# Yoshi- REST API Documentation for focus webserver

### Overview

The REST API facilitates interaction with the backend to configure and manage data streams for spectrogram visualization. It supports operations to query available channels, live fiber metadata, configure streaming jobs, and retrieve job-specific metadata.

---

### API Endpoints

---

#### 1. Get Available Channels

- **Endpoint**: `GET /channels`

**Description**:
Retrieves the lowest and highest available channel numbers from the live fiber data.

**Response**:

- ```
  {
    "lowest_channel": "<int32>",
    "highest_channel": "<int32>"
  }
  ```

**Status Codes**:

- `200 OK`: Successfully retrieved channel information.

---

#### 2. Get Live Metadata

- **Endpoint**: `GET /live_metadata`

**Description**:
Returns a JSON object containing all live fiber metadata, including the range of available channels, fiber spacing ( `dx` ), pulse repetition rate ( `prr` ), and other relevant information.

**Response**:

- ```
  {
    "dx": "<float>",
    "prr": "<float>",
    "fiber_start_meters": "<int>",
    "fiber_length_meters": "<int>",
  ```

```
  "sw_version": "<str>",
  "number_of_channels": "<int>",
  "fiber_description": "<str>"
}
```

**Status Codes**:

- `200 OK`: Successfully retrieved live metadata.
- `404 Not Found`: The system has no metadata for live fiber—possibly not initialized yet.

---

##### 3. Configure Streaming Job

- **Endpoint**: `POST /configure`

**Description**:

Configures a new streaming job with specified parameters for spectrogram or heatmap display.

**Request Body**:

- ```
  {
    "displayTimeAxisDuration": "<int>",
    "nfftSelection": "<int>",
    "displayInfo": {
      "height": "<int>"
    },
    "channels": {
      "min": "<int>",
      "max": "<int>"
    },
    "frequencyRange": {
      "min": "<int>",
      "max": "<int>"
    },
    "start_time": "<int32>",
    "end_time": "<int32>",
    "view_type": "<ViewType>"  // Options: 'multichannel', 'singlechannel', 'waterfall', 'heatmaps'
  }
  ```

`view_type` **Options:**

- `multichannel`:

  Default view. Displays multiple channels, each with its full spectrogram (time-frequency) content.
- `singlechannel`:

  Restricts the display to a single channel. `channels.min` and `channels.max` must be equal.
  ```

- `waterfall`:

  Similar to `multichannel` but compresses each channel to a single frequency-summed value per timeframe.
- `heatmaps`:

  New view type. Provides a 2D heatmap where each column represents a channel and each raw represents a time frame, with colors indicating energy levels.
  Useful for snapshot visualizations of the energy distribution across all channels and frequencies at a specific time or over averaged intervals.

> ℹ️ *Note: Future versions may introduce richer heatmap variations including frequency axes.*

**Success Response**:

```
{
  "status": "Config received",
  "frequencies_list": "<list of float16>",
  "lines_dt": "<int32>",
  "channel_to_stream_index": {
    "<channel_number>": "<data_stream_id>"
  },
  "stream_amount": "<int>",
  "job_id": "<string 15 hex chars>",
  "frequencies_amount": "<int>",
  "channel_amount": "<int>",
  "job_route": "<string>",
  "view_type": "<ViewType>"
}
```

**Status Codes**:

- `200 OK`: Configuration received and processed successfully.
- `400 Bad Request`: Invalid request format or parameters.

---

##### 4. Get Job Metadata

- **Endpoint**: `GET /metadata/{job_id}`

**Description**:
Retrieves the configuration details and metadata for a previously configured job.

**URL Parameters**:

- `job_id`: The identifier of the job to retrieve metadata for.

**Response**:

