Navixy IoT Logic API Documentation

Generated from https://developers.navixy.com/docs/iot-logic-api/overview

Overview

> *BETA Version!*
> Now the early access of IoT Logic's API is implemented, which means possible changes in the near future. Feel free to try the functionality, however, you may need to introduce changes in your applications reflecting the API functionality updates. Stay tuned!

Introduction

Navixy IoT Logic is a no-code/low-code tool that enables seamless IoT data processing and integration. Its API provides programmatic access to create, manage, and optimize data flows between IoT devices and destination systems without requiring extensive development resources.

Purpose and core capabilities

Navixy IoT Logic functions as a data flow manager that:

- Receives information from devices connected to the platform
- Decodes and converts data in real-time
- Sends processed data to other platforms and services
- Enables building complex flows with nodes responsible for specific data processing tasks
- Standardizes telematics data through the Navixy Generic Protocol

The **IoT Logic API** allows developers and system integrators to programmatically implement these capabilities, making it effective for organizations that need to:

- Work efficiently with decoded device data
- Apply flexible data transformation to match specific business needs
- Monitor and troubleshoot data streams

- Create consistent data flows across multiple devices and protocols

Key concepts

Navixy IoT Logic operates based on two fundamental components that work together to process device data:

Flow

A **Flow** is the foundation for all data logic in the product. It defines how data moves through stages of reception, enrichment, and transmission. Each flow consists of connected nodes that determine what happens to the data at each processing stage.

Key characteristics of flows: - Flows can be enabled or disabled to control data processing - Every flow requires at least one data source and one output endpoint - Each device can only be assigned to one flow at a time - Flows process data in real-time as it arrives from devices

Nodes

Nodes are the functional elements of a **flow**, with each node handling a specific stage of the data lifecycle. There are three primary types of nodes:

- **Data Source node**: Receives data from M2M devices and serves as the entry point for all device data
- **Initiate Attribute node**: Processes and enriches incoming data, including creating new calculated attributes trough mathematical operations in Navixy IoT Logic Expression Language
- **Output Endpoint node**: Transmits data to target systems using the Navixy Generic Protocol. This node can be configured to use different endpoint types:
- **Default endpoint**: Pre-configured destination for sending data to the Navixy platform
- **MQTT endpoint**: Configurable connection for sending data to third-party systems and services

Nodes are connected through transitions (`edges`) that define the path data follows through the flow.

Data flow architecture

The following screenshot from IoT Logic UI illustrates the basic architecture of a flow in IoT Logic:

This represents a simple linear flow where: 1. The **Data Source** node collects telemetry from selected devices 2. The **Initiate Attribute** node processes and enriches this data 3. The **Default Output Endpoint** node delivers the transformed data to its destination - Navixy platform

More complex architectures can be created by: - Adding multiple data source nodes to process different device types - Chaining multiple attribute nodes for multi-stage data processing - Including several output endpoints to deliver data to multiple destinations outside Navixy simultaneously

Quick start for IoT Logic API

To ensure a clear picture of the basic IoT Logic API capabilities, let's create your first flow.

The following example demonstrates how to create a complete flow with three nodes that sends data to Navixy. This flow will: 1. Collect data from specified devices 2. Calculate temperature in Fahrenheit from Celsius readings 3. Send the enriched data to the Navixy platform

Step 1: Authentication

First, authenticate to obtain a session token. To do it, send a POST request to the user authentication endpoint `{baseURL}/v2/user/auth` providing your account's login and passord as parameters:

```
curl -X POST "https://your.server.com/v2/user/auth" \
  -H "Content-Type: application/json" \
  -d '{
    "login": "your_email_or_username",
    "password": "your_password"
  }'
```

Response (example):

```
{
  "success": true,
  "hash": "22eac1c27af4be7b9d04da2ce1af111b"
}
```

Copy the `hash` value from the response.

> For more details on how to authenticate your requests, see Authentication.

Step 2: Create a complete flow with nodes and connections

Create a flow with all nodes and connections in a single request:

```
curl -X POST "https://your.server.com/iot/logic/flow/create" \
  -H "Content-Type: application/json" \
  -H "Authorization: NVX hash_value" \
  -d '{
    "flow": {
      "title": "Basic Temperature Monitoring",
      "enabled": true,
      "nodes": [
        {
          "id": 1,
          "type": "data_source",
          "enabled": true,
          "data": {
            "title": "Fleet Vehicles",
            "sources": [394892, 394893, 394894]
          },
          "view": {
            "position": {
              "x": 50,
              "y": 100
            }
          }
        },
        {
          "id": 2,
          "type": "initiate_attributes",
          "data": {
            "title": "Temperature Conversion",
            "items": [
              {
                "name": "temperature_f",
                "value": "value(\"temperature\")*1.8 + 32",
                "generation_time": "genTime(\"temperature\", 0, \"valid\")",
                "server_time": "now()"
              }
            ]
          },
```

```
          "view": {
            "position": {
              "x": 300,
              "y": 100
            }
          }
        },
        {
          "id": 3,
          "type": "output_endpoint",
          "enabled": true,
          "data": {
            "title": "Navixy Platform",
            "output_endpoint_type": "output_navixy"
          },
          "view": {
            "position": {
              "x": 550,
              "y": 100
            }
          }
        }
      ],
      "edges": [
        {
          "from": 1,
          "to": 2
        },
        {
          "from": 2,
          "to": 3
        }
      ]
    }
  }'
```

Response (example):

```
{
  "success": true,
  "id": 123
}
```

Parameters explained

- **Flow entity**: The main container defining a complete data processing pipeline
- `title`: Names your flow for easier identification
- `enabled`: When true, flow begins processing data immediately after creation
- **Nodes**: Functional components that each handle a specific step in data processing
- **Node 1 (data_source)**:
  - Entry point collecting data from devices (IDs: 394892, 394893, 394894)
  - Unique ID within the flow for connection references
  - Position coordinates control UI display location
- **Node 2 (initiate_attributes)**:
  - Transforms data with custom calculations
  - Creates new "temperature_f" attribute using formula
  - Uses timestamps for data validity tracking
- **Node 3 (output_endpoint)**:
  - Destination for processed data
  - Type "output_navixy" sends to Navixy platform
  - Final step in the processing pipeline
- **Edges**: Define connections between nodes
- Reference nodes by their IDs to create the processing sequence
- Create a clear path for data to follow from source to destination

> This single request creates a complete flow that: 1. Collects data from three specific devices (IDs: 394892, 394893, 394894) 2. Converts temperature values from Celsius to Fahrenheit 3. Transmits all data, including the new calculated attribute, to the Navixy platform
>
> The success response includes the ID of the newly created flow, which you can use for future operations like updating the flow or adding additional nodes.
>
> You can expand this example by adding more devices, creating additional calculated attributes, or configuring MQTT endpoints to send data to external systems.

Authentication

Authentication

The Navixy IoT Logic API supports two authentication methods:

1. **User session hash**: The basic, mandatory authentication method obtained through user login
2. **API key**: The recommended, more secure method for ongoing API access and integrations

1. User session hash (basic authentication)

A session hash is the fundamental authentication token in the Navixy system and serves as the mandatory starting point for all authentication workflows.

Use session hashes for:

- Initial system access
- Creating API keys
- User interface operations
- Short-term scripts

> Session hash limitations: - Expires after periods of inactivity (typically 30 minutes) - Invalidated when logging out or changing password - Requires periodic renewal for longer sessions - Less secure for automated processes

Obtaining a session hash

To get a session hash, send a POST request to the user authentication endpoint `{baseURL}/v2/user/auth` providing your account's login and passord as parameters:

```
curl -X POST "https://your.server.com/v2/user/auth" \
  -H "Content-Type: application/json" \
  -d '{
    "login": "your@email.com",
    "password": "your_password"
  }'
```

Successful response:

```
{
  "success": true,
  "hash": "22eac1c27af4be7b9d04da2ce1af111b"
}
```

Copy the `hash` parameter value and save it, you will use it for authenticating your further requests.

2. API keys (recommended authentication)

API keys are a more stable and secure authentication method, it is recommended for long-term use with all production integrations and automated systems. Here's a comparison with session hash to highlight API key advantages:

Feature	API Keys	Session Hashes
Expiration	Don't automatically expire	Expire after inactivity
Password changes	Not affected	Immediately invalidated
User logout	Not affected	Immediately invalidated
Credential storage	Only store the API key	Must store/access username & password
Revocation	Individual keys can be revoked	All sessions terminated together
Integration segmentation	One key per integration	All use same credentials
Periodic renewal	Not required	Required for long sessions

> API key limitations: - Every user account may have up to 20 API keys - IOnly account Owners have access to API keys functionality - Requires a valid session hash to initially create a key

Creating API keys

Method 1: Through the web interface

1. **Login to the Web Interface**:
    - Access the Navixy user interface via the web.
    - Use your credentials to log in.

2. **Navigate to the API Key Section**:
    - Once logged in, click on your username.
    - Find and select the `API Keys` section.
3. **Generate a New API Key**:
    - Click on the plus button on the top left corner.
    - Provide a name for the API key to easily identify it later.
    - Click `Save`.
4. **Copy the API Key Hash**:
    - Once the key is generated, you will see it in the API keys table, including the label, creation date, and the API key hash.
    - Use this hash in your API requests.

Method 2: Through the API (requires session hash)

Send a POST request to `{baseURL}/v2/user/api_key/create` providing your session `hash` and new API key `title` as parameters:

```
curl -X POST "https://api.eu.navixy.com/v2/user/api_key/create" \
  -H "Content-Type: application/json" \
  -d '{
    "hash": "your_session_hash",
    "title": "Integration API Key"
  }'
```

Successful response:

```
{
  "success": true,
  "value": "new_api_key_hash_here"
}
```

Using authentication in API requests

You can include your authentication credentials in requests trough three different methods. Each method has specific use cases and security considerations.

1. As a request header (recommended)

Include the hash in the `Authorization` header

```
Authorization: NVX your_hash_or_api_key
```

```
Authorization: NVX 22eac1c27af4be7b9d04da2ce1af111b
```

```
# missing space after NVX

Authorization: NVX22eac1c27af4be7b9d04da2ce1af111b
```

This is the most secure method and follows API best practices. It keeps authentication separate from request data and works well with all HTTP methods.

