Understanding the API Structure

Before we code, let's explore the key endpoints we'll use:

1. Get All Jobs

```
GET /api/v6.8/jobs
```

Returns a list of all jobs on the machine with basic information like start time, end time, and status.

2. Get Specific Job Details

```
GET /api/v6.8/jobs/{jobId}
```

Returns detailed information about a specific job, including:

- `timeStarted` : When the job began
- `timeEnded` : When the job finished (null if still running)
- `result` : Job status (`Pending` , `Successful` , `UserCanceled` , `Failed`)
- `durations` : Breakdown of exposure time, recoating time, etc.
- `material` , `task` , `layerInTaskCurrent` : Build parameters

3. Get user messages for a job

```
GET /api/v6.8/software/usermessages
```

Returns details about the user messages, including

A list of messages is returned or nothing if no messages exist.

- `severity` : Severity of the user message
- `jobId` : the unique identifier of the build job,
- `timeRaised` : the timestamp the messages was created,
- `message` : the message itself,

The API uses **OAuth2 authentication** and returns data in **JSON format**. All timestamps are in UTC.

To get a complete overview of the Web API interface, please refer to appendix Api Documentation

Setting Up the Environment

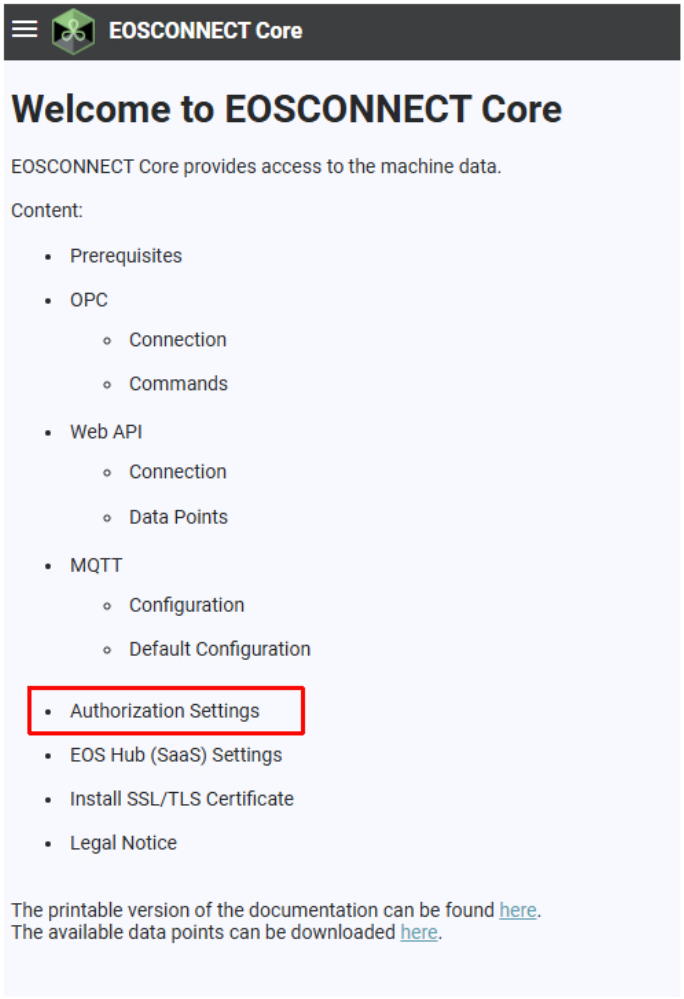
First, let's install the required Python packages and set up our imports.

