Django Message Broker

Release 0.1.0

Tanzo Creative Ltd

Mar 25, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Django Message Broker is a plugin written in Python for Django that provides an all-in-one message broker. It interfaces with Django Channels and Celery [1], and replaces the need for separate message brokers such as Redis and RabbitMQ.

The motivating use case for Django Message Broker is small site solutions where it is easier to deploy a single server containing all the required functionality rather than a multi-server solution. An example would be a small site running data science models, where executing the model is a long running process and needs to execute in the background so that it doesn't degrade the responsiveness of the user interface.

Potential scenarios for Django Message Broker include:

- Prototyping, Testing, Training
- Data science projects where the model complexity exceeds the capabilities of packages such as Shiny, Dash and Streamlit.
- Small systems with a low number of users.

The Django Message Broker is an installable component of the Django platform that provides an easy to install, all-in-one alternative for small scale solutions. It does not replace the need for high volume message brokers where message volume and redundancy are important.

Note:

1. The Celery interface is in development and not supported in this release.

1.1 Installation and configuration

1.1.1 Installing Django Message Broker

Install latest stable version into your python environment using pip:

```
pip install django-message-broker
```

Once installed add `django_message_broker` to your `INSTALLED_APPS` in settings.py:

```
INSTALLED_APPS = (
    'django_message_broker',
    ...
)
```

Django Message Broker should be installed as early as possible in the installed applications list and ideally before other applications such as Channels and Celery. The message broker forks a background process which should occur before other applications create new threads in the current process.

1.1.2 Configure Django Channels Layers

To configure the Django Message Broker as a Channels layer add the following to the `CHANNEL_LAYERS` setting in settings.py:

```
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'django_message_broker.ChannelsServerLayer',
    },
}
```

1.2 Reference

Public APIs support integration with other Django applications. The internal APIs are documented to facilitate integration with other applications.

1.2.1 Public Interfaces

Public APIs support integration with other Django applications.

Celery API

NOT IMPLEMENTED.

The Celery API is planned for implementation.

Django Channels API

This provides an implementation of the Django Channels layers API.

Channels installable channel backend using Django Message Broker

class django_message_broker.layer.**ChannelsServerLayer**(*expiry=60*, *group_expiry=86400*, *capacity=100*, *channel_capacity=None*, *ip_address='127.0.0.1'*, *port=5556*, ***kwargs*)

Django Message Broker channel layer backend.

property channels: List[bytes]

Return list of channel names.

Returns List of channel names.

Return type List[bytes]

async new_channel(*prefix: str = 'specific'*) → str

Returns a new channel name that can be used by something in our process as a specific channel.

Parameters **prefix** (*str, optional*) – Prefix to channel name. Defaults to "specific".

Returns Process specific channel name.

Return type str

async send(*channel: str*, *message: Dict*) → None
Send a message onto a (general or specific) channel.

Parameters

- **channel** (*str*) – Channel name
- **message** (*Dict*) – Message to send

async receive(*channel: str*) → Dict
Receive the first message that arrives on the channel. If more than one coroutine waits on the same channel, a random one of the waiting coroutines will get the result. Subscribe to the channel first to ensure that the Channels client receive messages from the server.

Parameters **channel** (*str*) – Channel name

Returns Received message.

Return type Dict

async flush() → None
Resets the server by flushing all messages from the message store, and groups from the group store.

async close() → None
Closes the client connection.

It's not clear what the purpose of this function should be? Perhaps it is intended to trigger removal of channel from messages store? But would only make sense if process specific.

async group_add(*group: str*, *channel: str*) → None
Adds the channel to a group. If the group doesn't exist then it is created. A subscription request is also sent to the server to ensure that messages are delivered locally.

Parameters

- **group** (*str*) – Name of the group to join.
- **channel** (*str*) – Channel joining the group.

async group_discard(*group: str*, *channel: str*) → None
Removes a channel from a group

Parameters

- **group** (*str*) – Group name
- **channel** (*str*) – Channel name

async group_send(*group: str*, *message: Dict[str, Any]*) → None
Sends a message to a group

Parameters

- **group** (*str*) – Group name
- **message** (*str*) – Message

Process Worker API

This provides a message interface for background process workers.

Implements a client for the Django Message Broker within a process

class django_message_broker.process_client.**ProcessClient**(*expiry=60*, *group_expiry=86400*, *ip_address='127.0.0.1'*, *port=5556*, ***kwargs*)

Implements a process client.

async new_channel(*prefix: str = 'specific'*) → str

Returns a new channel name that can be used by something in our process as a specific channel.

Parameters **prefix** (*str, optional*) – Prefix to channel name. Defaults to "specific".

Returns Process specific channel name.

Return type str

async send(*channel: str*, *message: Dict*) → None

Send a message onto a (general or specific) channel.

Parameters

- **channel** (*str*) – Channel name
- **message** (*Dict*) – Message to send

async receive(*channel: str*) → Dict

Receive the first message that arrives on the channel. If more than one coroutine waits on the same channel, a random one of the waiting coroutines will get the result. Subscribe to the channel first to ensure that the Channels client receive messages from the server.

Parameters **channel** (*str*) – Channel name

Returns Received message.

Return type Dict

async group_add(*group: str*, *channel: str*) → None

Adds the channel to a group. If the group doesn't exist then it is created. A subscription request is also sent to the server to ensure that messages are delivered locally.

Parameters

- **group** (*str*) – Name of the group to join.
- **channel** (*str*) – Channel joining the group.

async group_discard(*group: str*, *channel: str*) → None

Removes a channel from a group

Parameters

- **group** (*str*) – Group name
- **channel** (*str*) – Channel name

async group_send(*group: str*, *message: dict*) → None

Sends a message to a group

Parameters

- **group** (*str*) – Group name
- **message** (*str*) – Message

1.2.2 Internal Interfaces

The internal APIs are documented to facilitate integration with other applications.