2. In the request body

Include your authentication directly in the JSON body of your request as `hash` parameter:

```
{
  "hash": "your_hash_or_api_key",
  "param1": "value1"
}
```

This method works only with POST requests and mixes authentication with request parameters. It can be useful for testing but is less ideal for production use.

3. As a query parameter (testing only)

Append your authentication `hash` to the URL as a query parameter:

```
https://api.navixy.com/iot/logic/flow/list?hash=your_hash_or_api_key
```

> This method is for testing purposes only as it exposes your authentication credentials in URLs, server logs, and browser history. Never use this method in production environments.

Managing API keys

These endpoints allow you to manage your API keys through the API itself using a valid session hash.

Listing API keys

Get a complete list of all API keys associated with your account:

```
curl -X POST "https://api.eu.navixy.com/v2/user/api_key/list" \
  -H "Content-Type: application/json" \
  -d '{
    "hash": "your_session_hash"
  }'
```

This endpoint returns all your active API keys with their labels and creation dates, helping you manage and audit your integrations.

Deleting an API key

Immediately revoke an API key when it's no longer needed or if it may have been compromised:

```
curl -X POST "https://api.eu.navixy.com/v2/user/api_key/delete" \
  -H "Content-Type: application/json" \
  -d '{
    "hash": "your_session_hash",
    "api_key": "api_key_to_delete"
  }'
```

Any application using this key will immediately lose access and need to be reconfigured with a new key.

Recommended authentication flow

After understanding the available authentication methods, follow this recommended progression for integrating with the Navixy IoT Logic API:

1. **Initial authentication**

2. Authenticate with username/password to obtain a session hash
3. This step is mandatory and cannot be skipped

4. Session hashes provide temporary access needed to create API keys

5. **API key creation**

6. Use the session hash to create one or more API keys
7. Create separate API keys for different integrations

8. Give each key a descriptive name to identify its purpose

9. **Ongoing API access**

10. Use API keys for all subsequent API calls
11. Store API keys securely in your integration

12. Implement the API key in headers for maximum security

13. **API key management**

14. Periodically rotate API keys for security
15. Revoke compromised or unused API keys
16. Monitor key usage for unusual patterns

Authentication errors

Understanding authentication errors helps you troubleshoot issues and implement proper error handling in your integration:

```
{
  "success": false,
  "status": {
    "code": 3,
    "description": "Access denied"
  }
}
```

Common authentication error codes: - **Code 3: Access denied** - Your API key or session hash is invalid or has been revoked - **Code 4: Invalid session** - Your session hash has expired due to inactivity - **Code 7: User account blocked** - The associated user account is suspended or disabled

Your integration should handle these errors appropriately, possibly by prompting for re-authentication or alerting administrators about potential issues.

Best practices

Follow these guidelines to ensure secure and effective authentication:

1. **Start with session hash authentication** to access the system initially - this is a mandatory first step
2. **Create and use API keys** for all ongoing integration needs - they provide more stable, secure access
3. **Use separate keys** for different integrations for better management and security isolation
4. **Add descriptive labels** to easily identify the purpose of each key when viewing your key list
5. **Implement key rotation** as part of your security protocols to limit the impact of potential key exposure
6. **Revoke unused or compromised keys** promptly to maintain security of your account
7. **Monitor key usage** and investigate unexpected activities that might indicate unauthorized access
8. **Transmit authentication credentials** only over HTTPS to prevent interception of keys
9. **Store credentials securely** server-side, never in client-side code or public repositories
10. **Implement proper error handling** for authentication failures, including automatic recovery where appropriate

Technical reference

Technical reference

API environments

The Navixy IoT Logic API is available on multiple regional platforms to optimize performance and comply with data residency requirements.

Base URLs

Region	Base URL	Data Location
Europe	`https://api.eu.navixy.com`	European data centers
Americas	`https://api.us.navixy.com`	US-based data centers

Environment selection

Choose the environment that: 1. Is geographically closest to your operations (to minimize latency) 2. Complies with your data residency requirements 3. Matches your existing Navixy platform subscription

Both environments offer identical API functionality, but may differ in: - Response times based on your geographic location - Data storage location (important for compliance with regulations like GDPR) - Maintenance windows and update schedules

Authentication

Authentication for the Navixy IoT Logic API uses API keys or user session hashes. For detailed information about authentication methods, obtaining API keys, and best practices, please refer to the Authentication documentation.

Request formats

HTTP methods

Method	Usage	Examples
`GET`	Retrieving information	Flow listing, flow details

Method	Usage	Examples
`POST`	Creating, updating, deleting resources	Creating, updating and deleting flows and nodes

> Some read operations use POST requests with a request body instead of GET with query parameters. Always check the endpoint documentation for the correct method.

Content type

All requests and responses use JSON format:

```
Content-Type: application/json
```

All request bodies must be valid JSON objects. The API will return a 400 Bad Request response for malformed JSON.

Date and time formats

All timestamps in the API use ISO 8601 format in UTC timezone:

```
YYYY-MM-DDThh:mm:ssZ
```

Example: `2025-04-08T14:30:00Z` represents April 8, 2025, at 2:30 PM UTC.

Response structure

All API responses follow a consistent JSON format.

Success responses

Success responses always include a `success: true` field and may include additional data:

```
{
  "success": true,
  "value": {
    // Requested data or resource
```

```
  }
}
```

For list operations:

```
{
  "success": true,
  "list": [
    // Array of requested items
  ]
}
```

Error responses

Error responses include `success: false` and a `status` object with details:

```
{
  "success": false,
  "status": {
    "code": 2,
    "description": "Invalid parameters"
  }
}
```

Common error codes

Code	Description	Typical Causes
1	Database error	Internal server issue, data format problem
2	Invalid parameters	Missing required fields, incorrect data types
3	Access denied	Invalid API key or permissions
4	Resource not found	Incorrect ID for flow or endpoint
5	Rate limit exceeded	Too many requests in short time period
6	Validation error	Flow structure issues, invalid connections

Endpoint reference

> For OpenAPI reference, see Navixy IoT Logic API

The Navixy IoT Logic API provides the following endpoints for managing flows and endpoints:

Flow management endpoints

Endpoint	Method	Description	Key Parameters
`/iot/logic/flow/create`	POST	Create a new flow	`flow` object with title, nodes, edges
`/iot/logic/flow/read`	GET	Read an existing flow	`flow_id`
`/iot/logic/flow/update`	POST	Update an existing flow	`flow` object with id, title, nodes, edges
`/iot/logic/flow/delete`	POST	Delete a flow	`flow_id`
`/iot/logic/flow/list`	GET	List all flows	none

Node management endpoints

Endpoint	Method	Description	Key Parameters
`/iot/logic/flow/endpoint/create`	POST	Create a new endpoint	`endpoint` object with type, title, properties
`/iot/logic/flow/endpoint/read`	POST	Read an existing endpoint	`endpoint_id`
`/iot/logic/flow/endpoint/update`	POST	Update an existing endpoint	`endpoint` object with id and updated fields
`/iot/logic/flow/endpoint/delete`	POST	Delete an endpoint	`endpoint_id`
`/iot/logic/flow/endpoint/list`	POST	List all endpoints	none

Flow architecture

Flows in Navixy IoT Logic follow a directed graph architecture with specific requirements and constraints.

Basic requirements

Requirement	Description
Input nodes	Every flow must have at least one data source node
Output nodes	Every flow must have at least one output endpoint node
Node IDs	Each node has a unique ID within its flow (not globally unique)
Connections	Edges define directional data flow between nodes
Multiple connections	Nodes can have multiple incoming and outgoing connections
No cycles	Circular references are not supported

Flow validation

When creating or updating flows, the API performs several validation checks:

1. Node IDs must be unique within a flow
2. Each edge must reference valid node IDs
3. The flow must be acyclic (no circular references)
4. All nodes must be reachable from data sources
5. All required fields must be present for each node type

Flow states

State	Description	Effect
`enabled: true`	Active flow	The flow processes data in real-time
`enabled: false`	Inactive flow	Data is not processed by this flow

Individual nodes can also be enabled or disabled, allowing for partial flow activation.

Node positioning

The `view` property in nodes is used for visual representation in UI tools:

```
"view": {
  "position": {
    "x": 150,
    "y": 50
  }
}
```

While optional for API functionality, including this data helps maintain visual layout when editing flows later.

Endpoint management

Endpoint status options

Status	Description	Use Case
`active`	Fully operational	Normal operation, actively sending/receiving data
`suspend`	Temporarily paused	Maintenance periods, temporary disconnection
`disabled`	Permanently disabled	Decommissioned endpoints, security concerns

Rate limiting

To ensure system stability for all customers, the platform limits API requests to 50 requests per second per user and per IP address (for applications serving multiple users). These limits are applied based on user session hash and API keys.

> When a rate limit is exceeded, the API returns a 429 Too Many Requests status code with a specific error message

To avoid rate limiting issues: 1. Implement exponential backoff for retries 2. Batch operations where possible 3. Cache frequently accessed data 4. Distribute non-urgent requests over time

Node reference

1. Data source node (`data_source`)

This node specifies which devices will send data to your flow. It's the entry point of all data flows.

Data source node structure

```
{
  "id": 1,
  "type": "data_source",
  "title": "Your Title Here",
  "enabled": true,
  "data": {
    "sources": [12345, 67890]  // Device IDs
  },
  "view": {
    "position": { "x": 50, "y": 50 }
  }
}
```

Key properties

Property	Type	Required	Description
`id`	integer	Yes	Unique identifier within the flow
`type`	string	Yes	Must be `"data_source"`
`title`	string	Yes	Human-readable name for the node
`enabled`	boolean	Yes	Whether this node processes data
`data.sources`	array	Yes	Array of device IDs to collect data from

Usage notes

- The `data_source` node type is required in every flow
- Multiple devices can be specified in the `sources` array

- Each device is identified by its numeric ID in the Navixy system
- A flow can have multiple data source nodes for different device groups

2. Data processing node (`initiate_attribute`)

This node transforms raw data into meaningful information. It allows for creating new attributes or modifying existing ones through expressions.