- ```
  {
    "status": "Config received",
    "frequencies_list": "List of predicted frequencies",
    "lines_dt": "DT in seconds between two consecutive spectrogram calculations",
    "channel_to_stream_index": "Mapping of channel number to data stream ID",
    "stream_amount": "Number of data streams",
    "job_id": "Backend task ID",
    "frequencies_amount": "Number of frequencies per channel",
    "channel_amount": "Number of channels",
    "job_route": "String",
    "stream_port": "gRPC server port for this job",
    "stream_url": "The base URL to address the gRPC server machine"
  }
  ```

**Status Codes**:

- `200 OK`: Successfully retrieved job metadata.
- `404 Not Found`: The specified `job_id` does not exist.

---

### Data Types and Enumerations

- **Time Axis Duration Options**: `[60, 300, 600, 1200, 2400, 3600, 43200, 86400]` (in seconds)
- **NFFT Selection Options**: `[32, 64, 128, 256, 512, 1024, 2048]`
- **Frequency Range**: `{ "min": <int>, "max": <int> }`
- **Channel Range**: `{ "min": <int>, "max": <int> }`
- **View Types**:
  - `multichannel`
  - `singlechannel`
  - `waterfall`
  - `heatmaps` ← *(newly added)*

---

### Error Handling

We use the described HTTP codes and attached messages to provide information about the server's errors.

---
  ```

#### gRPC Streaming API (Multichannel, Single Channel, Waterfall, Heatmaps)

This section describes the server-streamed gRPC interface used to deliver visualization data to the client in real time. The server exposes multiple parallel streams per job (one per "stream shard" of channels). Each stream continuously emits 2D frames encoded as compact numeric matrices along with metadata to reconstruct and render the view.

- **Service**: `DataStreamService`
- **Method**: `StreamData(StreamDataRequest) returns (stream DataStream)`
- **Transport**: Insecure gRPC channel
- **Concurrency**: Open one `StreamData` call per stream ID (0-based) to receive all shards in parallel

##### Protocol definition

proto

```
syntax = "proto3";

package pandadatastream;

message DataStream {
  int32 start_channel = 1;
  int32 end_channel = 2;
  repeated int64 timestamp_in_milis = 3;
  repeated int32 row_index = 4;
  int32 data_shape_x = 5;
  int32 data_shape_y = 6;
  float global_maximum = 7;
  float global_minimum = 8;
  // main_data payload type by view:
  // - multichannel/singlechannel/waterfall: float16 buffer
  // - heatmaps: float32 buffer
  repeated bytes main_data = 9;
  optional int32 job_id = 10;
}

service DataStreamService {
  rpc StreamData (StreamDataRequest) returns (stream DataStream) {}
}

message StreamDataRequest {
  int32 stream_id = 1;  // 0..(stream_amount-1)
}
```

Ctrl+K to generate a command

powershell

Python

- Error during baby analyzer initialization
- Error handling in Kubernetes configuration
- Debugging 422 Unprocessable Entity error
- Refactor command format and differences
- Suggest a new name for Python file
- Integrate cursor with Confluence
-
-
-
-

Overview

The REST API facilitates interaction with the backend to configure and manage data streams for spectrogram visualization. It supports operations to query available channels, live fiber metadata, configure streaming jobs, and retrieve job-specific metadata.

API Endpoints

1. Get Available Channels

Endpoint: GET /channels

Description:
Retrieves the lowest and highest available channel numbers from the live fiber data.

Response:

```
{
  "lowest_channel": "<int32>",
  "highest_channel": "<int32>"
}
```

Status Codes:

200 OK: Successfully retrieved channel information.

2. Get Live Metadata

Endpoint: GET /live_metadata

Description:
Returns a JSON object containing all live fiber metadata, including the range of available channels, fiber spacing (dx), pulse repetition rate (prr), and other relevant information.

Response:

```
{
  "dx": "<float>",
  "prr": "<float>",
  "fiber_start_meters": "<int>",
  "fiber_length_meters": "<int>",
  "sw_version": "<str>",
  "number_of_channels": "<int>",
  "fiber_description": "<str>"
}
```

Status Codes:

200 OK: Successfully retrieved live metadata.

404 Not Found: The system has no metadata for live fiber—possibly not initialized yet.

3. Configure Streaming Job

Endpoint: POST /configure

Description:
Configures a new streaming job with specified parameters for spectrogram or heatmap display.

