Technical Report

WebSocket Integration in Django

Raciti Mario
Vitello Fabio Roberto

Abstract

Nowadays Web technologies have become more common as they improve the work of astronomers by easing, for example, the monitoring and analysing of data. The Django Python framework is one of the most widely used libraries for developing Web applications as it offers several advantages. However, the necessity of continuously deal with data in real time, such as tracking atmospheric parameters, analysing the evolution of the light curve during a transient event, displaying inline vector graphics for interactive plots and representation, has constantly grown in Astronomy and Astrophysics, and this has naturally involved in new challenges. Nevertheless the WebSocket protocol represents the best option to manage real-time data, but it is not supported by Django natively.
This report provides an overview of the WebSocket protocol and advances the integration of a WebSocket server as a loosely coupled service within a Django application by illustrating a simple and non-invasive methodology, within a proof-of-concept using open source software, which avoid switching to new deployment architectures, with all its consequences. Such proposed technique can be applied to any generic scenarios, such as done for the TMSS project included in the report as use case example.

Table of Contents

1 Introduction **3**

2 WebSocket Protocol Overview **3**

2.1 WebSocket Connections 4

2.2 WebSocket Messages 5

2.3 WebSocket Security 5

3 WebSocket State-of-the-Art **6**

4 WebSocket Integrations in Django **7**

4.1 Software Architecture 7

4.2 Implementation Details 7

4.2.1 UpdateSignal 8

4.2.2 WebSocketHandler 8

4.2.3 WSServerWrapper 8

4.2.4 WebsocketIntegrationConfig 8

5 Use Case: TMSS **11**

6 Conclusions **13**

References **13**

1. Introduction

Real-time Web applications have constantly grown during last years as of the increasing necessity of dealing with streams of data constantly updating, especially for sensor data in the IoT (Internet of Things) field. Furthermore, it is also common in Astronomy and Astrophysics the necessity of continuously monitoring and/or analyse data in real time, for example, to track atmospheric parameters and telemetry data, analyse the evolution of the light curve during a transient event, monitor antennas configurations, display inline vector graphics for interactive plots and representation, etc.
Traditionally, the approach to build Web applications that demand real-time communication between client and server has required an excessive use of the HTTP protocol[1] to continuously poll the server to fetch updates and send upstream data via distinct HTTP calls. In such protocol the client sends a request and the server returns a response. Typically, this response occurs immediately and the transaction is complete. Even if the network connection stays open, this will be used for a separate transaction of a request and a response. Therefore, a continuous polling involves in issues related to the nature of real-time data, i.e., it is not predictable when an event will occur thereby there will be unnecessary HTTP requests flooding the network. In addition, a Web server could also potentially run out of request threads and end up discarding further requests.

2. WebSocket Protocol Overview

The WebSocket Protocol[2], defined in RFC 6455, enables full-duplex communication channels between a client and a server over a single TCP connection. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of the WebSocket is to provide a mechanism for browser-based applications that need a two-way communication with servers that does not rely on opening multiple HTTP connections. This technique is particularly useful in situations where low-latency or server-initiated messages are required, and it can be adopted for a variety of Web applications, such as multiplayer games, stock tickers, shared document editing, instant messaging, real-time analytics, push notifications, et cetera.
Basically the protocol works as follow: once the client and server have both sent their handshakes, and the procedure was successful, then the data transfer part starts. The method results in a two-way communication channel where each side can, independently from the other, send data at will. After a successful handshake, clients and servers transfer data back and forth in conceptual units referred to in the WebSocket specification as ”messages”. On the wire, a message is composed of one or more frames. The WebSocket message does not necessarily correspond to a particular network layer framing, as a fragmented message may be coalesced or split by an intermediary.
The WebSocket protocol represent one of the most suitable solutions to overcome the issues illustrated in Sec. 1: in fact, its implementations are commonly used in modern Web applications for streaming data and other asynchronous traffic.