Data processing node structure

```
{
  "id": 2,
  "type": "initiate_attributes",
  "title": "Your Title Here",
  "data": {
    "items": [
      {
        "name": "attribute_name",
        "value": "expression",
        "generation_time": "now()",
        "server_time": "now()"
      }
    ]
  },
  "view": {
    "position": { "x": 150, "y": 50 }
  }
}
```

Key properties

Property	Type	Required	Description
`id`	integer	Yes	Unique identifier within the flow
`type`	string	Yes	Must be `"initiate_attributes"`
`title`	string	Yes	Human-readable name for the node
`data.items`	array	Yes	Array of attribute definitions
`data.items[].name`	string	Yes	The attribute identifier

Property	Type	Required	Description
`data.items[].value`	string	Yes	Mathematical or logical expression
`data.items[].generation_time`	string	Yes	When the data was generated
`data.items[].server_time`	string	Yes	When the server received the data

Expression language

For calculations IoT Logic API uses Navixy IoT Logic Expression Language. Here's a guick reference:

Feature	Operators/Examples	Description
Mathematical operators	`+`, `-`, `*`, `/`, `%`	Basic arithmetic operations
Functions	`now()`, `sqrt()`, `pow()`, `abs()`	Built-in functions
Attribute references	`speed`, `fuel_level`, `analog_1`	Reference to device attributes

Expression examples

Scenario	Expression	Description
Temperature conversion	`(temperature_f - 32) * 5/9`	Convert Fahrenheit to Celsius
Distance calculation	`sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2))`	Calculate distance between points
Time-based condition	`hour(time) >= 22 || hour(time) <= 6 ? 'night' : 'day'`	Determine day/night status

3. Output endpoint node (`output_endpoint`)

This node defines where your data will be sent. It's the termination point for data flow paths.

Output types

The output endpoint node supports different destination types:

Default output for sending data to Navixy

```
{
  "id": 3,
  "type": "output_endpoint",
  "title": "Your Title Here",
  "enabled": true,
  "data": {
    "output_endpoint_type": "output_navixy"
  },
  "view": {
    "position": { "x": 250, "y": 50 }
  }
}
```

MQTT output for sending data to to external systems

```
{
  "id": 4,
  "type": "output_endpoint",
  "title": "Your Title Here",
  "enabled": true,
  "data": {
    "output_endpoint_type": "output_mqtt_client",
    "output_endpoint_id": 45678
  },
  "view": {
    "position": { "x": 250, "y": 150 }
  }
}
```

Key properties

Property	Type	Required	Description
`id`	integer	Yes	Unique identifier within the flow
`type`	string	Yes	Must be `"output_endpoint"`
`title`	string	Yes	Human-readable name for the node

Property	Type	Required	Description
`enabled`	boolean	Yes	Whether this node processes data
`data.output_endpoint_type`	string	Yes	Type of output destination (`"output_navixy"` or `"output_mqtt_client"`)
`data.output_endpoint_id`	integer	For MQTT only	Reference to a previously created endpoint

Usage notes

- Every flow must have at least one output endpoint node to be functional
- The `output_navixy` type doesn't require a referenced endpoint (built-in)
- The `output_mqtt_client` type requires an `output_endpoint_id` referencing a previously created endpoint
- Multiple output nodes can be used to send the same data to different destinations
- Output nodes are "terminal" - they don't connect to any downstream nodes

Endpoint reference

Endpoint types

Type	Description	Use Case
`input_navixy`	Default input from Navixy platform	Receiving data from Navixy devices
`output_navixy`	Default output to Navixy platform	Sending processed data back to Navixy
`output_mqtt_client`	External MQTT broker connection	Integrating with third-party systems

MQTT endpoint properties

Property	Type	Required	Description	Example
`protocol`	string	Yes	Protocol of messages	`"NGP"` (Navixy Generic Protocol)
`domain`	string	Yes	MQTT broker domain/IP	`"mqtt.example.com"`
`port`	integer	Yes	MQTT port	`1883`
`client_id`	string	Yes	Client identifier	`"navixy-client-1"`
`qos`	integer	Yes	Quality of Service (0 or 1)	`1`
`topics`	array of strings	Yes	Topic names	`["iot/data"]`
`version`	string	Yes	MQTT version	`"5.0"` or `"3.1.1"`
`use_ssl`	boolean	Yes	Whether to use SSL	`true`
`mqtt_auth`	boolean	Yes	Whether auth is required	`true`
`user_name`	string	Only if `mqtt_auth: true`	MQTT username	`"mqtt_user"`
`user_password`	string	Only if `mqtt_auth: true`	MQTT password	`"mqtt_password"`

MQTT QoS levels

Level	Description	Use Case
QoS 0	"At most once" delivery (fire and forget)	High-volume, non-critical data where occasional loss is acceptable

Level	Description	Use Case
QoS 1	"At least once" delivery (acknowledged delivery)	Important messages that must be delivered, can handle duplicates
QoS 2	Not currently supported by the API	-

MQTT protocol versions

Version	Description	Features
MQTT 3.1.1	Widely supported version	Basic pub/sub functionality, broad broker compatibility
MQTT 5.0	Newer version with enhanced features	Message expiry, topic aliases, shared subscriptions, reason codes

Best practices

Flow design

1. **Plan your flow design** before implementation
2. Sketch your flow structure including all nodes and connections
3. Identify device sources and required data transformations
4. Determine appropriate output endpoints
5. **Create endpoints first** before referencing them in flows
6. MQTT endpoints must exist before they can be referenced
7. Test endpoint connectivity independently before adding to flows
8. **Use descriptive titles** for both flows and nodes
9. Clear naming helps with maintenance and troubleshooting
10. Include purpose or function in the title
11. **Test incrementally** by adding one component at a time

12. Start with a minimal viable flow and expand
13. Test each node's functionality before adding complexity
14. **Monitor data flow** after implementation
15. Verify data is flowing as expected
16. Check for proper attribute transformation

Data processing

1. **Keep expressions simple** where possible
2. Complex expressions are harder to debug and maintain
3. Consider using multiple nodes for complex transformations
4. **Use consistent naming conventions** for attributes
5. Follow a pattern like `category_attribute_unit`
6. For example: `engine_temperature_celsius`
7. **Document complex expressions** with clear comments
8. Keep documentation for non-obvious calculations
9. Include business logic explanations
10. **Consider data volume** when creating flows
11. High-frequency data may impact performance
12. Multiple outputs multiply processing requirements

Security

1. **Secure your MQTT connections** with SSL when possible
    - Enable `use_ssl: true` for production environments
    - Use secure ports (typically 8883 for MQTT over SSL)
2. **Rotate credentials** periodically for MQTT endpoints
    - Update credentials every 90 days or after personnel changes
    - Use strong, unique passwords for each endpoint

3. **Use unique client IDs** for each MQTT endpoint
    - Avoid connection conflicts by using distinctive IDs
    - Consider including account ID and purpose in the client ID
4. **Restrict topic access** on your MQTT broker
    - Limit permissions to only necessary topics
    - Use topic structures that enable precise access control

Maintenance

1. **Back up your flow configurations**
    - Store JSON responses from successful flow creation
    - Document flow purposes and interconnections
2. **Version your flows** with sequential titles
    - Include version numbers in flow titles
    - Maintain a changelog of modifications
3. **Regularly review active flows**
    - Audit flows to ensure they're still needed
    - Remove or disable unused flows to reduce overhead
4. **Keep a library of common patterns**
    - Reuse successful flow patterns across applications
    - Standardize approaches to similar problems

Data security considerations

When working with the Navixy IoT Logic API:

1. **API Keys**: Protect your API keys as they provide full access to your account
2. **MQTT Credentials**: Use strong passwords for MQTT authentication
3. **SSL**: Enable SSL (`use_ssl: true`) for MQTT connections whenever possible
4. **Data Privacy**: Be mindful of what device data you transmit to external systems
5. **Endpoint Security**: Regularly audit your endpoints and disable unused ones

Troubleshooting

Issue	Possible Solution
Flow not processing data	Check that both flow and all nodes have `enabled: true`
MQTT connection failing	Verify credentials, domain, port, and SSL settings
Data transformations not working	Test expressions in isolation to identify issues
Missing data in output	Ensure all required attributes are being processed
API errors	Double-check JSON syntax and required fields
Flow validation errors	Ensure node IDs are unique and edges connect valid nodes
Unexpected data format	Verify the calculation expressions in processing nodes
Missing fields in response	Check if you have all required permissions

Error handling

Common error codes and their solutions:

Error Code	Description	Solution
1	Database error	Check your data format and try again
2	Invalid parameters	Ensure all required fields are provided
3	Access denied	Check your API key and permissions
4	Resource not found	Verify IDs for flows and endpoints
5	Rate limit exceeded	Reduce request frequency
6	Validation error	Check flow structure for circular references

Error troubleshooting guide

When encountering API errors:

1. **Check request format**
2. Verify JSON syntax is correct
3. Ensure all required fields are included
4. **Validate authentication**
5. Confirm API key format includes "NVX " prefix
6. Check that your key has required permissions
7. **Review error response details**
8. The `description` field often provides specific guidance
9. Note the exact error code for documentation reference
10. **Test with minimal examples**
11. Reduce request complexity to isolate issues
12. Start with known working examples
13. **Implement proper error handling**
14. Design applications to handle API errors gracefully
15. Include retry logic with exponential backoff for rate limiting

Flow schemas

JSON-schema example

JSON-schema template

Here's an example of a JSON structure describing a complete flow.