Channels Client

Implements a channel client for the Django Message Broker.

class django_message_broker.server.channels_client.**ChannelsClient**(**args*, *ip_address: str = '127.0.0.1'*, *port: int = 5556*, ***kwargs*)

Client for Django Channels message server.

Implements a client to the network message server for Django channels using zero message queue (ZMQ) to support communication between channels registered within different processes.

The server opens two sequential ports:

- Data port (base port, default=5556): Transmission of data messages between the client and server.
- Signalling port (base port + 1, default=5557): Transmission of signalling messages between the client and server.

class DataCommands
Create registry of data commands using decorator.

class SignallingCommands
Create registry of signalling commands using decorator.

stop()
Stops the server and flushes all queues.

_flush_queues() → None
Periodic callback to flush queues where there are no subscribers or messages.

async _await_data_response(*message_id: bytes*) → None
Creates an event which is set when a confirmation message is received on the data channel from the server.

Parameters **message_id** (*bytes*) – Message id of the sent message for which a response is required.

async _await_signalling_response(*message_id: bytes*) → None
Creates an event which is set when a confirmation message is received on the signalling channel from the server.

Parameters **message_id** (*bytes*) – Message id of the sent message for which a response is required.

get_routing_id() → str
Returns the routing id from the zmq.DEALER socket used to route message from the server to the client. The routing id is a 32-bit unsigned integer which is appended to the string *zmq_id_* to provide a unique channels client identifier.

Returns Routing id

Return type str

get_subscriber_id() → bytes
Returns a unique subscriber id across all clients connected to the server. Uniqueness is obtained by combining the *routing_id* of the ZMQ socket with a uuid4 reference. This minimises the chance that any two subscribers will have the same reference.

Returns Unique subscriber reference.

Return type bytes

async _subscribe(*channel_name: bytes*, *subscriber_name: bytes*) → None
Subscribes to a channel to ensure that messages are delivered to the client.

Parameters **channel_name** (*bytes*) – Name of channel subscribed to.

async _unsubscribe(*channel_name: bytes*, *subscriber_name: bytes*) → None
Unsubscribes from a channel this drops all local messages and removes the subscription from the server. This may result in messages being lost.

Parameters **channel_name** (*bytes*) – Name of channel subscribed to.

async _receive(*subscriber_name: bytes*) → Dict
Receive the first message that arrives on the channel. If more than one coroutine waits on the same channel, a random one of the waiting coroutines will get the result.

Parameters **channel** (*bytes*) – Channel name

Returns Received message.

Return type Dict

async _send(*channel_name: bytes*, *message: Dict*, *time_to_live: float = 60*, *acknowledge=False*) → None
Sends a message to a channel.

Parameters

- **channel** (*Union[str, bytes]*) – Channel name
- **message** (*Dict*) – Message to send (as a dictionary)
- **time_to_live** (*float, optional*) – Time to live (seconds). Defaults to 60.
- **acknowledge** (*bool*) – Await server has processed message. Defaults to False.

async _send_to_group(*group_name: bytes*, *message: Dict*, *time_to_live: float = 60*) → None
Sends a message to a group.

Parameters

- **group** (*Union[str, bytes]*) – Group name
- **message** (*Dict*) – Message to send (as dictionary)
- **time_to_live** (*int, optional*) – Time to live (seconds). Defaults to 60.

async _group_add(*group_name: bytes*, *channel_name: bytes*)
Adds the channel to a group. If the group doesn't exist then it is created. A subscription request is also sent to the server to ensure that messages are delivered locally.

Parameters

- **group** (*bytes*) – Name of the group to join.
- **channel** (*bytes*) – Channel joining the group.

async _group_discard(*group_name: bytes*, *channel_name: bytes*) → None
Removes a channel from a group

Parameters

- **group** (*bytes*) – Group name
- **channel** (*bytes*) – Channel name

_receive_data(*multipart_message: Union[_asyncio.Future, List]*) → None
Callback that receives data messages from the server and dispatches them to the relevant handler method.

Parameters **multipart_message** (*Union[Future, List]*) – zmq multipart message

Raises

- **ImportError** – Error raised if the multitpart message cannot be parsed to a valid data message.
- ***MessageCommandUnknown*** – The command in the message is unknown.

_delivery(*message:* django_message_broker.server.data_message.DataMessage) → None
Receive a message for delivery to subscribers of a channel and pushes it onto a message queue for later collection by the client method.

Parameters **message** (DataMessage) – Data message.

_data_task_complete(*message:* django_message_broker.server.data_message.DataMessage) → None
Response from server indicating that the data command completed successfully.

Called when a data message confirms that actions have completed. This then sets an event to release the data send thread to execute subsequent instructions.

Parameters **message** (SignallingMessage) – Signalling message

_data_task_exception(*message:* django_message_broker.server.data_message.DataMessage) → None
Response from server indicating that the data command generated a caught exception.

Parameters **message** (SignallingMessage) – Signalling message

_receive_signalling(*multipart_message: Union[_asyncio.Future, List]*) → None
Callback that receives signalling messages from the server and dispatches them to the relevant handler method.

Parameters **multipart_message** (*Union[Future, List]*) – zmq multipart message

Raises

- **ImportError** – Error raised if the multitpart message cannot be parsed to a valid data message.
- ***MessageCommandUnknown*** – The command in the message is unknown.

_signalling_task_complete(*message:* django_message_broker.server.signalling_message.SignallingMessage) → None
Response from server indicating that the signalling command completed successfully.

Called when a signalling message confirms that actions have completed. This then sets an event to release the signalling send thread to execute subsequent instructions.

Parameters **message** (SignallingMessage) – Signalling message

_signalling_task_exception(*message:* django_message_broker.server.signalling_message.SignallingMessage) → None
Response from server indicating that the signalling command generated a caught exception.

