dasf-messaging-python

Daniel Eggert, Mike Sips, Philipp S. Sommer, Doris Dransch

Apr 15, 2024

TABLE OF CONTENTS

Data Analytics Software Framework

The python module wrapper for the data analytics software framework DASF

The **D**ata **A**nalytics **S**oftware **F**ramework DASF supports scientists to conduct data analysis in distributed IT infrastructures by sharing data analysis tools and data. For this purpose, DASF defines a remote procedure call (RPC) messaging protocol that uses a central message broker instance. Scientists can augment their tools and data with this protocol to share them with others or re-use them in different contexts.

CHAPTER

ONE

LANGUAGE SUPPORT

The DASF RPC messaging protocol is based on JSON and uses Websockets for the underlying data exchange. Therefore the DASF RPC messaging protocol in general is language agnostic, so all languages with Websocket support can be utilized. As a start DASF provides two ready-to-use language bindings for the messaging protocol, one for Python and one for the Typescript programming language.

CHAPTER

TWO

NEEDED INFRASTRUCTURE

DASF distributes messages via a central message broker. Currently we support a self-developed message broker called dasf-broker-django, as well as an 'off-the-shelf' solution called Apache Pulsar. Both are quite versatile and highly configurable depending on your requirements. (also see: *Choosing the message broker*)

2.1 Getting Started with DASF

For now DASF supports two application scenarios, python2python and typescript2python.

In case of python2python you 'simply' call a remote procedure (provided in python) from another python context. The typescript2python scenario basically does the same but introduces a context switch from the python based remote procedure to a web-based frontend. But let's start with a simple python2python example first.

2.1.1 Remotely call a python procedure from a python context

Remote procedure calls (RPCs) consist of two parts: a server and a client side. In this context the server exposes a certain function which is can be called by the client side. In the scope of DASF the server side exposing certain functions is called `backend module`.

The backend module (server side)

Install DASF

The python package name of the DASF python bindings is called `demessaging` and is provided via pypi. You can install it via `pip install demessaging[backend]`

Let's assume we have a python function that returns a `HelloWorld` string.

Listing 1: simple hello_world function

```
1 def hello_world():
2   return 'HelloWorld'
```

In order to expose this function through a DASF backend module all we have to do is import and call the `main` function from the `demessaging` package and register all functions that you want to expose via `__all__`. So our example above becomes:

Listing 2: hello_world function exposed via a HelloWorld backend module.

```python
from demessaging import main


__all__ = ["hello_world"]


def hello_world() -> str:
    return 'HelloWorld'


if __name__ == "__main__":
    main(
        messaging_config=dict(topic="hello-world-topic")
    )
```

Websocket Connection Parameters

The above example uses default connection values that will work with a local Apache Pulsar standalone Docker instance.

Consult the Pulsar documentation on how to setup custom values. In a nutshell:

- the websocket port can be set in the pulsar config files, e.g. standalone.conf
- the tenant and namespaces are setup through the pulsar admin interface (also see: https://pulsar.apache.org/docs/en/admin-api-tenants/#create)

In order to distinguish between different backend modules exposed through the same Pulsar instance you can create arbitrary tenant/namespace/topic combinations. Since topics are created on the fly, you might keep the tenant/namespace part static and register different backend modules via different topic names.

See: *Backend module configuration parameters*

Now we simply start the module with

```
python hello_world_backend.py listen
```

That's it, you exposed your first function through a DASF backend module. As a result you should see the following output:

```
connection attempt 1
connection to ws://localhost:8080/ws/v2/consumer/non-persistent/public/default/hello-
↪world-topic/backend-module-2022-01-14T17:26:55 established
waiting for incoming request
```

Now it's time to create the client side where we are going to call the exposed `hello_world` function.

The client side

Once we created our backend module we can use it to create the so called client stub. This is an auto-generated code stub that we can use in the client to call the functions exposed by the backend. For this we call:

```
python hello_world_backend.py generate >> hello_world_client_stub.py
```

Note: In case the backend module is already running/blocking the current terminal, you need to open a second one for this.

This will create a new file called `hello_world_client_stub.py` providing access to the `hello_world()` function exposed by the backend module. We now can use the client stub to call the function, e.g.

```
from hello_world_client_stub import hello_world

print(hello_world())
```

The execution of the above code should result in the following output:

```
connection to ws://localhost:8080/ws/v2/producer/non-persistent/public/default/hello-
↪world-topic established
connection to ws://localhost:8080/ws/v2/consumer/non-persistent/public/default/hello-
↪world-topic_UcribetS/python-backend-2022-01-14T18:11:06.768105 established
request successful
HelloWorld
```

That's it, you just called your first remote function exposed through a DASF backend module.

2.1.2 Remotely call a python procedure from a typescript context

The visualization of data and algorithmic results are an integral part of data science. Hence, DASF offers a web library called `dasf-web`, providing typescript bindings for the RPC messaging protocol, as well as some customizable visualization components. The general goal is to connect the results from computational methods exposed by a `backend module` directly to visualization components in web-based context. But for the sake of simplicity, let's pick up the `hello_world` backend module from the previous example and create a web application calling the backend module and displaying the returned value.

Setup a web-application project

First we need to setup a web-application project. The visualization components provided by the `dasf-web` package are using Vuetify/Vuejs. If you also want to use them in your application you have to setup a Vuetify/Vuejs based web application. The `dasf-web` package is published via the NPM registry (https://www.npmjs.com/package/dasf-web). You can add it to your projects dependencies with `npm i dasf-web --save`.

Application template

In case you are setting up a web-application from scratch and you want to use DASF, you can also checkout our nuxtjs application template by cloning the following repository: Repository Link

The client stub

After setting up the web application, we need to create the client stub (analog to the python only scenario above) that interfaces with the backend module. In contrast to the python client stub DASF does not support the generation of typescript client stubs yet. So you have to do this yourself, but don't worry, it's actually quite simple.

Listing 3: Typescript client stub for the HelloWorld backend module.

```
1  import PulsarConnection from 'dasf-web/lib/messaging/PulsarConnection'
2  import { PulsarModuleResponse, PulsarModuleRequest, PulsarModuleRequestReceipt } from
   ↪'dasf-web/lib/messaging//PulsarMessages'
3  import { DefaultPulsarUrlBuilder } from 'dasf-web/lib/messaging//PulsarUrlBuilder'
4
5  export default class HelloWorldClient {
6    private pulsarConnection: PulsarConnection;
7
8    public constructor () {
9      this.pulsarConnection = new PulsarConnection(new DefaultPulsarUrlBuilder('localhost',
   ↪ '8080', 'default', 'hello-world-topic'))
10   }
11
12   private createHelloWorldRequest (): PulsarModuleRequest {
13     const request = PulsarModuleRequest.createRequestMessage()
14     request.payload = btoa(JSON.stringify({ func_name: 'hello_world' }))
15
16     return request
17   }
18
19   public helloWorld (): Promise<string> {
20     return new Promise((resolve: (value: string) => void, reject: (reason: string) =>␣
   ↪void) => {
21       this.pulsarConnection.sendRequest(this.createHelloWorldRequest(),
22         (response: PulsarModuleResponse) => {
23           if (response.properties.status === 'success') {
24             resolve(JSON.parse(atob(response.payload)))
25           } else {
26             reject(atob(response.payload))
27           }
28         },
29         null,
30         (receipt: PulsarModuleRequestReceipt) => {
31           reject(receipt.errorMsg)
32         })
33     })
34   }
35 }
```

Let's break the shown client stub apart. First of all, we need to initialize a connection to the message broker. This is done in the constructor (line 9) via the `PulsarConnection` and `DefaultPulsarUrlBuilder` classes, where you also provide the connection parameters for your backend module.

Connection Establishment

The connection to the message broker is established as soon as the connection class is instantiated.

In order to send an actual request, we have to create an instance of `PulsarModuleRequest` which we do in the `createHelloWorldRequest` function (line 12). There you also set the function name and the function parameters (in this example there are none). The function request parameters are then stored as a b64 string in the requests payload field.

Finally we have to submit the created request and wait for the response. We do this by calling the `sendRequest` function (line 21), providing the request we created before and a callback function that is triggered once the response or an error arrives. In order to account for the async part we use the `Promise` proxy in this example, but in general you can deal with the received response in whatever way you prefer. In the show example the promise is resolved in case a `success` response is received. The payload of the response is then decoded from b64 and parsed into an Javascript object (the exact reverse of what we did to create the initial request) and finally passed to the resolve function of the promise. In case we received an error response or a negative request receipt, we `reject` the promise and pass the corresponding error message.

The vue component

Now that we created the client stub, we can use it in our web front-end context, e.g. like so:

Listing 4: Vue component 'visualizing' the data returned by the HelloWorld backend module

```
<template>
  <v-container
    style="margin-bottom: 35px; padding-top: 100px; padding-bottom: 20px"
  >
    <v-card :loading="response.length===0">
      <v-card-title>Hello World Example</v-card-title>
      <v-card-text>The hello world backend module responded with: <b>{{ response }}</b></
↪v-card-text>
    </v-card>
  </v-container>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator'
import HelloWorldClient from '~/lib/HelloWorldClient'

@Component({})
export default class HelloWorldVis extends Vue {
  protected response = '';

  protected created (): void {
    const hwClient: HelloWorldClient = new HelloWorldClient()

    hwClient.helloWorld().then((moduleResponse: string) => {
      this.response = moduleResponse
    }).catch((reason) => {
      this.response = reason
    })
  }
}
```

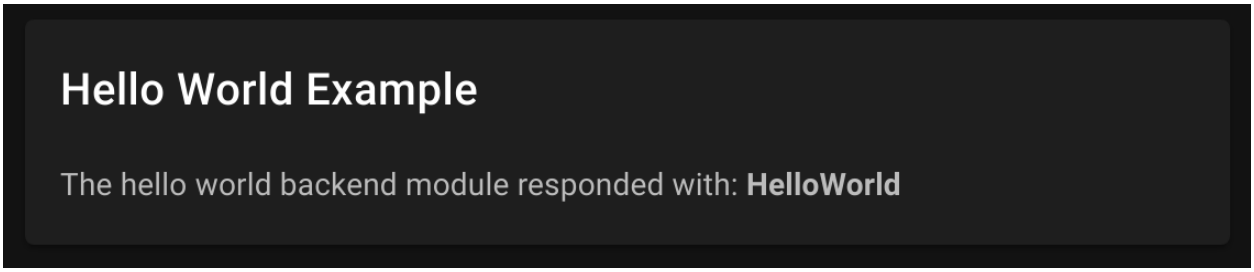
(continues on next page)

(continued from previous page)

```
</script>
```

Here we create a minimalistic vue component/page showing the response of the hello world request through the `HelloWorldClient` client stub we created above.

In case you used our application template you can execute it via `npm run dev` (don't forget to install the dependencies first via `npm install`). The logging output during compilation will show you the localhost url to access the web page, e.g. http://localhost:3000/dasf-app-template/ Depending on where you created your vue page, you will have to extend the url with the corresponding name. Assuming we put the `HelloWorldVis` under `pages/hello-world.vue`, you have to browse to `http://localhost:3000/dasf-app-template/hello-world` to open your hello world vue component. It should show a card stating the response text of the hello world backend module, e.g. like so:



That's it as far as the minimal hello world example goes. For more elaborate examples checkout the corresponding guides in the usage section.

You find the source code of the shown example in the following repository: https://codebase.helmholtz.cloud/dasf/demos/dasf-full-example

2.2 Usage

2.2.1 How to define and use custom data types

In the hello world example from the quick-start guide we only used the string data type. Here we will show you how to define more complex composite data types and use them either as function parameters or return values.

Therefore we extend the previous HelloWorld example.

Exposing classes

In order to support most use case scenarios you can not just expose individual functions via DASF, but entire classes. So before we dive into custom data types, let's convert our tiny `hello_world` function into a hello world class.

Listing 5: HelloWorld class

```python
from typing import List


class HelloWorld:
    def __init__(self, message: str):
        self.message = message

    def repeat_message(self, repeat: int) -> List[str]:
        return [self.message] * repeat
```

The given HelloWorld class defines a constructor that expects a string parameter called `message` and a single function called `repeat_message` that takes an integer and returns a list of strings. Now the idea is, that objects of this HelloWorld class are instantiated with a message string, that then will be used in its `repeat_message` along with the `repeat` parameter to generate the list of strings (repeating the message parameter `repeat` times) that are returned.

Now, to expose this class through a DASF backend module all we have to do is import and call the `main` function from the `demessaging` package and register the class via `__all__`. So our example above becomes:

Listing 6: HelloWorld class exposed via a HelloWorld backend module.

```python
from typing import List
from demessaging import main

__all__ = ["HelloWorld"]


class HelloWorld:
    def __init__(self, message: str):
        self.message = message

    def repeat_message(self, repeat: int) -> List[str]:
        return [self.message] * repeat


if __name__ == "__main__":
    main(
        messaging_config=dict(topic="hello-world-class-topic")
    )
```

Configuring exposed classes and functions through annotations

Sometimes you might not want to expose all functions of a class, like private/internal ones. This can be configured via the `@configure` annotation. Furthermore you might want to assert a certain value range for the method arguments or returns. This can also be configured via the `@configure` annotation.

Listing 7: Exposed HelloWorld class configured through @configure annotation.

```python
from typing import List
from demessaging import main, configure

__all__ = ["HelloWorld"]


@configure(methods=["repeat_message"])
class HelloWorld:
    def __init__(self, message: str):
        self.message = message

    @configure(field_params={"repeat": {"ge": 0}})
    def repeat_message(self, repeat: int) -> List[str]:
        return [self.message] * repeat

```

(continues on next page)

(continued from previous page)

```
    def unexposed_method(self) -> str:
        return self.message


if __name__ == "__main__":
    main(
        messaging_config=dict(topic="hello-world-class-topic")
    )
```

Class and function configuration parameters

For a comprehensive list of configuration parameters see: `demessaging.config.backend.ClassConfig`

Also refer to https://pydantic-docs.helpmanual.io/usage/schema/#field-customisation

Define custom data types

Let's extent our HelloWorld class even further by defining a custom data type/class that we are going to return in one of our exposed functions. In order to define the data type class we have to register it by using the `@registry.register_type` annotation. Let's register a `GreetResponse` as in the following example:

Listing 8: Custom data type class defined through @registry.register_type annotation.

```
from typing import List
import datetime
from pydantic import BaseModel
from demessaging import main, configure, registry

__all__ = ["HelloWorld"]

@registry.register_type
class GreetResponse(BaseModel):
    message: str
    greetings: List[str]
    greeting_time: datetime.datetime

@configure(methods=["repeat_message"])
class HelloWorld:
    def __init__(self, message: str):
        self.message = message

    @configure(field_params={"repeat": {"ge": 0}})
    def repeat_message(self, repeat: int) -> List[str]:
        return [self.message] * repeat

    def unexposed_method(self) -> str:
        return self.message


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    main(
        messaging_config=dict(topic="hello-world-topic")
    )
```

Note that the registered class has to inherit from the pydantic `BaseModel` class. Once registered we can use it as a function argument type or a return value type, like in the following `greet` function example.

Listing 9: Custom data type class defined through @registry.register_type annotation.

```
from typing import List
import datetime
from pydantic import BaseModel
from demessaging import main, configure, registry

__all__ = ["HelloWorld"]


@registry.register_type
class GreetResponse(BaseModel):
    message: str
    greetings: List[str]
    greeting_time: datetime.datetime


@configure(methods=["repeat_message", "greet"])
class HelloWorld:
    def __init__(self, message: str):
        self.message = message

    @configure(field_params={"repeat": {"ge": 0}})
    def repeat_message(self, repeat: int) -> List[str]:
        return [self.message] * repeat

    @configure(field_params={"repeat": {"ge": 0}})
    def greet(self, repeat: int, greet_message: str) -> GreetResponse:
        greetings: List[str] = [greet_message] * repeat
        return GreetResponse(
            message=self.message,
            greetings=greetings,
            greeting_time=datetime.datetime.now(),
        )

    def unexposed_method(self) -> str:
        return self.message


if __name__ == "__main__":
    main(
        messaging_config=dict(topic="hello-world-topic")
    )
```

2.2.2 Deploying Backend Modules

This is one of the strengths of DASF. You can deploy a backend module basically anywhere. The only requirement is, that the backend module is able to establish a connection to the used message broker instance.

Firewalls

This is especially useful for the deployment behind institutional firewalls. Since the backend module itself is establishing the connection and not the calling instance. So, as long as the firewall allows outward connections to the message broker instance the backend module can be used by all clients that are also able to connect to the message broker.

2.2.3 Connecting backend modules and web frontend components

One common aspect of data analytics is of course data visualization. DASF supports this by providing a variety of web frontend visualization components. Since we initially developed DASF in a geo-scientific context we start with some examples visualizing spatio-temporal data.

Flood Event Explorer

For a general impression of what is possible with DASF you can checkout the Digital Earth Flood Event Explorer. It was developed with DASF. In case you don't want to dive too deep into the individual workflows of the Flood Event Explorer, you might watch some videos showing the usage of the tool and providing some overview of what kind of visualizations are supported. The videos are linked in the applications landing page linked below or via the projects youtube channel.

