RUGBY-360

Smart monitoring for comprehensive management.

Cristian De Nicola, MAT: **744954**; Rosario Catalfamo, MAT: **744951**

1. Introduction

Rugby 360 is an application designed to help rugby teams to monitor, analyze, and manage player performance and team dynamics. In the fast-paced and physically demanding world of rugby, real-time data and advanced analytics are essential tools for coaches, trainers, and medical staff. *Rugby 360* provides a holistic solution that leverages cutting-edge technology to enhance player safety, optimize performance, and drive team success.

1.1 Key Features

- **Heatmap Tracking:** Rugby 360 includes sophisticated heatmap functionality to trace player movements throughout a game or training session. This feature provides valuable insights into player positioning, movement patterns, and overall field coverage, helping coaches develop strategic game plans.

- **Impact, Acceleration, and Heart Rate Calculation:** The application monitors critical player metrics such as the force of impacts, acceleration rates, and heart rate. These metrics are crucial for assessing player fatigue, monitoring physical stress, and ensuring player safety during both games and training sessions.

- **Comprehensive Dashboards**:
    - **Player Performance Dashboard:** This dashboard presents individual player performance metrics, including distance traveled, average speed, number of tackles made, playtime, and heart rate, among others. It enables coaches to quickly evaluate each player's strengths and areas for improvement.
    - **Team Analysis Dashboard:** This dashboard offers a macro view of team performance, displaying aggregated metrics such as total distance covered by the team, ball possession time, passing

accuracy, and goals scored/conceded. This high-level overview helps in assessing overall team effectiveness and cohesion.

- ○ **Advanced Analytics Dashboard:** To uncover deeper insights, the advanced analytics dashboard offers complex data visualizations, including scatter plots, heatmaps, and clustering analyses. This allows for the identification of patterns, trends, and correlations in player and team performance over time.

2. System Architecture

The system architecture of *Rugby 360* is designed to ensure seamless data flow, real-time processing, and robust storage, all within a scalable and maintainable ecosystem.

At the core of the system is a **Python** script responsible for simulating player and team performance data. This simulated data is transmitted via **MQTT**, a lightweight messaging protocol, enabling efficient communication between the Python script and the Node-RED application.

Node-RED serves as the primary platform for orchestrating the entire application, processing incoming data, performing real-time analysis, and driving the dashboards that provide insights into player and team performance. To ensure that all collected data is securely stored and easily retrievable, MongoDB is used as the database solution, offering flexibility in managing the various types of performance metrics and analysis results.

Docker containers are employed to create and manage the entire ecosystem, ensuring that each component—Python scripts, MQTT broker, Node-RED, and MongoDB—runs consistently across different environments, simplifying deployment and maintenance while ensuring system reliability and scalability.