Parameters **message** (SignallingMessage) – Signalling message

_flush_messages(*message:* django_message_broker.server.signalling_message.SignallingMessage) → None
Response from server indicating that the client should flush message which have not yet been received.

This method does not implement any actions. It is intended that when a client sends a flush all message to the server, the server will send a flush messages command to all clients to remove any undelivered messages from the clients.

Parameters **message** (SignallingMessage) – Signalling message

async _flush_all()
Resets the server by flushing all messages from the message store, and groups from the group store.

Channels Server

Implements the Django Message Broker Server.

class django_message_broker.server.channels_server.**ChannelsServer**(*ip_address: str = '127.0.0.1'*, *port: int = 5556*)

Message server for Django Channels.

Implements a network message server for Django channels using zero message queue (ZMQ) to support communication between channels registered within different processes.

The server opens two sequential ports:

- Data port (base port, default=5556): Transmission of data messages between the client and server.
- Signalling port (base port + 1, default=5557): Transmission of signalling messages between the client and server.

class DataCommands
Create registry of data commands using decorators.

class SignallingCommands
Create registry of signalling commands using decorators.

start() → None
Start the channels server.

Raises *ChannelsServerError* – If the server cannot be started.

stop() → None
Stop the channels server.

_receive_data(*multipart_message: Union[_asyncio.Future, List]*) → None
Callback raised when a message is received on the data port. Parses the message to identify the relevant command and dispatches the messages to the relevant command handler.

If the "ack" property is set then the server will send a response to the client. There are three potential responses:

- COMPLETE: The command completed successfully.
- EXCEPTION: An exception occurred in the server.
- COMMAND SPECIFIC: A specific response to the command generated by the server.

Parameters **multipart_message** (*Union[Future, List]*) – Multipart message received on data port.

_subscribe(*message:* django_message_broker.server.data_message.DataMessage) → None
Command handler for SUBCHANX: Subscribe to a channel.

Registers an endpoint as a subscriber to a channel. If the channel doesn't exist then it creates a new channel.

Parameters **message** (DataMessage) – Message received requesting subscription to the channel.

_unsubscribe(*message:* django_message_broker.server.data_message.DataMessage) → None
Command handler for USUBCHAN: Unsubscribe to a channel.

Removes an endpoint from a channel.

Parameters **message** (DataMessage) – Message received requesting subscription to the channel.

_send_to_channel(*message:* django_message_broker.server.data_message.DataMessage) → None
Command handler for SENDCHAN: Send message to a channel.

Creates a new queue if one does not exist. Adds the message to the queue.

If the "ttl" property is set on the message then set the expiry time of the message.

Parameters **message** (DataMessage) – Received data message.

_send_to_group(*message:* django_message_broker.server.data_message.DataMessage) → None
Command handler for SENDGRPX: Send message to a group.

Sends a message to a group of channels. The method creates a quick copy of the data message and pushes it to each channel. The shallow copy share the body of the message between all copies to reduce encoding/decoding times and memory utilisation.

Parameters **message** (DataMessage) – Received data message.

_receive_signalling(*multipart_message: Union[_asyncio.Future, List]*) → None
Callback raised when a message is received on the signalling port. Parses the message to identify the relevant command and dispatches the messages to the relevant command handler.

If the "ack" property is set then the server will send a response to the client. There are three potential responses:

- COMPLETE: The command completed successfully.
- EXCEPTION: An exception occurred in the server.
- COMMAND SPECIFIC: A specific response to the command generated by the server.

Parameters **multipart_message** (*Union[Future, List]*) – Multipart message received on data port.

Raises

- **ImportError** – Error raised if the multitpart message cannot be parsed to a valid data message.
- ***MessageCommandUnknown*** – The command in the message is unknown.

_group_add(*message:* django_message_broker.server.signalling_message.SignallingMessage) → None
Command handler for GROUPADD: Add channel to group.

Adds a channel to a group. Creates the group if it does not exist.

If the ttl property is set then sets the expiry time of the group (default is 1 day)

Parameters **message** (DataMessage) – Received data message.

_group_discard(*message:* django_message_broker.server.signalling_message.SignallingMessage) → None
Command handler for GROUPDIS: Remove channel from group.

Removes a channel from a group.

Parameters **message** (DataMessage) – Received data message.

_flush_all(*_: django_message_broker.server.signalling_message.SignallingMessage*) → None
Command handler for FLUSHXXX: Reset the server.

Remove all groups, subscriptions and messages from the server

Parameters **_** (SignallingMessage) – Received message (not used)

print(*: django_message_broker.server.signalling_message.SignallingMessage*) → None
Command handler for PRINTXXX: Print server contents.

Prints the contents of the message store and group store on the terminal.

Parameters **_** (SignallingMessage) – Received message (not used)

_flush_queues() → None
Periodic callback to flush queues where there are no subscribers or messages.

_flush_groups() → None
Periodic callback to flush groups that have expired.

Client Queue

Implements a message queue, used within the client for the Django Message Broker.

class django_message_broker.server.client_queue.**ClientQueue**(*channel_name: bytes = b''*, *time_to_live: Optional[float] = None*)
Client message queue.

When the client receives messages they are automatically appended to the channel queue by the receive data callback. Client functions are then able to wait on the queue until a message is available.

stop()
Stop periodic callbacks and remove future tasks from the event queue. This should be called prior to deleting the object.

_set_messages_available() → None
Set (or clear) asyncio event if there are message in the queue.

push(*message: django_message_broker.server.data_message.DataMessage*) → None
Pushes a message onto the key.

The message may arrive with an expiry time already set, this time should not be modified. If an expiry time has not been set, and a time to live is defined, then the expiry time for the message is determined and appended to the message.

Parameters **message** (DataMessage) – Data message.

async pull() → Optional[*django_message_broker.server.data_message.DataMessage*]
Pull a message from the queue.

Waits for a message to become available and then returns the first message received by the queue.