- Homepage: https://digitalearth-hgf.de/results/workflows/flood-event-explorer/
- Source Code Repository: https://codebase.helmholtz.cloud/digital-earth/flood-event-explorer
- Deployed Application: http://rz-vm154.gfz-potsdam.de:8080/de-flood-event-explorer/

Citation Reference: Eggert, Daniel; Rabe, Daniela; Dransch, Doris; Lüdtke, Stefan; Nam, Christine; Nixdorf, Erik; Wichert, Viktoria; Abraham, Nicola; Schröter, Kai; Merz, Bruno (2022): The Digital Earth Flood Event Explorer: A showcase for data analysis and exploration with scientific workflows. GFZ Data Services. https://doi.org/10.5880/GFZ.1.4.2022.001

The DASF map component: Demap

The geo-visualization backbone of DASF is a web map component called `Demap` which is part of the `dasf-web` library. The `De` part originates from the Digital Earth project through which the development was funded. It is a Vue component and based on the popular Openlayers (https://openlayers.org/) library. In order to use the components provided by `dasf-web` you have to include it in your vue-based web application (also see: *Setup a web-application project*).

As any other custom Vue component import the `Demap` and add it to the html template of your own page/component. The map component supports the following customization properties and events:

Props

Name	Type	Default	Description
zoom	number	9	Initial zoom level of the map
projection	string	'EPSG:3857'	EPSG-Code of the projection used by the map
center	[number, number]	[13.740107, 51.055168]	Coordinates to center the map in geographical coordinates (EPSG: 4326)
legend-collapsed	boolean	false	If set, the legend will be initially collapsed
no-legend	boolean	false	If set, the map component will disable it's integrated legend / layer-switcher component
force-actions-append	boolean	false	If set, custom layer actions registered via `window['default_raster_layer_actions']` are added to new layers, even if the internal legend is deactivated
map-view	ol.View	null	External `view` instance, can be used to create linked views (e.g. Pan&Zoom) between multiple `Demap` instances.
show-osm	boolean	false	If set, adds the default OpenStreetMap tile layer as a basemap
enable-rotation	boolean	false	If set, enables the map rotation feature (mobile: pinch; desktop: alt+shift+drag)
disable-auto-zoom-to-layer	boolean	false	If set, the map does not automatically pans and zooms to newly added layers
start-layers	ol.Layer[] | Promise<ol.Lay	null	An array of layers (or a Promise resolving to an array of layers) that will be added to the map right from the start (or as soon as the Promise resolves).

Events

Name	Description
item-selected	called with an array of ol.Feature that have been selected in the map
load-custom-file	called with a File that was drag'n'drop onto the map, but no internal file handler was found supporting the files format. (integrated file handlers: .geojson are .nc) also see: add-layer component
set-roi	called with a ol.Geometry that the user has selected as a region of interest through the roi layer action.
unset-roi	called when the user unselects the previously selected roi geometry.
layer-added	called with a ol.Layer instance that was just added to the map component via the `addLayer` function.

Api

In case you created an object reference (e.g. via the @Ref annotation) for the `Demap` component you can utilize the following api to interact with it.

Name	Description
getLayers	Returns an array of all layers registered with the map
updateSelectInteraction	Accepting an optional ol.StyleLike style and an optional FilterFunction used for the build-in feature selection interaction. Also refer to OL Select API
updateSelectedFeature	Accepting an instance of ol.Feature and selecting it, if the internal selection interaction is enabled, e.g. via 'updateSelectInteraction'
updateSelectedFeatures	Accepting an array of ol.Feature and selecting them, if the internal selection interaction is enabled, e.g. via 'updateSelectInteraction'
addBaseLayer	Accepting an instance of ol.TileLayer to be added as an additional base layer
startEditLayer	Accepting an instance of ol.Layer and initializing the build-in editing feature in case the provided layer has a ol.VectorSource. This is mainly used by the edit layer LayerAction.
stopEditLayer	Finishes the build-in editing feature. This is mainly used by the edit layer LayerAction.
createVectorLayer	Returns a new instance of an ol.VectorLayer, accepting a title string, an optional ol.VectorSource an optional array of DemapLayerActions
Demap.isVecto	Returns `true` if the given layer is an instance of ol.VectorLayer
addLayer	Accepting an instance of ol.Layer and adding it to the map, adding new layers to the map through this method also creates color scales for all numeric properties of the underlying data
panAndZoomToExtent	Accepting an instance of ol.Layer or ol.Extent and pans and zooms the map so the entire (layers) extent is visible.
getMap	Returns the internally used ol.Map instance
getThematicLayerGroup	Returns the ol.LayerGroup containing all added non-base layers
dispose	Disposes the ol.Map context by setting its target element to `null`

Common spatio-temporal datastructure

Now the initial goal was to connect data produced/provided by backend modules with a web visualization component, e.g. `Demap`. For this to work we need a common datastructure for the data. In case of spatio-temporal data we rely on the NETCDF format. In the scope of a python backend module the popular xarray library can be used to load, create or manipulate the data. Finally we send it to the front-end application for visualization. In order to interpret the netcdf data the `dasf-web` library provides a `NetcdfRasterSource` class, which can be wrapped into a `TemporalImageLayer` and directly added to the map component.

Network Common Data Format (NetCDF)

The NetCDF is a pretty generic format, so its practically impossible to interpret all possible data arrangements. Therefore we need to enforce certain conventions. For this we assume the NetCDF data to follow the commonly used CF conventions.

Especially important is the global `crs` attribute, defining the coordinate reference system for the spatial data. It has to

be a string containing the EPSG code of the CRS, e.g. `'EPSG:4326'`.

Demap example

The following example creates a web front-end utilizing the `Demap` component.

Listing 10: Vue page showing a map via the Demap component

```
<template>
  <v-container
    style="margin-bottom: 35px; padding-top: 100px; padding-bottom: 20px"
  >
    <v-card :loading="dataPending">
      <v-card-title>Spatio-Temporal Data Example</v-card-title>
      <demap
        ref="demap"
        show-osm
        legend-collapsed
        :center="[13.064923, 52.379539]"
        :zoom="16"
        style="height: 50vh; min-height: 300px;"/>
    </v-card>
  </v-container>
</template>

<script lang="ts">
import { Component, Vue, Ref } from 'vue-property-decorator'
import Demap from 'dasf-web/lib/map/Demap.vue'
import SpatioTemporalClient from '~/lib/SpatioTemporalClient'
import TemporalImageLayer from 'dasf-web/lib/map/model/TemporalImageLayer'

@Component({
  components: { Demap }
})
export default class SpatioTemporal extends Vue {
  @Ref('demap')
  private demap!: Demap

  private dataPending = true

  protected created (): void {
    const backend: SpatioTemporalClient = new SpatioTemporalClient()

    backend.getData().then((dataLayer: TemporalImageLayer) => {
      this.demap.addLayer(dataLayer)
      this.dataPending = false
    }).catch((reason) => {
      console.warn(reason)
    }
    )
  }
}
```

(continues on next page)

(continued from previous page)

```
45 </script>
```

Note that we create an object reference to the map via the `@Ref` annotation in line 28. We will use this reference later to programatically interact with the map. Event without the backend part (lines 34-43) the map component is fully functional. Since the `show-osm` property is set the map shows an OpenStreetMap base layer. User can add new layers either via drag'n'drop or via the built-in legend menu.

Backend providing spatio-temporal data

Now that we created the front-end part, let's create a backend module providing the data that the front-end will visualize later on. We keep it simple and expose a `get_data()` function returning a `xarray.Dataset` that we load from a local netcdf file. In general the data could also be a result of some algorithm or loaded from an URL, depending on you use case.

Listing 11: spatio-temporal Dataset exposed via get_data function

```
1  from demessaging import main, configure
2  from demessaging.validators.xarray import validate_dataset
3  from demessaging.serializers.xarray import encode_xarray
4  import xarray as xr
5
6  __all__ = ["get_data"]
7
8
9  @configure(
10     return_serializer=encode_xarray,
11     return_validators=[validate_dataset],
12 )
13 def get_data() -> xr.Dataset:
14     ds = xr.load_dataset('sample_data.nc')
15     return ds
16
17
18 if __name__ == "__main__":
19     main(
20         messaging_config=dict(topic="spatio-temporal")
21     )
```

Note that `xr.Dataset` is not a type that is easily serializable. That is why you have to tell `DASF` whow to serialize the data. For `xarray.DataArray` and `xarray.Dataset` you find this information in the `demessaging.validators.xarray` and `demessaging.serializers.xarray` modules. You may use this workflow as a template to implement your own validators and serializers.

For most websocket based message brokers there is a maximum message size of 1MB configured. So, depending on the size of your data, we might exceed this size. DASF automatically fragments messages that are too big. In case you want to avoid fragmented messages you need to increase message brokers message size as well as DASF payload size.

WebSocket Message Size

WebSocket connections define a maximum message size (usually: 1MB). DASF will fragment all messages exceeding a given threshold called `max_payload_size`. The current default is 500kb. In order to work, the `max_payload_size` must be smaller than the maximum message size of the used message broker. e.g. Apache Pulsars message size can be configured in the `standalone.conf` via the `webSocketMaxTextFrameSize` parameter. The `max_payload_size`

for a backend module can be configured in the config passed to the `main()` function (see: *demessaging.config.messaging.WebsocketURLConfig()*).

The spatio-temporal client stub

In order connect the backend and the `Demap` visualization, we are going to the need the typescript client stub for the backend module. This looks almost identical as our hello world client stub (see *The client stub*), except that we don't return the payload string, but convert it to a `TemporalImageLayer` based on a `NetcdfRasterSource`.

Listing 12: Typescript client stub for the spatio-temporal backend module.

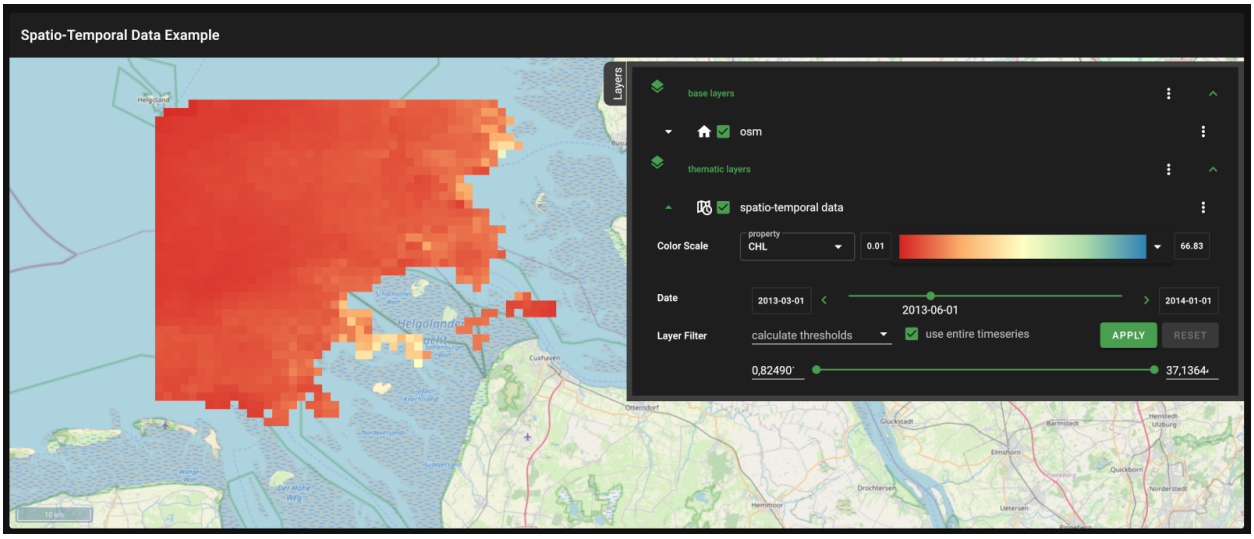
```
import PulsarConnection from 'dasf-web/lib/messaging/PulsarConnection'
import { PulsarModuleResponse, PulsarModuleRequest, PulsarModuleRequestReceipt } from
↪'dasf-web/lib/messaging//PulsarMessages'
import { DefaultPulsarUrlBuilder } from 'dasf-web/lib/messaging//PulsarUrlBuilder'
import TemporalImageLayer from 'dasf-web/lib/map/model/TemporalImageLayer'
import NetcdfRasterSource from 'dasf-web/lib/map/model/NetcdfRasterSource'
import b64a from 'base64-arraybuffer'

export default class SpatioTemporalClient {
  private pulsarConnection: PulsarConnection

  public constructor () {
    this.pulsarConnection = new PulsarConnection(new DefaultPulsarUrlBuilder('localhost',
↪ '8080', 'default', 'spatio-temporal'))
  }

  private createGetDataRequest (): PulsarModuleRequest {
    const request = PulsarModuleRequest.createRequestMessage()
    request.payload = btoa(JSON.stringify({ func_name: 'get_data' }))

    return request
  }

  public getData (): Promise<TemporalImageLayer> {
    return new Promise((resolve: (value: TemporalImageLayer) => void, reject: (reason:␣
↪string) => void) => {
      this.pulsarConnection.sendRequest(this.createGetDataRequest(),
        (response: PulsarModuleResponse) => {
          if (response.properties.status === 'success') {
            // parse the payload
            const netcdfB64 = JSON.parse(atob(response.payload))
            // convert the b64 into an arraybuffer and parse the buffer into an ol.
↪Source object
            NetcdfRasterSource.create({
              data: b64a.decode(netcdfB64)
            }).then((src: NetcdfRasterSource) => {
              // wrap ntcdf source into a layer
              const imageLayer = new TemporalImageLayer({
                title: 'spatio-temporal data',
                source: src
```

(continues on next page)

(continued from previous page)

```
37              })
38
39              resolve(imageLayer)
40            })
41          } else {
42            reject(atob(response.payload))
43          }
44        },
45        null,
46        (receipt: PulsarModuleRequestReceipt) => {
47          reject(receipt.errorMsg)
48        })
49    })
50  }
51}
```

Finally we can add the `TemporalImageLayer` returned by the client stub to our `Demap` via the `addLayer` method (line 37 of *Demap example*). The resulting visualization will look like the following:



While you can explore the spatial dimension with the maps pan&zoom functionality, the selection of the rendered data variable and the used color scale are possible through the legend component. In order to browse through the temporal dimension, the legend component provides an intuitive time-slider.

2.2.4 Backend module configuration parameters

You have three possibilities to configure the backend module. You can include them in the code via the *`main` function*, provide them as *command-line arguments* or *use environment variables*.

Note: Command line arguments take precedence over parameters to the `main` function which in turn takes precendece over environment variables!

Using the main function

When calling the *main()* function, you can pass provide various options to call for configuration (see the *API reference*).

Providing command line arguments

All parameters can also be provided via command line arguments.

```
usage: python my_module.py [-h] -t MESSAGING_CONFIG.TOPIC
                           [--header {"authorization": "Token ..."}]
                           [-m MODULE_NAME] [-d DOC] [--debug]
                           [-H PULSAR_CONFIG.HOST] [-p PULSAR_CONFIG.PORT]
                           [--persistent PULSAR_CONFIG.PERSISTENT]
                           [--tenant PULSAR_CONFIG.TENANT]
                           [--namespace PULSAR_CONFIG.NAMESPACE]
                           [--max-workers MESSAGING_CONFIG.MAX_WORKERS]
                           [--queue_size MESSAGING_CONFIG.QUEUE_SIZE]
                           [--max_payload_size MESSAGING_CONFIG.MAX_PAYLOAD_SIZE]
                           [--producer-keep-alive MESSAGING_CONFIG.PRODUCER_KEEP_ALIVE]
                           [--producer-connection-timeout MESSAGING_CONFIG.PRODUCER_
↪CONNECTION_TIMEOUT]
                           [--websocket-url WEBSOCKETURL_CONFIG.WEBSOCKET_URL]
                           [--producer-url WEBSOCKETURL_CONFIG.PRODUCER_URL]
                           [--consumer-url WEBSOCKETURL_CONFIG.CONSUMER_URL]
                           [--members member [member ...]]
                           [--log-config LOG_CONFIG.CONFIG_FILE]
                           [--log-level {10,20,30,40,50}]
                           [--log-file LOG_CONFIG.LOGFILE]
                           [--merge-log-config]
                           [--log-overrides LOG_CONFIG.CONFIG_OVERRIDES]
                           {test-connect,listen,schema,send-request,shell,generate}
                           ...
```

Named Arguments

-m, --module	Name of the backend module. Default: "__main__" Default: "__main__"
-d, --description	The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute. Default: ""
--debug	Run the backend module in debug mode (creates more verbose error messages). Default: False
--members	List of members for this module Default: [] Default: []

Connection options

General options for the backend module

-t, --topic The topic identifier under which to register at the pulsar. Default: "__NOTSET" This option is required!

Default: "__NOTSET"

--header Header parameters for the request Default: {}

Default: {}

--max-workers (optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.

--queue_size (optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.

--max_payload_size (optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb). Default: 512000

Default: 512000

--producer-keep-alive The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged. Default: 120

Default: 120

--producer-connection-timeout The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed. Default: 30

Default: 30

Pulsar connection options

Arguments for connecting to a pulsar. This is the default connection method unless you specify a websocket-url (see below).