here a graphic representation of the architecture -
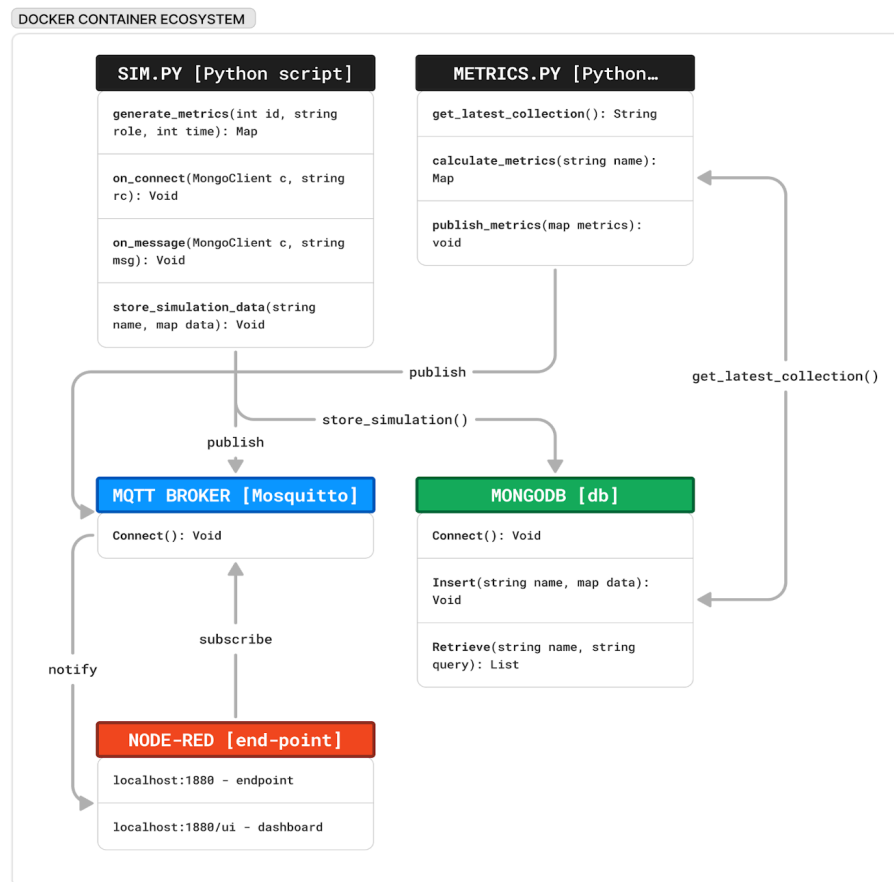
2.1 Docker

Before continuing, we wanted to open a small parenthesis on why we use **Docker**.

Docker is used in this project to ensure a consistent development and production environment, simplifying the application deployment and management process. With Docker containers, every component of the system-from the Python script for data simulation, to the MQTT broker, the Node-RED application, and the MongoDB database-can run in an isolated, standardized environment. This approach reduces issues related to software dependencies and configuration, allowing the entire ecosystem to be easily replicated on any machine. In addition,

Docker facilitates application scalability and maintenance, as it allows individual components to be upgraded or replaced without impacting the entire system.

Being that we had problems on the VM, it seemed like a good compromise.

In order to mount the container, the command will be `docker-compose -f path_to_docker_file up -d`.

3. Application Components

Let us now go into the details of the project, and look specifically at its component parts.

3.1 Python Scripts

We chose Python as the main language for the project because of its simplicity, which facilitates rapid development and code maintenance. In addition, Python offers a wide range of external libraries that extend its versatility, enabling easy integration with other systems and technologies, such as MQTT for communication and MongoDB for data management.

3.1.1 sim.py

The `sim.py` file is one of the main components of the system and plays a key role in simulating the behavior of rugby players. This script generates a data stream that emulates sensor metrics (mouth guard, GPS, and wristband) for each player, such as heart rate, GPS speed, body temperature, and blood pressure, over a period that simulates an entire 80-minute match (at a scale of 1:60). The generated data are published to an MQTT broker, allowing real-time distribution to other system components, and are also stored in a MongoDB database for later analysis.

The system dynamically assigns data to players, realistically simulating the impact of fatigue, collisions, and physical performance.

First, global variables are set, which will be used for proper connection to MQTT and mongo.

```python
# MQTT settings
MQTT_BROKER = 'localhost'
MQTT_PORT = 1883
MQTT_TOPIC_TEMPLATE = 'rugby/players/{}/sensors'
MQTT_COORDINATES_TOPIC_TEMPLATE =
'rugby/players/{}/sensors/coordinates'

# MongoDB settings
MONGO_URI = "mongodb://localhost:27017/"
DATABASE_NAME = "rugbyDB"
BASE_COLLECTION_NAME = "simulations"

# MongoDB client
client = MongoClient(MONGO_URI)
db = client[DATABASE_NAME]

# Definition of roles and player distribution in rugby
ROLES = {
    1: 'prop',
    2: 'prop',
    3: 'hooker',
    4: 'lock',
    5: 'lock',
    6: 'flanker',
    7: 'flanker',
    8: 'number_eight',
    9: 'scrum_half',
    10: 'fly_half',
    11: 'center',
    12: 'center',
    13: 'wing',
    14: 'wing',
    15: 'full_back'
}
```

After setting all the variables that will be needed, the two callback functions required by MQTT are implemented: `on_connect()` and `on_message()`.