In [ ]:
```python
import requests
import time
import json
from datetime import datetime, timezone
from typing import Dict, Optional, Tuple
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from dotenv import load_dotenv
import os
from datetime import timedelta

# For demonstration purposes, we'll disable SSL warnings
# In production, use proper SSL certificates!
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

print("✓ All packages loaded successfully!")
```

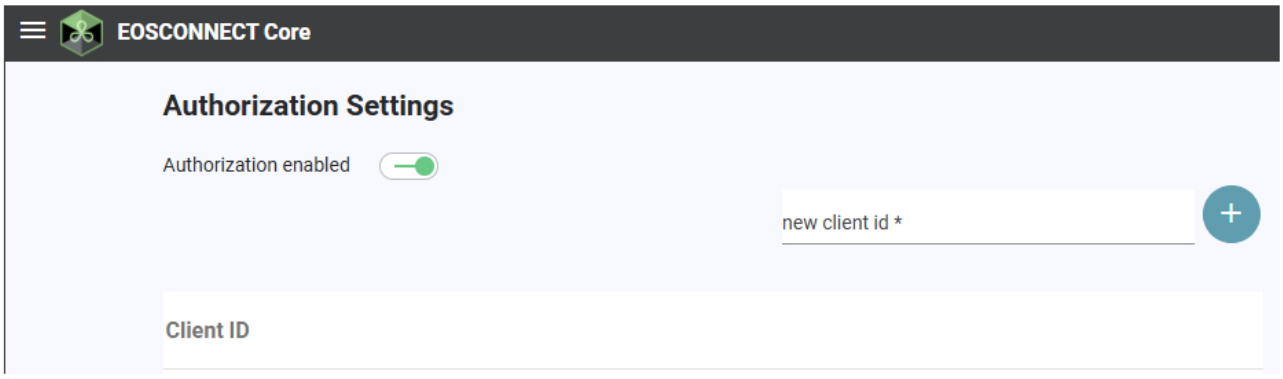
Configuration

Let's set up the connection parameters. In a real production environment, you'd use environment variables or a secure configuration file for credentials.

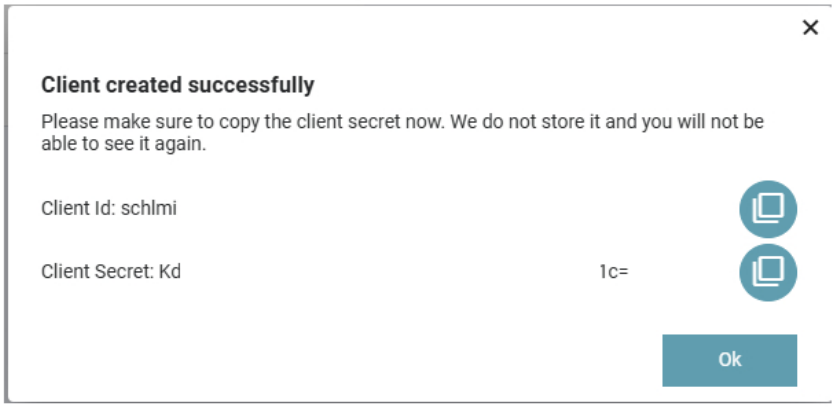
To access the machine interface, we need to create API credentials. This can be done conveniently through the EOSCONNECT Core interface. Open your printer's web page in a browser - in our case, that would be `https://si16120019/`.



In many cases, the browser will display a warning that the site is not secure - don't worry, this warning appears because EOS machines come with self-signed certificates by default (see also Understanding Self-Signed Certificates in the appendix).



Under the Authorization Settings menu, we can generate a new Client ID and Secret pair using the (+) button.



The recently generated API credentials are stored in the `.env` file.

Why use a `.env` file?

Storing sensitive credentials like API keys and passwords directly in your code is a security risk - especially if you share your notebooks or commit them to version control systems like Git. A `.env` file provides a clean solution:

- **Security**: Credentials are kept separate from your code
- **Convenience**: Easy to update credentials without modifying your code
- **Flexibility**: Different environments (development, production) can use different `.env` files
- **Best Practice**: Follows the "12-factor app" methodology for configuration management

Setting the Variables

In the following, we determine the version of the EOSCONNECT Core Web API that we want to use. EOSCONNECT Core offers backward compatibility for the Web API - meaning that machines with both current and older software can be queried via the Web API. We read the latest version directly from the machine - at the time of this post, the latest available version was version `v6.8`