-H, --host The remote host of the pulsar. Default: "localhost"

Default: "localhost"

-p, --port The port of the pulsar at the given `host`. Default: "8080"

Default: "8080"

--persistent Default: "non-persistent"

Default: "non-persistent"

--tenant Default: "public"

Default: "public"

--namespace Default: "default"

Default: "default"

Websocket URL group

Arguments for connecting to an arbitrary websocket service.

--websocket-url The fully qualified URL to the websocket.

Default: “”

--producer-url An alternative URL to use for producers. If None, the *websocket_url* will be used.

Default: “”

--consumer-url An alternative URL to use for consumers. If None, the *websocket_url* will be used.

Default: “”

Logging Configuration group

Arguments for configuring the logging within DASF.

--log-config Path to the logging configuration. Default: /home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yam

Default: /home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/log

--log-level Possible choices: 10, 20, 30, 40, 50

Level for the logger. Setting this will override any levels specified in the logging config file. The lower the value, the more verbose the logging. Typical levels are 10 (DEBUG), 20 (INFO), 30 (WARNING), 40 (ERROR) and 50 (CRITICAL).

--log-file A path to use for logging. If this is specified, we will add a RotatingFileHandler that loggs to the given path and add this handler to any logger in the logging config.

--merge-log-config If this is True, the specified logging configuration file will be merged with the default one at /home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yam

Default: False

--log-overrides Any valid YAML string, YAML file or JSON file that is merged into the logging configuration. This option can be used to quickly override some default logging functionality.

Commands

command Possible choices: test-connect, listen, schema, send-request, shell, generate

Sub-commands

test-connect

Connect the backend module to the pulsar message handler.

```
python my_module.py test-connect [-h]
```

listen

Connect the backend module to the pulsar message handler.

```
python my_module.py listen [-h]
```

schema

Print the schema for the backend module.

```
python my_module.py schema [-h] [-i INDENT]
```

Named Arguments

-i, --indent Indent the JSON dump.

send-request

Test a request via the pulsar messaging system.

```
python my_module.py send-request [-h] request
```

Positional Arguments

request A JSON-formatted file with the request.

shell

This command starts an IPython shell where you can access and work with the generated pydantic Model. This model class is available via the `Model` variable in the shell.

```
python my_module.py shell [-h]
```

generate

This command generates an API module that connects to the backend module via the pulsar and can be used on the client side. We use isort and black to format the generated python file.

```
python my_module.py generate [-h] [-l LINE_LENGTH] [--no-formatters]
                             [--no-isort] [--no-black] [--no-autoflake]
```

Named Arguments

-l, --line-length	The line-length for the output API. Default: 79
	Default: 79
--no-formatters	Do not use any formatters (isort, black or autoflake) for the generated code.
	Default: True
--no-isort	Do not use isort for formatting.
	Default: True
--no-black	Do not use black for formatting.
	Default: True
--no-autoflake	Do not use autoflake for formatting.
	Default: True

Using environment variables

Parameters for the messaging config (i.e. for *PulsarConfig* and *WebsocketURLConfig*) can also be provided via environment variables. Add the following prefix to the exported parameter: `DE_BACKEND_`, e.g. `DE_BACKEND_HOST` to set the `host` parameter.

The same works for the logging setup. The *LoggingConfig* class listens to environment variables prefixed with `DE_LOGGING_` (e.g. `DE_LOGGING_LEVEL`).

2.3 API Reference

2.3.1 demessaging package

Data Analytics Software Framework

python module wrapper for the data analytics software framework DASF

Classes:

BackendModule([root])	A base class for a backend module.

Functions:

configure([js, merge])	Configuration decorator for function or modules.
main([module_name])	Main function for starting a backend module from the command line.

class demessaging.**BackendModule**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootModel

A base class for a backend module.

Do not directly instantiate from this class, rather use the *create_model()* method.

Parameters

root (typing.Union[demessaging.backend.function.BackendFunction, demessaging.backend.class_.BackendClass]) – None

Attributes:

backend_config	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
pulsar	
root	

Methods:

compute()	Send this request to the backend module and compute the result.
create_model([module_name, members, config, ...])	Generate a module for a backend module.
generate([line_length, use_formatters, ...])	Generate the code for the frontend module.
get_api_info()	Get the API info on the module.
handle_message(request_msg)	
listen()	Connect to the message pulsar.
model_json_schema(*args, **kwargs)	Generates a JSON schema for a model class.
send_request(request)	Test a request to the backend.
shell()	Start a shell with the module defined.
test_connect()	Connect to the message pulsar.

backend_config: ClassVar[BackendModuleConfig]

compute() → BaseModel

Send this request to the backend module and compute the result.

This method updates the model inplace.

classmethod create_model(*module_name: str | None = None*, *members: List[Type[*BackendFunction*] | Type[*BackendClass*] | Callable | str | Type[object]] | None = None*, *config:* ModuleConfig *| None = None*, *class_name: str | None = None*, ***config_kws*) → Type[*BackendModule*]

Generate a module for a backend module.

Parameters

- **module_name** (*str*) – The name of the module to import. If none is given, the *members* must be specified
- **members** (*list of members*) – The list of members that shall be added to this module. It can be a list of
  - *BackendFunction* classes (generated with *create_model()*)
  - *BackendClass* classes (generated with *create_model()*)
  - functions (that will then be transformed using *create_model()*)
  - classes (that will then be transformed using *create_model()*)
  - strings, in which case they point to the member of the given *module_name*
- **config** (ModuleConfig, *optional*) – The configuration for the module. If this is not given, you must provide `config_kws` or define a `backend_config` variable within the module corresponding to *module_name*
- **class_name** (*str, optional*) – The name for the generated subclass of `pydantic.BaseModel`. If not given, the name of *Class* is used
- ****config_kws** – An alternative way to specify the configuration for the backend module.

Returns

The newly generated class that represents this module.

Return type

Subclass of *BackendFunction*

classmethod generate(*line_length: int = 79*, *use_formatters: bool = True*, *use_autoflake: bool = True*, *use_black: bool = True*, *use_isort: bool = True*) → str

Generate the code for the frontend module.

classmethod get_api_info() → *ModuleAPIModel*

Get the API info on the module.

classmethod handle_message(*request_msg*)

classmethod listen()

Connect to the message pulsar.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Union[BackendFunction, BackendClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod model_json_schema(**args*, ***kwargs*) → Dict[str, Any]

Generates a JSON schema for a model class.

Parameters

- **by_alias** – Whether to use attribute aliases or not.
- **ref_template** – The reference template.
- **schema_generator** – To override the logic used to generate the JSON schema, as a subclass of *GenerateJsonSchema* with your desired modifications
- **mode** – The mode in which to generate the schema.

Returns

The JSON schema for the given model class.

pulsar: ClassVar[*MessageConsumer*]

root: *BackendFunction* | *BackendClass*

classmethod send_request(*request:* BackendModule | *IO* | *Dict[str, Any]*) → BaseModel

Test a request to the backend.

Parameters

request (*dict or file-like object*) – A request to the backend module.

classmethod shell()

Start a shell with the module defined.

classmethod test_connect()

Connect to the message pulsar.

demessaging.**configure**(*js: str | None = None*, *merge: bool = True*, ***kwargs*) → Callable[[T], T]

Configuration decorator for function or modules.

Use this function as a decorator for classes or functions in the backend module like so:

```
>>> @configure(field_params={"a": {"gt": 0}}, returns={"gt": 0})
... def sqrt(a: float) -> float:
...     import math
...
...     return math.sqrt(a)
```

The available parameters for this function vary depending on what you are decorating. If you are decorating a class, your parameters must be valid for the `ClassConfig`. If you are decorating a function, your parameters must be valid for a `FunctionConfig`.

Parameters

- **js** (*Optional[str]*) – A JSON-formatted string that can be used to setup the config.
- **merge** (*bool*) – If True (default), then the configuration will be merged with the existing configuration for the function (if existing)
- ****kwargs** – Any keyword argument that can be used to setup the config.

Notes

If you are specifying any `kwargs`, your first argument (*js*) should be `None`.

demessaging.**main**(*module_name: str = '__main__'*, **args*, ***config_kws*) → Type[*BackendModule*]

Main function for starting a backend module from the command line.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding `__doc__` attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the module for the generated API.
- **messaging_config** (*Union[*demessaging.config.messaging.PulsarConfig, demessaging.config.messaging.WebsocketURLConfig*]*) – Configuration on how to connect to the message broker.
- **log_config** (demessaging.config.logging.LoggingConfig) – Configuration for the logging.
- **debug** (*bool*) – Run the backend module in debug mode (creates more verbose error messages).
- **members** (*List[Union[str, Callable, Type[object], Any]]*) – List of members for this module
- **imports** (*str*) – Imports that should be added to the generate API module.
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function

Subpackages

demessaging.backend package

Core module for generating a de-messaging backend module.

This module defines the base classes to serve a general python module as a backend module in the DASF.

The most important members are:

main([module_name])	Main function for starting a backend module from the command line.
BackendModule([root])	A base class for a backend module.
BackendFunction()	A base class for a function model.
BackendClass()	A basis for class models

Functions:

main([module_name])	Main function for starting a backend module from the command line.

demessaging.backend.**main**(*module_name: str = '__main__'*, **args*, ***config_kws*) → Type[*BackendModule*]

Main function for starting a backend module from the command line.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the module for the generated API.
- **messaging_config** (*Union[*demessaging.config.messaging.PulsarConfig, demessaging.config.messaging.WebsocketURLConfig*]*) – Configuration on how to connect to the message broker.
- **log_config** (demessaging.config.logging.LoggingConfig) – Configuration for the logging.
- **debug** (*bool*) – Run the backend module in debug mode (creates more verbose error messages).
- **members** (*List[Union[str, Callable, Type[object], Any]]*) – List of members for this module
- **imports** (*str*) – Imports that should be added to the generate API module.
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function

Submodules

demessaging.backend.class_ module

Transform a python class into a corresponding pydantic model.

The *BackendClass* model in this module generates subclasses based upon a python class (similarly as the *BackendFunction* does it for functions).

Classes:

BackendClass()	A basis for class models
BackendClassConfig(*[, doc, registry, ...])	Configuration class for a backend module class.
ClassAPIModel(*, name, rpc_schema, methods)	A class in the API suitable for RPC via DASF

class demessaging.backend.class_.**BackendClass**

Bases: BaseModel

A basis for class models

Do not directly instantiate from this class, rather use the *create_model()* method.

Attributes:

backend_config	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
return_model	The return model of the member function.

Methods:

create_model(Class[, config, methods, ...])	Generate a pydantic model from a class.
get_api_info()	Get the API info on the function.
get_constructor_fields(Class, config, class_name)	
model_json_schema(*args, **kwargs)	Generates a JSON schema for a model class.

backend_config: ClassVar[*BackendClassConfig*]

classmethod create_model(*Class*, *config:* ClassConfig | *None* = *None*, *methods:* *List[Type[*BackendFunction*]* | *Callable* | *str]* | *None* = *None*, *class_name: str* | *None* = *None*, ***kwargs: Any*) → Type[*BackendClass*]

Generate a pydantic model from a class.

Parameters

- **func** (*type*) – A class
- **config** (*ClassConfig, optional*) – The configuration to use. If given, this overrides the `__pulsar_config__` of the given *Class*
- **methods** (*list of methods, optional*) – A list of methods or model classes generated with `FunctionModel()`. This overrides the methods in *config* or the `__pulsar_config__` attribute of *Class*
- **class_name** (*str, optional*) – The name for the generated subclass of `pydantic.BaseModel`. If not given, the name of *Class* is used
- ****kwargs** – Any other parameter for the `pydantic.create_model()` function

Returns

The newly generated model that represents this class.

Return type

Subclass of *BackendClass*

function: *BackendFunction*

classmethod get_api_info() → *ClassAPIModel*

Get the API info on the function.

classmethod get_constructor_fields(*Class*, *config:* ClassConfig, *class_name: str* | *None*) → Tuple[Dict[str, Any], *BackendClassConfig*]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod model_json_schema(**args*, ***kwargs*) → Dict[str, Any]

Generates a JSON schema for a model class.

Parameters

- **by_alias** – Whether to use attribute aliases or not.
- **ref_template** – The reference template.
- **schema_generator** – To override the logic used to generate the JSON schema, as a subclass of *GenerateJsonSchema* with your desired modifications
- **mode** – The mode in which to generate the schema.

Returns

The JSON schema for the given model class.

property return_model: Type[BaseModel]

The return model of the member function.

class demessaging.backend.class_.**BackendClassConfig**(*, *doc: str = ''*, *registry:* ApiRegistry = *None*, *template:* Template = *Template(name='class_.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_bui suffix='.jinja2', context={})*, *name: str = ''*, *init_doc: str = ''*, *signature: Signature | None = None*, *methods: List[str] = None*, *validators: Dict[str, Any] = None*, *serializers: Dict[str, Any] = None*, *field_params: Dict[str, Dict[str, Any]] = None*, *annotations: Dict[str, Any] = None*, *reporter_args: Dict[str, BaseReport] = None*, *json_schema_extra: Dict[str, Any] = None*, *models: Dict[str, Type[*BackendFunction*]] = None*, *Class: Type[object]*, *class_name: str*)

Bases: *ClassConfig*

Configuration class for a backend module class.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the class for the generated API.

- **name** (*str*) – The name of the function. If empty, this will be taken from the classes `__name__` attribute.
- **init_doc** (*str*) – The documentation of the function. If empty, this will be taken from the classes `__init__` method.
- **signature** (*Optional[inspect.Signature]*) – The calling signature for the function. If empty, this will be taken from the function itself.
- **methods** (*List[str]*) – methods to use within the backend modules
- **validators** (*Dict[str, Any]*) – custom validators for the constructor parameters
- **serializers** (*Dict[str, Any]*) – A mapping from function argument to serializer that is of instance `pydantic.functional_serializers.PlainSerializer` or `pydantic.functional_serializers.WrapSerializer`.
- **field_params** (*Dict[str, Dict[str, Any]]*) – custom Field overrides for the constructor parameters. See `pydantic.Fields.Field()`
- **annotations** (*Dict[str, Any]*) – custom annotations for constructor parameters
- **reporter_args** (*Dict[str, deprogressapi.base.BaseReport]*) – Arguments that use the dasf-progress-api
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function
- **models** (*Dict[str, Type[demessaging.backend.function.BackendFunction]]*) – Mapping of method name to the function model for the methods of this class
- **Class** (*Type[object]*) – The class that corresponds to this config.
- **class_name** (*str*) – Name of the model class

Attributes:

Class	
class_name	
method_configs	Get a list of the method configs.
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
models	

Methods:

update_from_cls()	Update the config from the corresponding function.

Class: Type[object]

annotations: Dict[str, Any]

class_name: str

doc: str

field_params: Dict[str, Dict[str, Any]]

init_doc: str

json_schema_extra: Dict[str, Any]

property method_configs: List[*BackendFunctionConfig*]

Get a list of the method configs.

methods: List[str]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'Class': FieldInfo(annotation=Type[object], required=True, description='The class that corresponds to this config.'), 'annotations': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='custom annotations for constructor parameters'), 'class_name': FieldInfo(annotation=str, required=True, description='Name of the model class'), 'doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the object. If empty, this will be taken from the corresponding ``__doc__`` attribute.'), 'field_params': FieldInfo(annotation=Dict[str, Dict[str, Any]], required=False, default_factory=dict, description='custom Field overrides for the constructor parameters. See :func:`pydantic.Fields.Field`'), 'init_doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the function. If empty, this will be taken from the classes ``__init__`` method.'), 'json_schema_extra': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='Any extra parameter for the JSON schema export for the function'), 'methods': FieldInfo(annotation=List[str], required=False, default_factory=list, description='methods to use within the backend modules'), 'models': FieldInfo(annotation=Dict[str, Type[BackendFunction]], required=False, default_factory=dict, description='Mapping of method name to the function model for the methods of this class'), 'name': FieldInfo(annotation=str, required=False, default='', description='The name of the function. If empty, this will be taken from the classes ``__name__`` attribute.'), 'registry': FieldInfo(annotation=ApiRegistry, required=False, default_factory=_get_registry, description='Utilities for imports and encoders.'), 'reporter_args': FieldInfo(annotation=Dict[str, BaseReport], required=False, default_factory=dict, description='Arguments that use the dasf-progress-api'), 'serializers': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='A mapping from function argument to serializer that is of instance :class:`pydantic.functional_serializers.PlainSerializer` or :class:`pydantic.functional_serializers.WrapSerializer`.'), 'signature': FieldInfo(annotation=Union[Signature, NoneType], required=False, default=None, description='The calling signature for the function. If empty, this will be taken from the function itself.'), 'template': FieldInfo(annotation=Template, required=False, default=Template(name='class_.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'), suffix='.jinja2', context={}), description='The :class:`demessaging.template.Template` that is used to render the class for the generated API.'), 'validators': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='custom validators for the constructor parameters')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

models: Dict[str, Type[*BackendFunction*]]

name: str

registry: *ApiRegistry*

reporter_args: Dict[str, BaseReport]

serializers: Dict[str, Any]

signature: inspect.Signature | None

template: *Template*

update_from_cls() → None

Update the config from the corresponding function.

validators: Dict[str, Any]

class demessaging.backend.class_.**ClassAPIModel**(*, *name: str*, *rpc_schema: Dict[str, Any]*, *methods: List[*FunctionAPIModel*]*)

Bases: BaseModel

A class in the API suitable for RPC via DASF

Attributes:

methods	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
name	
rpc_schema	

methods: List[*FunctionAPIModel*]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'methods': FieldInfo(annotation=List[FunctionAPIModel], required=True, description='The list of methods that this class provides.'), 'name': FieldInfo(annotation=str, required=True, description='The name of the class that is used as identifier in the RPC.'), 'rpc_schema': FieldInfo(annotation=Dict[str, Any], required=True, description='The JSON Schema for the constructor of the class.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

rpc_schema: JsonSchemaValue

demessaging.backend.function module

Transform a python function into a corresponding pydantic model.