Request Body:

```
{
  "displayTimeAxisDuration": "<int>",
  "nfftSelection": "<int>",
  "displayInfo": {
    "height": "<int>"
  },
  "channels": {
    "min": "<int>",
    "max": "<int>"
  },
  "frequencyRange": {
```

    "min": "<int>",
    "max": "<int>"
  },
  "start_time": "<int32>",
  "end_time": "<int32>",
  "view_type": "<ViewType>"  // Options: 'multichannel', 'singlechannel', 'waterfall', 'heatmaps'
}


view_type Options:

multichannel:
Default view. Displays multiple channels, each with its full spectrogram (time-frequency) content.

singlechannel:
Restricts the display to a single channel. channels.min and channels.max must be equal.

waterfall:
Similar to multichannel but compresses each channel to a single frequency-summed value per timeframe.

heatmaps:
New view type. Provides a 2D heatmap where each column represents a channel and each raw represents a time frame, with colors indicating energy levels.
Useful for snapshot visualizations of the energy distribution across all channels and frequencies at a specific time or over averaged intervals.

ℹ️ Note: Future versions may introduce richer heatmap variations including frequency axes.

Success Response:


{
  "status": "Config received",
  "frequencies_list": "<list of float16>",
  "lines_dt": "<int32>",
  "channel_to_stream_index": {
    "<channel_number>": "<data_stream_id>"
  },
  "stream_amount": "<int>",
  "job_id": "<string 15 hex chars>",
  "frequencies_amount": "<int>",
  "channel_amount": "<int>",

```
  "job_route": "<string>",
  "view_type": "<ViewType>"
}
```

Status Codes:

200 OK: Configuration received and processed successfully.

400 Bad Request: Invalid request format or parameters.

4. Get Job Metadata

Endpoint: GET /metadata/{job_id}

Description:
Retrieves the configuration details and metadata for a previously configured job.

URL Parameters:

job_id: The identifier of the job to retrieve metadata for.

Response:

```
{
  "status": "Config received",
  "frequencies_list": "List of predicted frequencies",
  "lines_dt": "DT in seconds between two consecutive spectrogram calculations",
  "channel_to_stream_index": "Mapping of channel number to data stream ID",
  "stream_amount": "Number of data streams",
  "job_id": "Backend task ID",
  "frequencies_amount": "Number of frequencies per channel",
  "channel_amount": "Number of channels",
  "job_route": "String",
  "stream_port": "gRPC server port for this job",
  "stream_url": "The base URL to address the gRPC server machine"
}
```

Status Codes:

200 OK: Successfully retrieved job metadata.

404 Not Found: The specified job_id does not exist.

Data Types and Enumerations

Time Axis Duration Options: [60, 300, 600, 1200, 2400, 3600, 43200, 86400] (in seconds)

NFFT Selection Options: [32, 64, 128, 256, 512, 1024, 2048]

Frequency Range: { "min": <int>, "max": <int> }

Channel Range: { "min": <int>, "max": <int> }

View Types:

multichannel

singlechannel

waterfall

heatmaps ← (newly added)

Error Handling

We use the described HTTP codes and attached messages to provide information about the server's errors.

This is my confluence page
I want to add section for grpc server that opens stream for multichannel single channel waterfall and heatmaps
Can you create this section for me

Thoughtfor 1m 6s

#### gRPC Streaming API (Multichannel, Single Channel, Waterfall, Heatmaps)

This section describes the server-streamed gRPC interface used to deliver visualization data to the client in real time. The server exposes multiple parallel streams per job (one per "stream shard" of channels). Each stream continuously emits 2D frames encoded as compact float16 matrices along with metadata to reconstruct and render the view.