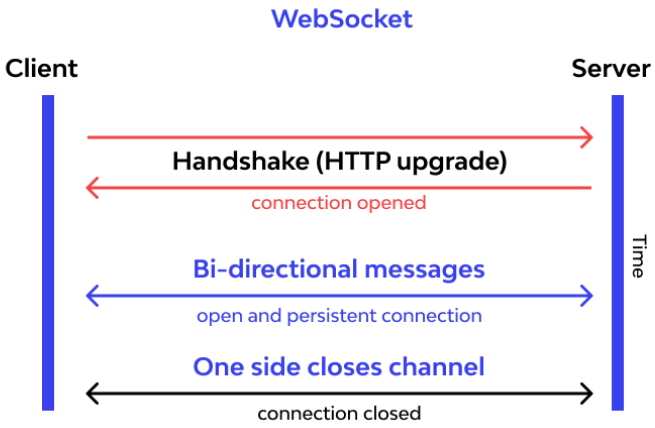

Fig. 1. WebSocket Protocol

2.1 WebSocket Connections

WebSocket connections are typically long-lived as they normally stay open and idle until either the client or the server is ready to send a message. Supposing a WebSocket server is properly configured and listening for incoming connections, a WebSocket client can normally use client-side JavaScript to create a connection to the server, as follows:

```
let ws = new WebSocket("wss://my-website.com/chat");
```

The connection is established by the client and the server performing a WebSocket handshake over HTTP. The client issues a WebSocket handshake request like the following:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

On success, the server then returns a WebSocket handshake response as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

At the end of the handshake, the network connection remains open and can be used to send WebSocket messages in either direction until one party sends a close frame.

2.2 WebSocket Messages

Once a WebSocket connection has been established upon a successful handshake, the client and server can send messages asynchronously in either direction, as messages are not transactional in nature. In principle, WebSocket messages can contain UTF-8 strings[3] or binary data and, to this regard, in modern applications it is very common for JSON[4] to be used to send structured data – or BSON (Binary JSON) when data serialisation is involved – within WebSocket messages.
Below two examples of simple messages sent from the browser using client-side JavaScript:

```
// Send message as a string
const msg = "Observation succeeded!";
ws.send(msg);

// Send message as a JSON object
const data = {"status": "ready", "data": {"parameters": ["x1", "x2"]}};
ws.send(data);
```

2.3 WebSocket Security

By default, the employment of the WebSocket protocol may expose an application to several risks[5]: in fact the protocol does not consider some security implications by design, therefore some measures need to be adopted in order to avoid undesired incidents. In this section we provide a concise overview regarding WebSocket security and best-practices.
In the first place, it is extremely important to properly validate client inputs and server data to avoid potential XSS (Cross-Site-Scripting) attacks et similia, such as SQL injection or XML external entity injection. Futhermore similarly to HTTP, it is recommended to add a layer of encryption by using the TLS protocol thus opting for the WSS (WebSocket Secure) protocol instead of the regular WS – this ensures the communication is encrypted and also data integrity can be guaranteed. However, this action does not imply an authentication method which, together with authorisation, is inherently completely lacking in the WebSocket specification. It is also important to avoid relying on the Origin header field, defined from the standard, as it is essentially advisory and does not represent a proof of authentication. The preferred way to overcome this problem is to implement a ticket-based authentication mechanism. Eventually, it is also recommended to adopt the following steps: avoid tunneling, as it may allow an escalation of attacks into a complete remote breach; implement rate limiting to prevent denial of service (DoS) attacks; protect the WebSocket handshake message against cross-site request forgery (CSRF) that may come from a cross-site WebSocket hijacking attack.

3. WebSocket State-of-the-Art

In general, there is a variety of implementations of the WebSocket protocol for different programming languages. As we already disclosed in Sec. 2.1, among the Web APIs and interfaces (object types) that JavaScript natively comes with, we have the WebSocket Web APIs[6] to create connections with a WebSocket server and exchange messages with. However, the state-of-the-art tools for the WebSocket protocol also include client-side implementations in addition to server-side.
Below the most well-known and widespread used tools:

- **ws**[7] is a simple to use, blazing fast and thoroughly tested WebSocket client and server for Node.js;
- **Socket.IO**[8] is the wide-spread and most used module that enables real-time bidirectional event-based communication between clients and a server. The official implementations are written in JavaScript and Node.js, but there are several community-maintained packages which provide equivalent implementations for other languages (i.e., C++, Java, Python, Golang, Rust, etc.). It is important to highlight that this module is not a WebSocket implementation, since it adds additional metadata to each packet when using WebSocket as a transport;
- **µWebSockets**[9] is a simple to use yet thoroughly optimized, standards compliant and secure implementation of WebSockets (and HTTP) for C++11 and Node.js. It comes with built-in pub/sub support, URL routing, TLS 1.3, SNI, IPv6, permessage-deflate and is battle tested as one of the most popular implementations, reaching many millions of end-users daily;
- **Ratchet**[10] a loosely coupled PHP library providing developers with tools to create real time, bi-directional applications between clients and servers over WebSockets;
- **Django Channels**[11] wraps Django's native asynchronous view support, allowing Django projects to handle not only HTTP, but protocols that require long-running connections too - WebSockets, MQTT, chatbots, amateur radio, and more;
- **Flask-SocketIO**[12] gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application can use any of the SocketIO client libraries in Javascript, Python, C++, Java and Swift, or any other compatible client to establish a permanent connection to the server;
- **Gorilla WebSocket**[13] is a Go implementation of the WebSocket protocol.

4. WebSocket Integrations in Django

According to the state-of-the-art tools, Django Channels might seem the obvious choice for an integration of the WebSocket protocol within Django[14], as it is fully supported by the framework. However, in some cases the adoption of this module requires switching to a new deployment architecture and such modifications may involve in further issues, e.g., a project replan, code changes, and so forth. Below we describe an alternative to Channels in order to deploy a separate WebSocket server along with a Django project without revolutionising its structure. This technique is well suited when you need to add a small set of real-time features, such as a notification service, to an HTTP application. In either case, it is convenient to also use a helpful Django library, namely *Signals*[15], which facilitates "decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain senders to notify a set of receivers that some action has taken place". This simplifies the events handling and enforces the loose coupling between the different applications or, even more, between different parts of the code.
There are several WebSocket implementations in Python and this approach can be applied independently, upon the appropriate changes. To facilitate our explanation we adopt the *SimpleWebSocketServer*[16] module, which implements the WebSocket protocol in a simple and direct way without the need of any interfaces such as ASGI, WSGI or others.

4.1 Software Architecture

In order to implement a WebSocket service within Django, we firstly need a Django application. Since the procedure of creating such application is out of scope here, we assume a Django project has already been created and set up properly according to the default and standard Django architecture. The WebSocket-related code can be gathered into a single file – in our case "websocket.py" – and it is composed of two classes: *WebSocketHandler* and *WSServerWrapper*. The aim of the former is, as suggested by its name, to manage the WebSocket interations with the clients (i.e., send messages, handle received messages, etc.), whilst the wrapper class is used just to wrap and run a *SimpleWebSocketServer* specifically configured for the handler class. At this point, the integration is completed by starting the WebSocket service in a new thread. This will ensure the WebSocket server to have the proper Django context and its own execution life within the Django application.

4.2 Implementation Details

A simple and working proof-of-concept is included in the Github repository WebSocket-in-Django[17], demonstrating a scenario where the clients receive a WebSocket message whenever a specific signal is triggered. For the sake of simplicity, such signal is sent when the index view is accessed and the message that comes within the signal itself is replicated and broadcasted via WebSocket protocol to all the connected clients.
Below we discuss the details of such implementation.

4.2.1 UpdateSignal

The implementation of the custom signal, in the ”signals.py” file, and its use in the index view is quite simple, as Fig. 2 illustrates. The *update_signal* is an instance of the *Signal* class and its *send* method is invoked, with a custom message, from the index view whenever the latter is requested. Such message can be retrieved from a *@receiver* decorated function listening to *update_signal*.

4.2.2 WebSocketHandler

The class *WebSocketHandler*, as depicted in Fig. 3, extends the *WebSocket* class implementend by the module and overrides its three handling methods, respectively, for the message reception, incoming connection and closing. To keep the handler as simple as possible, it just results in an echo server which returns the message received from a client back and logs the IP address of the clients on connection/disconnection. The core part relevant for the integration with Django is exemplified by the *onUpdateSignal* method: it is a *@receiver* decorated method which listens for the *update_signal* which, in turn, triggers the execution of the method itself, resulting in calling the *sendBroadcast* method that, as suggested by its name, broadcasts a message to all the connected WebSocket clients.