These functions are essential for managing MQTT communication in the project. The `on_connect()` function is executed when the client connects to the MQTT broker. If the connection is successful, it automatically subscribes to the sensor-related topics of all players, thus ensuring real-time data monitoring. The `on_message()` function, on the other hand, is called whenever a message is received on one of the subscribed topics.

After this, we implemented the data generation function `generate_metrics()`, which through coefficients and modifiers based on role, match time etc, try to simulate the data as best as possible. ([check github for full code.](#))

```python
def generate_metrics(player_id, role, elapsed_time):
    # ...
    # NB. we avoided putting all the code in this snapshot
    # putting in all modifiers and coefficients was too long
    metrics = {
        "timestamp": datetime.now(rome_timezone).isoformat(),
        "elapsed_time": elapsed_time,
        "player_id": player_id,
        "role": role,
        "heart_rate": {"heart_rate": heart_rate},
        "temperature": {"body_temperature": body_temperature},
        "blood_pressure": {
            "systolic": systolic,
            "diastolic": diastolic
        },
        "calories_consumed": {"calories": round(calorie_counters[player_id], 1)},
        "gps": {
            "x": random.randint(0, 120),
            "y": random.randint(0, 50),
            "unic": random.randint(1706, 5118),
            "velocity": gps_velocity,
            "top_speed": top_speed[player_id]
        },
        "impacts": {
            "impact_count": impact_counters[player_id],
            "impact_force": impact_force
        }
    }
    # ...
    return metrics
```

Post data simulation, a second function called `store_simulation_data()` will use the global variables previously settled in order to save on the mongo the data. Lastly, here is the `main()` function of `sim.py`, the function will simulate the data, publish them using MQTT and then save the simulation in the mongodb.

```python
def main():
    mqtt_client = mqtt.Client(protocol=mqtt.MQTTv311)  # Specify MQTT protocol version
    mqtt_client.on_connect = on_connect
    mqtt_client.on_message = on_message
    mqtt_client.connect(MQTT_BROKER, MQTT_PORT)

    mqtt_client.loop_start()  # Start a background thread to handle MQTT events

    try:
        elapsed_time = 0  # Initialize the simulation elapsed time
        simulation_name = datetime.now(rome_timezone).strftime("%Y%m%d_%H%M%S")

        while elapsed_time <= 80:
            for player_id in range(1, 16):
                role = ROLES[player_id]
                payload = generate_metrics(player_id, role, elapsed_time)

                mqtt_topic = MQTT_TOPIC_TEMPLATE.format(player_id)
                mqtt_client.publish(mqtt_topic, json.dumps(payload))
                store_simulation_data(simulation_name, payload)
                print(f"Published sensor data for Player {player_id} to topic '{mqtt_topic}'")

            elapsed_time += 1
            time.sleep(1)

    except KeyboardInterrupt:
        print("\nStopping sensor simulation...")
        mqtt_client.loop_stop()  # Stop the MQTT thread
        mqtt_client.disconnect()
```

3.1.2 realtime_metrics.py

The `realtime_metrics.py` simulates the collection and publication of real-time metrics on rugby players using a MongoDB database and the MQTT protocol. The code connects to a database to retrieve data from the most recent simulations, calculates metrics such as speed, calories expended, blood pressure, and impacts suffered by players, and then publishes this data to an MQTT broker for real-time monitoring. This data is used for the post-match performance study, where through the data collected by `sim.py`, various metrics will be extrapolated.