- **Service**: `DataStreamService`
- **Method**: `StreamData(StreamDataRequest) returns (stream DataStream)`
- **Transport**: Insecure gRPC channel
- **Concurrency**: Open one `StreamData` call per stream ID (0-based) to receive all shards in parallel

#### Protocol definition

proto

```proto
syntax = "proto3";

package pandadatastream;

message DataStream {
  int32 start_channel = 1;
  int32 end_channel = 2;
  repeated int64 timestamp_in_milis = 3;
  repeated int32 row_index = 4;
  int32 data_shape_x = 5;
  int32 data_shape_y = 6;
  float global_maximum = 7;
  float global_minimum = 8;
  repeated bytes main_data = 9;      // Single float16 buffer per message
  optional int32 job_id = 10;        // Equal to stream_id for reference
}

service DataStreamService {
  rpc StreamData (StreamDataRequest) returns (stream DataStream) {}
}

message StreamDataRequest {
  int32 stream_id = 1;               // 0..(stream_amount-1)
}
```

#### Connecting to a job

- After POST `/configure`, use:
- **stream_url** and **stream_port**: gRPC endpoint host and port
- **stream_amount**: number of streams to open (IDs `0..stream_amount-1`)
- **channel_to_stream_index**: mapping for channels → stream shard
- Open one gRPC call per `stream_id` to receive its shard.

#### Message layout and decoding

- **Data buffer**: `main_data[0]` is a single contiguous float16 array (little-endian IEEE 754) of length `data_shape_x * data_shape_y` in C-order (row-major).
- **Reshape**: `buffer → np.float16 → reshape((data_shape_x, data_shape_y), order='C')`

- Axis X: concatenated “frequency bins per channel” across all channels in this stream
- Axis Y: time lines (new rows per message are appended)
- **Row indices**: `row_index[j]` gives the absolute line index for column `j` in the reshaped matrix; use it to stitch the global time axis across messages.
- **Timestamps**: `timestamp_in_milis[j]` is the wall-clock timestamp for line `row_index[j]` (typically one timestamp repeated per incoming chunk).
- **Channel range**: `start_channel..end_channel` tells which channels this stream contains.
- **Normalization**: `global_minimum` and `global_maximum` are job-scoped running extrema. For consistent coloring:
- normalized = clamp((data - global_minimum) / (global_maximum - global_minimum), 0, 1)

Per-view semantics

- **Multichannel (spectrogram)**:
- `data_shape_x = channels_in_stream * frequencies_per_channel`
- Reconstruct per-channel spectra per time line by:
- split axis X into contiguous blocks of size `frequencies_per_channel`, one block per channel in `[start_channel..end_channel]`
- **Singlechannel (spectrogram)**:
- `start_channel == end_channel`
- `data_shape_x = frequencies_per_channel` (single channel’s spectrum per time line)
- **Waterfall**:
- Frequency content is aggregated to one value per channel
- `data_shape_x = channels_in_stream` (one intensity per channel per time line)
- **Heatmaps**:
- Current encoding matches waterfall (one value per channel per time line) for a channel-time heatmap

- Note: future versions may add a frequency axis (then `data_shape_x` will become `channels_in_stream * frequencies_per_channel` similar to multichannel)

#### Sample Python client

python

```

```

```

```

```python
import grpc
import numpy as np
from pzpy.recording.backends.protocols.panda_datastream import pandadatastream_pb2_grpc, pandadatastream_pb2

def consume_stream(stream_url, stream_port, stream_id):
    channel = grpc.insecure_channel(f"{stream_url}:{stream_port}")
    stub = pandadatastream_pb2_grpc.DataStreamServiceStub(channel)
    request = pandadatastream_pb2.StreamDataRequest(stream_id=stream_id)

    for msg in stub.StreamData(request):
        buf = np.frombuffer(msg.main_data[0], dtype=np.float16)
        mat = buf.reshape((msg.data_shape_x, msg.data_shape_y), order='C')

        # Use msg.row_index[j] and msg.timestamp_in_milis[j] per column j
        # Normalize for coloring if desired:
        if msg.global_maximum > msg.global_minimum:
            mat_norm = np.clip((mat - msg.global_minimum) / (msg.global_maximum - msg.global_minimum), 0.0, 1.0)
        else:
            mat_norm = mat  # Degenerate case

        # Interpret by view:
        # - Multichannel/singlechannel: split X into per-channel frequency blocks
        # - Waterfall/heatmaps: X is channels, one value per channel per time line
        # Render mat or mat_norm according to your canvas layout

# Example usage for all shards:
# for sid in range(stream_amount):
#     start a thread calling consume_stream(stream_url, stream_port, sid)
```

```

```

#### Stream sharding and ordering

- The server splits channels across `stream_amount` shards; each shard maps to a `stream_id`.
- Use `start_channel..end_channel` to place each shard in the correct horizontal position.