4.2.3 WSServerWrapper

The class *WSServerWrapper* defines a WebSocket server instance and a *run* method which is invoked to be executed into a separated thread. As shown in Fig. 4, also an instance of the *Event*[18] class is defined and set whenever the *run* method is called. Note that if you want to switch to the WSS protocol, it suffices to switch from the *SimpleWebSocketServer* instance to the *SimpleSSLWebSocketServer* class and specify the path of the TLS certificate and private key by using the *os.path.join*[19] method (e.g., *certfile=os.path.join(BASE_DIR, 'cert.pem')* assuming the files are in the base directory of the Django project), otherwise an exception is triggered from Django.

4.2.4 WebsocketIntegrationConfig

The class *WebsocketIntegrationConfig* extends the default Django *AppConfig*[20] class and specifies the run of the WebSocket server into a separated thread, as illustrated in Fig. 5. The application configurations represents a good choice to achieve this goal, in particular its *ready* method which can be used to perform initialisation tasks – including the registering of signals – and it is called once from Django as soon as the registry is fully populated. A check is performed by using an *Event* instance that is set when the WebSocket server is actually started – if this does not happen within 10 seconds, then an exception is raised from the *ready* overridden method.

```
# signals.py

update_signal = Signal()

# views.py

def index(request):
    update_signal.send('', msg='I was wondering if after all these years you\'d like to meet')
    return HttpResponse("Hello, it\'s me...")
```

Fig. 2. Update Signal Snippet

```
13 class WebSocketHandler(WebSocket):
14
15     @staticmethod
16     def sendBroadcast(msg):
17         # Broadcast a message to connected clients
18         for ws in WSServerWrapper.ws_server.connections.values():
19             ws.sendMessage(msg)
20
21     # Signal handling methods
22
23     @staticmethod
24     @receiver(update_signal)
25     def onUpdateSignal(**kwargs):
26         logger.info('Received a signal')
27         msg = kwargs.get('msg', '')
28         # Broadcast the message received from update_signal
29         WebSocketHandler.sendBroadcast(msg)
30
31     # WebSocket handling methods
32
33     def handleMessage(self):
34         logger.info('Received msg "%s" from %s' % (self.data, self.address[0]))
35         self.sendMessage(self.data)  # Echo message back to client
36
37     def handleConnected(self):
38         logger.info('New client connected %s' % self.address[0])
39
40     def handleClose(self):
41         logger.info('Client disconnected %s' % self.address[0])
```

Fig. 3. WebSocket Handler Snippet

```
class WSServerWrapper():
    ws_started_event = Event()
    ws_server = SimpleWebSocketServer('', DEFAULT_WS_PORT, WebSocketHandler)

    @staticmethod
    def run():
        logger.info('Starting WebSocket server')
        WSServerWrapper.ws_started_event.set()
        WSServerWrapper.ws_server.serveforever()
```

Fig. 4. WebSocket Server Snippet

```
class WebsocketIntegrationConfig(AppConfig):
    name = 'websocketIntegration'

    def ready(self):
        # Implicitly connect signal handlers decorated with @receiver
        from websocketIntegration import signals

        # Start the WebSocket server in a new thread
        from websocketIntegration.websocket import WSServerWrapper
        self.t = Thread(target=WSServerWrapper.run, daemon=True)
        self.t.start()
        if not WSServerWrapper.ws_started_event.wait(10):
            raise RuntimeError("Could not start websocket server on port %s"%DEFAULT_WS_PORT)
```