The *BackendFunction* model in this module generates subclasses based upon a python class (similarly as the *BackendClass* does it for classes).

Classes:

BackendFunction()	A base class for a function model.
BackendFunctionConfig(*[, doc, registry, ...])	Configuration class for a backend module function.
FunctionAPIModel(*, name, rpc_schema, ...)	A class in the API suitable for RPC via DASF
ReturnModel([root])	

Functions:

get_return_model(docstring, config)	Generate field for the return property.

class demessaging.backend.function.**BackendFunction**

Bases: BaseModel

A base class for a function model.

Don't use this model, rather use *create_model()* method to generate new models.

Attributes:

backend_config	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
return_model	

Methods:

create_model(func[, config, class_name])	Create a new pydantic Model from a function.
get_api_info()	Get the API info on the function.
model_json_schema(*args, **kwargs)	Generates a JSON schema for a model class.

backend_config: ClassVar[*BackendFunctionConfig*]

classmethod create_model(*func: Callable*, *config:* FunctionConfig | *None* = *None*, *class_name=None*, ***kwargs*) → Type[*BackendFunction*]

Create a new pydantic Model from a function.

Parameters

- **func** (*callable*) – A function or method
- **config** (FunctionConfig, *optional*) – The configuration to use. If given, this overrides the __pulsar_config__ of the given *func*
- **class_name** (*str*, *optional*) – The name for the generated subclass of pydantic.BaseModel. If not given, the name of *func* is used
- ****kwargs** – Any other parameter for the pydantic.create_model() function

Returns
The newly generated class that represents this function.

Return type
Subclass of *BackendFunction*

func_name: str

classmethod get_api_info() → *FunctionAPIModel*
Get the API info on the function.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'validate_assignment': True}
Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}
Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod model_json_schema(args*, ***kwargs*) → Dict[str, Any]**
Generates a JSON schema for a model class.

Parameters

- **by_alias** – Whether to use attribute aliases or not.
- **ref_template** – The reference template.
- **schema_generator** – To override the logic used to generate the JSON schema, as a subclass of *GenerateJsonSchema* with your desired modifications
- **mode** – The mode in which to generate the schema.

Returns
The JSON schema for the given model class.

return_model: ClassVar[Type[BaseModel]]

class demessaging.backend.function.**BackendFunctionConfig**(*, *doc: str = ''*, *registry: ApiRegistry = None*, *template: Template = Template(name='function.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/us suffix='.jinja2', context={})*, *name: str = ''*, *signature: Signature | None = None*, *validators: Dict[str, List[ImportString | Callable]] = None*, *serializers: Dict[str, ImportString | Callable] = None*, *return_validators: List[ImportString | Callable] | None = None*, *return_serializer: ImportString | Callable | None = None*, *field_params: Dict[str, Dict[str, Any]] = None*, *returns: Dict[str, Any] = None*, *return_annotation: Any | None = None*, *annotations: Dict[str, Any] = None*, *reporter_args: Dict[str, BaseReport] = None*, *json_schema_extra: Dict[str, Any] = None*, *function: Any*, *class_name: str*)

Bases: *FunctionConfig*

Configuration class for a backend module function.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the function for the generated API.
- **name** (*str*) – The name of the function. If empty, this will be taken from the functions __name__ attribute.
- **signature** (*Optional[inspect.Signature]*) – The calling signature for the function. If empty, this will be taken from the function itself.
- **validators** (*Dict[str, List[Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')]]]]*) – Custom validators for function arguments. This parameter is a mapping from function argument name to a list of callables that can be used as validator.
- **serializers** (*Dict[str, Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')]]]*) – A mapping from function argument to serializing function that is then used for the pydantic.functional_serializers.PlainSerializer.
- **return_validators** (*Optional[List[Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')]]]]*) – Validators for the return value. This parameter is a list of callables that can be used as validator for the return value.

- **return_serializer** (*Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')], NoneType]*) – A function that is used to serialize the return value.
- **field_params** (*Dict[str, Dict[str, Any]]*) – custom Field overrides for the constructor parameters. See `pydantic.Fields.Field()`
- **returns** (*Dict[str, Any]*) – custom returns overrides.
- **return_annotation** (*Optional[Any]*) – The annotation for the return value.
- **annotations** (*Dict[str, Any]*) – custom annotations for function parameters
- **reporter_args** (*Dict[str, deprogressapi.base.BaseReport]*) – Arguments that use the dasf-progress-api
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function
- **function** (*Any*) – The function to call.
- **class_name** (*str*) – Name of the model class

Attributes:

class_name	
function	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

Methods:

update_from_function()	Update the config from the corresponding function.

annotations: Dict[str, Any]

class_name: str

doc: str

field_params: Dict[str, Dict[str, Any]]

function: Any

json_schema_extra: Dict[str, Any]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'annotations': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='custom annotations for function parameters'), 'class_name': FieldInfo(annotation=str, required=True, description='Name of the model class'), 'doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the object. If empty, this will be taken from the corresponding ``__doc__`` attribute.'), 'field_params': FieldInfo(annotation=Dict[str, Dict[str, Any]], required=False, default_factory=dict, description='custom Field overrides for the constructor parameters. See :func:`pydantic.Fields.Field`'), 'function': FieldInfo(annotation=Any, required=True, description='The function to call.'), 'json_schema_extra': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='Any extra parameter for the JSON schema export for the function'), 'name': FieldInfo(annotation=str, required=False, default='', description='The name of the function. If empty, this will be taken from the functions ``__name__`` attribute.'), 'registry': FieldInfo(annotation=ApiRegistry, required=False, default_factory=_get_registry, description='Utilities for imports and encoders.'), 'reporter_args': FieldInfo(annotation=Dict[str, BaseReport], required=False, default_factory=dict, description='Arguments that use the dasf-progress-api'), 'return_annotation': FieldInfo(annotation=Union[Any, NoneType], required=False, default=None, description='The annotation for the return value.'), 'return_serializer': FieldInfo(annotation=Union[ImportString, Annotated[Callable, PlainSerializer], NoneType], required=False, default=None, description='A function that is used to serialize the return value.'), 'return_validators': FieldInfo(annotation=Union[List[Union[ImportString, Annotated[Callable, PlainSerializer]]], NoneType], required=False, default=None, description='Validators for the return value. This parameter is a list of callables that can be used as validator for the return value.'), 'returns': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='custom returns overrides.'), 'serializers': FieldInfo(annotation=Dict[str, Union[ImportString, Annotated[Callable, PlainSerializer]]], required=False, default_factory=dict, description='A mapping from function argument to serializing function that is then used for the :class:`pydantic.functional_serializers.PlainSerializer`.'), 'signature': FieldInfo(annotation=Union[Signature, NoneType], required=False, default=None, description='The calling signature for the function. If empty, this will be taken from the function itself.'), 'template': FieldInfo(annotation=Template, required=False, default=Template(name='function.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'), suffix='.jinja2', context={}), description='The :class:`demessaging.template.Template` that is used to render the function for the generated API.'), 'validators': FieldInfo(annotation=Dict[str, List[Union[ImportString, Annotated[Callable, PlainSerializer]]]], required=False, default_factory=dict, description='Custom validators for function arguments. This parameter is a mapping from function argument name to a list of callables that can be used as validator.')}

Metadata about the fields defined on the model, mapping of field names to [*Field-*

Info][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

registry: *ApiRegistry*

reporter_args: Dict[str, BaseReport]

return_annotation: Any | None

return_serializer: Optional[Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]

return_validators: Optional[List[Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]]

returns: Dict[str, Any]

serializers: Dict[str, Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]

signature: inspect.Signature | None

template: *Template*

update_from_function() → None

Update the config from the corresponding function.

validators: Dict[str, List[Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]]

class demessaging.backend.function.**FunctionAPIModel**(*, *name: str*, *rpc_schema: Dict[str, Any]*, *return_schema: Dict[str, Any]*)

Bases: BaseModel

A class in the API suitable for RPC via DASF

Attributes:

model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
name	
return_schema	
rpc_schema	

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'name': FieldInfo(annotation=str, required=True, description='The name of the function that is used as identifier in the RPC.'), 'return_schema': FieldInfo(annotation=Dict[str, Any], required=True, description='The JSON Schema for the return value.'), 'rpc_schema': FieldInfo(annotation=Dict[str, Any], required=True, description='The JSON Schema for the function.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

return_schema: JsonSchemaValue

rpc_schema: JsonSchemaValue

class demessaging.backend.function.**ReturnModel**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootModel

Attributes:

model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=~RootModelRootType, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: RootModelRootType

demessaging.backend.function.**get_return_model**(*docstring: Docstring*, *config:* BackendFunctionConfig) → Type[BaseModel]

Generate field for the return property.

Parameters
docstring (*docstring_parser.Docstring*) – The parser that analyzed the docstring

Returns
The pydantic field

Return type
Any

demessaging.backend.module module

Backend module to transform a python module into a pydantic model.

This module defines the main model in the demessaging framework. It takes a list of members, or a module, and creates a new Model that can be used to generate code, connect to the pulsar, and more. See *BackendModule* for details.

Classes:

BackendModule([root])	A base class for a backend module.
BackendModuleConfig(*[, doc, registry, ...])	Configuration class for a backend module.
ModuleAPIModel(*, classes, functions, rpc_schema)	An model that represants the API of a backend module.

class demessaging.backend.module.**BackendModule**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootModel

A base class for a backend module.

Do not directly instantiate from this class, rather use the *create_model()* method.

Parameters
root (typing.Union[demessaging.backend.function.BackendFunction, demessaging.backend.class_.BackendClass]) – None

Attributes:

backend_config	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
pulsar	
root	

Methods:

compute()	Send this request to the backend module and compute the result.
create_model([module_name, members, config, ...])	Generate a module for a backend module.
generate([line_length, use_formatters, ...])	Generate the code for the frontend module.
get_api_info()	Get the API info on the module.
handle_message(request_msg)	
listen()	Connect to the message pulsar.
model_json_schema(*args, **kwargs)	Generates a JSON schema for a model class.
send_request(request)	Test a request to the backend.
shell()	Start a shell with the module defined.
test_connect()	Connect to the message pulsar.

backend_config: ClassVar[*BackendModuleConfig*]

compute() → BaseModel

Send this request to the backend module and compute the result.

This method updates the model inplace.

classmethod create_model(*module_name: str | None = None*, *members: List[Type[*BackendFunction*] | Type[*BackendClass*] | Callable | str | Type[object]] | None = None*, *config:* ModuleConfig *| None = None*, *class_name: str | None = None*, ***config_kws*) → Type[*BackendModule*]

Generate a module for a backend module.

Parameters

- **module_name** (*str*) – The name of the module to import. If none is given, the *members* must be specified
- **members** (*list of members*) – The list of members that shall be added to this module. It can be a list of
  - *BackendFunction* classes (generated with *create_model()*)
  - *BackendClass* classes (generated with *create_model()*)
  - functions (that will then be transformed using *create_model()*)
  - classes (that will then be transformed using *create_model()*)
  - strings, in which case they point to the member of the given *module_name*
- **config** (ModuleConfig, *optional*) – The configuration for the module. If this is not given, you must provide `config_kws` or define a `backend_config` variable within the module corresponding to *module_name*
- **class_name** (*str, optional*) – The name for the generated subclass of `pydantic.BaseModel`. If not given, the name of *Class* is used
- ****config_kws** – An alternative way to specify the configuration for the backend module.

Returns

The newly generated class that represents this module.

Return type
Subclass of *BackendFunction*

classmethod generate(*line_length: int = 79*, *use_formatters: bool = True*, *use_autoflake: bool = True*, *use_black: bool = True*, *use_isort: bool = True*) → str

Generate the code for the frontend module.

classmethod get_api_info() → *ModuleAPIModel*

Get the API info on the module.

classmethod handle_message(*request_msg*)

classmethod listen()

Connect to the message pulsar.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Union[BackendFunction, BackendClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod model_json_schema(**args*, ***kwargs*) → Dict[str, Any]

Generates a JSON schema for a model class.

Parameters

- **by_alias** – Whether to use attribute aliases or not.
- **ref_template** – The reference template.
- **schema_generator** – To override the logic used to generate the JSON schema, as a subclass of *GenerateJsonSchema* with your desired modifications
- **mode** – The mode in which to generate the schema.

Returns
The JSON schema for the given model class.

pulsar: ClassVar[*MessageConsumer*]

root: *BackendFunction* | *BackendClass*

classmethod send_request(*request:* BackendModule | *IO* | *Dict[str, Any]*) → BaseModel

Test a request to the backend.

Parameters
request (*dict or file-like object*) – A request to the backend module.

classmethod shell()

Start a shell with the module defined.

classmethod test_connect()

Connect to the message pulsar.

class demessaging.backend.module.**BackendModuleConfig**(*, *doc: str = ''*, *registry: ApiRegistry = None*, *template: Template = Template(name='module.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_bu suffix='.jinja2', context={})*, *messaging_config: PulsarConfig | WebsocketURLConfig*, *log_config: LoggingConfig = LoggingCon- fig(config_file=PosixPath('/home/docs/checkouts/readthedocs.org/ level=None, logfile=None, config_overrides=None, merge_config=False)*, *debug: bool = False*, *members: List[str | Callable | Type[object] | Any] = None*, *imports: str = ''*, *json_schema_extra: Dict[str, Any] = None*, *models: List[Any] = None*, *module: Any*, *class_name: str*)

Bases: *ModuleConfig*

Configuration class for a backend module.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the module for the generated API.
- **messaging_config** (*Union[*demessaging.config.messaging.PulsarConfig, demessaging.config.messaging.WebsocketURLConfig*]*) – Configuration on how to connect to the message broker.
- **log_config** (demessaging.config.logging.LoggingConfig) – Configuration for the logging.
- **debug** (*bool*) – Run the backend module in debug mode (creates more verbose error messages).
- **members** (*List[Union[str, Callable, Type[object], Any]]*) – List of members for this module
- **imports** (*str*) – Imports that should be added to the generate API module.
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function
- **models** (*List[Any]*) – a list of function or class models for the members of the backend module
- **module** (*Any*) – The imported backend module (or none, if there is none)
- **class_name** (*str*) – Name of the model class

Attributes:

class_name	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
models	
module	

class_name: `str`

debug: `bool`

doc: `str`

imports: `str`

json_schema_extra: `Dict[str, Any]`

log_config: *LoggingConfig*

members: `List[str | Callable | Type[object] | Any]`

messaging_config: *PulsarConfig* | *WebsocketURLConfig*

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'class_name': FieldInfo(annotation=str, required=True, description='Name of the model class'), 'debug': FieldInfo(annotation=bool, required=False, default=False, description='Run the backend module in debug mode (creates more verbose error messages).'), 'doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the object. If empty, this will be taken from the corresponding ``__doc__`` attribute.'), 'imports': FieldInfo(annotation=str, required=False, default='', description='Imports that should be added to the generate API module.'), 'json_schema_extra': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='Any extra parameter for the JSON schema export for the function'), 'log_config': FieldInfo(annotation=LoggingConfig, required=False, default=LoggingConfig(config_file=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yaml'), level=None, logfile=None, config_overrides=None, merge_config=False), description='Configuration for the logging.'), 'members': FieldInfo(annotation=List[Union[str, Callable, Type[object], Any]], required=False, default_factory=list, description='List of members for this module'), 'messaging_config': FieldInfo(annotation=Union[PulsarConfig, WebsocketURLConfig], required=True, description='Configuration on how to connect to the message broker.'), 'models': FieldInfo(annotation=List[Any], required=False, default_factory=list, description='a list of function or class models for the members of the backend module'), 'module': FieldInfo(annotation=Any, required=True, description='The imported backend module (or none, if there is none)'), 'registry': FieldInfo(annotation=ApiRegistry, required=False, default_factory=_get_registry, description='Utilities for imports and encoders.'), 'template': FieldInfo(annotation=Template, required=False, default=Template(name='module.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'), suffix='.jinja2', context={}), description='The :class:`demessaging.template.Template` that is used to render the module for the generated API.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

models: List[Any]

module: Any

registry: *ApiRegistry*

template: *Template*

class demessaging.backend.module.**ModuleAPIModel**(*, *classes: List[*ClassAPIModel*]*, *functions: List[*FunctionAPIModel*]*, *rpc_schema: Dict[str, Any]*)

Bases: BaseModel

An model that represants the API of a backend module.