If the queue is terminated without a message (because it has been flushed due to inactivity) then return None.

Returns Data message

Return type Optional[*DataMessage*]

property is_empty: bool
Returns true if the queue is empty

Returns Returns True if the queue is empty.

Return type bool

property clients_waiting: bool
Returns true if there are clients waiting to receive a message.

Returns True if clients waiting for a message.

Return type bool

property can_be_flushed: bool
True if the channel can be flushed from the message store.

A channel can be flushed when there are no messages and no subscribers to the queue, and the expiry date has been passed.

Returns True if the channel can be flushed.

Return type bool

_flush_messages() → None
Remove expired messages from the queue.

Periodic callback to remove expired messages from the queue.

Data Message

Implements a data message as a structured dataclass. Used to pass data messages between the Django Message Broker clients and server.

class django_message_broker.server.data_message.**DataMessageCommands**
Valid message commands for data messages.

Client to server:

- SUBSCRIBE (b"SUBCHANX") - Subscribe to a channel.
- UNSUBSCRIBE (b"USUBCHAN") - Unsubscribe from a channel.
- SEND_TO_CHANNEL (b"SENDCHAN") - Send this message to a channel.
- SEND_TO_GROUP (b"SENDGRPX") - Send this message to a group.

Server to client:

- DELIVERY (b"DELIVERY") - Deliver this message to a client.
- COMPLETE (b"COMPLETE") - Command complete
- EXCEPTION (b"EXCEPTON") - Command raised an exception.
- SUBSCRIPTION_ERROR (b"ESUBCHAN") - Error subscribing to channel.

Both ways:

- KEEPALIVE (b"HARTBEAT") - Intermittent message confirming client/server alive.

class django_message_broker.server.data_message.**DataMessage**(*endpoints: typing.List = <factory>*, *channel_name: bytes = b'Default'*, *command: bytes = b'XXXXXXXX'*, *id: bytes = <factory>*, *sequence: int = 0*, *properties: typing.Dict = <factory>*, *encoded_body: typing.Optional[bytes] = b'{}'*, *body: typing.Optional[typing.Dict] = None*)

Message is formatted on wire as n + 1 + 6 frames. Where n is the number of routing_ids added to the front of a message on zmq.DEALER to zmq.ROUTER messages; 1 is the null frame to separate routing_ids from the message; and 6 is the number of frames in the message.

- frame -1:-n: Routing IDs (list of zmq.DEALER identities)
- frame 0: b"" blank frame
- frame 1: channel_name - Name of channel or group to which message is sent
- frame 2: command - Message type (publish to channel, to group, subscribe to channel)
- frame 3: id - Unique universal identifier identifying messages in the same sequence of communication.
- frame 4: sequence - Sequence number of this message from this publisher.
- frame 5: properties - Properties appended to message
- frame 6: body (serialisable object) - Body of message (as dictionary) Maximum size 1MB.

__getitem__(*key: Union[int, str]*) → Any

Returns values stored in the properties frame of the message.

Parameters **key** (*Union[int, str]*) – Property key

Returns Value stored in the property.

Return type Optional[Any]

Raises **KeyError** – If the key doesn't exist.

get(*key: Union[int, str]*, *default: Optional[Any] = None*) → Optional[Any]

Returns values stored in the properties frame of the message.

Parameters

- **key** (*Union[int, str]*) – Property key.
- **default** (*Any, optional*) – Default value if key doesn't exist.

Returns Property value if key exists, default or None if it doesn't.

Return type Optional[Any]

__setitem__(*key: Union[int, str]*, *value: Any*) → None

Sets the value of a property in the properties frame of the message.

Parameters

- **key** (*Union[int, str]*) – Property key
- **value** (*Any*) – Value of the property (must be serialisable by msgspec)

get_body() → Dict

Returns the value of the message, unpacking the message if the body is encoded.

Note: The default is NOT to decode the body of the message when it is received. This reduces the number of times the body (which can be up to 1MB in size) is decoded/encoded when traversing servers and routers.

Accessing the body of the message using the message attribute is potentially unsafe if the body of the message has not be decoded. Therefore, accessing the body should be using the get_body() method.

Returns Body of the message.

Return type Dict

__repr__() → str
Returns a printable string of data message contents.

Returns:

- Endpoint list
- Channel name
- Unique message id
- Command
- Size of data body
- Properties
- Data body

Returns Data message contents.

Return type str

quick_copy() → *django_message_broker.server.data_message.DataMessage*
Returns a new instance of the message with all new envelope items and shared (between instances) body object. The purpose of providing a new envelope with shared body is to support copying the message for transmission to multiple addressees where the envelope must change but the body of the message does not. Sharing the body object reduces copying time, memory utilisation.

Returns Copy of the message with shared body object.

Return type *DataMessage*

copy() → *django_message_broker.server.data_message.DataMessage*
Returns a new instance of the message deepcopying all objects. This returns an copy of the message which is independent of the original message.

Returns [description]

Return type *DataMessage*

async send(*socket: Union[zmq.sugar.socket.Socket, zmq.eventloop.zmqstream.ZMQStream]*) → None
Sends a message on the defined socket or stream.

Parameters **socket** (*Union[zmq.Socket, ZMQStream]*) – Socket or stream on which message sent.

Raises

- ***MessageTooLarge*** – The body of the message exceeds the maximum permissible.
- ***ChannelsSocketClosed*** – Socket or stream closed.
- ***MessageFormatException*** – Error occurred sending the message.

classmethod recv(*socket: zmq.sugar.socket.Socket*) → *django_message_broker.server.data_message.DataMessage*
Reads message from socket, returns new message instance.

classmethod from_msg(*multipart_message: Union[_asyncio.Future, List]*, *unpack_body=True*) → *django_message_broker.server.data_message.DataMessage*

Returns a new message instance from zmq multipart message frames.