Fig. 5. WebSocket Service Thread Snippet

5. Use Case: TMSS

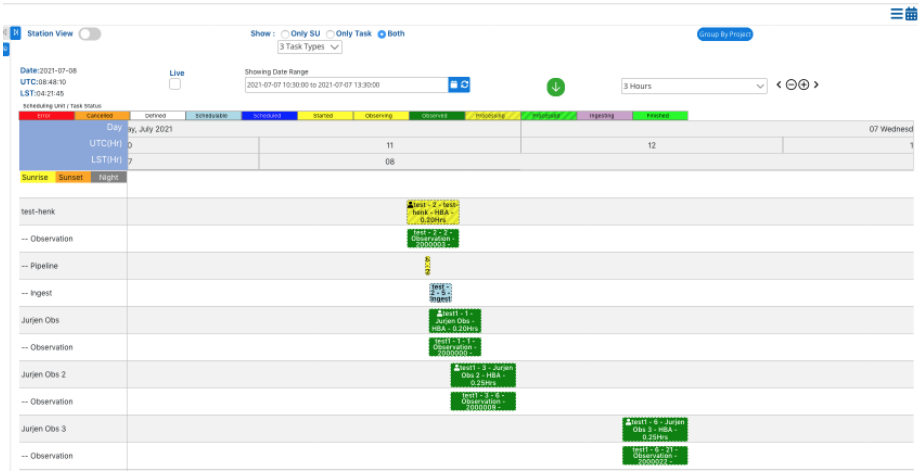

Fig. 6. TMSS WebSocket Timeline View

LOFAR (Low Frequency Array)[21] is an international telescope, designed and built by ASTRON (Netherlands Institute for Radio Astronomy), which is devoted to the global astronomical community and managed by the International LOFAR Telescope (ILT)[22] foundation. The project LOFAR 2.0 has involved, among new changes and challenges, in the design and implementation of the TMSS (Telescope Manager Specification System)[23] project, namely a new platform for specifying, managing and planning LOFAR observations. Essentially, TMSS provides a dynamic scheduling system, other specification and data flow enhancements that improve the efficiency and automation of LOFAR operations. Without going into detail, the software system is mainly composed of a frontend and a backend that are developed, respectively, using the Prime React framework and the Django framework – also various microservices communicating each other and with the Django application are included in the midst of the backend components.

Among all the goals and features expected in TMSS, users need the possibility to observe a real-time panel – the timeline view is shown in Fig. 6 – where different pieces of information are shown, i.e., scheduled observations, ongoing tasks, reservations, and so forth. The time This scenario, characterised by multiple elements and associated events, suited well for the application of the WebSocket protocol, even though the communication is de facto one-side – the client is not required to send any data to the server – as the server broadcasts the messages to the clients.

In Fig. 7 a summary of the code from the official TMSS repository[24] is depicted. Basically, a WebSocket server has been implemented as a microservice to broadcast real-time notifications about some models state changes to the connected clients. This data is then displayed in a timeline that gives a real-time overview of what is happening on the system. The state changes are listened from the LOFAR RabbitMQ message bus by the *TMSSBuslistener* that, similarly to *Signals*, reacts to relevant events and sends such data via a Web-

Socket broadcast method. In this use case note that, despite the implementation is slightly different from the approach proposed in Sec. 4 – the behaviour of Django Signals has been replaced by the LOFAR RabbitMQ message bus and the WebSocket service has been included as a microservice outside the context of the Django *AppConfig.ready* method – the principle still remains unvaried. Even more so, this results illustrated in Fig. 7 within the main details: the server is started on a separated thread and comes with a simple broadcast method that forwards a message to each of the connected clients. Eventually, there are several event listeners that reacts by calling the *_post_update_on_websocket* method whenever the event they are subscribed to is triggered.

```
class TMSSEventMessageHandlerForWebsocket(TMSSEventMessageHandler):

    def __init__(self, websocket_port: int=DEFAULT_WEBSOCKET_PORT):
        super().__init__(log_event_messages=True)
        self.websocket_port = websocket_port
        self._run_ws = True

    def start_handling(self):
        socket_started_event = Event()

        # Create and run a simple ws server
        def start_ws_server():
            self._ws_server = SimpleWebSocketServer('', self.websocket_port, WebSocket)
            socket_started_event.set()
            while self._run_ws: # Run the server till the stop_handling
                self._ws_server.serveonce()

        self.t = Thread(target=start_ws_server)
        self.t.start()
        if not socket_started_event.wait(10):
            raise RuntimeError("Could not start websocket server on port %s"%self.websocket_port)
        super().start_handling()

    def stop_handling(self):
        super().stop_handling()
        self._run_ws = False    # Stop the ws server
        self.t.join()

    def _broadcast_notify_websocket(self, msg):
        # Send a broadcast message to all connected ws clients
        for ws in self._ws_server.connections.values():
            ws.sendMessage(JSONdumps(msg))    # Stringify msg, so clients can parse it as JSON

    def _post_update_on_websocket(self, id, object_type, action):
        # Prepare the json_blob_template
        json_blob = {'object_details': {'id': id}, 'object_type': object_type.value, 'action': action.value}

        # ...some other data aggregation... #

        # Send the json_blob as a broadcast message to all connected ws clients
        self._broadcast_notify_websocket(json_blob)

    def onSubTaskCreated(self, id: int):
        self._post_update_on_websocket(id, self.ObjTypes.SUBTASK, self.ObjActions.CREATE)

    # ...other event listeners as "onObjectAction"... #
```