Attributes:

classes	
functions	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
rpc_schema	

classes: List[*ClassAPIModel*]

functions: List[*FunctionAPIModel*]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'classes': FieldInfo(annotation=List[ClassAPIModel], required=True, description='The RPC-enabled classes that this module contains.'), 'functions': FieldInfo(annotation=List[FunctionAPIModel], required=True, description='The RPC-enabled functions that this module contains.'), 'rpc_schema': FieldInfo(annotation=Dict[str, Any], required=True, description='The aggregated JSON schema for an RPC call to this module.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

rpc_schema: JsonSchemaValue

demessaging.backend.utils module

Utitlity functions for the backend framework.

Classes:

AsyncIoThread(func, args, kwargs)	A thread that runs an async function.
ImportMixin()	Mixin class for `isort.identify.Import`.

Functions:

camelize(w)	Camelize a word by making the first letter upper case.
get_desc(docstring)	Get the description of an object.
get_fields(name, sig, docstring, config)	Get the model fields from a function signature.
get_kws(sig, obj)	Get keywords from a signature and a base model.
get_module_imports(mod)	Get all the imports from a module
run_async(func, *args, **kwargs)	Run an async function and wait for the result.
snake_to_camel(*words)	Transform a list of words into its camelized version.

class demessaging.backend.utils.**AsyncIoThread**(*func: Callable*, *args: Tuple*, *kwargs: Dict*)

Bases: Thread

A thread that runs an async function.

See: func: *run_async* for the implementation.

Methods:

run()	Method representing the thread's activity.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class demessaging.backend.utils.**ImportMixin**

Bases: object

Mixin class for `isort.identify.Import`.

A response to https://github.com/PyCQA/isort/issues/1641.

Methods:

statement()	

statement() → str

demessaging.backend.utils.**camelize**(*w: str*) → str

Camelize a word by making the first letter upper case.

demessaging.backend.utils.**get_desc**(*docstring: docstring_parser.Docstring*) → str

Get the description of an object.

Parameters
docstring (*docstring_parser.Docstring*) – The parser that analyzed the docstring.

Returns
The description of the callable.

Return type
str

demessaging.backend.utils.**get_fields**(*name: str*, *sig: inspect.Signature*, *docstring: docstring_parser.Docstring*, *config:* FunctionConfig | ClassConfig) → Dict[str, Tuple[Any, Any]]

Get the model fields from a function signature.

Parameters

- **name** (*str*) – The name of the function or class
- **sig** (*inspect.Signature*) – The signature of the callable
- **docstring** (*docstring_parser.Docstring*) – The parser that analyzed the docstring
- **config** (FunctionConfig *or* ClassConfig) – The configuration for the callable

Returns

A mapping from field name to field parameters to be used in `pydantic.create_model()`.

Return type

dict

demessaging.backend.utils.**get_kws**(*sig*, *obj*) → Dict[str, Any]

Get keywords from a signature and a base model.

demessaging.backend.utils.**get_module_imports**(*mod: Any*) → str

Get all the imports from a module

Parameters

mod (*module*) – The module to use

demessaging.backend.utils.**run_async**(*func: Callable*, **args: Any*, ***kwargs: Any*) → Any

Run an async function and wait for the result.

This function works within standard python scripts, and during a running jupyter session.

demessaging.backend.utils.**snake_to_camel**(**words: str*) → str

Transform a list of words into its camelized version.

demessaging.config package

Configuration classes for DASF.

Submodules

demessaging.config.api module

DASF API utilties for to access the configuration.

Functions:

configure([js, merge])	Configuration decorator for function or modules.

demessaging.config.api.**configure**(*js: str | None = None*, *merge: bool = True*, ***kwargs*) → Callable[[T], T]

Configuration decorator for function or modules.

Use this function as a decorator for classes or functions in the backend module like so:

```
>>> @configure(field_params={"a": {"gt": 0}}, returns={"gt": 0})
... def sqrt(a: float) -> float:
...     import math
...
...     return math.sqrt(a)
```

The available parameters for this function vary depending on what you are decorating. If you are decorating a class, your parameters must be valid for the `ClassConfig`. If you are decorating a function, your parameters must be valid for a `FunctionConfig`.

Parameters

- **js** (*Optional[str]*) – A JSON-formatted string that can be used to setup the config.
- **merge** (*bool*) – If True (default), then the configuration will be merged with the existing configuration for the function (if existing)
- ****kwargs** – Any keyword argument that can be used to setup the config.

Notes

If you are specifying any `kwargs`, your first argument (*js*) should be `None`.

demessaging.config.api.**registry = ApiRegistry(imports={}, objects=[])**

registry for the stuff that should be available in the generated client stub

demessaging.config.backend module

Configuration classes for the de-messaging backend module.

Classes:

BaseConfig(*[, doc, registry, template])	Configuration base class for functions, modules and classes.
ClassConfig(*[, doc, registry, template, ...])	Configuration class for a backend module class.
FunctionConfig(*[, doc, registry, template, ...])	Configuration class for a backend module function.
ModuleConfig(*[, doc, registry, template, ...])	Configuration class for a backend module.

class demessaging.config.backend.**BaseConfig**(*, *doc: str = ''*, *registry:* ApiRegistry *= None*, *template:* Template *= Template(name='empty', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/ch suffix='.jinja2', context={})*)

Bases: `BaseModel`

Configuration base class for functions, modules and classes.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding `__doc__` attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render this object for the generated API.

Attributes:

doc	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
registry	
template	

Methods:

render(**context)	Generate the code to call this function in the frontend.

doc: `str`

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the object. If empty, this will be taken from the corresponding ``__doc__`` attribute.'), 'registry': FieldInfo(annotation=ApiRegistry, required=False, default_factory=_get_registry, description='Utilities for imports and encoders.'), 'template': FieldInfo(annotation=Template, required=False, default=Template(name='empty', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'), suffix='.jinja2', context={}), description='The :class:`demessaging.template.Template` that is used to render this object for the generated API.')}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

registry: *ApiRegistry*

render(***context*) → str

Generate the code to call this function in the frontend.

template: *Template*

class demessaging.config.backend.**ClassConfig**(*, *doc: str = ''*, *registry: ApiRegistry = None*, *template: Template = Template(name='class_.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/c suffix='.jinja2', context={})*, *name: str = ''*, *init_doc: str = ''*, *signature: Signature | None = None*, *methods: List[str] = None*, *validators: Dict[str, Any] = None*, *serializers: Dict[str, Any] = None*, *field_params: Dict[str, Dict[str, Any]] = None*, *annotations: Dict[str, Any] = None*, *reporter_args: Dict[str, BaseReport] = None*, *json_schema_extra: Dict[str, Any] = None*)

Bases: *BaseConfig*

Configuration class for a backend module class.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the class for the generated API.
- **name** (*str*) – The name of the function. If empty, this will be taken from the classes __name__ attribute.
- **init_doc** (*str*) – The documentation of the function. If empty, this will be taken from the classes __init__ method.
- **signature** (*Optional[inspect.Signature]*) – The calling signature for the function. If empty, this will be taken from the function itself.
- **methods** (*List[str]*) – methods to use within the backend modules
- **validators** (*Dict[str, Any]*) – custom validators for the constructor parameters
- **serializers** (*Dict[str, Any]*) – A mapping from function argument to serializer that is of instance pydantic.functional_serializers.PlainSerializer or pydantic.functional_serializers.WrapSerializer.
- **field_params** (*Dict[str, Dict[str, Any]]*) – custom Field overrides for the constructor parameters. See pydantic.Fields.Field()
- **annotations** (*Dict[str, Any]*) – custom annotations for constructor parameters
- **reporter_args** (*Dict[str, deprogressapi.base.BaseReport]*) – Arguments that use the dasf-progress-api
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function

Attributes:

annotations	
field_params	
init_doc	
json_schema_extra	
methods	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
name	
reporter_args	
serializers	
signature	
template	
validators	

annotations: Dict[str, Any]

doc: str

field_params: Dict[str, Dict[str, Any]]

init_doc: str

json_schema_extra: Dict[str, Any]

methods: List[str]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'annotations':
FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict,
description='custom annotations for constructor parameters'), 'doc':
FieldInfo(annotation=str, required=False, default='', description='The documentation
of the object. If empty, this will be taken from the corresponding ``__doc__``
attribute.'), 'field_params':  FieldInfo(annotation=Dict[str, Dict[str, Any]],
required=False, default_factory=dict, description='custom Field overrides for the
constructor parameters. See :func:`pydantic.Fields.Field`'), 'init_doc':
FieldInfo(annotation=str, required=False, default='', description='The documentation
of the function. If empty, this will be taken from the classes ``__init__``
method.'), 'json_schema_extra':  FieldInfo(annotation=Dict[str, Any],
required=False, default_factory=dict, description='Any extra parameter for the JSON
schema export for the function'), 'methods':  FieldInfo(annotation=List[str],
required=False, default_factory=list, description='methods to use within the backend
modules'), 'name':  FieldInfo(annotation=str, required=False, default='',
description='The name of the function. If empty, this will be taken from the classes
``__name__`` attribute.'), 'registry':  FieldInfo(annotation=ApiRegistry,
required=False, default_factory=_get_registry, description='Utilities for imports
and encoders.'), 'reporter_args':  FieldInfo(annotation=Dict[str, BaseReport],
required=False, default_factory=dict, description='Arguments that use the
dasf-progress-api'), 'serializers':  FieldInfo(annotation=Dict[str, Any],
required=False, default_factory=dict, description='A mapping from function argument
to serializer that is of instance
:class:`pydantic.functional_serializers.PlainSerializer` or
:class:`pydantic.functional_serializers.WrapSerializer`.'), 'signature':
FieldInfo(annotation=Union[Signature, NoneType], required=False, default=None,
description='The calling signature for the function. If empty, this will be taken
from the function itself.'), 'template':  FieldInfo(annotation=Template,
required=False, default=Template(name='class_.py', folder=PosixPath('/home/docs/
checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'),
suffix='.jinja2', context={}), description='The
:class:`demessaging.template.Template` that is used to render the class for the
generated API.'), 'validators':  FieldInfo(annotation=Dict[str, Any],
required=False, default_factory=dict, description='custom validators for the
constructor parameters')}
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

```
name:  str
```

```
registry:  ApiRegistry
```

```
reporter_args:  Dict[str, BaseReport]
```

```
serializers:  Dict[str, Any]
```

```
signature:  inspect.Signature | None
```

```
template:  Template
```

```
validators:  Dict[str, Any]
```

class demessaging.config.backend.**FunctionConfig**(*, *doc: str = ''*, *registry: ApiRegistry = None*, *template: Template = Template(name='function.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/d suffix='.jinja2', context={})*, *name: str = ''*, *signature: Signature | None = None*, *validators: Dict[str, List[ImportString | Callable]] = None*, *serializers: Dict[str, ImportString | Callable] = None*, *return_validators: List[ImportString | Callable] | None = None*, *return_serializer: ImportString | Callable | None = None*, *field_params: Dict[str, Dict[str, Any]] = None*, *returns: Dict[str, Any] = None*, *return_annotation: Any | None = None*, *annotations: Dict[str, Any] = None*, *reporter_args: Dict[str, BaseReport] = None*, *json_schema_extra: Dict[str, Any] = None*)

Bases: *BaseConfig*

Configuration class for a backend module function.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the function for the generated API.
- **name** (*str*) – The name of the function. If empty, this will be taken from the functions __name__ attribute.
- **signature** (*Optional[inspect.Signature]*) – The calling signature for the function. If empty, this will be taken from the function itself.
- **validators** (*Dict[str, List[Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')]]]]*) – Custom validators for function arguments. This parameter is a mapping from function argument name to a list of callables that can be used as validator.
- **serializers** (*Dict[str, Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')]]]*) – A mapping from function argument to serializing function that is then used for the pydantic.functional_serializers.PlainSerializer.
- **return_validators** (*Optional[List[Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')]]]]*) – Validators for the return value. This parameter is a list of callables that can be used as validator for the return value.
- **return_serializer** (*Union[pydantic.types.ImportString, Annotated[Callable, PlainSerializer(func=<function object_to_string at 0x7f24517675b0>, return_type=PydanticUndefined, when_used='always')], NoneType]*) – A function that is used to serialize the return value.

- **field_params** (*Dict[str, Dict[str, Any]]*) – custom Field overrides for the constructor parameters. See `pydantic.Fields.Field()`
- **returns** (*Dict[str, Any]*) – custom returns overrides.
- **return_annotation** (*Optional[Any]*) – The annotation for the return value.
- **annotations** (*Dict[str, Any]*) – custom annotations for function parameters
- **reporter_args** (*Dict[str, deprogressapi.base.BaseReport]*) – Arguments that use the dasf-progress-api
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function

Attributes:

annotations	
field_params	
json_schema_extra	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
name	
reporter_args	
return_annotation	
return_serializer	
return_validators	
returns	
serializers	
signature	
template	
validators	

annotations: `Dict[str, Any]`

doc: `str`

field_params: Dict[str, Dict[str, Any]]

json_schema_extra: Dict[str, Any]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'annotations': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='custom annotations for function parameters'), 'doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the object. If empty, this will be taken from the corresponding ``__doc__`` attribute.'), 'field_params': FieldInfo(annotation=Dict[str, Dict[str, Any]], required=False, default_factory=dict, description='custom Field overrides for the constructor parameters. See :func:`pydantic.Fields.Field`'), 'json_schema_extra': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='Any extra parameter for the JSON schema export for the function'), 'name': FieldInfo(annotation=str, required=False, default='', description='The name of the function. If empty, this will be taken from the functions ``__name__`` attribute.'), 'registry': FieldInfo(annotation=ApiRegistry, required=False, default_factory=_get_registry, description='Utilities for imports and encoders.'), 'reporter_args': FieldInfo(annotation=Dict[str, BaseReport], required=False, default_factory=dict, description='Arguments that use the dasf-progress-api'), 'return_annotation': FieldInfo(annotation=Union[Any, NoneType], required=False, default=None, description='The annotation for the return value.'), 'return_serializer': FieldInfo(annotation=Union[ImportString, Annotated[Callable, PlainSerializer], NoneType], required=False, default=None, description='A function that is used to serialize the return value.'), 'return_validators': FieldInfo(annotation=Union[List[Union[ImportString, Annotated[Callable, PlainSerializer]]], NoneType], required=False, default=None, description='Validators for the return value. This parameter is a list of callables that can be used as validator for the return value.'), 'returns': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='custom returns overrides.'), 'serializers': FieldInfo(annotation=Dict[str, Union[ImportString, Annotated[Callable, PlainSerializer]]], required=False, default_factory=dict, description='A mapping from function argument to serializing function that is then used for the :class:`pydantic.functional_serializers.PlainSerializer`.'), 'signature': FieldInfo(annotation=Union[Signature, NoneType], required=False, default=None, description='The calling signature for the function. If empty, this will be taken from the function itself.'), 'template': FieldInfo(annotation=Template, required=False, default=Template(name='function.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'), suffix='.jinja2', context={}), description='The :class:`demessaging.template.Template` that is used to render the function for the generated API.'), 'validators': FieldInfo(annotation=Dict[str, List[Union[ImportString, Annotated[Callable, PlainSerializer]]]], required=False, default_factory=dict, description='Custom validators for function arguments. This parameter is a mapping from function argument name to a list of callables that can be used as validator.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

registry: *ApiRegistry*

reporter_args: Dict[str, BaseReport]

return_annotation: Any | None

return_serializer: Optional[Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]

return_validators: Optional[List[Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]]

returns: Dict[str, Any]

serializers: Dict[str, Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]

signature: inspect.Signature | None

template: *Template*

validators: Dict[str, List[Union[ImportString, Annotated[Callable, PlainSerializer(object_to_string)]]]]

class demessaging.config.backend.**ModuleConfig**(*, *doc: str = ''*, *registry:* ApiRegistry *= None*, *template:* Template *= Template(name='module.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf*
suffix='.jinja2', context={}), *messaging_config:* PulsarConfig | WebsocketURLConfig, *log_config:* LoggingConfig *= LoggingConfig(config_file=PosixPath('/home/docs/checkouts/readthedocs.org/user_bui*
level=None, logfile=None, config_overrides=None, merge_config=False), *debug: bool = False*, *members: List[str | Callable | Type[object] | Any] = None*, *imports: str = ''*, *json_schema_extra: Dict[str, Any] = None*)

Bases: *BaseConfig*

Configuration class for a backend module.

Parameters

- **doc** (*str*) – The documentation of the object. If empty, this will be taken from the corresponding __doc__ attribute.
- **registry** (demessaging.config.registry.ApiRegistry) – Utilities for imports and encoders.
- **template** (demessaging.template.Template) – The *demessaging.template.Template* that is used to render the module for the generated API.
- **messaging_config** (*Union[*demessaging.config.messaging.PulsarConfig, demessaging.config.messaging.WebsocketURLConfig*]*) – Configuration on how to connect to the message broker.