Example schema

```json
json_schema
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Navixy IoT Gateway Flow",
  "description": "A schema for defining IoT data flows in the Navixy platform",
  "type": "object",
  "required": [ "id", "title", "enabled", "nodes", "edges" ],
  "properties": {
    "id": { "type": "integer", "description": "Unique identifier for the flow", "examples": [543] },
    "title": { "type": "string", "description": "Name of the flow", "examples": ["Temperature Monitoring Flow", "Vehicle Tracking Flow"] },
    "enabled": { "type": "boolean", "description": "Whether the flow is active or not", "default": true },
    "nodes": { "type": "array", "description": "Collection of nodes in the flowchart", "items": { "type": "object", "oneOf": [ { "$ref": "#/definitions/dataSourceNode" }, { "$ref": "#/definitions/initiateAttributesNode" }, { "$ref": "#/definitions/outputEndpointNode" } ] } },
    "edges": { "type": "array", "description": "Connections between nodes in the flowchart", "items": { "$ref": "#/definitions/edge" } } },
  "definitions": {
    "edge": { "type": "object", "description": "Represents a connection between two nodes", "required": ["from", "to"], "properties": { "from": { "type": "integer", "description": "ID of the source node", "examples": [1] }, "to": { "type": "integer", "description": "ID of the destination node", "examples": [2] } } },
    "nodeView": { "type": "object", "description": "Visual properties of a node in the flowchart UI", "properties": { "position": { "type": "object", "description": "Position of the node's top-left corner in the UI", "properties": { "x": { "type": "integer", "description": "X coordinate (horizontal position)", "examples": [25] }, "y": { "type": "integer", "description": "Y coordinate (vertical position)", "examples": [25] } }, "required": ["x", "y"] } } },
    "dataSourceNode": { "type": "object", "description": "Input endpoint node that defines the source of data for the flow", "required": ["id", "type", "title", "enabled", "data"], "properties": { "id": { "type": "integer", "description": "Unique identifier for the node within the flow", "examples": [1] }, "type": { "type": "string", "description": "Type of node, must be 'data_source' for input endpoints", "enum": ["data_source"] }, "title": { "type": "string", "description": "Name of the node", "examples": ["GPS Tracker Input", "Sensor Data Input"] }, "enabled": { "type": "boolean", "description": "Whether the node is active within the flow", "default": true }, "data": { "type": "object", "description": "Configuration data specific to this node type", "required": ["sources"], "properties": { "sources": { "type": "array", "description": "Collection of source device IDs to receive data from", "items": { "type": "integer", "description": "ID of a source device", "examples": [123458] } } } }, "view": { "$ref": "#/definitions/nodeView" } } },
    "initiateAttributesNode": { "type": "object", "description": "Node that creates or modifies data attributes in the flow", "required": ["id", "type", "title", "data"], "properties": { "id": { "type": "integer", "description": "Unique identifier for the node within the flow", "examples": [2] }, "type": { "type": "string", "description": "Type of node, must be 'initiate_attributes' for attribute manipulation nodes", "enum": ["initiate_attributes"] }, "title": { "type": "string", "description": "Name of the node", "examples": ["Calculate Fuel Consumption", "Convert Temperature Units"] }, "data": { "type": "object", "description": "Configuration data specific to this node type", "required": ["items"], "properties": { "items": { "type": "array", "description": "Collection of
```

attribute definitions/transformations", "items": { "type": "object", "required": ["name", "value"], "properties": { "name": { "type": "string", "description": "Name of the attribute to create or modify", "examples": ["fuel_tank_2", "temperature_celsius"] }, "value": { "type": "string", "description": "Expression that defines the attribute value, can reference other attributes or functions", "examples": ["(analog_1 + 100)/2", "temp_f * 5/9 - 32"] }, "generation_time": { "type": "string", "description": "Expression for when the data was generated, often uses functions like now()", "examples": ["now()", "timestamp"] }, "server_time": { "type": "string", "description": "Expression for when the data was received by the server, often uses functions like now()", "examples": ["now()"] } } } } } }, "view": { "$ref": "#/definitions/nodeView" } } }, "outputEndpointNode": { "type": "object", "description": "Terminating node that defines where the processed data will be sent", "required": ["id", "type", "title", "enabled", "data"], "properties": { "id": { "type": "integer", "description": "Unique identifier for the node within the flow", "examples": [3] }, "type": { "type": "string", "description": "Type of node, must be 'output_endpoint' for data output nodes", "enum": ["output_endpoint"] }, "title": { "type": "string", "description": "Name of the node", "examples": ["Navixy Output", "MQTT Broker Output"] }, "enabled": { "type": "boolean", "description": "Whether the node is active within the flow", "default": true }, "data": { "type": "object", "description": "Configuration data specific to this node type", "oneOf": [{ "$ref": "#/definitions/outputEndpointDataNavixy" }, { "$ref": "#/definitions/outputEndpointDataMqtt" }] }, "view": { "$ref": "#/definitions/nodeView" } } }, "outputEndpointDataNavixy": { "type": "object", "description": "Configuration for sending data to the Navixy platform", "required": ["output_endpoint_type"], "properties": { "output_endpoint_type": { "type": "string", "description": "Type of output endpoint, must be 'output_navixy' for Navixy platform", "enum": ["output_navixy"] } } }, "outputEndpointDataMqtt": { "type": "object", "description": "Configuration for sending data to an MQTT broker", "required": ["output_endpoint_type", "output_endpoint_id"], "properties": { "output_endpoint_type": { "type": "string", "description": "Type of output endpoint, must be 'output_mqtt_client' for MQTT brokers", "enum": ["output_mqtt_client"] }, "output_endpoint_id": { "type": "integer", "description": "ID of the predefined MQTT endpoint configuration", "examples": [44551] } } } } }

```
## Example flow

```json
{
  "id": 1001,
  "title": "Vehicle Telematics Processing Flow",
  "enabled": true,
  "nodes": [
    {
```
```

```
    "id": 1,
    "type": "data_source",
    "title": "Vehicle Tracker Input",
    "enabled": true,
    "data": {
      "sources": [
        123458,
        123459,
        123460
      ]
    },
    "view": {
      "position": {
        "x": 50,
        "y": 50
      }
    }
  },
  {
    "id": 2,
    "type": "initiate_attributes",
    "title": "Calculate Fuel Metrics",
    "data": {
      "items": [
        {
          "name": "fuel_level_percent",
          "value": "(analog_1 / 1024) * 100",
          "generation_time": "now()",
          "server_time": "now()"
        },
        {
          "name": "fuel_consumption_rate",
          "value": "analog_2 * 0.25",
          "generation_time": "now()",
          "server_time": "now()"
        },
        {
          "name": "estimated_range_km",
          "value": "fuel_level_percent * 5",
          "generation_time": "now()",
          "server_time": "now()"
        }
      ]
    },
    "view": {
      "position": {
        "x": 250,
```

```
        "y": 50
      }
    }
  },
  {
    "id": 3,
    "type": "initiate_attributes",
    "title": "Calculate Engine Metrics",
    "data": {
      "items": [
        {
          "name": "engine_temp_celsius",
          "value": "analog_3 * 0.5 - 40",
          "generation_time": "now()",
          "server_time": "now()"
        },
        {
          "name": "engine_load",
          "value": "(analog_4 / 1024) * 100",
          "generation_time": "now()",
          "server_time": "now()"
        },
        {
          "name": "maintenance_due",
          "value": "mileage > 10000",
          "generation_time": "now()",
          "server_time": "now()"
        }
      ]
    },
    "view": {
      "position": {
        "x": 250,
        "y": 200
      }
    }
  },
  {
    "id": 4,
    "type": "output_endpoint",
    "title": "Navixy Platform Output",
    "enabled": true,
    "data": {
      "output_endpoint_type": "output_navixy"
    },
    "view": {
      "position": {
```

```
          "x": 450,
          "y": 125
        }
      }
    },
    {
      "id": 5,
      "type": "output_endpoint",
      "title": "MQTT Broker Output",
      "enabled": true,
      "data": {
        "output_endpoint_type": "output_mqtt_client",
        "output_endpoint_id": 44551
      },
      "view": {
        "position": {
          "x": 450,
          "y": 250
        }
      }
    }
  ],
  "edges": [
    {
      "from": 1,
      "to": 2
    },
    {
      "from": 1,
      "to": 3
    },
    {
      "from": 2,
      "to": 4
    },
    {
      "from": 3,
      "to": 4
    },
    {
      "from": 3,
      "to": 5
    }
  ]
}
```

Example flow explanation

The example template shows a flow that:

1. Collects data from vehicle tracking devices
2. Processes the data in two parallel paths:
3. Calculating fuel-related metrics (level, consumption, range)
4. Calculating engine metrics (temperature, load, maintenance status)
5. Sends the processed data to:
6. The Navixy platform for tracking and visualization
7. An external MQTT broker for integration with other systems

Guides

Sending device data to an external MQTT system

Sending device data to an external system

Let's create a flow that sends your device data to an external system through MQTT. Rather than creating multiple endpoints separately, we can accomplish this in one single request.