Fig. 7. WebSocket Service Implementation in TMSS

6. Conclusions

Real-time applications are very useful in Astronomy and Astrophysics, among all the other fields, and the WebSocket protocol remains the best option to manage real-time data, but it is not supported by Django natively. This report provides an overview of the WebSocket protocol and describes a simple and non-invasive technique, within a proof-of-concept using open source software, for the integration of a WebSocket service within a Django application with the aim of supporting generic scenarios which can be exemplified in most astronomical and astrophysics applications. This is also demonstrated by the proposed case study from the TMSS project which eventually illustrates a real-world implementation of the methodology proposed in this report.

References

[1] Nielsen H, Mogul J, Masinter LM, Fielding RT, Gettys J, Leach PJ, Berners-Lee T (1999) Hypertext Transfer Protocol – HTTP/1.1, RFC 2616. https://doi.org/10.17487/RFC2616. Available at https://www.rfc-editor.org/info/rfc2616

[2] Melnikov A, Fette I (2011) The WebSocket Protocol, RFC 6455. https://doi.org/10.17487/RFC6455. Available at https://www.rfc-editor.org/info/rfc6455

[3] Yergeau F (2003) UTF-8, a transformation format of ISO 10646, RFC 3629. https://doi.org/10.17487/RFC3629. Available at https://www.rfc-editor.org/info/rfc3629

[4] Json (javascript object notation). Available at https://www.json.org/json-en.html.

[5] Portswigger - testing for websockets security vulnerabilities. Available at https://portswigger.net/web-security/websockets.

[6] Websocket web apis. Available at https://developer.mozilla.org/en-US/docs/Web/API/WebSocket.

[7] ws. Available at https://www.npmjs.com/package/ws.

[8] Socket.io. Available at https://socket.io/.

[9] uwebsockets. Available at https://github.com/uNetworking/uWebSockets.

[10] Ratchet. Available at http://socketo.me/.

[11] Django channels. Available at https://channels.readthedocs.io/en/stable/.

[12] Flask-socket.io. Available at https://flask-socketio.readthedocs.io/en/stable/.

[13] Gorilla websocket. Available at https://github.com/gorilla/websocket.

[14] Django framework. Available at https://www.djangoproject.com/.

[15] Django signals. Available at https://docs.djangoproject.com/en/4.0/topics/signals/.

[16] Simplewebsocketserver. Available at https://github.com/dpallot/simple-websocket-server.

[17] Raciti M WebSocket Integration in Django. Available at https://github.com/tsumarios/WebSocket-in-Django.

[18] Python threading event objects. Available at https://docs.python.org/3/library/threading.html#event-objects.

[19] Python os path. Available at https://docs.python.org/3/library/os.path.html.
[20] Django applications. Available at https://docs.djangoproject.com/en/4.0/ref/applications/.
[21] Lofar (low frequency array). Available at https://www.astron.nl/telescopes/lofar/.
[22] Vermeulen RC, van Haarlem M (2011) The international lofar telescope (ilt). *2011 XXXth URSI General Assembly and Scientific Symposium*, , pp 1–1. https://doi.org/10.1109/URSIGASS.2011.6051244
[23] Tmss (telescope manager specification system). Available at https://tinyurl.com/tmssproject.
[24] Tmss websocket gitlab repository. Available at https://git.astron.nl/ro/lofar/-/blob/master/SAS/TMSS/backend/services/websocket/lib/websocket_service.py.