- **log_config** (demessaging.config.logging.LoggingConfig) – Configuration for the logging.
- **debug** (*bool*) – Run the backend module in debug mode (creates more verbose error messages).
- **members** (*List[Union[str, Callable, Type[object], Any]]*) – List of members for this module
- **imports** (*str*) – Imports that should be added to the generate API module.
- **json_schema_extra** (*Dict[str, Any]*) – Any extra parameter for the JSON schema export for the function

Attributes:

debug	
imports	
json_schema_extra	
log_config	
members	
messaging_config	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
pulsar_config	DEPRECATED! Get the messaging configuration.
template	

debug: bool

doc: str

imports: str

json_schema_extra: Dict[str, Any]

log_config: *LoggingConfig*

members: List[str | Callable | Type[object] | Any]

messaging_config: *PulsarConfig* | *WebsocketURLConfig*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'debug': FieldInfo(annotation=bool, required=False, default=False, description='Run the backend module in debug mode (creates more verbose error messages).'), 'doc': FieldInfo(annotation=str, required=False, default='', description='The documentation of the object. If empty, this will be taken from the corresponding ``__doc__`` attribute.'), 'imports': FieldInfo(annotation=str, required=False, default='', description='Imports that should be added to the generate API module.'), 'json_schema_extra': FieldInfo(annotation=Dict[str, Any], required=False, default_factory=dict, description='Any extra parameter for the JSON schema export for the function'), 'log_config': FieldInfo(annotation=LoggingConfig, required=False, default=LoggingConfig(config_file=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yaml'), level=None, logfile=None, config_overrides=None, merge_config=False), description='Configuration for the logging.'), 'members': FieldInfo(annotation=List[Union[str, Callable, Type[object], Any]], required=False, default_factory=list, description='List of members for this module'), 'messaging_config': FieldInfo(annotation=Union[PulsarConfig, WebsocketURLConfig], required=True, description='Configuration on how to connect to the message broker.'), 'registry': FieldInfo(annotation=ApiRegistry, required=False, default_factory=_get_registry, description='Utilities for imports and encoders.'), 'template': FieldInfo(annotation=Template, required=False, default=Template(name='module.py', folder=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates'), suffix='.jinja2', context={}), description='The :class:`demessaging.template.Template` that is used to render the module for the generated API.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

property pulsar_config: *PulsarConfig* | *WebsocketURLConfig*

DEPRECATED! Get the messaging configuration.

Please use the `messaging_config` attribute of this class.

registry: *ApiRegistry*

template: *Template*

demessaging.config.logging module

Configuration module for logging in DASF.

This module defines the configuration class for logging within the DASF Framework.

Classes:

LoggingConfig([_case_sensitive, ...])	Configuration for logging.

class demessaging.config.logging.**LoggingConfig**(*_case_sensitive: bool | None = None*, *_env_prefix: str | None = None*, *_env_file: DotenvType | None = PosixPath('.')*, *_env_file_encoding: str | None = None*, *_env_ignore_empty: bool | None = None*, *_env_nested_delimiter: str | None = None*, *_env_parse_none_str: str | None = None*, *_secrets_dir: str | Path | None = None*, ***, *config_file: Path = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/l*
level: int | None = None, *logfile: Path | None = None*, *config_overrides: Dict | None = None*, *merge_config: bool = False*)

Bases: `BaseSettings`

Configuration for logging.

Parameters

- **config_file** (*pathlib.Path*) – Path to the logging configuration.
- **level** (*Optional[Annotated[int, Gt(gt=0)]]*) – Level for the logger. Setting this will override any levels specified in the logging config file. The lower the value, the more verbose the logging. Typical levels are 10 (DEBUG), 20 (INFO), 30 (WARNING), 40 (ERROR) and 50 (CRITICAL).
- **logfile** (*Optional[pathlib.Path]*) – A path to use for logging. If this is specified, we will add a RotatingFileHandler that loggs to the given path and add this handler to any logger in the logging config.
- **config_overrides** (*Optional[Dict]*) – A dictionary to override the configuration specified in the logging configuration file.
- **merge_config** (*bool*) – If this is True, the specified logging configuration file will be merged with the default one at /home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yaml

Attributes:

config_dict	Configuration dictionary for the logging.
config_file	
config_overrides	
level	
logfile	
merge_config	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

Methods:

configure_logging()	Configure the loggers based upon the given config.

property config_dict: Dict

Configuration dictionary for the logging.

config_file: Path

config_overrides: Dict | None

configure_logging()

Configure the loggers based upon the given config.

level: int | None

logfile: Path | None

merge_config: bool

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'env_file': None, 'env_file_encoding': None, 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_none_str': None, 'env_prefix': 'de_logging_', 'extra': 'forbid', 'json_file': None, 'json_file_encoding': None, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'config_file': FieldInfo(annotation=Path, required=False, default=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yaml'), description='Path to the logging configuration.', metadata=[PathType(path_type='file')]), 'config_overrides': FieldInfo(annotation=Union[Dict, NoneType], required=False, default=None, description='A dictionary to override the configuration specified in the logging configuration file.'), 'level': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='Level for the logger. Setting this will override any levels specified in the logging config file. The lower the value, the more verbose the logging. Typical levels are 10 (DEBUG), 20 (INFO), 30 (WARNING), 40 (ERROR) and 50 (CRITICAL).'), 'logfile': FieldInfo(annotation=Union[Path, NoneType], required=False, default=None, description='A path to use for logging. If this is specified, we will add a RotatingFileHandler that loggs to the given path and add this handler to any logger in the logging config.'), 'merge_config': FieldInfo(annotation=bool, required=False, default=False, description='If this is True, the specified logging configuration file will be merged with the default one at /home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/config/logging.yaml')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

demessaging.config.messaging module

Messaging configuration classes for DASF.

Classes:

BaseMessagingConfig([_case_sensitive, ...])	Base class for messaging configs.
PulsarConfig([_case_sensitive, _env_prefix, ...])	A configuration class to connect to the pulsar messaging framework.
WebsocketURLConfig([_case_sensitive, ...])	A configuration for a websocket.

class demessaging.config.messaging.**BaseMessagingConfig**(*_case_sensitive: bool | None = None*, *_env_prefix: str | None = None*, *_env_file: DotenvType | None = PosixPath('.')*, *_env_file_encoding: str | None = None*, *_env_ignore_empty: bool | None = None*, *_env_nested_delimiter: str | None = None*, *_env_parse_none_str: str | None = None*, *_secrets_dir: str | Path | None = None*, ***, *topic: str*, *header: Dict[str, Any] | Dict[str, Any] = None*, *max_workers: int | None = None*, *queue_size: int | None = None*, *max_payload_size: int = 512000*, *producer_keep_alive: int = 120*, *producer_connection_timeout: int = 30*)

Bases: `BaseSettings`

Base class for messaging configs.

Parameters

- **topic** (*str*) – The topic identifier under which to register at the pulsar.
- **header** (*Union[Annotated[Dict[str, Any], Json], Dict[str, Any]]*) – Header parameters for the request
- **max_workers** (*Optional[Annotated[int, Gt(gt=0)]]*) – (optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.
- **queue_size** (*Optional[Annotated[int, Gt(gt=0)]]*) – (optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.
- **max_payload_size** (*int*) – (optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb).
- **producer_keep_alive** (*int*) – The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged.
- **producer_connection_timeout** (*int*) – The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed.

Methods:

get_topic_url(topic[, subscription])	Build the URL to connect to a websocket.
model_post_init(__context)	This function is meant to behave like a BaseModel method to initialise private attributes.
validate_queue_size()	Check that the queue_size is smaller than the max_workers.

Attributes:

header	
max_payload_size	
max_workers	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
producer	The connected producer for the messaging config
producer_connection_timeout	
producer_keep_alive	
queue_size	
topic	

get_topic_url(*topic: str*, *subscription: str | None = None*) → str

Build the URL to connect to a websocket.

header: Json[Dict[str, Any]] | Dict[str, Any]

max_payload_size: int

max_workers: PositiveInt | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'env_file': None, 'env_file_encoding': None, 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_none_str': None, 'env_prefix': 'de_backend_', 'extra': 'forbid', 'json_file': None, 'json_file_encoding': None, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'header': FieldInfo(annotation=Union[Annotated[Dict[str, Any], Json], Dict[str, Any]], required=False, default_factory=dict, description='Header parameters for the request'), 'max_payload_size': FieldInfo(annotation=int, required=False, default=512000, description="(optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb)."), 'max_workers': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='(optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.'), 'producer_connection_timeout': FieldInfo(annotation=int, required=False, default=30, description='The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed.'), 'producer_keep_alive': FieldInfo(annotation=int, required=False, default=120, description='The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged.'), 'queue_size': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='(optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.'), 'topic': FieldInfo(annotation=str, required=True, description='The topic identifier under which to register at the pulsar.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

property producer: *MessageProducer*

The connected producer for the messaging config

producer_connection_timeout: int

producer_keep_alive: int

queue_size: PositiveInt | None

topic: str

validate_queue_size()

Check that the queue_size is smaller than the max_workers.

class demessaging.config.messaging.**PulsarConfig**(*_case_sensitive: bool | None = None*, *_env_prefix: str | None = None*, *_env_file: DotenvType | None = PosixPath('.')*, *_env_file_encoding: str | None = None*, *_env_ignore_empty: bool | None = None*, *_env_nested_delimiter: str | None = None*, *_env_parse_none_str: str | None = None*, *_secrets_dir: str | Path | None = None*, ***, *topic: str*, *header: Dict[str, Any] | Dict[str, Any] = None*, *max_workers: int | None = None*, *queue_size: int | None = None*, *max_payload_size: int = 512000*, *producer_keep_alive: int = 120*, *producer_connection_timeout: int = 30*, *host: str = 'localhost'*, *port: str = '8080'*, *persistent: str = 'non-persistent'*, *tenant: str = 'public'*, *namespace: str = 'default'*)

Bases: *BaseMessagingConfig*

A configuration class to connect to the pulsar messaging framework.

Parameters

- **topic** (*str*) – The topic identifier under which to register at the pulsar.
- **header** (*Union[Annotated[Dict[str, Any], Json], Dict[str, Any]]*) – Header parameters for the request
- **max_workers** (*Optional[Annotated[int, Gt(gt=0)]]*) – (optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.
- **queue_size** (*Optional[Annotated[int, Gt(gt=0)]]*) – (optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.
- **max_payload_size** (*int*) – (optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb).
- **producer_keep_alive** (*int*) – The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged.
- **producer_connection_timeout** (*int*) – The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed.
- **host** (*str*) – The remote host of the pulsar.
- **port** (*str*) – The port of the pulsar at the given *host*.
- **persistent** (*str*) – None
- **tenant** (*str*) – None
- **namespace** (*str*) – None

Methods:

get_topic_url(topic[, subscription])	Build the URL to connect to a websocket.
model_post_init(_ModelMetaclass__context)	We need to both initialize private attributes and call the user-defined model_post_init method.

Attributes:

host	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
namespace	
persistent	
port	
tenant	

get_topic_url(*topic: str*, *subscription: str | None = None*) → str

Build the URL to connect to a websocket.

header: Json[Dict[str, Any]] | Dict[str, Any]

host: str

max_payload_size: int

max_workers: PositiveInt | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'env_file': None, 'env_file_encoding': None, 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_none_str': None, 'env_prefix': 'de_backend_', 'extra': 'forbid', 'json_file': None, 'json_file_encoding': None, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'header': FieldInfo(annotation=Union[Annotated[Dict[str, Any], Json], Dict[str, Any]], required=False, default_factory=dict, description='Header parameters for the request'), 'host': FieldInfo(annotation=str, required=False, default='localhost', description='The remote host of the pulsar.'), 'max_payload_size': FieldInfo(annotation=int, required=False, default=512000, description="(optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb)."), 'max_workers': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='(optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.'), 'namespace': FieldInfo(annotation=str, required=False, default='default'), 'persistent': FieldInfo(annotation=str, required=False, default='non-persistent'), 'port': FieldInfo(annotation=str, required=False, default='8080', description='The port of the pulsar at the given :attr:`host`.'), 'producer_connection_timeout': FieldInfo(annotation=int, required=False, default=30, description='The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed.'), 'producer_keep_alive': FieldInfo(annotation=int, required=False, default=120, description='The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged.'), 'queue_size': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='(optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.'), 'tenant': FieldInfo(annotation=str, required=False, default='public'), 'topic': FieldInfo(annotation=str, required=True, description='The topic identifier under which to register at the pulsar.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

namespace: str

persistent: str

port: str

producer_connection_timeout: int

producer_keep_alive: int

queue_size: PositiveInt | None

tenant: str

topic: str

class demessaging.config.messaging.**WebsocketURLConfig**(*_case_sensitive: bool | None = None*, *_env_prefix: str | None = None*, *_env_file: DotenvType | None = PosixPath('.')*, *_env_file_encoding: str | None = None*, *_env_ignore_empty: bool | None = None*, *_env_nested_delimiter: str | None = None*, *_env_parse_none_str: str | None = None*, *_secrets_dir: str | Path | None = None*, ***, *topic: str*, *header: Dict[str, Any] | Dict[str, Any] = None*, *max_workers: int | None = None*, *queue_size: int | None = None*, *max_payload_size: int = 512000*, *producer_keep_alive: int = 120*, *producer_connection_timeout: int = 30*, *websocket_url: str = ''*, *producer_url: str | None = None*, *consumer_url: str | None = None*)

Bases: *BaseMessagingConfig*

A configuration for a websocket.

Parameters

- **topic** (*str*) – The topic identifier under which to register at the pulsar.
- **header** (*Union[Annotated[Dict[str, Any], Json], Dict[str, Any]]*) – Header parameters for the request
- **max_workers** (*Optional[Annotated[int, Gt(gt=0)]]*) – (optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.
- **queue_size** (*Optional[Annotated[int, Gt(gt=0)]]*) – (optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.
- **max_payload_size** (*int*) – (optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb).
- **producer_keep_alive** (*int*) – The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged.
- **producer_connection_timeout** (*int*) – The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed.
- **websocket_url** (*str*) – The fully qualified URL to the websocket.
- **producer_url** (*Optional[str]*) – An alternative URL to use for producers. If None, the *websocket_url* will be used.
- **consumer_url** (*Optional[str]*) – An alternative URL to use for consumers. If None, the *websocket_url* will be used.

Attributes:

consumer_url	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
producer_url	
websocket_url	

Methods:

get_topic_url(topic[, subscription])	Build the URL to connect to a websocket.
model_post_init(_ModelMetaclass__context)	We need to both initialize private attributes and call the user-defined model_post_init method.

consumer_url: `str | None`

get_topic_url(*topic: str*, *subscription: str | None = None*) → str

Build the URL to connect to a websocket.

header: `Json[Dict[str, Any]] | Dict[str, Any]`

max_payload_size: `int`

max_workers: `PositiveInt | None`

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'env_file': None, 'env_file_encoding': None, 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_none_str': None, 'env_prefix': 'de_backend_', 'extra': 'forbid', 'json_file': None, 'json_file_encoding': None, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'consumer_url': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, description='An alternative URL to use for consumers. If None, the `websocket_url` will be used.'), 'header': FieldInfo(annotation=Union[Annotated[Dict[str, Any], Json], Dict[str, Any]], required=False, default_factory=dict, description='Header parameters for the request'), 'max_payload_size': FieldInfo(annotation=int, required=False, default=512000, description="(optional) maximum payload size, must be smaller than pulsars 'webSocketMaxTextFrameSize', which is configured e.g.via 'pulsar/conf/standalone.conf'.default: 512000 (500kb)."), 'max_workers': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='(optional) number of concurrent workers for handling requests, default: number of processors on the machine, multiplied by 5.'), 'producer_connection_timeout': FieldInfo(annotation=int, required=False, default=30, description='The amount of time that we grant producers to establish a connection to the message broker in order to send a response. If a connection cannot be established in this time, the response will not be sent and the connection will be closed.'), 'producer_keep_alive': FieldInfo(annotation=int, required=False, default=120, description='The amount of time that the websocket connection to a producer should be kept open. By default, 2 minutes (120 seconds). On each outgoing message, the timer will be reset. Set this to 0 to immediately close the connection when a message has been sent and acknowledged.'), 'producer_url': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, description='An alternative URL to use for producers. If None, the `websocket_url` will be used.'), 'queue_size': FieldInfo(annotation=Union[Annotated[int, Gt], NoneType], required=False, default=None, description='(optional) size of the request queue, if MAX_WORKERS is set, this needs to be at least as big as MAX_WORKERS, otherwise an AttributeException is raised.'), 'topic': FieldInfo(annotation=str, required=True, description='The topic identifier under which to register at the pulsar.'), 'websocket_url': FieldInfo(annotation=str, required=False, default='', description='The fully qualified URL to the websocket.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

producer_connection_timeout: int

producer_keep_alive: int

producer_url: str | None

queue_size: PositiveInt | None

topic: str

websocket_url: str

demessaging.config.registry module

API registry class for demessaging.

Classes:

ApiRegistry(*[, imports, objects])	A registry for imports and encoders

class demessaging.config.registry.**ApiRegistry**(*, *imports: Dict[str, str] = None*, *objects: List[str] = None*)

Bases: BaseModel

A registry for imports and encoders

Parameters

- **imports** (*Dict[str, str]*) – Modules to import at the top of every file. The first item is the module, the second is the alias
- **objects** (*List[str]*) – Source code for objects that should be inlined in the generated Python API.