Creating a complete flow with integrated MQTT node

The simplest approach is to define both your data sources and MQTT output endpoint directly in your flow creation request. To do it, send a request to the following endpoint:

POST /iot/logic/flow/create

Request example:

```
curl -X POST "https://api.{region}.navixy.com/v2/iot/logic/flow/create" \
  -H "Content-Type: application/json" \
  -H "Authorization: NVX your_token_here" \
  -d '{
  "flow": {
    "title": "Fleet data to external system",
```

```
  "enabled": true,
  "nodes": [
    {
      "id": 1,
      "type": "data_source",
      "enabled": true,
      "data": {
        "title": "Fleet vehicles",            // Title must be in the data object
        "sources": [12345, 12346, 12347]      // Your actual vehicle IDs
      },
      "view": {
        "position": { "x": 50, "y": 50 }
      }
    },
    {
      "id": 2,
      "type": "output_endpoint",
      "enabled": true,
      "data": {
        "title": "External MQTT System",       // Title must be located in the data objec
        "output_endpoint_type": "output_mqtt_client",  // Defines this as an MQTT output
        "output_endpoint_id": 45678,           // Required ID (can be any unique number)
        "properties": {
          "protocol": "Navixy Generic Protocol (NGP)", // Navixy Generic Protocol
          "domain": "mqtt.mycompany.com",      // Your MQTT broker address
          "port": 1883,                        // Standard MQTT port
          "client_id": "navixy-integration",   // Identifier for this client
          "qos": 1,                            // Quality of Service level
          "topics": ["fleet/vehicles/data"],   // Topics to publish to
          "version": "5.0",                    // MQTT protocol version
          "use_ssl": true,                     // Secure connection
          "mqtt_auth": true,                   // Authentication required
          "user_name": "mqtt_username",        // Your MQTT credentials
          "user_password": "mqtt_password"
        }
      },
      "view": {
        "position": { "x": 250, "y": 50 }
      }
    }
  ],
  "edges": [
    {
      "from": 1,  // Connect the data source node (id: 1)
      "to": 2     // to the MQTT output node (id: 2)
    }
  ]
```

```
  }
}'
```

The response will include the flow ID:

```
{
  "success": true,
  "id": 1234
}
```

Try it out

```EUserver json http { "method": "post", "url": "https://api.eu.navixy.com/v2/iot/logic/flow/create", "headers": { "Authorization": "NVX your_hash_here" }, "body": { "flow": { "title": "Fleet data to external system", "enabled": true, "nodes": [ { "id": 1, "type": "data_source", "title": "Fleet vehicles", "enabled": true, "data": { "title": "Fleet vehicles", "sources": [12345, 12346, 12347] }, "view": { "position": { "x": 50, "y": 50 } } }, { "id": 2, "type": "output_endpoint", "enabled": true, "data": { "title": "External MQTT System", "output_endpoint_type": "output_mqtt_client", "output_endpoint_id": "45678" "properties": { "protocol": "NGP", "domain": "mqtt.mycompany.com", "port": 1883, "client_id": "navixy-integration", "qos": 1, "topics": ["fleet/vehicles/data"], "version": "5.0", "use_ssl": true, "mqtt_auth": true, "user_name": "mqtt_username", "user_password": "mqtt_password" } }, "view": { "position": { "x": 250, "y": 50 } } } ], "edges": [ { "from": 1, "to": 2 } ] } } }
```

```
```USserver json http
{
  "method": "post",
  "url": "https://api.us.navixy.com/v2/iot/logic/flow/create",
  "headers": {
    "Authorization": "NVX your_hash_here"
  },
  "body":
  {
  "flow": {
    "title": "Fleet data to external system",
    "enabled": true,
    "nodes": [
      {
        "id": 1,
        "type": "data_source",
        "title": "Fleet vehicles",
```
```

```
        "enabled": true,
        "data": {
          "title": "Fleet vehicles",
          "sources": [12345, 12346, 12347]
        },
        "view": {
          "position": { "x": 50, "y": 50 }
        }
      },
      {
        "id": 2,
        "type": "output_endpoint",
        "enabled": true,
        "data": {
          "title": "External MQTT System",
          "output_endpoint_type": "output_mqtt_client",
          "output_endpoint_id": "45678"
          "properties": {
            "protocol": "NGP",
            "domain": "mqtt.mycompany.com",
            "port": 1883,
            "client_id": "navixy-integration",
            "qos": 1,
            "topics": ["fleet/vehicles/data"],
            "version": "5.0",
            "use_ssl": true,
            "mqtt_auth": true,
            "user_name": "mqtt_username",
            "user_password": "mqtt_password"
          }
        },
        "view": {
          "position": { "x": 250, "y": 50 }
        }
      }
    ],
    "edges": [
      {
        "from": 1,
        "to": 2
      }
    ]
  }
}
}
```

> **Congratulations!**
> You've now set up a flow that creates a complete end-to-end data pipeline in a single API call. This flow: - Connects to multiple vehicles in your fleet through a data source endpoint - Sends the device data to your external MQTT system - Uses your custom MQTT broker settings for secure data transfer

Managing your flows and endpoints

Managing your flows and endpoints

After you create one or multiple flows, you can proceed with managing thm. This guide demonstrates how to view, read details, and update existing IoT Logic flows using the Navixy IoT Logic API. You'll learn how to list all your flows, examine specific flow configurations including nodes and connections, and modify flows by adding new data processing rules. The examples show practical scenarios like managing fleet vehicle data flows, adding calculated attributes for business metrics, and connecting to MQTT endpoints for external system integration.

Prerequisites

For this example, let's presume that we have already:

1. Created an MQTT output endpoint with ID 45678 using `/iot/logic/flow/endpoint/create`
2. Created a flow with ID 1234 using `/iot/logic/flow/create` with the following components:
3. A data source node (ID: 1) that captures data from devices 12345, 12346, and 12347
4. An attribute calculation node (ID: 2) for basic metrics
5. An output endpoint node (ID: 3) that sends data to the MQTT endpoint

Viewing your flows

GET /iot/logic/flow/list

The flow list endpoint provides a quick overview of all IoT Logic flows in your account. This is useful for getting flow IDs and titles before performing detailed operations. Each flow in the

response includes its unique identifier and descriptive title, allowing you to identify which flows you want to examine or modify further.

To see all your existing flows, send the request:

```
curl -X GET "https://api.eu.navixy.com/v2/iot/logic/flow/list" \
  -H "Authorization: NVX your_session_token_here" \
  -H "Content-Type: application/json"
```

You will receive `id` and `title` parameters in the response:

```
{
  "success": true,
  "list": [
    {
      "id": 1234,
      "title": "Fleet Data to External System"
    }
  ]
}
```

Try it out

```EUserver json http { "method": "post", "url": "https://api.eu.navixy.com/v2/iot/logic/flow/list", "headers": { "Authorization": "NVX your_hash_here" } }

```USserver json http
{
  "method": "post",
  "url": "https://api.us.navixy.com/v2/iot/logic/flow/list",
  "headers": {
    "Authorization": "NVX your_hash_here"
  }
}
```

Viewing flow details

POST /iot/logic/flow/read
```

The flow read endpoint retrieves complete configuration details for a specific flow, including all nodes, their properties, and the connections between them. This detailed view shows you the entire data processing pipeline - from data sources through transformation nodes to output endpoints. Use this when you need to understand the current flow structure before making modifications or troubleshooting data processing issues.

To see the details of a specific flow, copy the `id` value of the needed flow from GET /iot/logic/flow/list response. Add it in the respective field of this request:

```
curl -X POST "https://api.eu.navixy.com/v2/iot/logic/flow/read" \
  -H "Authorization: NVX your_session_token_here" \
  -H "Content-Type: application/json" \
  -d '{
    "flow_id": 1234
  }'
```

You will receive the complete structure of the flow in the response:


```
{
  "success": true,
  "value": {
    "id": 1234,
    "title": "Fleet Data to External System",
    "description": null,
    "enabled": true,
    "default_flow": false,
    "nodes": [
      {
        "id": 1,
        "type": "data_source",
        "view": {
          "position": { "x": 50, "y": 50 }
        },
        "data": {
          "title": "Fleet Vehicles",
          "source_ids": [12345, 12346, 12347]
        }
      },
      {
        "id": 2,
        "type": "initiate_attributes",
        "view": {
          "position": { "x": 200, "y": 50 }
```

```
        },
        "data": {
          "title": "Calculate Business Metrics",
          "items": [
            {
              "name": "fuel_efficiency",
              "value": "value(\"distance_traveled\") / value(\"fuel_consumed\")",
              "generation_time": "genTime(\"distance_traveled\", 0, \"valid\")",
              "server_time": "now()"
            },
            {
              "name": "idle_time_percent",
              "value": "(value(\"idle_time\") / (value(\"idle_time\") + value(\"moving_time
              "generation_time": "genTime(\"idle_time\", 0, \"valid\")",
              "server_time": "now()"
            },
            {
              "name": "vehicle_status",
              "value": "value(\"speed\") gt 0 ? \"moving\" : \"stopped\"",
              "generation_time": "genTime(\"speed\", 0, \"valid\")",
              "server_time": "now()"
            }
          ]
        }
      },
      {
        "id": 3,
        "type": "output_endpoint",
        "view": {
          "position": { "x": 350, "y": 50 }
        },
        "data": {
          "title": "Send to External System",
          "output_endpoint_type": "output_mqtt_client",
          "output_endpoint_id": 45678
        }
      }
    ],
    "edges": [
      {
        "from": 1,
        "to": 2
      },
      {
        "from": 2,
        "to": 3
      }
```

```
    ]
  }
}
```

Try it out

```EUserver json http { "method": "post", "url": "https://api.eu.navixy.com/v2/iot/logic/flow/read", "headers": { "Authorization": "NVX your_hash_here" }, "body": { "flow_id": 1234 } }

```USserver json http
{
  "method": "post",
  "url": "https://api.us.navixy.com/v2/iot/logic/flow/read",
  "headers": {
    "Authorization": "NVX your_hash_here"
  },
  "body": {
    "flow_id": 1234
  }
}
```

Updating a flow

POST /iot/logic/flow/update

The flow update endpoint allows you to modify existing flows by changing node configurations, adding new processing rules, or adjusting connections. When updating a flow, you must provide the complete flow structure including all nodes and edges, even if you're only modifying one element. This ensures data consistency and prevents accidental deletion of existing components. In this example, we're adding a new calculated attribute to convert engine temperature from Fahrenheit to Celsius while preserving all existing functionality.

Copy the flow obgect structure from POST /iot/logic/flow/read response and add the new attribute to the Initiate Atribute node (`id": 2`). Then paste the resulting object in the body of this request:

```
curl -X POST "https://api.eu.navixy.com/v2/iot/logic/flow/update" \
  -H "Authorization: NVX your_session_token_here" \
  -H "Content-Type: application/json" \
```
```

```
-d '{
  "flow": {
    "id": 1234,
    "title": "Fleet Data to External System",
    "description": null,
    "enabled": true,
    "default_flow": false,
    "nodes": [
      {
        "id": 1,
        "type": "data_source",
        "view": {
          "position": { "x": 50, "y": 50 }
        },
        "data": {
          "title": "Fleet Vehicles",
          "source_ids": [12345, 12346, 12347]
        }
      },
      {
        "id": 2,
        "type": "initiate_attributes",
        "view": {
          "position": { "x": 200, "y": 50 }
        },
        "data": {
          "title": "Calculate Business Metrics",
          "items": [
            {
              "name": "fuel_efficiency",
              "value": "value(\"distance_traveled\") / value(\"fuel_consumed\")",
              "generation_time": "genTime(\"distance_traveled\", 0, \"valid\")",
              "server_time": "now()"
            },
            {
              "name": "idle_time_percent",
              "value": "(value(\"idle_time\") / (value(\"idle_time\") + value(\"moving_t
              "generation_time": "genTime(\"idle_time\", 0, \"valid\")",
              "server_time": "now()"
            },
            {
              "name": "engine_temp_celsius",
              "value": "(value(\"engine_temp_f\") - 32) * 5/9",
              "generation_time": "genTime(\"engine_temp_f\", 0, \"valid\")",
              "server_time": "now()"
            }
          ]
```

```
          }
        },
        {
          "id": 3,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 50 }
          },
          "data": {
            "title": "Send to External System",
            "output_endpoint_type": "output_mqtt_client",
            "output_endpoint_id": 45678
          }
        }
      ],
      "edges": [
        {
          "from": 1,
          "to": 2
        },
        {
          "from": 2,
          "to": 3
        }
      ]
    }
  }'
```

You will receive thes request status in response:

```
{
  "success": true
}
```

Try it out

```EUserver json http { "method": "post", "url": "https://api.eu.navixy.com/v2/iot/logic/flow/update", "headers": { "Authorization": "NVX your_hash_here" }, "body": { "flow": { "id": 1234, "title": "Fleet Data to External System", "description": null, "enabled": true, "default_flow": false, "nodes": [ { "id": 1, "type": "data_source", "view": { "position": { "x": 50, "y": 50 } }, "data": { "title": "Fleet Vehicles", "source_ids": [12345, 12346, 12347] } }, { "id": 2, "type": "initiate_attributes", "view": {
```

"position": { "x": 200, "y": 50 } }, "data": { "title": "Calculate Business Metrics", "items": [{ "name": "fuel_efficiency", "value": "value(\"distance_traveled\") / value(\"fuel_consumed\")", "generation_time": "genTime(\"distance_traveled\", 0, \"valid\")", "server_time": "now()" }, { "name": "idle_time_percent", "value": "(value(\"idle_time\") / (value(\"idle_time\") + value(\"moving_time\"))) * 100", "generation_time": "genTime(\"idle_time\", 0, \"valid\")", "server_time": "now()" }, { "name": "engine_temp_celsius", "value": "(value(\"engine_temp_f\") - 32) * 5/9", "generation_time": "genTime(\"engine_temp_f\", 0, \"valid\")", "server_time": "now()" }] } }, { "id": 3, "type": "output_endpoint", "view": { "position": { "x": 350, "y": 50 } }, "data": { "title": "Send to External System", "output_endpoint_type": "output_mqtt_client", "output_endpoint_id": 45678 } }], "edges": [{ "from": 1, "to": 2 }, { "from": 2, "to": 3 }] } } }

```USserver json http
{
  "method": "post",
  "url": "https://api.us.navixy.com/v2/iot/logic/flow/update",
  "headers": {
    "Authorization": "NVX your_hash_here"
  },
  "body": {
    "flow": {
      "id": 1234,
      "title": "Fleet Data to External System",
      "description": null,
      "enabled": true,
      "default_flow": false,
      "nodes": [
        {
          "id": 1,
          "type": "data_source",
          "view": {
            "position": { "x": 50, "y": 50 }
          },
          "data": {
            "title": "Fleet Vehicles",
            "source_ids": [12345, 12346, 12347]
          }
        },
        {
          "id": 2,
          "type": "initiate_attributes",
          "view": {
            "position": { "x": 200, "y": 50 }
          },
          "data": {
```

```
            "title": "Calculate Business Metrics",
            "items": [
              {
                "name": "fuel_efficiency",
                "value": "value(\"distance_traveled\") / value(\"fuel_consumed\")",
                "generation_time": "genTime(\"distance_traveled\", 0, \"valid\")",
                "server_time": "now()"
              },
              {
                "name": "idle_time_percent",
                "value": "(value(\"idle_time\") / (value(\"idle_time\") + value(\"moving_t
                "generation_time": "genTime(\"idle_time\", 0, \"valid\")",
                "server_time": "now()"
              },
              {
                "name": "engine_temp_celsius",
                "value": "(value(\"engine_temp_f\") - 32) * 5/9",
                "generation_time": "genTime(\"engine_temp_f\", 0, \"valid\")",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 3,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 50 }
          },
          "data": {
            "title": "Send to External System",
            "output_endpoint_type": "output_mqtt_client",
            "output_endpoint_id": 45678
          }
        }
      ],
      "edges": [
        {
          "from": 1,
          "to": 2
        },
        {
          "from": 2,
          "to": 3
        }
      ]
    }
```

```
  }
}
```

> **Congratulations!**
> You've now successfully enhanced your data flow by: - Adding an engine temperature conversion calculation (Fahrenheit to Celsius) - Maintaining your existing business metrics calculations - Updating your flow while preserving the connection to your fleet vehicles and MQTT endpoint

Advanced configurations

Advanced configurations

API resource relationships

Before diving into advanced configurations, it's important to understand the relationship between API resources:

```
User Account
    └── Flows (multiple)
        ├── Nodes/Endpoints (multiple per flow)
        └── Edges (connections between nodes)
```

Each Flow belongs to a user account, and Nodes/Endpoints are resources that can be shared across multiple flows.

Flow and node schemas

The API defines several schemas that are important to understand:

- `FlowDraft` : Used when creating a new flow (without ID)
- `Flow` : Used when updating an existing flow (includes ID)
- `Node` : The base type for all nodes in a flow
- `Edge` : Defines connections between nodes

Each node has a specific subtype (`NodeDataSource` , `NodeInitiateAttributes` , `NodeOutputEndpoint`) with different required properties.

Multiple output destinations

This scenario demonstrates how to configure a flow to send data to multiple destinations simultaneously. This is useful when you need to distribute processed IoT data to different systems, for example:

- Storing data in Navixy platform for tracking and analytics.
- Sending real-time updates to an external system for immediate alerting or reporting.

The flow processes data once and then branches it to multiple output endpoints, ensuring data consistency across all destinations.

Prerequisites

For this example, let's presume that we have already: - Created a flow with ID 1234 containing nodes 1, 2, and 3 - Created an MQTT output endpoint with ID 44551 - Node 1 is a data source node - Node 2 and 3 are transformation nodes that process temperature data - The existing flow has edges connecting node 1 to 2, and node 2 to 3

Updating a flow with multiple output endpoints

POST /iot/logic/flow/update

The flow update operation allows you to modify the entire flow structure, including adding new output endpoints and creating the necessary connections. In this example, we're adding two output endpoint nodes that will receive the same processed data from the transformation chain. This creates a branching pattern where data goes through the processing nodes and then splits to multiple destinations simultaneously.

> You can also create a completely new flow with this configuration by using POST /iot/logic/flow/create

To update an existing flow with additional output destinations, send the following request:

```
curl -X POST "https://api.eu.navixy.com/v2/iot/logic/flow/update" \
  -H "Authorization: NVX your_session_token_here" \
  -H "Content-Type: application/json" \
  -d '{
```

```
  "flow": {
    "id": 1234,
    "title": "Temperature Monitoring Flow",
    "description": null,
    "enabled": true,
    "default_flow": false,
    "nodes": [
      {
        "id": 1,
        "type": "data_source",
        "view": {
          "position": { "x": 50, "y": 50 }
        },
        "data": {
          "title": "Temperature Sensors",
          "source_ids": [123458]
        }
      },
      {
        "id": 2,
        "type": "initiate_attributes",
        "view": {
          "position": { "x": 150, "y": 50 }
        },
        "data": {
          "title": "Basic Calculations",
          "items": [
            {
              "name": "temp_celsius",
              "value": "(value(\"raw_temp\") - 32) * 5/9",
              "generation_time": "genTime(\"raw_temp\", 0, \"valid\")",
              "server_time": "now()"
            }
          ]
        }
      },
      {
        "id": 3,
        "type": "initiate_attributes",
        "view": {
          "position": { "x": 250, "y": 50 }
        },
        "data": {
          "title": "Advanced Calculations",
          "items": [
            {
              "name": "temp_status_numeric",
```

```
                "value": "value(\"temp_celsius\") / 10",
                "generation_time": "genTime(\"temp_celsius\", 0, \"valid\")",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 4,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 25 }
          },
          "data": {
            "title": "Send to Navixy",
            "output_endpoint_type": "output_default"
          }
        },
        {
          "id": 5,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 75 }
          },
          "data": {
            "title": "Send to MQTT",
            "output_endpoint_type": "output_mqtt_client",
            "output_endpoint_id": 44551
          }
        }
      ],
      "edges": [
        {"from": 1, "to": 2},
        {"from": 2, "to": 3},
        {"from": 3, "to": 4},
        {"from": 3, "to": 5}
      ]
    }
  }'
```

You will receive thes request status in response:

```
{
  "success": true
}
```

Try it out

```EUserver json http { "method": "post", "url": "https://api.eu.navixy.com/v2/iot/logic/flow/update", "headers": { "Authorization": "NVX your_hash_here" }, "body": { "flow": { "id": 1234, "title": "Temperature Monitoring Flow1", "description": null, "enabled": true, "default_flow": false, "nodes": [ { "id": 1, "type": "data_source", "view": { "position": { "x": 50, "y": 50 } }, "data": { "title": "Temperature Sensors", "source_ids": [123458] } }, { "id": 2, "type": "initiate_attributes", "view": { "position": { "x": 150, "y": 50 } }, "data": { "title": "Basic Calculations", "items": [ { "name": "temp_celsius", "value": "(value(\"raw_temp\") - 32) * 5/9", "generation_time": "genTime(\"raw_temp\", 0, \"valid\")", "server_time": "now()" } ] } }, { "id": 3, "type": "initiate_attributes", "view": { "position": { "x": 250, "y": 50 } }, "data": { "title": "Advanced Calculations", "items": [ { "name": "temp_status_numeric", "value": "value(\"temp_celsius\") / 10", "generation_time": "genTime(\"temp_celsius\", 0, \"valid\")", "server_time": "now()" } ] } }, { "id": 4, "type": "output_endpoint", "view": { "position": { "x": 350, "y": 25 } }, "data": { "title": "Send to Navixy", "output_endpoint_type": "output_default" } }, { "id": 5, "type": "output_endpoint", "view": { "position": { "x": 350, "y": 75 } }, "data": { "title": "Send to MQTT", "output_endpoint_type": "output_mqtt_client", "output_endpoint_id": 44551 } } ], "edges": [ {"from": 1, "to": 2}, {"from": 2, "to": 3}, {"from": 3, "to": 4}, {"from": 3, "to": 5} ] } } }