The body of the message is decoded by default, though this can be over-ridden if the body does not need decoded. This may be advantageous where the message is being stored and forwarded to the final destination (e.g. in a server or router).

Parameters

- **multipart_message** (*Union[Future, List]*) – A list (or list in a Future) of zmq multipart frames.
- **unpack_body** (*bool, optional*) – Decode the body. Defaults to True.

Raises *MessageFormatException* – If the number of frames in the multipart message doesn't match the schema.

Returns New instance of the message.

Return type *DataMessage*

Exceptions

Django Message Broker custom exceptions.

exception django_message_broker.server.exceptions.**MessageFormatException**
The message format is incorrect.

exception django_message_broker.server.exceptions.**MessageCommandUnknown**
The command in the received message is unknown.

exception django_message_broker.server.exceptions.**MessageTooLarge**
The messsage is too large.

exception django_message_broker.server.exceptions.**SubscriptionError**
Error whilst subscribing to channel

exception django_message_broker.server.exceptions.**ChannelsServerError**
General server exception.

exception django_message_broker.server.exceptions.**ChannelQueueFull**
Channel queue is full.

exception django_message_broker.server.exceptions.**ChannelFlushed**
Channel was flushed from the message broker and no longer exists.

exception django_message_broker.server.exceptions.**ChannelsSocketClosed**
Exception raised when message pushed to closed socket.

Server Queue

Implements a message queue within the Django Message Broker server.

Includes:

- Endpoint - A class containing zero message queue routing back to the subscriber's client.
- RoundRobinDict - Implements a dictionary of messages over which the channel queue iterates, returning to the first item in the list when the end is reached.
- ChannelQueue - Implements a channel queue within the server.

class django_message_broker.server.server_queue.**Endpoint**(*socket: typing.Union[zmq.sugar.socket.Socket, zmq.eventloop.zmqstream.ZMQStream], dealer: typing.List = <factory>, subscriber_name: bytes = b'', is_process_channel: bool = False, time_to_live: typing.Optional[int] = 86400, expiry: typing.Optional[datetime.datetime] = None*)

Provides routing information for subscribers

socket
zmq.ROUTER socket or stream to which messages are sent.

Type zmq.Socket, ZMQStream

dealer
Identity of zmq.DEALER to which messages are sent.

Type bytes

is_process_channel
The channel is a process channel (can have only one delivery endpoint).

Type bool

time_to_live
Number of seconds before the endpoint is discarded.

Type int

expiry
Time when the endpoint will expire

Type Optional[datetime]

property id: FrozenSet
Returns a unique endpoint id from the routing information and channel name.

Returns Unique endpoint id.

Return type FrozenSet

class django_message_broker.server.server_queue.**RoundRobinDict**(**args*, ***kwargs*)

Dictionary of endpoints that returns the next endpoint when a next value is requested and continues from the start of the list when the end of the list is reached.

Notes:

1. Returns the value of the dictionary entry, not the key
2. StopIteration exception is not raised when the dictionary rolls over to the start.

__next__() → Optional[*django_message_broker.server.server_queue.Endpoint*]
Gets the next value of the next entry in the dictionary, returns to the first entry once the end of the dictionary is reached. Values are returned by insertion order.

Returns Value of the next dictionary entry.

Return type Optional[*Endpoint*]

first_key() → Optional[Any]
Returns the first key in the dictionary by insertion order.

Returns Value of the first key in the dictionary.

Return type Optional[int]

class django_message_broker.server.server_queue.**ChannelQueue**(*channel_name: bytes = b''*, *max_length: Optional[int] = None*)

Provides a managed queue for storing and forwarding messages to endpoints. Messages are pulled from the queue and delivered to endpoints according to a round robin algorithm. The ordering of the round robin list is updated each time endpoints subscribe to the queue, with most recently updated channels going to the end of the list.

Channel queues may be identified as Process Channels by the inclusion of an '!' within the channel name. These channels only have one endpoint. Attempts to change the endpoint (channel_name or dealer_id) or add additional endpoints result in a Subscription Exception.

Once initialised, the queue starts an event loop which monitors the queue and sends messages in the queue once there are subscribers.

Two additional callbacks are created, one to periodically scan unsent messages in the queue and remove those messages which are unsent after their expiry time. The second periodically scans subscribers and removes those who have passed their expiry time.

Raises ***SubscriptionError*** – Attempt to add multiple endpoints or change the endpoint of a Process Channel

stop()

Stop periodic callbacks and remove future tasks from the event queue. This should be called prior to deleting the object.

_set_subscribers_available() → None

Set (or clear) asyncio event if there are subscribers to the queue.

_set_messages_available() → None

Set (or clear) asyncio event if there are messages in the queue.

is_empty() → bool

Returns true if there are no subscribers and no messages.

Returns True if the queue is empty and has no subscribers.

Return type bool

async event_loop() → None

Queue event loop. Waits for subscribers, and if there are message in the queue pull them from the queue and send them to the subscriber.

subscribe(*endpoint:* django_message_broker.server.server_queue.Endpoint) → None

Add subscribers to the channel. Subscribers are identified by the dealer that transmits the request. Multiple endpoints may be registered for normal channels (without the ! type indicator), only one endpoint may be registered for a process channel (indicated by the ! type indicator).

Parameters **endpoint** (Endpoint) – Endpoint defining the receiver.

Raises ***SubscriptionError*** – Attempt to change or add multiple endpoints for a Process Channel

push(*message:* django_message_broker.server.data_message.DataMessage, *time_to_live: Optional[float] = 60*)

Pushes a message onto the message queue.

Parameters

- **message** (DataMessage) – Message to send to channel

- **time_to_live** (*float*) – Number of seconds until message expires

async pull_and_send() → None
Pulls a message from the queue and sends to an endpoints. Endpoints are chosen based upon a round-robin algorithm. If the mesage cannot be sent then it is add back to the end of the queue and the queue is penalised 100mSec to reduce rate of outbound messaging.

discard(*endpoint:* django_message_broker.server.server_queue.Endpoint)
Discards an endpoint from the queue.