In [ ]:
```python
# Load environment variables from .env file
load_dotenv()

# Load API credentials from environment variables
HOSTNAME = os.getenv("HOSTNAME")
API_VERSION_INFO_URL = f"https://{HOSTNAME}/api/supportedVersions"

# Fetch available API versions

# Check if API_VERSION is set in environment variables
API_VERSION = os.getenv("API_VERSION")

if API_VERSION:
    # Use the version from environment variable
    print(f"✓ Using API version from environment variable: {API_VERSION}")

else:
    # Fetch API versions from the endpoint
    print(f"\n🔍 Fetching available API versions from {API_VERSION_INFO_URL}...")
    response = requests.get(API_VERSION_INFO_URL, verify=False, timeout=10)
    versions_data = response.json()
    # Extract version strings from the response
    available_versions = [f"v{v['majorVersion']}.{v['minorVersion']}" for v in versions_data]

    if available_versions:
        # Sort versions to get the latest
        API_VERSION = sorted(available_versions, key=lambda v: [int(x) for x in v.lstrip('v').split('.')])[-1]
        print(f"✓ Available API versions: {', '.join(available_versions)}")
        print(f"✓ Using latest version: {API_VERSION}")
    else:
        # Fallback to default version
        API_VERSION = "v6.0"
        print(f"⚠️  No versions found in response, using default: {API_VERSION}")
```

Loading API Credentials

Now we read the API credentials from the `.env` file into memory

In [ ]:
```python
from dotenv import load_dotenv
import os
from datetime import timedelta

# Load environment variables from .env file
load_dotenv()

API_BASE_URL = f"https://{HOSTNAME}/api/{API_VERSION}"
API_CLIENT_ID = os.getenv("API_CLIENT_ID")
API_CLIENT_SECRET = os.getenv("API_CLIENT_SECRET")

print("✓ Environment variables loaded")
print(f"  - Client ID: {API_CLIENT_ID}")
print(f"  - Client Secret: {'*' * len(API_CLIENT_SECRET) if API_CLIENT_SECRET else 'Not set'}")
print(f"  - Base URL: {API_BASE_URL}")
```

Retrieving the Access Token

To communicate with the EOSCONNECT Web API, we need an **Access Token**. This token serves as a digital key that:

- **Authentication**: Confirms that we are authorized to access the API
- **Authorization**: Defines which actions we are permitted to perform (based on client permissions)
- **Security**: Protects the machine from unauthorized access

The token is requested via the **OAuth2 Client Credentials Flow** - a standardized procedure for machine-to-machine communication. The API credentials (Client ID and Secret) that we previously created in the EOSCONNECT Core interface are exchanged for a time-limited access token.

Important: The token has a limited validity period (typically 1 hour). For longer-running applications, the token must be renewed before it expires.