Methods:

can_import_import(imports)	
hard_code(python_code)	Register some code to be implemented in the generated module.
register_import(module[, alias])	Register a module that needs to be imported in generated API files.
register_type(obj)	Register a class or function to be available in the generated API.

Attributes:

imports	
model_computed_fields	A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.
model_config	Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].
model_fields	Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].
objects	

classmethod can_import_import(*imports: Dict[str, str]*) → Dict[str, str]

hard_code(*python_code: str*) → None

Register some code to be implemented in the generated module.

Parameters
python_code (*str*) – The code that is supposed to be executed on a module level.

imports: Dict[str, str]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'imports': FieldInfo(annotation=Dict[str, str], required=False, default_factory=dict, description='Modules to import at the top of every file. The first item is the module, the second is the alias'), 'objects': FieldInfo(annotation=List[str], required=False, default_factory=list, description='Source code for objects that should be inlined in the generated Python API.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

objects: List[str]

register_import(*module: str*, *alias: str | None = None*) → None

Register a module that needs to be imported in generated API files.

Parameters
module (*str*) – The name of the module, e.g. matplotlib.pyplot

register_type(*obj: T*) → T

Register a class or function to be available in the generated API.

Use this function if you want to have certain functions of classes available in the generated API module, but they should not be transformed to a call to the backend module.

demessaging.messaging package

DASF Methods to connect to the Message Broker.

Submodules

demessaging.messaging.connection module

Base module for a websocket connection.

Classes:

WebsocketConnection(pulsar_config)	Base class to connect to a message broker using a websocket.

Functions:

get_random_letters(length)	

class demessaging.messaging.connection.**WebsocketConnection**(*pulsar_config:* BaseMessagingConfig)

Bases: ABC

Base class to connect to a message broker using a websocket.

Methods:

create_websocketapp([subscription, topic])	
generate_response_topic([topic])	
on_close(ws_app, close_status_code, close_msg)	
on_message(ws_app, msg)	
on_ping(ws_app, payload)	
on_pong(ws_app, payload)	
open_socket([subscription, topic])	

create_websocketapp(*subscription: str | None = None*, *topic: str | None = None*, ***app_kws*) → WebSocketApp

generate_response_topic(*topic: str | None = None*) → str

on_close(*ws_app: WebSocketApp*, *close_status_code*, *close_msg*)

on_message(*ws_app: WebSocketApp*, *msg*)

on_ping(*ws_app: WebSocketApp*, *payload*)

on_pong(*ws_app: WebSocketApp*, *payload*)

open_socket(*subscription: str | None = None*, *topic: str | None = None*, ***connection_kws*) → WebSocket

demessaging.messaging.connection.**get_random_letters**(*length: int*) → str

demessaging.messaging.constants module

Enums within the DASF Messaging Framework.

Classes:

MessageType(value)	Supported message types.
PropertyKeys(value)	Property keys for a message to the message broker.
Status(value)	Status flag of a request.

Functions:

generate_enum_doc(class_)	Utility function to generate the docstring for an enum.

class demessaging.messaging.constants.**MessageType**(*value*)

Bases: str, Enum

Supported message types.

The following values are valid:

- **PING** ("ping")
- **PONG** ("pong")
- **REQUEST** ("request")
- **RESPONSE** ("response")
- **LOG** ("log")
- **INFO** ("info")
- **PROGRESS** ("progress")
- **API_INFO** ("api_info")

Attributes:

API_INFO
INFO
LOG
PING
PONG
PROGRESS
REQUEST
RESPONSE

API_INFO = 'api_info'

INFO = 'info'

LOG = 'log'

PING = 'ping'

PONG = 'pong'

PROGRESS = 'progress'

REQUEST = 'request'

RESPONSE = 'response'

class demessaging.messaging.constants.**PropertyKeys**(*value*)

Bases: str, Enum

Property keys for a message to the message broker.

The following values are valid:

- **REQUEST_CONTEXT** ("requestContext")
- **RESPONSE_TOPIC** ("response_topic")
- **SOURCE_TOPIC** ("source_topic")
- **REQUEST_MESSAGEID** ("requestMessageId")
- **MESSAGE_TYPE** ("messageType")
- **FRAGMENT** ("fragment")
- **NUM_FRAGMENTS** ("num_fragments")
- **STATUS** ("status")

Attributes:

FRAGMENT
MESSAGE_TYPE
NUM_FRAGMENTS
REQUEST_CONTEXT
REQUEST_MESSAGEID
RESPONSE_TOPIC
SOURCE_TOPIC
STATUS

FRAGMENT = 'fragment'

MESSAGE_TYPE = 'messageType'

NUM_FRAGMENTS = 'num_fragments'

REQUEST_CONTEXT = 'requestContext'

REQUEST_MESSAGEID = 'requestMessageId'

RESPONSE_TOPIC = 'response_topic'

SOURCE_TOPIC = 'source_topic'

STATUS = 'status'

class demessaging.messaging.constants.**Status**(*value*)

Bases: str, Enum

Status flag of a request.

The following values are valid:

- **SUCCESS** ("success")
- **ERROR** ("error")
- **RUNNING** ("running")

Attributes:

ERROR
RUNNING
SUCCESS

ERROR = 'error'

RUNNING = 'running'

SUCCESS = 'success'

demessaging.messaging.constants.**generate_enum_doc**(*class_: Type[Enum]*) → Type[Enum]

Utility function to generate the docstring for an enum.

demessaging.messaging.consumer module

Consumer for messages submitted via the message broker.

Classes:

MessageConsumer(pulsar_config, handle_request)	Consumer for messages submitted via the message broker.

class demessaging.messaging.consumer.**MessageConsumer**(*pulsar_config:* BaseMessagingConfig, *handle_request*, *handle_response=None*, *module_info: dict* | *None* = *None*, *api_info:* ModuleAPIModel | *None* = *None*)

Bases: *WebsocketConnection*

Consumer for messages submitted via the message broker.

Attributes:

RECONNECT_TIMEOUT_SLEEP
SOCKET_PING_INTERVAL

Methods:

acknowledge(msg)	
close_websocket_app(ws_app[, reason])	
disconnect()	
extract_context(msg)	
extract_message_type(msg)	
extract_response_topic(msg)	
handle_api_info(api_info_request)	Show the api of the module.
handle_info(info_request)	
handle_pong(request)	
handle_request_via_queue(msg)	
is_valid_request(request_message)	
is_valid_value(value)	
on_message(ws_app, msg)	
on_producer_message(ws_app, msg)	
open_producer_app(topic, **kwargs)	
reset_close_timer(ws_app)	
send_error(request, error_message)	
send_pong(request)	
send_response(request[, response_payload, ...])	
setup_subscription()	
wait_for_request()	
wait_for_websocket_connection(ws_app[, timeout])	

RECONNECT_TIMEOUT_SLEEP = 10

SOCKET_PING_INTERVAL = 60

acknowledge(*msg*)

close_websocket_app(*ws_app: WebSocketApp*, *reason: str = 'unspecified'*)

disconnect()

static extract_context(*msg*) → str | None

static extract_message_type(*msg*)

static extract_response_topic(*msg*)

handle_api_info(*api_info_request*)

Show the api of the module.

handle_info(*info_request*)

handle_pong(*request*)

handle_request_via_queue(*msg*)

static is_valid_request(*request_message*)

static is_valid_value(*value*)

on_message(*ws_app: WebSocketApp*, *msg*)

on_producer_message(*ws_app: WebSocketApp*, *msg: str*)

open_producer_app(*topic*, ***kwargs*)

producer_locks: Dict[WebSocketApp, allocate_lock]

producer_threads: Dict[WebSocketApp, Thread]

producer_timer: Dict[WebSocketApp, Timer]

producers: Dict[str, WebSocketApp]

request_semaphore: BoundedSemaphore | None

reset_close_timer(*ws_app: WebSocketApp*)

send_error(*request*, *error_message*)

send_pong(*request*)

send_response(*request*, *response_payload=None*, *msg_type=MessageType.RESPONSE*, *response_properties=None*)

setup_subscription()

wait_for_request()

wait_for_websocket_connection(*ws_app: WebSocketApp*, *timeout=10*)

demessaging.messaging.producer module

Producer of messages submitted for the message broker.

Classes:

MessageProducer(pulsar_config[, topic])	Producer class to send requests to a registered backend module (topic)

class demessaging.messaging.producer.**MessageProducer**(*pulsar_config:* BaseMessagingConfig, *topic:* *str* | *None* = *None*)

Bases: *WebsocketConnection*

Producer class to send requests to a registered backend module (topic)

Attributes:

RECONNECT_TIMEOUT_SLEEP	
SOCKET_PING_INTERVAL	
in_app	
is_connected	Check if the websocket apps are connected.
out_app	

Methods:

connect()	
disconnect()	Disconnect in- and out-sockets.
on_in_message(ws_app, response)	Message handler for the incoming websocket connection.
on_out_message(ws_app, ack)	Message handler for the outgoing websocket connection.
send_request(request_msg)	Sends the given request to the backend module bound to the topic provided in the pulsar configuration.
setup_subscription()	
wait_for_connection([timeout])	Wait until the websockets are connected

RECONNECT_TIMEOUT_SLEEP = 5

SOCKET_PING_INTERVAL = 60

connect()

disconnect()

Disconnect in- and out-sockets.

in_app: WebSocketApp

property is_connected: bool

Check if the websocket apps are connected.

on_in_message(*ws_app: WebSocketApp*, *response*)

Message handler for the incoming websocket connection.

on_out_message(*ws_app: WebSocketApp*, *ack*)

Message handler for the outgoing websocket connection.

out_app: WebSocketApp

response_topic: str

async send_request(*request_msg*) → Any

Sends the given request to the backend module bound to the topic provided in the pulsar configuration. In order to increase re-usability the destination topic can be overridden with the optional topic argument.

Parameters

- **request_msg** – dictionary providing a 'property' dictionary, a payload string, or both
- **topic** – overrides the used topic for this request

Returns

received response from the backend module

setup_subscription()

subscription_name: str

wait_for_connection(*timeout=10*)

Wait until the websockets are connected

demessaging.serializers package

Common serializers for DASF backend modules.

Submodules

demessaging.serializers.xarray module

demessaging.validators package

Common validators for DASF backend modules.

Submodules

demessaging.validators.xarray module

Submodules

demessaging.NetCDFDecoder module

demessaging.PulsarConnection module

Classes:

PulsarConnection(*args, **kwargs)	

class demessaging.PulsarConnection.**PulsarConnection**(**args*, ***kwargs*)

Bases: *WebsocketConnection*

demessaging.PulsarMessageConstants module

demessaging.PulsarMessageConsumer module

Classes:

PulsarMessageConsumer(*args, **kwargs)	

class demessaging.PulsarMessageConsumer.**PulsarMessageConsumer**(**args*, ***kwargs*)

Bases: *MessageConsumer*

demessaging.PulsarMessageProducer module

Classes:

PulsarMessageProducer(*args, **kwargs)	

class demessaging.PulsarMessageProducer.**PulsarMessageProducer**(**args*, ***kwargs*)

Bases: *MessageProducer*

demessaging.cli module

Command-line options for the backend module.

Classes:

MessagingArgumentParser([prog, usage, ...])	An `argparse.ArgumentParser` for the messaging framework.

Functions:

get_parser([module_name, config])	Generate the command line parser.
split_namespace(ns)	Split a namespace into multiple namespaces.

class demessaging.cli.**MessagingArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

Bases: ArgumentParser

An argparse.ArgumentParser for the messaging framework.

Methods:

parse_known_args([args, namespace])	

parse_known_args(*args=None*, *namespace=None*)

demessaging.cli.**get_parser**(*module_name: str = '__main__'*, *config: ModuleConfig | None = None*) → ArgumentParser

Generate the command line parser.

demessaging.cli.**split_namespace**(*ns: Namespace*) → Tuple[Namespace, Dict[str, Namespace]]

Split a namespace into multiple namespaces.

This utility function takes a namespace and looks for attributes with a `.` inside. These are then splitted and returned as the second attribute.

Example

Consider the following namespace:

```
>>> from argparse import Namespace
>>> ns = Namespace(**{"main": 1, "sub.main": 2})
>>> split_namespace(ns)
(Namespace(main=1), {'sub': Namespace(main=2)})
```

demessaging.template module

Classes:

Template(name, folder, suffix, context, ...)	A convenience wrapper to render a jinja2 template.

class demessaging.template.**Template**(*name: str*, *folder: ~pathlib.Path = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessag*
suffix: str = '.jinja2', *context: ~typing.Dict[str, ~typing.Any] = <factory>*)

Bases: object

A convenience wrapper to render a jinja2 template.

Attributes:

context
folder
name
path
renderer
suffix

Methods:

render(**context)

context: Dict[str, Any]

folder: Path = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/dasf/checkouts/latest/demessaging/templates')

name: str

property path: Path

render(***context*) → str

property renderer: Any

suffix: str = '.jinja2'

demessaging.utils module

Utilities for the demessaging module.

Functions:

append_parameter_docs(model)	Append the parameters section to the docstring of a model.
build_parameter_docs(model)	Build the docstring for the parameters of a model.
merge_config(base, to_merge)	Merge two configuration dictionaries.
object_to_string(obj)	
type_to_string(type_)	

demessaging.utils.**append_parameter_docs**(*model: Type[BaseModel]*) → Type[BaseModel]

Append the parameters section to the docstring of a model.

demessaging.utils.**build_parameter_docs**(*model: Type[BaseModel]*) → str

Build the docstring for the parameters of a model.

demessaging.utils.**merge_config**(*base: Dict*, *to_merge: Dict*) → Dict

Merge two configuration dictionaries.

Parameters

- **base** (*Dict*) – The base dictionary that *to_merge* shall be merged into.
- **to_merge** (*Dict*) – The dictionary to merge.

Returns

base merged with *to_merge*

Return type

Dict

Notes

base is modified in-place!

demessaging.utils.**object_to_string**(*obj: Any*)

demessaging.utils.**type_to_string**(*type_: Any*)

2.4 Developers Guide

If you want to learn more about the internals of the DASF messaging framework, then welcome!

This part of the docs introduces you more deeply into

2.4.1 Remote Procedure Calls

The DASF Messaging Framework enables you to do so-called *Remote Procedure Calls (RPC)*, meaning that you call a function on your local computer and this function is then executed on a remote machine.

In the terminology of RPC, your local computer is the so-called *client stub*, whereas the remote computer is the so-called *server stub*.

Because most people never heard of RPC, we often just name the *client stub* client, and we call the *server stub* the *backend module*.

2.4.2 Abstraction and Serialization of the backend module

Setting up a backend module within the DASF Framework can be done by every newbie who just started with python. The basic API is very simple and a lot happens under the hood. So this here is already a backend module

Listing 13: `hello_world` function exposed via a HelloWorld backend module.

```
from demessaging import main


__all__ = ["hello_world"]


def hello_world(name: str) -> str:
    return 'Hello World, ' + name


if __name__ == "__main__":
    main()
```

You are probably familiar with these things (unless you are completely new to python), but let's go through it and explain a bit, what happens here in the background:

- `from demessaging import main`:

  This and the call of the *`main()`* function is the only DASF specific part in this framework. We will come to this function below

- `__all__ = ["hello_world"]`

  This variable specifies, what should be available for the remote procedure call (see *Specifying module members*).

- `def hello_world(name:  str) -> str:`

  This is the definition of a procedure that we want to make available as RPC through our framework. It's just a python function with type annotations (see *Abstracting procedures* for more details.)

- `return 'Hello World, ' + name`

  This is the computing part that happens in the server stub. In our case we just combine two strings, but you could do anything here that python let's you do.

- `main()`

  Now the magic happens. The *`main()`* function that takes control now. It interprets your `hello_world` function and transforms it into something, that can be used to execute remote procedure calls (a subclass of pydantics `BaseModel`, we'll come to that later). It also generates a command-line interface that you can use to configure several aspects in DASF, see *the backend config*.

More in-depth details that describe what is happening behind the scenes are provided in the following sections.

Abstracting backend modules

A remote procedure call in DASF happens via a single message to the endpoint where the server stub is listening to, i.e. the topic at the message broker.

To call the `hello_world` function of *our example code*, for instance, the consumer expects the following payload in a message sent to the topic:

```
{
    "func_name": "hello_world",
    "name": "Peter"
}
```

Obviously the "func_name" corresponds to the name of the function to call, and the remaining parameters (here only "name") are the input for the function.

After this section we describe in more details

1. how we decide what procedure is available for the RPC call (i.e. what the valid values for "func_name" is, see *Specifying module members*)
2. how we verify that the request is a valid request for the procedure that should be called via RPC (see *Abstracting procedures*).

But let us anticipate the final result: What we end up with is one single *pydantic* root model[1] that represents the models of the individual procedures (aka functions and classes in the backend module).

Our *main()* function that we use in our example code is in fact only a shortcut that equips the *create_model()* classmethod of the *BackendModule* class with a command-line interface. The purpose of this function is to create a subclass of the *BackendModule* that accepts our request from above, and is able to map it to the hello_world function.