Parameters **endpoint** (Endpoint) – id of the dealer to discard.

_flush_messages() → None
Flush expired messsages from the queue.

_flush_subscribers() → None
Flush expired subscribers from the queue.

Signalling Message

Implements a signalling message as a structured dataclass. Used to pass signaling message between the Django Message Broker clients and server.

class django_message_broker.server.signalling_message.**SignallingMessageCommands**
Valid message commands for signalling messages.

Client to server

- GROUP_ADD (b"GROUPADD") - Add a channel to a group.
- GROUP_DISCARD (b"GROUPDIS") - Remove a channel from a group.
- FLUSH (b"FLUSHXXX") - Remove all groups, subscriptions and messages from the server.
- PRINT (b"PRINTXXX") - Print contents of message store and group store to terminal.
- PERFORMANCE (b"PERFMNCE") - Request performance report

Server to client

- COMPLETE (b"COMPLETE") - Command complete
- EXCEPTION (b"EXCEPTON") - Command raised an exception.
- PERFORMANCE_REPORT (b"PERFRPRT") - Transmit performance report
- FLUSH_CLIENT (b"FLUSHCXX") - Flush client message store.

Both ways

- KEEPALIVE (b"HARTBEAT") - Intermittent message confirming client/server alive.

class django_message_broker.server.signalling_message.**SignallingMessage**(*endpoints: typing.List = <factory>, id: bytes = <factory>, command: bytes = b'XXXXXXXX', properties: typing.Dict = <factory>*)
Message is formatted on wire as n + 1 + 3 frames. Where n is the number of routing_ids added to the front of a message on zmq.DEALER to zmq.ROUTER messages; 1 is the null frame to separate routing_ids from the message; and 3 is the number of frames in the message.

- frame -1:-n: Routing IDs (list of zmq.DEALER identities)
- frame 0: b"" blank frame
- frame 1: id - Unique universal identifier for tracking commands
- frame 2: command - Message type (publish to channel, to group, subscribe to channel)
- frame 3: properties - Properties appended to message

__getitem__(*key: Union[int, str]*) → Any
Returns values stored in the properties frame of the message.

Parameters **key** (*Union[int, str]*) – Property key

Returns Value stored in the property.

Return type Optional[Any]

Raises **KeyError** – If the key doesn't exist.

get(*key: Union[int, str]*, *default: Optional[Any] = None*) → Optional[Any]
Returns values stored in the properties frame of the message.

Parameters

- **key** (*Union[int, str]*) – Property key.
- **default** (*Any, optional*) – Default value if key doesn't exist.

Returns Property value if key exists, default or None if it doesn't.

Return type Optional[Any]

__setitem__(*key: Union[int, str]*, *value: Any*) → None
Sets the value of a property in the properties frame of the message.

Parameters

- **key** (*Union[int, str]*) – Property key
- **value** (*Any*) – Value of the property (must be serialisable by msgspec)

__repr__() → str
Returns a printable string of signalling message contents.

Returns:

- Endpoint list
- Unique message id
- Command
- Properties

Returns Signalling message contents.

Return type str

async send(*socket: Union[zmq.sugar.socket.Socket, zmq.eventloop.zmqstream.ZMQStream]*) → None
Sends a message on the defined socket or stream.

Parameters **socket** (*Union[zmq.Socket, ZMQStream]*) – Socket or stream on which message sent.

Raises

- *MessageTooLarge* – The body of the message exceeds the maximum permissible.
- *ChannelsSocketClosed* – Socket or stream closed.
- *MessageFormatException* – Error occurred sending the message.

classmethod recv(*socket: zmq.sugar.socket.Socket*) → *django_message_broker.server.signalling_message.SignallingMessage*

Reads message from socket, returns new message instance.

classmethod from_msg(*multipart_message: Union[_asyncio.Future, List]*) → *django_message_broker.server.signalling_message.SignallingMessage*

Returns a new message instance from zmq multipart message frames.

Args: multipart_message (Union[Future, List]): A list (or list in a Future) of zmq multipart frames.

Raises *MessageFormatException* – If the number of frames in the multipart message doesn't match the schema.

Returns New instance of the message.

Return type *DataMessage*

Socket Manager

Wrapper around Zero Message Queue to manage sockets.

class django_message_broker.server.socket_manager.**SocketManager**(*context: Optional[zmq.sugar.context.Context] = None*, *bind_ip_address: str = '127.0.0.1'*, *port: int = 5555*, *is_server: bool = False*, *io_loop: Optional[tornado.ioloop.IOLoop] = None*)

Creates a zero-messsage-queue (ZMQ) socket.

get_socket() → zmq.sugar.socket.Socket

Get the ZMQ socket.

Raises *ChannelsServerError* – Raises error if there is no socket.

Returns ZMQ socket.

Return type zmq.Socket

get_stream() → zmq.eventloop.zmqstream.ZMQStream

Get the ZMQ stream.

Raises *ChannelsServerError* – Raises error if there is no stream.

Returns ZMQ stream.

Return type ZMQStream

get_routing_id() → Optional[bytes]

Returns the routing ID if the sockets is a ZMQ.Dealer (client socket).

Returns Routing ID of the socket.

Return type Optional[bytes]

start() → None
Opens the socket, sets up streams and configures callbacks.

stop() → None
Closes the socket, streams and clears configured callbacks.

set_receive_callback(*callback: Callable[[Union[_asyncio.Future, List[Any]]], None]*) → None
Sets the callback when a message is received on the socket.

Parameters **callback** (*Callable[[Union[Future, List[Any]]], None]*) – Callback method accepting Future or List

Note: The callback needs to accept a multipart_message which could be either of the following types:

- List: Multi-part message with each frame expressed as an element in the list.
- asyncio Future: An asyncio Future containing the above list.