In [ ]:
```python
def get_oauth_token(client_id: str, client_secret: str) -> Optional[Dict]:
    """
    Fetch OAuth2 token using client credentials flow.

    Args:
        client_id: OAuth2 client ID
        client_secret: OAuth2 client secret

    Returns:
        Dictionary containing token information or None if request fails
    """
    token_url = f"https://{HOSTNAME}/auth/connect/token"

    # Prepare the request data for client credentials grant
    data = {
        "grant_type": "client_credentials",
        "client_id": client_id,
        "client_secret": client_secret
    }

    headers = {
        "Content-Type": "application/x-www-form-urlencoded"
    }

    try:
        response = requests.post(token_url, data=data, headers=headers, verify=False, timeout=10)
        response.raise_for_status()
        token_data = response.json()
        return token_data
    except requests.exceptions.RequestException as e:
        print(f"❌ Error fetching OAuth token: {e}")
        return None


def refresh_token() -> bool:
    """
    Refresh the OAuth2 token and update global HEADERS.

    Returns:
        True if token refresh was successful, False otherwise
    """
    global API_TOKEN, HEADERS

    token_data = get_oauth_token(API_CLIENT_ID, API_CLIENT_SECRET)
```

```
    if token_data:
        API_TOKEN = token_data.get("access_token", "")
        token_type = token_data.get("token_type", "Bearer")

        HEADERS = {
            "Authorization": f"{token_type} {API_TOKEN}",
            "Accept": "application/json"
        }

        return True
    else:
        print("❌ Failed to refresh token")
        return False

# Fetch the token
token_response = get_oauth_token(API_CLIENT_ID, API_CLIENT_SECRET)

if token_response:
    # Extract token details
    API_TOKEN = token_response.get("access_token", "")
    token_type = token_response.get("token_type", "Bearer")
    expires_in = token_response.get("expires_in", 0)

    # Update the headers with the new token
    HEADERS = {
        "Authorization": f"{token_type} {API_TOKEN}",
        "Accept": "application/json"
    }

    # Calculate expiry time
    current_time = datetime.now(timezone.utc)
    expiry_time = current_time + timedelta(seconds=expires_in)

    # Display token information
    print("🔑 Token Details:")
    print(f"  Token Type: {token_type}")
    print(f"  Expires In: {expires_in} seconds ({expires_in / 3600:.2f} hours)")
    print(f"  Current Time (UTC): {current_time.strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"  Expiry Time (UTC): {expiry_time.strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"  Token (first 50 chars): {API_TOKEN[:50]}...")
    print(f"\n✓ Authorization headers updated")
else:
    print("⚠️  Failed to retrieve token. Please check your credentials.")
```

Step 1: Connecting to the API

Let's create functions to fetch job details and user messages from the API. This demonstrates how simple it is to interact with the EOSCONNECT interface!

In [ ]:

```
def get_last_job() -> Optional[Dict]:
    """
    Fetch the most recent job from the printer.

    Returns:
        Dictionary containing the last job's details or None if request fails
    """
    url = f"{API_BASE_URL}/jobs/last"
    try:
        response = requests.get(url, headers=HEADERS, verify=False, timeout=10)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"❌ Error fetching last job: {e}")
        return None

def get_user_message(jobId : str, limit: int = 20) -> Optional[Dict]:
    """
    Fetch the current user message displayed on the printer.

    Returns:
        Dictionary containing the user message or None if request fails
    """
    url = f"{API_BASE_URL}/software/usermessages"
    params = {"jobId": jobId, "limit":limit}
    try:
        response = requests.get(url, headers=HEADERS, params=params, verify=False, timeout=10)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"❌ Error fetching user message: {e}")
        return None
```

```python
def get_all_jobs(limit: int = 10, from_date: Optional[str] = None, to_date: Optional[str] = None) -> Optional[list]:
    """
    Fetch a list of recent jobs.

    Args:
        limit: Maximum number of jobs to retrieve
        from_date: Filter jobs starting from this date (ISO 8601 format, e.g., '2024-01-01T00:00:00Z')
        to_date: Filter jobs up to this date (ISO 8601 format, e.g., '2024-12-31T23:59:59Z')

    Returns:
        List of job summaries or None if request fails
    """
    url = f"{API_BASE_URL}/jobs"
    params = {"take": limit}

    # Add optional date filters if provided
    if from_date:
        params["from"] = from_date
    if to_date:
        params["to"] = to_date

    try:
        response = requests.get(url, headers=HEADERS, params=params, verify=False, timeout=10)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"❌ Error fetching jobs list: {e}")
        return None


print("✓ API functions defined")
```

Step 2: Testing the Connection

Let's test our connection by fetching recent jobs from the printer. This will help us understand the data structure.

The `fetch_jobs` function queries the machine for all jobs that were started within the time range between `from_date` and `to_date`. We use the previously defined API function `get_all_jobs` for this purpose. The result from the API function is formatted and then displayed in a Gradio UI. The code for the graphical representation has nothing to do with EOSCONNECT Core and has been moved to the `ui.py` file for the sake of clarity.