The script consists of 4 main functions: `get_latest_collection()`, `calculate_metrics()`, `publish_metrics()`, and `main()`. `get_latest_collection()` simply connects to the mongo db, and fetches the last available collection (based on run date). This collection will be fed to the `calculate_metrics()` function, which will extract post-match metrics from it.

```python
def calculate_metrics(collection_name):
    try:
        collection = db[collection_name]

        metrics = {}

        for player_id in range(1, 16):

            pipeline = [
                {"$match": {"player_id": player_id}},
                {"$sort": {"timestamp": DESCENDING}},
                {"$limit": 1}
            ]
            result = list(collection.aggregate(pipeline))

            if result:
                player_data = result[0]

                velocity = player_data["gps"]["velocity"]
                elapsed_time = player_data["elapsed_time"]
                calories = player_data["calories_consumed"]["calories"]
                heart_rate = player_data["heart_rate"]["heart_rate"]
                body_temperature = player_data["temperature"]["body_temperature"]
                systolic = player_data["blood_pressure"]["systolic"]
                diastolic = player_data["blood_pressure"]["diastolic"]
                impact_count = player_data["impacts"]["impact_count"]
                impact_force = player_data["impacts"]["impact_force"]

                distance_traveled_km = round(velocity * (elapsed_time / 60.0), 2)
                average_impact_force = round(impact_force / impact_count, 2) if impact_count > 0 else 0

                impact_to_play_ratio = round(random.uniform(0.1, 2.0), 2)
                velocity_variability = round(random.uniform(0.5, 3.0), 2)
                max_heart_rate = int(random.uniform(160, 210))
                impact_severity_index = round(random.uniform(1, 10), 1)
```

```python
                metrics[player_id] = {
                    "player_id": player_id,
                    "average_velocity": round(velocity, 2),
                    "distance_traveled_km": distance_traveled_km,
                    "calories_consumed": round(calories, 2),
                    "heart_rate": heart_rate,
                    "body_temperature": round(body_temperature, 1),
                    "blood_pressure": {
                        "systolic": systolic,
                        "diastolic": diastolic
                    },
                    "impacts": {
                        "impact_count": impact_count,
                        "average_impact_force": average_impact_force
                    },
                    "impact_to_play_ratio": impact_to_play_ratio,
                    "velocity_variability": velocity_variability,
                    "max_heart_rate": max_heart_rate,
                    "impact_severity_index": impact_severity_index
                }
            else:
                metrics[player_id] = {
                    "average_velocity": 0.0,
                    "distance_traveled_km": 0.0,
                    "calories_consumed": 0.0,
                    "heart_rate": 0,
                    "body_temperature": 0.0,
                    "blood_pressure": {
                        "systolic": 0,
                        "diastolic": 0
                    },
                    "impacts": {
                        "impact_count": 0,
                        "average_impact_force": 0.0
                    },
                    "impact_to_play_ratio": 0.0,
                    "velocity_variability": 0.0,
                    "max_heart_rate": 0,
                    "impact_severity_index": 0.0
                }

        return metrics

    except Exception as e:
        print(f"Error querying MongoDB: {e}")
        return None
```

After calculating the metrics, the same method used in `sim.py` is called, but changing the MQTT topic. (To read in detail about the MQTT hierarchy used, refer to the file "**MQTT topic Hierarchy**").

Finally, the whole thing is called in the `main()` function.

```python
def main():
    latest_collection = get_latest_collection()
    if latest_collection:
        # Calculate metrics from the latest collection
        metrics = calculate_metrics(latest_collection)
        if metrics:
            # Publish metrics via MQTT
            publish_metrics(metrics)
        else:
            print("No metrics calculated.")
    else:
        print("No latest collection found.")
```

3.2 MQTT

The MQTT protocol was chosen to ensure real-time sharing of simulated data with the different dashboards, while providing high security in communications. To manage the MQTT network, the Mosquitto broker was used, configured to operate on localhost, using port 1883 (default MQTT port), as defined in the global variables of the code. This setup enables efficient and secure transmission of data between the various system components, making it possible to monitor simulated metrics in real time without exposing the information flow to external risks. Here is an example of a log of an MQTT connection to NODE-RED -

```
2024-08-29 15:05:06 1724936706: mosquitto version 2.0.18 starting
2024-08-29 15:05:06 1724936706: Config loaded from /mosquitto/config/mosquitto.conf.
2024-08-29 15:05:06 1724936706: Opening ipv4 listen socket on port 1883.
2024-08-29 15:05:06 1724936706: Opening ipv6 listen socket on port 1883.
2024-08-29 15:05:06 1724936706: mosquitto version 2.0.18 running
2024-08-29 15:05:10 1724936710: New connection from 172.20.0.5:48402 on port 1883.
2024-08-29 15:05:10 1724936710: New client connected from 172.20.0.5:48402 as nodered_03526a0a345653c3 (p2, c1, k60).
```