```USserver json http
{
  "method": "post",
  "url": "https://api.us.navixy.com/v2/iot/logic/flow/update",
  "headers": {
    "Authorization": "NVX your_hash_here"
  },
  "body": {
    "flow": {
      "id": 1234,
      "title": "Temperature Monitoring Flow1",
      "description": null,
      "enabled": true,
      "default_flow": false,
      "nodes": [
        {
          "id": 1,
          "type": "data_source",
          "view": {
            "position": { "x": 50, "y": 50 }
          },
          "data": {
```
```

```
            "title": "Temperature Sensors",
            "source_ids": [123458]
          }
        },
        {
          "id": 2,
          "type": "initiate_attributes",
          "view": {
            "position": { "x": 150, "y": 50 }
          },
          "data": {
            "title": "Basic Calculations",
            "items": [
              {
                "name": "temp_celsius",
                "value": "(value(\"raw_temp\") - 32) * 5/9",
                "generation_time": "genTime(\"raw_temp\", 0, \"valid\")",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 3,
          "type": "initiate_attributes",
          "view": {
            "position": { "x": 250, "y": 50 }
          },
          "data": {
            "title": "Advanced Calculations",
            "items": [
              {
                "name": "temp_status_numeric",
                "value": "value(\"temp_celsius\") / 10",
                "generation_time": "genTime(\"temp_celsius\", 0, \"valid\")",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 4,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 25 }
          },
          "data": {
```

```
            "title": "Send to Navixy",
            "output_endpoint_type": "output_default"
          }
        },
        {
          "id": 5,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 75 }
          },
          "data": {
            "title": "Send to MQTT",
            "output_endpoint_type": "output_mqtt_client",
            "output_endpoint_id": 44551
          }
        }
      ],
      "edges": [
        {"from": 1, "to": 2},
        {"from": 2, "to": 3},
        {"from": 3, "to": 4},
        {"from": 3, "to": 5}
      ]
    }
  }
}
```

Verifying the flow configuration

You can then validate the configuration of the updated flow using the `read` endpoint: POST /iot/logic/flow/read

> **Congratulations!**
> You have successfully configured a flow to send processed data to multiple destinations simultaneously. This setup allows your IoT data to be processed once and then distributed to both Navixy platform and an external system.

Complex data transformations

This scenario demonstrates how to create a flow with multiple data transformation steps for more advanced processing needs. Complex transformations are essential when working with sophisticated IoT applications that require multi-stage data processing, sequential computation paths, and the combination of multiple sensor inputs into derived metrics. This

pattern shows how to build processing pipelines that can handle chained data streams, perform sequential calculations on different attributes, and then combine the results for comprehensive analysis.

Prerequisites

For this example, let's presume that we have already: - Created a flow with ID 5678 - Added a data source node with ID 1 that reads from sensor ID 98765

Adding multiple transformation nodes

POST /iot/logic/flow/update

This example updates a flow with sequential branches that process temperature and humidity data in series before combining them into advanced analytics. The flow demonstrates several important patterns: sequential processing of different sensor attributes, chained transformation steps, and the accumulation of multiple data transformations into unified analysis nodes. This approach is particularly useful for environmental monitoring, industrial sensors, or any scenario where multiple related measurements need to be processed through different algorithms in a specific order before being combined.

> You can also create a completely new flow with this configuration by using POST /iot/logic/flow/create

To update an existing flow with additional output destinations, send the following request:

```
curl -X POST "https://api.eu.navixy.com/v2/iot/logic/flow/update" \
  -H "Authorization: NVX your_session_token_here" \
  -H "Content-Type: application/json" \
  -d '{
    "flow": {
      "id": 5678,
      "title": "Sensor Data Processing Flow",
      "description": null,
      "enabled": true,
      "default_flow": false,
      "nodes": [
        {
          "id": 1,
          "type": "data_source",
          "view": {
            "position": { "x": 50, "y": 50 }
          },
```

```
      "data": {
        "title": "Environmental Sensors",
        "source_ids": [98765]
      }
    },
    {
      "id": 2,
      "type": "initiate_attributes",
      "view": {
        "position": { "x": 150, "y": 25 }
      },
      "data": {
        "title": "Temperature Processing",
        "items": [
          {
            "name": "temp_celsius",
            "value": "(value(\"analog_1\") - 32) * 5/9",
            "generation_time": "genTime(\"analog_1\", 0, \"valid\")",
            "server_time": "now()"
          }
        ]
      }
    },
    {
      "id": 3,
      "type": "initiate_attributes",
      "view": {
        "position": { "x": 150, "y": 75 }
      },
      "data": {
        "title": "Humidity Processing",
        "items": [
          {
            "name": "humidity_adjusted",
            "value": "value(\"analog_2\") * 1.05",
            "generation_time": "genTime(\"analog_2\", 0, \"valid\")",
            "server_time": "now()"
          }
        ]
      }
    },
    {
      "id": 4,
      "type": "initiate_attributes",
      "view": {
        "position": { "x": 250, "y": 50 }
      },
```

```
          "data": {
            "title": "Combined Analysis",
            "items": [
              {
                "name": "heat_index",
                "value": "value(\"temp_celsius\") + (0.05 * value(\"humidity_adjusted\"))",
                "generation_time": "genTime(\"temp_celsius\", 0, \"valid\")",
                "server_time": "now()"
              },
              {
                "name": "comfort_score",
                "value": "value(\"heat_index\") / 10",
                "generation_time": "genTime(\"heat_index\", 0, \"valid\")",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 5,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 50 }
          },
          "data": {
            "title": "Send to Navixy",
            "output_endpoint_type": "output_default"
          }
        }
      ],
      "edges": [
        {"from": 1, "to": 2},
        {"from": 2, "to": 3},
        {"from": 3, "to": 4},
        {"from": 4, "to": 5}
      ]
    }
  }'
```

You will receive thes request status in response:

```
{
  "success": true
}
```

Try it out

```EUserver json http { "method": "post", "url": "https://api.eu.navixy.com/v2/iot/logic/flow/update", "headers": { "Authorization": "NVX your_hash_here" }, "body": { "flow": { "id": 5678, "title": "Sensor Data Processing Flow", "description": null, "enabled": true, "default_flow": false, "nodes": [ { "id": 1, "type": "data_source", "view": { "position": { "x": 50, "y": 50 } }, "data": { "title": "Environmental Sensors", "source_ids": [98765] } }, { "id": 2, "type": "initiate_attributes", "view": { "position": { "x": 150, "y": 25 } }, "data": { "title": "Temperature Processing", "items": [ { "name": "temp_celsius", "value": "(value(\"analog_1\") - 32) * 5/9", "generation_time": "now()", "server_time": "now()" } ] } }, { "id": 3, "type": "initiate_attributes", "view": { "position": { "x": 150, "y": 75 } }, "data": { "title": "Humidity Processing", "items": [ { "name": "humidity_adjusted", "value": "value(\"analog_2\") * 1.05", "generation_time": "now()", "server_time": "now()" } ] } }, { "id": 4, "type": "initiate_attributes", "view": { "position": { "x": 250, "y": 50 } }, "data": { "title": "Combined Analysis", "items": [ { "name": "heat_index", "value": "value(\"temp_celsius\") + (0.05 * value(\"humidity_adjusted\"))", "generation_time": "now()", "server_time": "now()" } ] } }, { "id": 5, "type": "output_endpoint", "view": { "position": { "x": 350, "y": 50 } }, "data": { "title": "Send to Navixy", "output_endpoint_type": "output_default" } } ], "edges": [ {"from": 1, "to": 2}, {"from": 2, "to": 3}, {"from": 3, "to": 4}, {"from": 4, "to": 5} ] } } }

```USserver json http
{
  "method": "post",
  "url": "https://api.us.navixy.com/v2/iot/logic/flow/update",
  "headers": {
    "Authorization": "NVX your_hash_here"
  },
  "body": {
    "flow": {
      "id": 5678,
      "title": "Sensor Data Processing Flow",
      "description": null,
      "enabled": true,
      "default_flow": false,
      "nodes": [
        {
          "id": 1,
          "type": "data_source",
          "view": {
            "position": { "x": 50, "y": 50 }
          },
          "data": {
```
```

```
            "title": "Environmental Sensors",
            "source_ids": [98765]
          }
        },
        {
          "id": 2,
          "type": "initiate_attributes",
          "view": {
            "position": { "x": 150, "y": 25 }
          },
          "data": {
            "title": "Temperature Processing",
            "items": [
              {
                "name": "temp_celsius",
                "value": "(value(\"analog_1\") - 32) * 5/9",
                "generation_time": "now()",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 3,
          "type": "initiate_attributes",
          "view": {
            "position": { "x": 150, "y": 75 }
          },
          "data": {
            "title": "Humidity Processing",
            "items": [
              {
                "name": "humidity_adjusted",
                "value": "value(\"analog_2\") * 1.05",
                "generation_time": "now()",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 4,
          "type": "initiate_attributes",
          "view": {
            "position": { "x": 250, "y": 50 }
          },
          "data": {
```

```
            "title": "Combined Analysis",
            "items": [
              {
                "name": "heat_index",
                "value": "value(\"temp_celsius\") + (0.05 * value(\"humidity_adjusted\"))",
                "generation_time": "now()",
                "server_time": "now()"
              }
            ]
          }
        },
        {
          "id": 5,
          "type": "output_endpoint",
          "view": {
            "position": { "x": 350, "y": 50 }
          },
          "data": {
            "title": "Send to Navixy",
            "output_endpoint_type": "output_default"
          }
        }
      ],
      "edges": [
        {"from": 1, "to": 2},
        {"from": 2, "to": 3},
        {"from": 3, "to": 4},
        {"from": 4, "to": 5}
      ]
    }
  }
}
```

Verifying the flow configuration

You can then validate the configuration of the updated flow using the `read` endpoint: POST /iot/logic/flow/read

> **Congratulations!**
> You have successfully built a complex data transformation chain that processes multiple sensor inputs through a series of calculations. This flow demonstrates advanced patterns including:
>
> 1. Sequential processing of different sensor attributes (temperature and humidity)

> 2. Chaining processed data through multiple transformation steps into a unified analysis
> 3. Creating derived values that reference previously calculated attributes from earlier nodes
>
> This type of multi-step transformation is powerful for implementing complex business logic and data processing requirements within your IoT system, allowing each processing stage to build upon the results of previous calculations.