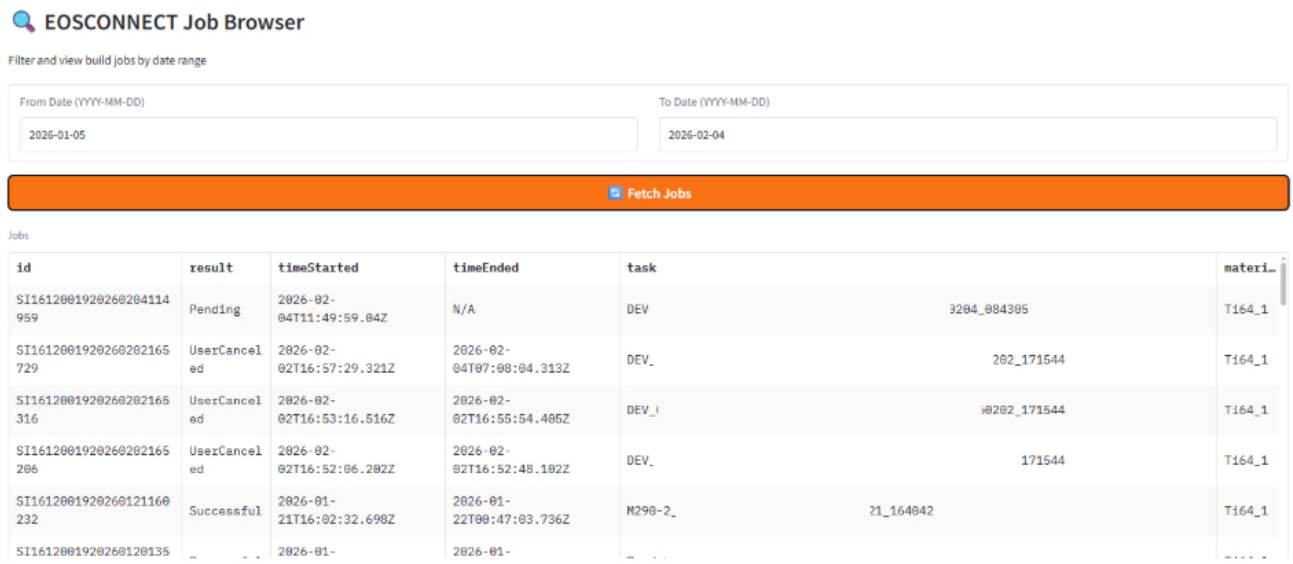
Let's look at the job details:

- The `id` is a unique identifier for the job. Unfortunately, it provides no information about our task (recipe), as it merely represents a combination of the machine serial number and a timestamp.
- The `result` is a property we're interested in. We want to be notified whether a job has completed and whether it was successful or not.

In [ ]:
```python
import pandas as pd
from datetime import datetime, timedelta
from ui import create_job_browser_ui

def fetch_jobs(from_date, to_date):
    """
    Fetch jobs within the specified date range and display them in a table.

    Args:
        from_date: Start date for filtering jobs
        to_date: End date for filtering jobs

    Returns:
        Pandas DataFrame with job information
    """
    # Convert dates to ISO 8601 format with time
    from_datetime = f"{from_date}T00:00:00.000Z" if from_date else None
    to_datetime = f"{to_date}T23:59:59.999Z" if to_date else None

    # Fetch jobs from API
    refresh_token()
    jobs = get_all_jobs(limit=500, from_date=from_datetime, to_date=to_datetime)

    if not jobs:
        return pd.DataFrame(columns=['id', 'result', 'timeStarted', 'timeEnded', 'task', 'material'])
```

```
    # Extract relevant fields
    job_data = []
    for job in jobs:
        job_data.append({
            'id': job.get('id', 'N/A'),
            'result': job.get('result', 'N/A'),
            'timeStarted': job.get('timeStarted', 'N/A'),
            'timeEnded': job.get('timeEnded', 'N/A'),
            'task': job.get('task', 'N/A'),
            'material': job.get('material', 'N/A')
        })

    # Create DataFrame
    df = pd.DataFrame(job_data)
    return df

# Launch the interface
demo = create_job_browser_ui(fetch_jobs)
demo.launch(share=False, inline=True)
```

Example: this is a sample output



We now see an overview of the most recently built jobs. Now, depending on the job result, we can decide whether we want to retrieve further details or not. In our case, we are only interested in additional details—specifically the user messages—if a job was canceled or ended due to an error.