3.2.1 MQTT config functions

As mentioned earlier, MQTT needs two config functions to be implemented to work properly: `on_connect()` and `on_message()`.

The `on_connect()` function manages the process of connecting to the MQTT broker. Upon a successful connection, it subscribes to the necessary topics for each player to gather their sensor data. If the connection is unsuccessful, an error message is displayed.

The `on_message()` function handles the incoming MQTT messages. It verifies whether the message's topic corresponds to a player's sensor data topic. If it matches, it extracts and simplifies the information, keeping only the key GPS coordinates. The simplified data is then published to a separate topic dedicated to coordinates.

```python
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print(f"Connected to MQTT broker with result code {rc}")
        # Subscribe to topics for all players
        for player_id in range(1, 16):
            client.subscribe(MQTT_TOPIC_TEMPLATE.format(player_id))
    else:
        print(f"Failed to connect to MQTT broker with result code {rc}")

def on_message(client, userdata, msg):
    for player_id, role in ROLES.items():
        topic = MQTT_TOPIC_TEMPLATE.format(player_id)
        if msg.topic == topic:
            data = json.loads(msg.payload.decode())

            simplified_data = {
                "player_id": data["player_id"],
                "role": data["role"],
                "gps": {
                    "x": data["gps"]["x"],
                    "y": data["gps"]["y"]
                }
            }

            coordinates_topic =
MQTT_COORDINATES_TOPIC_TEMPLATE.format(player_id)
            client.publish(coordinates_topic, json.dumps(simplified_data))
            print(f"Published coordinates for Player {player_id} to topic
'{coordinates_topic}'")
```
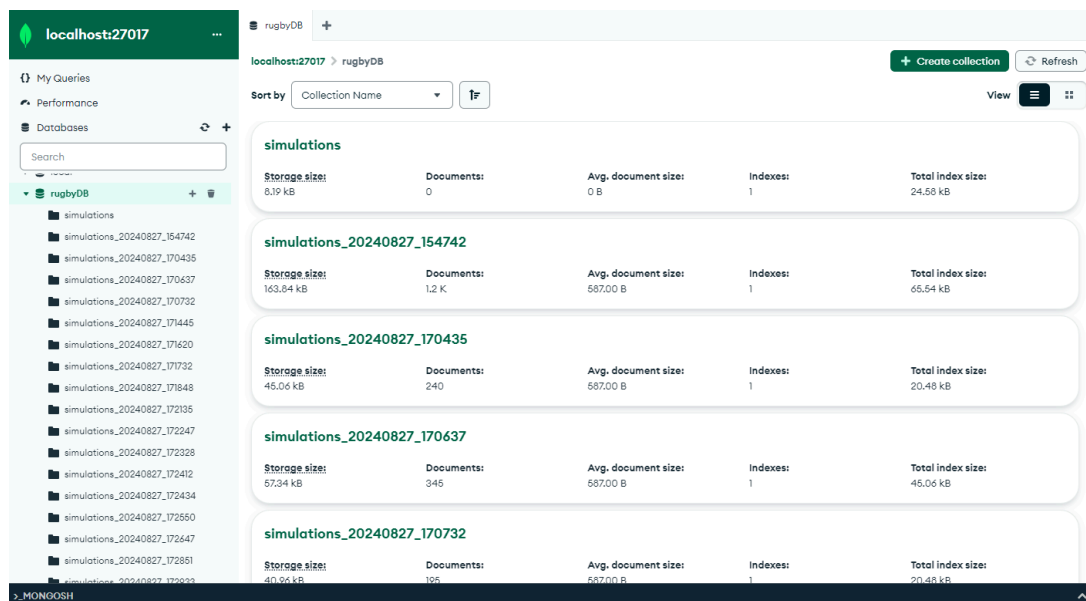
3.3 MONGODB

