aviso REST

This component is a REST frontend that allows notification providers to submit notifications to the store via a REST interface. Internally it uses Aviso Python API as if it was a client towards the store.

Install it by, from the main project directory:

```
pip install -e .
pip install -e aviso-server/monitoring
pip install -e aviso-server/rest
```

The *aviso* and *aviso-monitoring* packages are required by *aviso-rest*.

Launch it by:

```
aviso-rest
```

## 19.3 Aviso Auth

Aviso Auth is a web application implementing a proxy responsible for authenticating the end-users' requests directed to the store. This allows to not rely on the store native authentication and authorisation capability while using ECMWF centralised resources. It follows a 2-steps process:

1. The request is validated against the ECMWF authentication server by checking the token associated to the request.
2. The user associated to the token is checked if he can access to the resource is asking notifications for. This is performed by requesting the allowed resources associated to the user from the ECMWF authorisation server.

If both steps are successful the request is forwarded to the store.

**Note:** Currently only the `listen` command is allowed by this component. Any other operation is not authorised.

Install it by, from the main project directory:

```
pip install -e aviso-server/monitoring
pip install -e aviso-server/auth
```

The *aviso-monitoring* package is required by *aviso-auth*.

Launch it by:

```
aviso-auth
```

## 19.4 Aviso Admin

This component performs maintenance operations to the store in order to keep it at constant size. Currently the implementation is specific for an etcd store. This store requires the following operations:

- Compaction, this operation removes the history older than a certain date
- Deletion, this operation deletes all the keys older than a certain date

This component also uses the _monitoring_ package to run a UDP server to receive telemetries from all the other components on the server. It runs a periodic aggregation and evaluation of these telemetries and it then communicates the status of the components to the ECMWF monitoring server.

Install it by, from the main project directory:

```
pip install -e aviso-server/monitoring
pip install -e aviso-server/admin
```

The *aviso-monitoring* package is required by *aviso-admin*.

Launch it by:

```
aviso-admin
```

## 19.5 Monitoring

The package called `aviso_monitoring` allows the implementation of the monitoring system designed for the Aviso Server. It is a library that any other components can use either for:

- Collecting telemetries inside the component application, aggregate them and send them via UDP package
- Collecting telemetries from other components via a UDP server, aggregate and evaluate them and send them to a monitoring server.

The first capability is currently used by the components Aviso Rest and Aviso Auth. The second capability is used by the Aviso Admin component.

Install it by, from the main project directory:

```
pip install -e aviso-server/monitoring
```

CHAPTER

# TWENTY

# LICENSE

*Aviso* is available under the open source Apache License. In applying this licence, ECMWF does not waive the privileges and immunities granted to it by virtue of its status as an intergovernmental organisation nor does it submit to any jurisdiction.