Step 3: Job Monitoring Logic

Now let's implement the core monitoring logic. This function checks whether the most recently built job completed successfully or not. If the job was aborted or terminated due to an error, we then retrieve the user messages to gather any relevant information about the cancellation.

In [ ]:
```
def check_last_job_status(last_job_id: Optional[str] = None) -> Optional[Dict]:
    """
    Check the status of the last job.

    Args:
        last_job_id: Optional job ID to compare against. If provided and matches
                     the current last job, returns None (no change detected)

    Returns:
        Dictionary containing:
        - 'result': Job result status (Pending, Successful, Failed, UserCanceled)
        - 'jobId': The job ID
        - 'task': The task name
        - 'timeStarted': When the job started
        - 'timeEnded': When the job ended (None if still running)
        - 'userMessages': List of user messages (only for failed jobs)

        Returns None if last_job_id matches the current last job ID
    """
    refresh_token()
    # Fetch the last job
    last_job = get_last_job()
```

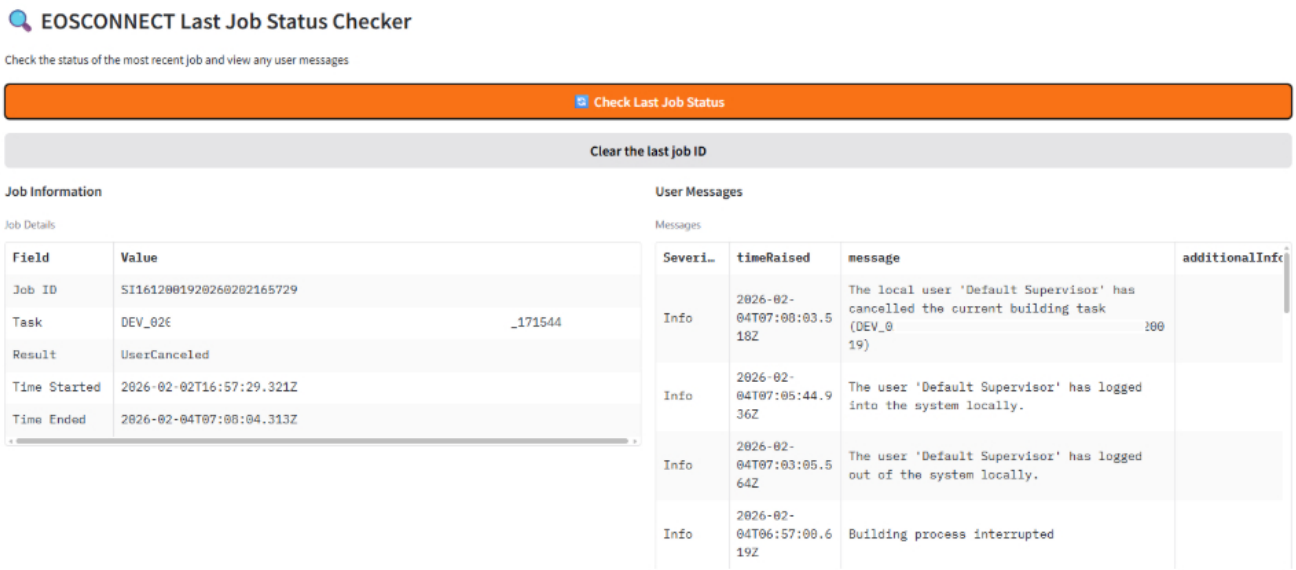
```
    if not last_job:
        return {
            'result': 'Error',
            'jobId': None,
            'task': None,
            'timeStarted': None,
            'timeEnded': None,
            'userMessages': None,
            'error': 'Failed to fetch last job'
        }

    job_id = last_job.get('id', 'Unknown')

    # If last_job_id is provided and matches current job, return None
    if last_job_id is not None and job_id == last_job_id:
        return None

    result = last_job.get('result', 'Unknown')
    task = last_job.get('task', 'N/A')
    time_started = last_job.get('timeStarted', 'N/A')
    time_ended = last_job.get('timeEnded', 'N/A')

    # Prepare base response
    response = {
        'result': result,
        'jobId': job_id,
        'task': task,
        'timeStarted': time_started,
        'timeEnded': time_ended,
        'userMessages': None
    }

    # If job failed or was canceled, fetch user messages
    if result in ['Failed', 'UserCanceled']:
        user_messages = get_user_message(job_id, limit=20)
        response['userMessages'] = user_messages if user_messages else []

    return response


print("✓ Last job status check function defined")
```