MongoDB, a document-oriented NoSQL database designed to handle large amounts of data in a flexible and scalable way, was chosen as the database. Unlike traditional relational databases, MongoDB stores data in JSON documents that can contain nested structures and vary from record to record. This structure makes it easy to adapt to unstructured or semi-structured data. (As in our case). Below is a snippet of the database as seen by mongo compass -

As can be seen, the db is populated by many collections which in turn have documents inside, representing the data from our simulations.

The `realtime_metrics.py` script retrieves from here, based on the timestamp of the simulation, the one to be used for calculating the post-match metrics.

Here the global variables used to connect to mongodb -

```
MONGO_URI = "mongodb://localhost:27017/"
DATABASE_NAME = "rugbyDB"
BASE_COLLECTION_NAME = "simulations"

client = MongoClient(MONGO_URI)
db = client[DATABASE_NAME]
```

3.4 Node-Red

The last tool used in this project is Node-RED, In our case, Node-RED was used to simulate a rugby activity, managing the collection and processing of real-time data from devices connected to players. Node-RED orchestrated the flow of this data, simulating a distributed sensor network and integrating the various inputs into a single workflow. This data was then forwarded to the MQTT broker, allowing updated real-time metrics to be displayed on dashboards.
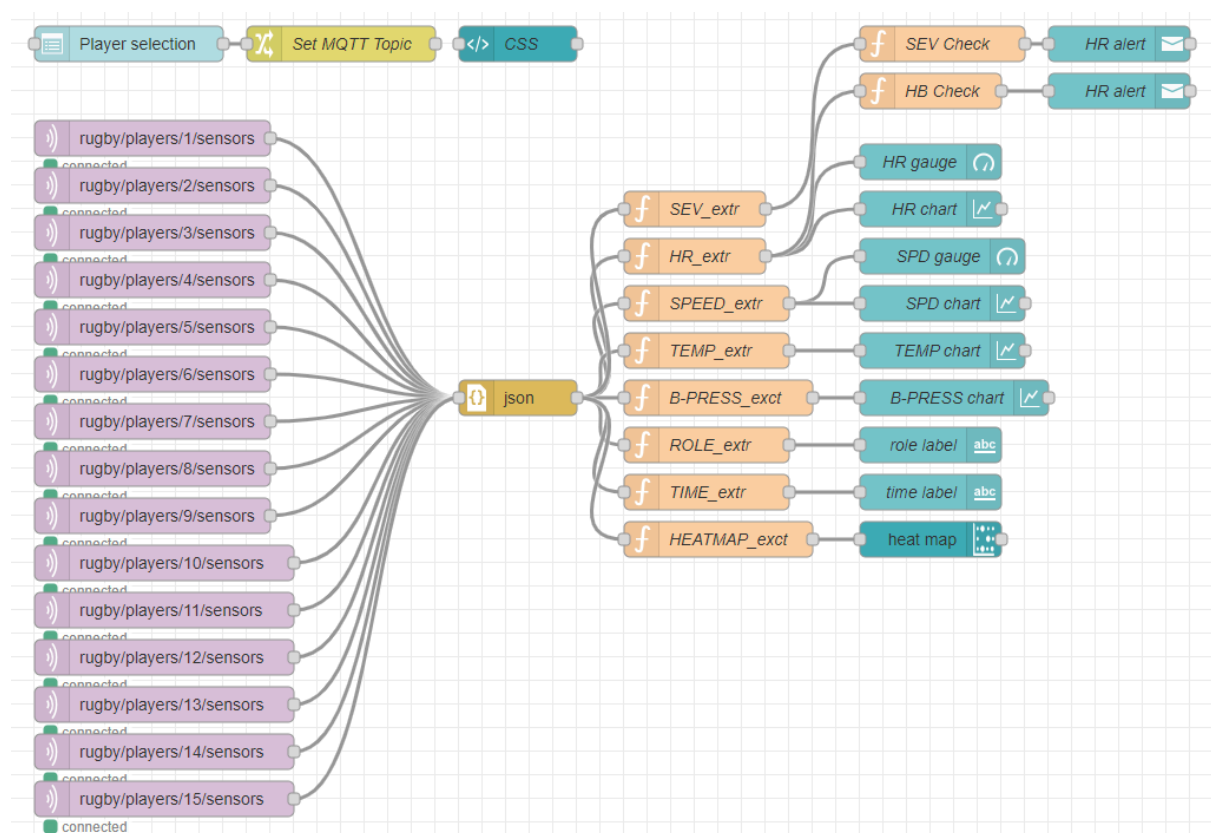
The image above shows one of the flows implemented in Node-Red, starting from left to right, the first element to be analyzed are the MQTT brokers (called MQTT in), each of which is connected on the channel of the specific player.