Websocket access for Data Stream Analyzer

Websocket access to Data Stream Analyzer

Overview

The WebSocket functionality provides real-time, event-driven access to device data. It enables you to subscribe to various event types—such as location updates, device state changes, and telemetry streams—without the need for continuous polling.

WebSocket communication is ideal for applications that require immediate data delivery, such as monitoring dashboards, alert systems, or integrations that rely on timely IoT events.

For more details, see WebSocket in Navixy API This guide covers connection setup, authentication, event types, and message formats.

The iot_monitor event stream is designed for Data Stream Analyzer (DSA) and delivers real-time telemetry from selected devices. It provides both the latest data and short-term history, allowing you to track attribute changes over time.

This feature supports use cases like diagnostics, trend visualization, and alerting—where understanding recent attribute patterns is as important as receiving the newest values. Its structured format also helps you distinguish between missing and inactive data, improving clarity and reliability in IoT monitoring.

Data Stream Analyzer's "iot_monitor" event subscription

> This is a DSA-focused snippet showcasing subscription parameters. For complete information, see WebSocket Subscription

This subscription type is intended for the Data Stream Analyzer tool, which allows viewing the attribute values of the last N messages received from a tracker. The server stores attributes for no more than the last 12 messages, sorted in descending order by message timestamp. For each attribute, data is stored in two queues:

- Without data gaps – only messages where the specific attribute was present (not null).
- With data gaps – if an attribute was missing in one of the last messages, a null value is recorded in the queue.

After subscribing to "iot_monitor", server will send values of attributes from the latest messages for all non-blocked trackers included in the subscription in a single packet. Receiver must be able to parse data from different devices in this packet. New data will arrive in real-time in the event message, but no more frequently than the specified `rate_limit`.

```
{
  "action": "subscribe",
  "hash": "6c638c52cb40729d5e6181a48f868649",
  "requests": [
    {
      "type": "iot_monitor",
      "target": {
        "type": "selected",
        "tracker_ids": [
          21080
        ]
      },
      "rate_limit": "5s"
    }
  ]
}
```

Request fields:

- `type` – required, text: *"iot_monitor"*. Event type.
- `target` – required, target. Trackers to subscribe. Maximum 10 trackers.
- `rate_limit` – optional, string. A timespan for batching.

Response

Response parameters:

- `type` – required, text: *"response"*.
- `action` – required, text: *"subscription/subscribe"*.
- `events` – required, array of enum, without nulls. List of the subscribed events types ("", "" or "iot_monitor").
- `data` – required, map . Map with events subscription result. One key per subscribed event.
  - `state` – present if the "state" subscription requested, see sub response below.
  - `state_batch` – present if the "state_batch" subscription requested, see sub response below.
  - `readings_batch` – present if the "readings_batch" subscription requested, see sub response below.

Sub response: * `success` – required, boolean. * `value` – required, map , present if success. The current status of requested trackers.

Keys is a tracker IDs, values – one of the item:

- `normal` – non-blocked, normal status. State events for this tracker will be delivered to client.
- `blocked` – tracker blocked. State events for this tracker `will *not* be delivered to client. Lifecycle events will be delivered. After unblocking, current tracker state will be sent automatically.
- `unknown` – tracker ID missed in the database or not belong to current user.
- `disallowed` – subscription for this tracker not allowed by the current session.

Response sample:

```
{
  "type": "response",
  "action": "subscription/subscribe",
  "events": ["state"],
  "data": {
    "state": {
      "value": {
        "15564": "normal",
        "15565": "blocked",
        "15568": "unknown"
      },
```

```
      "success": true
    }
  }
}
```

Data Stream Analyzer event

> This is a DSA-focused snippet showcasing only its event. For complete information, see WebSocket Events

You can receive Data Stream Analyser messages through websocket. These messages are coming from server if client subscribed to the `iot_monitor` events of the specific tracker that is not blocked. These packets contain values of attributes from the latest messages sent by the selected tracker. It occurs in the next cases:

- Immediately after subscription.
- The latest attribute values are updated. But no more frequently than the `rate_limit`.

Message fields:

- `type` – "event".
- `event` – "iot_monitor".
- `data`:
- `iot_last_values` – list of objects:
  - `tracker_id` – tracker ID.
  - `nonnull_fields` – list of objects. Queue without data gaps – only messages where the specific attribute was present (not null).
  - `<field_name>` – name of the attribute.
    - `value` – value of attribute.
    - `msg_time` – message time.
    - `srv_time` – server time.
  - `all_fields` – list of objects. Queue with data gaps – if an attribute was missing in one of the last messages, a null value is recorded in the queue.
  - `<field_name>` – name of the attribute.
    - `value` – value of attribute.
    - `msg_time` – message time.
    - `srv_time` – server time.

Message sample:

```
{
  "event": "iot_monitor",
  "type": "event",
  "data": {
    "iot_last_values": [
      {
        "tracker_id": 21080,
        "nonnull_fields": {
          "satellites": [
            {
              "msg_time": "2025-02-11T15:12:58Z",
              "srv_time": "2025-02-11T15:12:58Z",
              "value": 10
            },
            {
              "msg_time": "2025-02-11T15:12:52Z",
              "srv_time": "2025-02-11T15:12:53Z",
              "value": 10
            }
          ],
          "heading": [
            {
              "msg_time": "2025-02-11T15:12:58Z",
              "srv_time": "2025-02-11T15:12:58Z",
              "value": 7
            },
            {
              "msg_time": "2025-02-11T15:12:52Z",
              "srv_time": "2025-02-11T15:12:53Z",
              "value": 6
            }
          ],
          "latitude": [
            {
              "msg_time": "2025-02-11T15:12:58Z",
              "srv_time": "2025-02-11T15:12:58Z",
              "value": "5.9654133"
            },
            {
              "msg_time": "2025-02-11T15:12:52Z",
              "srv_time": "2025-02-11T15:12:53Z",
              "value": "5.9646583"
            }
          ],
          "longitude": [
            {
```

```
            "msg_time": "2025-02-11T15:12:58Z",
            "srv_time": "2025-02-11T15:12:58Z",
            "value": "6.5171267"
          },
          {
            "msg_time": "2025-02-11T15:12:52Z",
            "srv_time": "2025-02-11T15:12:53Z",
            "value": "6.5169383"
          }
        ],
        "lls_level_1": [
          {
            "msg_time": "2025-02-11T15:12:58Z",
            "srv_time": "2025-02-11T15:12:58Z",
            "value": "262.9138"
          },
          {
            "msg_time": "2025-02-11T15:12:52Z",
            "srv_time": "2025-02-11T15:12:53Z",
            "value": "262.9945"
          }
        ],
        "speed": [
          {
            "msg_time": "2025-02-11T15:12:58Z",
            "srv_time": "2025-02-11T15:12:58Z",
            "value": 43
          },
          {
            "msg_time": "2025-02-11T15:12:52Z",
            "srv_time": "2025-02-11T15:12:53Z",
            "value": 48
          }
        ]
      },
      "all_fields": {
        "satellites": [
          {
            "msg_time": "2025-02-11T15:12:58Z",
            "srv_time": "2025-02-11T15:12:58Z",
            "value": 10
          },
          {
            "msg_time": "2025-02-11T15:12:52Z",
            "srv_time": "2025-02-11T15:12:53Z",
            "value": 10
          }
```

```
    ],
    "heading": [
      {
        "msg_time": "2025-02-11T15:12:58Z",
        "srv_time": "2025-02-11T15:12:58Z",
        "value": 7
      },
      {
        "msg_time": "2025-02-11T15:12:52Z",
        "srv_time": "2025-02-11T15:12:53Z",
        "value": 6
      }
    ],
    "latitude": [
      {
        "msg_time": "2025-02-11T15:12:58Z",
        "srv_time": "2025-02-11T15:12:58Z",
        "value": "5.9654133"
      },
      {
        "msg_time": "2025-02-11T15:12:52Z",
        "srv_time": "2025-02-11T15:12:53Z",
        "value": "5.9646583"
      }
    ],
    "longitude": [
      {
        "msg_time": "2025-02-11T15:12:58Z",
        "srv_time": "2025-02-11T15:12:58Z",
        "value": "6.5171267"
      },
      {
        "msg_time": "2025-02-11T15:12:52Z",
        "srv_time": "2025-02-11T15:12:53Z",
        "value": "6.5169383"
      }
    ],
    "lls_level_1": [
      {
        "msg_time": "2025-02-11T15:12:59Z",
        "srv_time": "2025-02-11T15:12:59Z",
        "value": null
      },
      {
        "msg_time": "2025-02-11T15:12:52Z",
        "srv_time": "2025-02-11T15:12:53Z",
        "value": "262.9945"
```

```
            }
          ],
          "speed": [
            {
              "msg_time": "2025-02-11T15:12:58Z",
              "srv_time": "2025-02-11T15:12:58Z",
              "value": 43
            },
            {
              "msg_time": "2025-02-11T15:12:52Z",
              "srv_time": "2025-02-11T15:12:53Z",
              "value": 48
            }
          ]
        }
      }
    ]
  }
}
```