In our example code, the resulting model for the backend module, the HelloWorldBackendModule, is comparable to

```
from demessaging.backend import BackendModule, BackendFunction
from typing import Literal

class HelloWorldModel(BackendFunction):

    func_name: Literal["hello_world"] = "hello_world"
    name: str

class HelloWorldBackendModule(BackendModule):

    root: HelloWorldModel
```

and our hello_world function can be called with

```
# create the model
model = HelloWorldBackendModule.parse_json("""
{
    "func_name": "hello_world",
    "name": "Peter"
}
""")
# call the function
assert hello_world("Peter") == model()
```

But, let's now start with understanding, how we select the members for the remote procedure call.

---

[1] You will find more details about these kind of models in *Abstracting procedures*.

Specifying module members

The first step in our creation of the backend module is to specify what objects we want to make available for the remote procedure call. You find that workflow in the source code of the backend modules *create_model()* method.

In our example we used the `__all__` variable:

```
__all__ = ["hello_world"]
```

In fact, there are two possible ways to specify what should be available for the RPC:

1. using the `__all__` module variable (as we do above). The `__all__` variable is commonly used when using import statements like `from mymodule import *` (see the python docs), but it's also a good way to tell other developers about the important functions and methods in your code.
2. The second possibility is specifying the members in the call to the *main()* function, e.g. `main(members=["hello_world"])` or `main(members=[hello_world])`. This would then take precedence over the `__all__` variable of the module.

Abstracting procedures

Basic requirements

A procedure in our framework can be a module-level function or a class method. In our example above, we define a module-level function with

```
def hello_world(name: str) -> str:
    return 'Hello World, ' + name
```

and make this available for the RPC. It's a standard python function, so, nothing special here. **But** there is one important aspect: the so-called *type annotation* `name: str` and `-> str`. This feature of the python language, introduced with python 3.5, is crucial to the DASF messaging framework. In standard python, type annotations have no impact during runtime. So you can call `hello_world(["ok"])` and it would run the function (and produce an error of course, as it cannot combine `"Hello World, " + ["ok"]`). Only if you would use a static type checker such as `mypy`, it would produce an error without even prior to calling the function, as it realizes that the `hello_world` function accepts a string, and not a list. Within DASF however, we **do use** the type annotation **during runtime**. It is an important part in any web-based framework, **that you need to verify the input**. But instead of adding an additional verification system, we use the type annotations here[1]. If you want to understand the internals of the DASF messaging framework, you should familiarize yourself a bit with type annotations in python.

How the abstraction works

Our framework takes standard python functions or classes and makes them available for a remote procedure call. As a web-based framework, this means that

1. Our framework needs to be able to deserialize JSON requests
2. Our framework needs to be able to validate the requests
3. Our framework needs to be able to serialize JSON requests

Let's take our `hello_world` function from above. If you would want to call this function without our framework, you'd need to implement some wrapper that deserializes the request, validates the input and serializes the output, i.e. something like

[1] nevertheless, you can also add your own validators using the *configure()* function

```python
import json

def hello_world_request_wrapper(request: str):
    data = json.loads(request)  # deserialize the request
    assert isinstance(data, dict)  # make sure we get a dictionary
    assert "name" in data  # check that the name is in the data
    assert isinstance(data["name"], str)  # make sure the name is a string
    result = hello_world(name=data["name"])  # call our function
    return json.dumps(result)  # serialize and return the result
```

You can see that such kind of serialization and validation becomes quite verbose. And now imagine you have more complicated procedure calls then our very simple example. A hard-coded verification would add a lot of overhead to the code, just to make the procedures available for RPCs.

The abstraction in the DASF works differently and (usually) without additional code. The idea is that when you specify a function such as `def hello_world(name: str) -> str:` we know what we expect. Everything is here: The function, the arguments, the inputs, etc. So what we are doing is, that we take pythons built-in `inspect` module to analyse the function, and docstring-parser for interpreting the docstring of the procedure. Then we use this information and create a subclass of pydantics `BaseModel` that represents this function (or class or method).

Note: *pydantic* is a python package that uses pythons type annotations for runtime validation of data. Moreover, it can be used to serialize and de-serialize input based on the annotations. You can find many examples in their docs.

We recommend that you go through the overview and familiarize yourself a bit with it before continuing.

We'll stick now to our `hello_world` function and look how this is abstracted for the remote procedure call. Classes are handled pretty much the same, but we'll discuss this later in *Abstracting classes for RPC*.

Abstracting functions for RPC

When you call the *`main()`* function, it loads the procedures that you make available for the remote procedure call (see *Specifying module members*). If it encounters a function, it will call the *`create_model()`* classmethod of the *`demessaging.backend.function.BackendFunction()`* class. This method creates a subclass of the *`BackendFunction()`* using pydantics `create_model` function that represents our `hello_world` function in DASF.

In other words, you end up with

```python
from demessaging.backend.function import BackendFunction
HelloWorldModel = BackendFunction.create_model(hello_world)
```

The `HelloWorldModel` class that has been created through this method is comparable to the following class:

```python
from demessaging.backend.function import BackendFunction
from typing import Literal

class HelloWorldModel(BackendFunction):

    func_name: Literal["hello_world"] = "hello_world"
    name: str
```

It's a class with two attributes, `func_name` and `name`.

`func_name` is an attribute that is always added to these kind of models. It can take exactly one value, and this is the name of the function that it serialized (in this case, `"hello_world"`). The second attribute, `name`, comes from the

function that has been called. The `hello_world` function accepts exactly one argument, the `name: str`, and so this has been added as an attribute to our model.

The *create_model()* classmethod does create another model, the return model. Our `hello_world` function returns a string and also needs to be serialized, when sent to the client stub. Our return model is a *__root__* model of pydantic and comparable to

```
from pydantic import BaseModel

class HelloWorldReturnModel(BaseModel):

    root: str
```

and is accessible as `HelloWorldModel.return_model`.

Calling the procedure

Our framework creates an instance of the created `HelloWorldModel` from the request. It then executes the underlying model function via

```
request = HelloWorldModel(name="Peter")
result_model = request()  # calls the `hello_world` function. `result_model` is an␣
↪instance of HelloWorldReturnModel

assert result_model.root == hello_world(name="Peter")
```

But the thing is, that we can now also create and return JSON from this:

```
request = HelloWorldModel.parse_json('{"name": "Peter"}')
result_model = request()
result_model.model_dump_json()  # gives 'Hello World, Peter'
```

Configuring the Function Model Creation

You can also change how things are converted from the python function to the pydantic model using the *configure()* function. Passing arguments to this function would be equivalent to setting the `config` parameter to the call of *create_model()*. You can give any keyword argument here that is available for the initialization of a *FunctionConfig* instance. You can, for instance, add custom `validators` or `field_params`.

Abstracting classes for RPC

Instead of using a function as described above, one can also use a method of a python class. Here is an example:

```
class HelloClass:

    def __init__(self, text: str ="Hello"):
        self._text = text

    def hello_world(self, name: str) -> str:
        return self._text + " World, " + name
```

calling `hello_world("Peter")` is equivalent to calling `HelloClass().hello_world("Peter")`. When we encounter a class in the members that you specified for the *`main()`* function (see *Specifying module members*), then we will not use the *`BackendFunction`* to create a new model, but instead we use the *`BackendClass`*.

The *`create_model()`* classmethod of the *`BackendClass`* uses the same procedure as described in the previous section (*Abstracting functions for RPC*) to abstract the `__init__` method of the class. And it does an abstraction for all the methods of the class.

The pydantic representation of our class, i.e. the `HelloClassModel` in

```
from demessaging.backend.class_ import BackendClass
HelloClassModel = BackendClass.create_model(HelloClass)
```

is comparable to:

```
from demessaging.backend.class_ import BackendClass
from demessaging.backend.function import BackendFunction

class HelloClassHelloWorldModel(BackendFunction):

    func_name: Literal["hello_world"] = "hello_world"
    name: str

class HelloClassModel(BackendClass):

    class_name: Literal["HelloClass"] = "HelloClass"
    text: str = "Hello"
    function: HelloClassHelloWorldModel
```

and you would create an instance of this method via

```
request = HelloClassModel(
    text="Hello",
    function={"func_name": "hello_world", "name": "Peter"}
)
```

Hence, you specify the method that you want to call. Running `request()` now creates an instance of `HelloClass` and runs it's `hello_world` method, i.e.

```
response = HelloClassModel()
response.root == HelloClass(text="Hello").hello_world(name="Peter")
```

Configuring the Class Model Creation

You can configure classes as you can *configure functions*, just decorate your class with the *`configure()`* function and pass the parameters for the *`ClassConfig`*. You can, for instance, use the `methods` parameter to specify what methods will be available for the RPC. You can also decorate any method of the class with the *`configure()`* function as you do with normal functions.

2.4.3 Messaging and Message Brokers

Messaging

In standard web-based frameworks, communication works via HTTP requests. The client (e.g. the frontend of a website) makes a GET or POST to server (backend) and receives a response. This standard framework, however, exposes your backend to a lot of security issues and it takes a lot of effort making it secure.

DASF uses a slightly different approach. Here, client and backend cannot communicate with each other directly. Instead, we use a so-called message broker. The client stub (see *above*) sends the request to a so-called message broker which forwards the request to the server stub.

Fig. 1: Request to the backend module

Then the server processes the request and sends back a response to the client, again via the message broker.

Fig. 2: Response from the backend module

It is somehow comparable to a chat system. Assume Alice wants to chat with Bob using their mobile phones. The mobile phones do not know each other but instead they both register at a server (the message broker). When Alice wants to send a message to Bob, she actually sends it to the server and the server forwards it to Bobs mobile phone.

The advantage of such a framework is that neither the client nor the server need to be accessible from the web. This is especially important if you want to use protected computing resources such as internal servers or HPC clusters. The only thing you have to do is to protect and secure the message broker, this already takes most of the burden to secure your *backend module*.

Message Brokers

Concept

Message brokers distinguish between *consumers* and *producers*. As the names suggest, *producers* produce a message and *consumers* consume the message. When a client stub makes a request, the client is the *producer*, and the server is the *consumer*.

Fig. 3: Request to the backend module. The client **produces** a message, the server **consumes**.

And for the response, the client is the *consumer* and the server is the *producer*.

To connect producer and consumer, we use so-called *topics*. You can think of a topic to be the address of the *consumer* at the message broker. A client that wants to send a request to a backend module, needs to know the topic that the backend module is listening to (i.e. that the backend module is *consuming*). Likewise, the backend module needs to know where it should send the response to (i.e. it needs to know the topic that the client is *consuming*).

Within the DASF Framework, you specify the topic when you start listening for incoming messages. This is specified with the `-t` option

```
python ExampleMessageConsumer.py -t my-personal-topic listen
```

or you specify it in the call of the *`main()`* function,

Fig. 4: Response from the backend module. The server **produces** a message, the client **consumes**.

```
if __name__ == "__main__":
    main(messaging_config=dict(topic="my-personal-topic"))
```

The response topic, i.e. the topic that the client is consuming, is specified within the properties of the request (see *demessaging.messaging.constants.PropertyKeys*). From this, the backend module automatically knows where to send the response to.

Choosing the message broker

There are a variety of message brokers around, such as Apache Pulsar or MQTT. And we are working on a Django-based message broker in the DASF context.

We aim to be agnostic to what message broker is used. Currently, however, we only tested our workflows with the Apache Pulsar and our Django-based message broker. An important point is that, by design, our connection to the message broker does not use provided client libraries, such as the `pulsar-client`, etc.. Instead, we use the websocket protocol to connect to the message broker. This is necessary to guarantee flexibility of the framework. So the choice of your message broker depends on the application that you want to build.

2.4.4 Generating the client stub

Within DASF we provide the possibility to automatically generate the client stub from the server stub (see *Remote Procedure Calls* if you want to know more about these terms). This is done in the *generate()* method of the *BackendModule* class and can also be invoked from the command line, e.g.

```
python ExampleMessageConsumer.py generate
```

The generation of these files work via so-called *jinja2* templates[1]. Each config class, i.e. *FunctionConfig*, *ClassConfig* and *ModuleConfig* holds the reference to a *Template*. This template is used to render the corresponding function, class or module in the client stub. You can find these templates in the `demessaging/templates` folder of the source code.

Note: currently we only support the automatic creation of python client stubs. But we aim for the support of typescript in the near future.

If there is missing something, please open an issue in the gitlab repository (e.g. via the service desk). There are no dump questions, just bad documentation!

[1] https://pypi.org/project/Jinja2/

CHAPTER

THREE

HOW TO CITE THIS SOFTWARE

Please do cite this software!

APA

```
Eggert D., Sips M., Sommer P.S., Dransch D. Data Analytics Software Framework URL:␣
↪https://codebase.helmholtz.cloud/dasf/dasf-messaging-python
```

BibTex

```
@misc{YourReferenceHere,
author = {Eggert, Daniel and Sips, Mike and Sommer, Philipp S. and Dransch, Doris},
title = {Data Analytics Software Framework},
url = {https://codebase.helmholtz.cloud/dasf/dasf-messaging-python}
}
```

RIS

```
TY  - GEN
AB  - python module wrapper for the data analytics software framework DASF

AU  - Eggert, Daniel
AU  - Sips, Mike
AU  - Sommer, Philipp S.
AU  - Dransch, Doris
KW  - digital-earth
KW  - dasf
KW  - pulsar
KW  - gfz
KW  - hzg
KW  - hereon
KW  - hgf
KW  - helmholtz
```

(continues on next page)

(continued from previous page)

```
KW  - remote procedure call
KW  - rpc
KW  - django
KW  - python
KW  - websocket
TI  - Data Analytics Software Framework
UR  - https://codebase.helmholtz.cloud/dasf/dasf-messaging-python
ER
```

Endnote

```
%0 Generic
%A Eggert, Daniel
%A Sips, Mike
%A Sommer, Philipp S.
%A Dransch, Doris
%K digital-earth
%K dasf
%K pulsar
%K gfz
%K hzg
%K hereon
%K hgf
%K helmholtz
%K remote procedure call
%K rpc
%K django
%K python
%K websocket
%T Data Analytics Software Framework
%U https://codebase.helmholtz.cloud/dasf/dasf-messaging-python
```

CFF

```
# SPDX-FileCopyrightText: 2019-2024 Helmholtz Centre Potsdam GFZ German Research Centre␣
↪for Geosciences
# SPDX-FileCopyrightText: 2021-2024 Helmholtz-Zentrum hereon GmbH
#
# SPDX-License-Identifier: CC0-1.0

authors:
  # list author information. see
  # https://github.com/citation-file-format/citation-file-format/blob/main/schema-guide.
↪md#definitionsperson
  # for metadata
  - family-names: "Eggert"
```

(continues on next page)

(continued from previous page)

```
    given-names: "Daniel"
    affiliation: "Helmholtz Centre Potsdam GFZ German Research Centre for Geosciences"
    orcid: https://orcid.org/0000-0003-0251-4390
    email: "daniel.eggert@gfz-potsdam.de"
  - family-names: "Sips"
    given-names: "Mike"
    affiliation: "Helmholtz Centre Potsdam GFZ German Research Centre for Geosciences"
    orcid: https://orcid.org/0000-0003-3941-7092
  - family-names: "Sommer"
    given-names: "Philipp S."
    affiliation: "Helmholtz Centre Potsdam GFZ German Research Centre for Geosciences"
    orcid: https://orcid.org/0000-0001-6171-7716
    email: "philipp.sommer@hereon.de"
  - family-names: "Dransch"
    given-names: "Doris"
    affiliation: "Helmholtz Centre Potsdam GFZ German Research Centre for Geosciences"
    # orcid: null
cff-version: 1.2.0
message: "If you use this software, please cite it using these metadata."
title: "Data Analytics Software Framework"
keywords:
  - "digital-earth"
  - "dasf"
  - "pulsar"
  - "gfz"
  - "hzg"
  - "hereon"
  - "hgf"
  - "helmholtz"
  - "remote procedure call"
  - "rpc"
  - "django"
  - "python"
  - "websocket"

license: 'Apache-2.0'
repository-code: "https://codebase.helmholtz.cloud/dasf/dasf-messaging-python"
url: "https://codebase.helmholtz.cloud/dasf/dasf-messaging-python"
contact:
  # list maintainer information. see
  # https://github.com/citation-file-format/citation-file-format/blob/main/schema-guide.
↪md#definitionsperson
  # for metadata
  - family-names: "Sommer"
    given-names: "Philipp S."
    affiliation: "Helmholtz Centre Potsdam GFZ German Research Centre for Geosciences"
    # orcid: null
    email: "philipp.sommer@hereon.de"
abstract: |
  python module wrapper for the data analytics software framework DASF
```

CHAPTER

FOUR

OPEN SOURCE AND OPEN SCIENCE

4.1 License information

4.2 Repository

The individual DASF modules are developed via the following git group. Feel free to checkout the source code or leave comment via the service desk or directly via the issue tracker.

Gitlab Repository URL

https://codebase.helmholtz.cloud/dasf

4.3 Acknowledgment

DASF is developed at the GFZ German Research Centre for Geosciences (https://www.gfz-potsdam.de) in collaboration with the Helmholtz-Zentrum Hereon https://www.hereon.de and was funded by the Initiative and Networking Fund of the Helmholtz Association through the Digital Earth project (https://www.digitalearth-hgf.de/).

PYTHON MODULE INDEX

d

INDEX

D

I

J

L

M

N

O

P

Q

R

S

T

U

V

W