This data is then processed in JSON format (in case it was not already) and after that fed to manipulation functions, which will have the task of extracting only the data needed for that specific UI component.

Other important components are alerts, which will notify coaches in two cases: 1) If a player's heart rate exceeds a threshold limit (**210** BPM); 2) If a concussion occurs during an impact (impact index > **200**).

Finally, a dropdown is made available for the selection of the player to be analyzed (only in this specific view).

There are in total 4 flows: *DATA-GATHERING*, *PLAYER*, *TEAM* and *END GAME*:

- **DATA-GATHERING**: test flow to see how node-red could access the mongo db without going through MQTT, not needed for the purposes of the project but we thought it was nice to leave it.

- **PLAYER** (**LIVE**): flow analyzed in the previous image, allows you to choose a specific player to analyze and observe his metrics.

- **TEAM** (**LIVE**): general team flow, here you can analyze all players' metrics in real time.

- **END GAME** (**POST-MATCH**): flow that works with the metrics derived from the simulation. It allows you to observe the IDX and STIME of each player after the activity.

NB. flows labeled "*LIVE*" work with the `sim.py` script, those with "*POST-MATCH*" work using `realtime_metrics.py`.

4. Conclusions

The successful creation and integration of the simulation system for monitoring rugby players' movements, vital signs, and energy expenditure highlight the effectiveness and adaptability of current IoT technologies and data management solutions. By utilizing MQTT for efficient real-time data transfer, Node-RED for interactive data display, and MongoDB for reliable data storage and analysis, this project demonstrates a thorough approach to sports analytics that can greatly benefit coaches, players, and fans.

Key Achievements:

- **Scalability and Flexibility:** The system's modular design, which separates components for data generation, communication, visualization, and storage, ensures that it can be easily scaled and adapted. New features can be added with little disruption, and the system can be expanded to include more players or different sports.

- **Real-Time Processing and Feedback:** The system's real-time processing capabilities allow users to receive instant feedback on game and player performance, which is crucial for making timely decisions during matches and for comprehensive post-match evaluations.

- **User-Friendly Interface:** The GUI, driven by Node-RED, makes complex data easy to understand and use. This user-friendly interface allows coaches and analysts to interpret the data effectively, making it practical for training and strategic planning.

Future Directions:

- **Enhanced Analytics:** Future upgrades could involve more sophisticated analytical models and machine learning algorithms to predict player performance and potential injuries, providing valuable proactive insights.

- **Integration with Wearable Technologies:** Adding wearable sensors could yield even more precise and detailed data, improving the simulation's accuracy and usefulness.

In conclusion, this project showcases the transformative impact of combining IoT, real-time data processing, and advanced analytics in sports. It opens new possibilities for improving performance, strategic planning, and fan engagement, contributing to the progress of sports technology. The thoughtful design and integration of various tools ensure that the system is not only a technological milestone but also a practical solution for modern sports analytics.

[CHECK FULL CODE IN THE REPO https://github.com/cristiandenicola/RUGBY-360]