rohrpost Documentation

AX semantics

Aug 31, 2018

Contents:

1 Protocol 3

2 Installation 5

3 Routing 7

4 Adding Handlers 9

5 Using the mixins 11

6 Utility functions 13

Welcome to the rohrpost documentation!

rohrpost is a small library that aims to make protocol development for Django's sub-project channels (which provides WebSockets capabilities) easy and fun. We also have a client library called rohrpost.js.

It features a light weight, JSON based protocol, including an exemplary handler implementing a ping/pong method. It also comes with a variety of helper functions, and Django model mixins that allow to automatically send updates when any user updates, deletes, or creates an object.

CHAPTER 1

Protocol

The rohrpost protocol sits on top of channels inside the `text` component of a channels message. rohrpost expects this `text` component to be valid JSON with

- An `id` field that rohrpost sends back in the response.
- A `type` field that contains a string defining the message type (and hence, which handler will process the message).
- An optional `data` field containing whatever data rohrpost should pass to the handler. Please note that rohrpost will pass all request data to the handler, and the naming of the `data` field is convention, not a rule.

A typical message would look like this:

```
{
    "id": 1234,
    "type": "ping",
    "data": [1, 2, 3, 4]
}
```

Both server and client messages look like this. When the server receives a message, it will hand off the message to the appropriate handler named in the `type` field. If there is no such handler or the handler fails, the server responds with a message containing at least an `error` field, and optionally other data in the `data` field.

```
{
    "id": 1234,
    "error": "No registered handler called 'ping'."
}
```

CHAPTER 2

Installation

From the command line:

```
pip install rohrpost
```

Or add this line to your *requirements.txt*:

```
rohrpost==1.x
```

CHAPTER 3

Routing

Once you have installed *rohrpost*, you'll need to add the main *rohrpost* handler to your *routing.py*. You can find details on this in Channels' routing documentation.

```
from channels import route
from rohrpost.main import handle_rohrpost_message

channel_routing = [
    route('websocket.receive', handle_rohrpost_message, path=r'/rohrpost/$'),
]
```

CHAPTER 4

Adding Handlers

rohrpost provides a decorator `rohrpost_handler`, that accepts both a string and a list to register a method as the handler for incoming messages. This is how the ping method works, that rohrpost provides out of the box:

```
from rohrpost.message import send_message
from rohrpost.registry import rohrpost_handler


@rohrpost_handler('ping')
def handle_ping(message, request):
 response_kwargs = {
     'message': message,
     'message_id': request['id'],
     'handler': 'pong'
 }
 if 'data' in request:
     response_kwargs['data'] = request['data']
 send_message(**response_kwargs)
```

CHAPTER 5

Using the mixins

We have four relevant Django model mixins in `rohrpost.mixins`: `NotifyOnCreate`, `NotifyOnUpdate`, `NotifyOnDelete` and `NotifyOnChange` which inherits from the previous three classes.

When using these mixins, you'll need to set some fields or fill in some methods:

- `get_group_name(self, message_type)` or `group_name`, with the method having preference over the attribute. This method or attribute should return a string that denotes the group receiving the message. All users in that group will receive a message. This gives you the possibility to build per-object, per-class or global groups. If neither the message nor the attribute are present, the group name is the lower cased class name combined with the object's ID: f'{object.__class__.__name__.lower()}-{object.pk}'
- `get_push_notification_data(self, updated_fields, message_type)` returning a dictionary (or any data structure) containing the data you wish to send to the client. rohrpost will update the serialized object updated with an `updated_fields` attribute with a list *if* you do not set `updated_fields` in `get_push_notification_data()` *and* if you set it when calling the `save()` method that lead to the notification. The fallback value if you do not fill in this method is `{"id": obj.id}`.

The message will look like this:

```
{
    "id": <some id>,
    "type": "subscription-update",
    "data": {
        "type": <create|update|delete>,
        "group": <group-name>,
        "object": <serialized object>,
    }
}
```

You will have to put users who should receive these notifications in the group specified by `get_group_name` or `group_name`. Since authentication and registration works differently in every use case, rohrpost does not include a standard handler for this.

CHAPTER 6

Utility functions

rohrpost provides three main helper functions for message sending in `rohrpost.message`:

- `rohrpost.message.send_message`
  - `message`: The original message you are replying to (**required**).
  - `handler`: The string identifying your handler (**required**).
  - `message_id`: The message ID (any simple datatype allowed). If you do not provide any, an integer will be randomly chosen.
  - `close`: Set to `True` if you want to close the connection.
  - `error`: Include an error message or error content
  - `data`: A dict that will appear in the message as `data` (converted to a JSON object).
  - `**additional_data`: Any other keyword argument will appear in the message's `data` field as a JSON object. This is deprecated.
- `rohrpost.message.send_error` sends an error message explicitly, takes the same arguments as `send_message`.