The callback function should test whether the returned multipart message is a Future and then extract the multipart list as follow:

```
from asyncio.futures import Future

def callback(multipart_message: Union[Future, List]):
    if isinstance(multipart_message, Future):
        multipart_list = multipart_message.result()
    else:
        multipart_list = multipart_message
```

clear_receive_callback() → None
Clear the callback that is called when a message is received.

Utility Functions

Additional functionality used within the Django Message Broker:

- IntegerSequence - Iterator returning the next integer when accessed.
- WeakPeriodicCallback - Modifies Tornado Periodic Callback with weak method references.
- MethodProxy - Creates a register of weak references to methods.
- WaitFor - Asyncio method to wait upon multiple coroutines or events.
- MethodRegistry - Implements a decorator in a class to auto-create a registor of methods.

class django_message_broker.server.utils.**IntegerSequence**
Generates iterators of integer sequences.

new_iterator(*start: int = 0*) → Iterator[int]
Returns an iterator which generates the next integer in a sequence when called.

Parameters **start** (*int, optional*) – Starting number of the sequence. Defaults to 0.

Yields *Iterator[int]* – Iterator generating the next integer in a sequence.

class django_message_broker.server.utils.**WeakPeriodicCallback**(*callback: Callable[[], Optional[Awaitable]], *args, **kwargs*)
Implementation of Tornado PeriodCallback establishing a weak reference to the callable function.

A weak reference is required when the periodic callback is established in the initialization of a class to a callback method within the same class. In this situation the periodic callback holds a reference to an object within the class and the class can neither be deleted or garbage collected. This can lead to a memory leak. Establishing a weak link between the periodic callback and the callback method allows the callback to be removed from the event loop and the class deleted.

async _run() → None
Method called after the time that future is scheduled to run.

class django_message_broker.server.utils.**MethodProxy**
Creates a registry to proxy strong function references with weak ones.

classmethod register(*callback: Callable*) → Callable
Registers a callback in a registry of callable methods and returns a callback with a weakref to the original callback method.

Parameters **callback** (*Callable*) – Original callback method.

Raises **ReferenceError** – If the referenced callable has already been garbage the terminates future and propagates into enclosing waiting code.

Returns Weakref callable to the original callback method.

Return type Callable

classmethod unregister(*callback: Callable*) → None
Removes a callable method from the registry.

Parameters **callback** (*Callable*) – Callable method to unregister

class django_message_broker.server.utils.**WaitFor**(*events: List[asyncio.locks.Event]*)
Waits for a combination of events to occur before continuing code execution.

The code creates weak referenced callbacks which are triggered when an event in the list is set. The class exposes two asyncio methods:

- one_event() - Which is triggered when any one of the events is set.
- all_events() - Which is triggered when all the events are set.

_callback(*task: _asyncio.Task*) → None
Callback when an event is set. The callback is proxied so that the method reference is weak, enabling this class to be garbage collected before the task is complete.

Parameters **task** (*_type_*) – Asyncio callbacks are passed with the task as argument.

async all_events() → None
Waits until all events are set.

async one_event() → None
Waits until any one event is set.

__del__() → None
Remove any outstanding tasks from the event loop.

class django_message_broker.server.utils.**MethodRegistry**
Meta class plug-in adding a dictionary of callable methods indexed by command name.

The class exposes a decorator that registers the method in a dictionary indexed by a named command (as byte string). The dictionary of callable methods is created when the class is imported; however, methods can only be bound to an instance of the class when the class is instantiated.

When an instance of the class is instantiated then a copy of the dictionary containing bound callable methods can be created by calling the *get_bound_callables* method passing the instance of the class as a parameter.

Example usage:

```
class MathsByName:
    # Create a registry of math functions
    class MathFunctions(MethodRegistry):
        pass

    def __init__(self):
        # Bind methods to instance
        self.maths = MathsByName.MathFunctions.get_bound_callables(self)

    @MathFunctions.register(command=b"plusOne")
    def f1(self, a):
        return a + 1

    @MathFunctions.register(command=b"sumTwo")
    def f2(self, a, b):
        return a + b

myMaths = MathsByName()
plus_one = myMaths.maths[b"plusOne"](1)  # result = 2
sum_two = myMaths.maths[b"sumTwo"](1, 2)  # result = 3
```

classmethod register(*command: Optional[bytes] = None*) → Callable
Decorator to register a named function in the dictionary of callables.

Parameters **command** (*bytes, optional*) – Name of the command.

Raises

- **Exception** – If no command is given.
- **Exception** – If the command has already been defined.

classmethod get_bound_callables(*self*) → Dict[bytes, Callable]
Given an instance of the class, returns a dictionary of named methods.

Returns Dictionary of callable methods.

Return type Dict[bytes, Callable]

1.2.3 Messaging

Messaging between the client and server uses Zero Message Queue (ZMQ) as the transport protocol. Two communication channels are established, one to communicate signalling information and the other for data. Signalling is carried over a separate channel to minimise the chance that data messages block communication between the client and server. The data channel defaults to port 5556 and signalling to port 5557.

Note: The client-server communication is implemented using a ZMQ Dealer at the client and a ZMQ Router at the server. This allows multiple clients to connect to the server. Each message from the client has a routing ID prepended by the ZMQ Dealer which needs to be removed at the server prior to processing the message. If there are additional routing nodes between the client and the server then ZMQ may add multiple routing IDs to the message. All routing IDs are removed and recorded at the server so that messages can be routed back to the originating client, even where multi-hop ZMQ networks are used. Whilst this doesn't apply over localhost networks for which the message broker is designed, there may be future developments which allow the message broker to be deployed on a different server from the client and with additional routing nodes to support multi-server deployments.

The following signalling sequences are supported:

Data message: Send to channel

Messages sent from the client to a channel on the server. If the channel doesn't exist on the server then a channel will be created.