- Use `row_index` to place each received time column at the correct vertical index; messages may batch multiple consecutive lines.

#### Keepalive and timeouts

- The server enforces a keepalive timeout:
- Historic jobs: ~180s; Live jobs: ~30s (plus ~15s startup grace)
- As long as a client is actively consuming `StreamData`, the timeout is extended.
- If no active consumption occurs and the timeout elapses, the server stops the job and closes streams.
- Consume promptly to avoid queue buildup; one client per `stream_id` is recommended.

#### Data types

- `main_data`: float16 binary (little-endian), reshape using `data_shape_x`, `data_shape_y`
- `timestamp_in_milis`: int64 epoch milliseconds
- `row_index`: int32 monotonically increasing line indices within the job
- `global_minimum`, `global_maximum`: float32 extrema for normalization

—

- Added a concise gRPC section covering the service, proto, connection details, message schema, per-view interpretation, normalization, sharding, keepalive, and a Python client example.

On heatmap it will be float32

Thoughtfor 11s

#### gRPC Streaming API (Multichannel, Single Channel, Waterfall, Heatmaps)

This section describes the server-streamed gRPC interface used to deliver visualization data to the client in real time. The server exposes multiple parallel streams per job (one per "stream shard" of channels). Each stream continuously emits 2D frames encoded as compact numeric matrices along with metadata to reconstruct and render the view.

- **Service**: `DataStreamService`
- **Method**: `StreamData(StreamDataRequest) returns (stream DataStream)`

- **Transport**: Insecure gRPC channel
- **Concurrency**: Open one `StreamData` call per stream ID (0-based) to receive all shards in parallel

#### Protocol definition

proto

```
syntax = "proto3";

package pandadatastream;

message DataStream {
  int32 start_channel = 1;
  int32 end_channel = 2;
  repeated int64 timestamp_in_milis = 3;
  repeated int32 row_index = 4;
  int32 data_shape_x = 5;
  int32 data_shape_y = 6;
  float global_maximum = 7;
  float global_minimum = 8;
  // main_data payload type by view:
  // - multichannel/singlechannel/waterfall: float16 buffer
  // - heatmaps: float32 buffer
  repeated bytes main_data = 9;
  optional int32 job_id = 10;
}

service DataStreamService {
  rpc StreamData (StreamDataRequest) returns (stream DataStream) {}
}

message StreamDataRequest {
  int32 stream_id = 1;  // 0..(stream_amount-1)
}
```

#### Connecting to a job

- After POST `/configure`, use:
- **stream_url**, **stream_port**: gRPC endpoint host and port
- **stream_amount**: number of streams to open (IDs `0..stream_amount-1`)
- **channel_to_stream_index**: mapping for channels → stream shard
- **view_type**: use this to decode `main_data` correctly (dtype and layout)
- Open one gRPC call per `stream_id` to receive its shard.

#### Message layout and decoding

- **Data buffer (`main_data[0]`) type by view**:

- **Multichannel / Singlechannel / Waterfall**: float16
- **Heatmaps**: float32
- **Reshape**: `buffer → dtype → reshape((data_shape_x, data_shape_y), order='C')`
- Axis X: concatenated “frequency bins per channel” across all channels in this stream (waterfall/heatmaps: one value per channel)
- Axis Y: time lines (new rows per message are appended)
- **Row indices**: `row_index[j]` gives the absolute line index for column `j`
- **Timestamps**: `timestamp_in_milis[j]` is the wall-clock timestamp (epoch ms)
- **Channel range**: `start_channel..end_channel` specifies which channels the shard covers
- **Normalization**: use `global_minimum` / `global_maximum` for consistent coloring