In [ ]:

```
import pandas as pd
from ui import create_job_status_ui

# Track the last job ID
last_job_id_tracker = None

def fetch_job_status():
    """
    Fetch the last job status and format it for display.
    Returns a tuple of (job_info_df, user_messages_df)
    """
    global last_job_id_tracker

    result = check_last_job_status(last_job_id_tracker)

    if result is None:
        # No new job detected
        return (
            pd.DataFrame([{"Info": f"No new job detected or job status unchanged. Tracked ID: {last_job_id_tracker}"}]),
            pd.DataFrame()
        )

    # Update tracker
    last_job_id_tracker = result.get('jobId')

    # Create job info DataFrame
    job_info = {
        'Field': ['Job ID', 'Task', 'Result', 'Time Started', 'Time Ended'],
        'Value': [
            result.get('jobId', 'N/A'),
            result.get('task', 'N/A'),
            result.get('result', 'N/A'),
            result.get('timeStarted', 'N/A'),
            result.get('timeEnded', 'N/A')
        ]
    }
    job_info_df = pd.DataFrame(job_info)
```

```python
    # Create user messages DataFrame
    user_messages = result.get('userMessages')
    if user_messages:
        messages_data = []
        for msg in user_messages:
            messages_data.append({
                'Severity': msg.get('severity', 'N/A'),
                'timeRaised': msg.get('timeRaised', 'N/A'),
                'message': msg.get('message', 'N/A'),
                'additionalInfo': msg.get('additionalInfo', 'N/A')
            })
        user_messages_df = pd.DataFrame(messages_data)
    else:
        user_messages_df = pd.DataFrame([{"Info": "No user messages available"}])

    return job_info_df, user_messages_df

def clear_last_job_id_tracker():
    global last_job_id_tracker
    last_job_id_tracker = None
    return [],[]


# Create Gradio interface for job status
job_status_demo = create_job_status_ui(fetch_job_status, clear_last_job_id_tracker)

# Launch the interface
job_status_demo.launch(share=False, inline=True)
```

Example: this is a sample output



In this example, we can see that the most recently built job did not complete successfully. A look at the user messages shows that the machine operator aborted the job.

Appendix:

Viewing EOSCONNECT Core API Documentation with OpenAPI Editor

You can use the following official Swagger Editor: https://editor.swagger.io/ to render the `swagger.json` file. This gives you an overview of all available API calls provided by EOSCONNECT Core. A similar interface is also offered directly by EOSCONNECT Core itself and can be found under the Web API / Data Points section.

1. Navigate to the official Swagger Editor: https://editor.swagger.io/
2. Click on **File → Import URL**
3. Select the `swagger.json` file

Understanding Self-Signed Certificates

When you connect to a website, your browser checks if the site's security certificate is signed by a trusted authority (like a well-known organization).

Self-signed certificates are created by the website owner themselves, not by a trusted authority. It's like writing your own ID card instead of getting one from the government.

Your browser shows a warning because it can't verify if the certificate is trustworthy—it could be legitimate (like with EOS printers) or potentially dangerous. In the case of EOS machines, it's safe because you're connecting to your own local printer, not a random website on the internet.

Think of it as: The printer is saying "Trust me, I am who I say I am" without a third party vouching for it.