If the channel is full and `ack` is `True` then the server will send an exception back to the client. If `ack` is `False` then the message will be dropped silently.

The client can specify two properties:

- **ttl (float)**: Time in seconds after the server receives the message that it will be deleted. If the message is forwarded to a client from the server then the time that the message is due to expire it must be passed to the client as a property (`expiry`) with the message.
- **ack (bool)**: `True` if the server should acknowledge the message. The client must wait until the server has acknowledged the message. If ack is `False` then the server must process the message silently without reporting any exceptions to the client.

If `ack` is `true` then the send function will block the thread until a response is received from the server. If `ack` is `false` then execution will continue without waiting for a response.

Action	Client -> Server	Server -> Client
Initial action	**Message**: `b"SENDCHAN"` **Properties**: `ttl (float)`: Time to live in seconds. `ack (bool)`: Acknowledge task complete.	Add message to queue. If no queue, create queue. If `ack` = `False`: -> No response. If `ack` = `True`: -> Respond complete. -> Respond exception.
Respond complete		**Message**: `b"COMPLETE"`
Respond exception		**Message**: `b"EXCEPTON"` **Properties**: `exception (str)`: Error message

Data message: Send to group

Messages sent from the client to a group of channels on the server. If the group doesn't exist on the server or no channels have been added to the group then the message must be silently dropped.

The client can specify one property:

- `ttl (float)` : Time in seconds after the server receives the message that it will be deleted. If the message is forwarded to a client from the server then the time that the message is due to expire is must be passed to the client as a property (`expiry`) with the message.

Action	Client -> Server	Server -> Client
Initial action	**Message**: `b"SENDCHAN"` **Properties**: `ttl (float)`: Time to live in seconds.	Send message to channels in group.

Data message: Subscribe and receive

This messaging flow allows the client to subscribe to a channel and then automatically receive messages in a local subscriber queue. The application can then pull messages from this queue as required. This approach removes messages from the server and stores them locally to reduce load on the server and reduce latency when pulling messages.

The messaging flow comprises two sub-flows:

- Messaging flow for the client to subscribe to the channel
- Messaging flow for the server to forward messages to local client storage

Subscribe to channel

Send a message to the server requesting subscription to a channel. The client provides the name of the local queue set up to store and forward messages to the local subscriber.

If the `expiry` property is set within a received message then the message must be removed from the local queue once the expiry time is reached.

To support Django Channels, if the channel name includes an `!` then the channel is classed as local and can only have one subscriber.

Action	Client -> Server	Server -> Client
Initial action	**Message**: `b"SUBCHANX"` **Channel name**: Channel name on the server **Properties**: `subscriber_name (str)` = Local subscriber name.	No further action.

Forward message to client

Forward a message from a channel queue on the server to a local queue at the client. A message is sent from the server when there is at least one message in the queue and at least one client subscribed to the queue. If the channel is a local channel (name contains `!`) then only one subscriber is permitted to the channel queue at the server. If there is more than one subscriber to the queue then messages are delivered to each queue in turn using the *round robin* pattern.

Action	Server -> Client	Client -> Server
Triggered action	**Message**: `b"DELIVERY"` **Channel name**: Local subscriber name **Properties**: `expiry (DateTime)`: Time the message expires.	No further action.

Signalling message: Flush

Flushes all data from the client. Sends a message to all recently connected clients to flush channels and messages from their stores.

This message blocks the client thread until a response is received from the server.

Action	Client -> Server	Server -> Client
Initial action	**Message**: `b"FLUSHXXX"`	Remove channel from group. Respond: -> Flush client

The following message is sent from the server to all connected clients once the server message store has been flushed. The clients must reset their message store to the initial state.

Action	Server -> Client	Client -> Server
Flush client	**Message**: `b"FLUSHCXX"`	No further response.

Signalling message: Add to group

Adds a channel to a group. The channel name is the server channel name. The group may be set to expire after a certain amount of time (`ttl`) in seconds. The expiry time is reset each time a channel is added to the group. If channel is already part of the group then adding it again will not add the channel a second time, but will reset the expiry time.

This message blocks the client thread until a response is received from the server.

Action	Client -> Server	Server -> Client
Initial action	**Message**: `b"GROUPADD"` **Properties**: `group_name (bytes)`: Name of group. `channel_name (bytes)`: Name of channel. `ttl (float)`: Time until group expires.	Add channel to group. Respond: -> Complete -> Exception
Complete	**Message**: `b"COMPLETE"`	No further action.
Exception	**Message**: `b"EXCEPTON"` **Properties**: `exception (str)`: Exception description.	No further action.

Signalling message: Remove from group

Removes a channel from a group. The channel name is the server channel name.

This message blocks the client thread until a response is received from the server.

Action	Client -> Server	Server -> Client
Initial action	**Message**: `b"GROUPDIS"` **Properties**: `group_name (bytes)`: Name of group. `channel_name (bytes)`: Name of channel.	Remove channel from group. Respond: -> Complete -> Exception
Complete	**Message**: `b"COMPLETE"`	No further action.
Exception	**Message**: `b"EXCEPTON"` **Properties**: `exception (str)`: Exception description.	No further action.

1.3 Upgrading and change log

Recent versions of Django Message Broker have a corresponding git tag for each version released to pypi.

1.3.1 Changelog

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

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

[0.2.1] - 2022-03-25: Package upload to PyPi

[0.2.0] - 2022-03-25: Documentation and testing

Added

- Interrogate for documentation coverage.
- Sphinx documentation for Read The Docs.
- Add test cases

Changed

- Nothing

[0.1.0] - 2021-12-18: Initial commit

Added

- Initial code prior to packaging for PyPi.
- Interface for django channels.
- Interface for background worker.

Changed

- Nothing

PYTHON MODULE INDEX

d

INDEX

Symbols