Automation Contexts

Overview

This document explains how automation contexts are constructed, composed, and made available across every execution layer of the FLUX automation application: **flows**, **tasks**, **scripts**, **models**, and **services (email-msg)**.

Contexts provide a structured way to pass data and configuration throughout the automation pipeline, ensuring that each component has access to the appropriate information needed for execution.

Think of context as a "backpack of information" that moves through the execution. It starts with base device data and gets enriched as the process moves forward. The main idea: **each step can reuse what previous steps already discovered**.

Base Element Context

Function: `element_context_generate(element)`

The element context serves as the foundation of every execution. It is generated once per element at the beginning of each task or flow execution step and is passed throughout the entire pipeline.

Source Location: `apps/inventory/utils.py → element_context_generate()`

Context Structure

The base element context contains the following components:

```
{
    # ── Group Variables ──────────────────────────────────────────────
    # One key per GroupVariable configured in any Group assigned to the element.
    # Variables with dots in their name are expanded into nested dictionaries.
    # Example: "if.description" → context["if"]["description"] = value
    "<variable_name>": "<value>",
```

```
"<nested.variable>": "<value>",   # expanded: context["nested"]["variable"]

# ── Element Settings (ElementSetting) ──────────────────────────────────
# One key per parameter configured in the element's Profile.
# Key format: [group.variable_root.][instance.]parameter.variable_name
# Example: "net.0.hostname" → context["net"]["0"]["hostname"]
"<parameter_variable_name>": "<value>",

# ── Element Serialization ──────────────────────────────────────────────
"element": {                      # Serialized Element (all database fields)
    "id": 1,
    "name": "router-01",
    "uuid": "...",
    "status": True,               # operational status
    "status_change": "...",       # last status change datetime
    "prev_status_change": "...",
    "is_active": True,
    "debug": False,
    "sync": False,
    "reconf": False,
    "internal": False,
    "location": {                 # nested – all Location fields
        "id": 1,
        "name": "DataCenter A",
        ...
    },
    "latitude": None,
    "longitude": None,
    "profile": 3,                 # Foreign Key ID
    "organization": 1,            # Foreign Key ID
    "is_public": False,
    "group": [2, 5],              # list of Foreign Key IDs
    "certificate": None,
    "key": None,
    "password": None,
    "elevated_priv_password": None,
    "management_address": "192.168.1.1",
    "management_gateway": None,
    "transport_settings": None,
```

```
        "software_version": "17.3.4",
        "hardware_version": None,
        "serial_number": "FTX1234ABCD",
        "serial_number_alt": None,
        "ip_address": None,            # Foreign Key ID
        "ip_network": None,
        "dfgw_address": None,
        "dns_servers": None,
        "mac_address": None,
        "created": "...",
        "created_by": 1,
        "last_change": "...",
        "first_execution": "...",
        "last_execution": "...",
        "description": "Core router",
        "description2": None,
        "description3": None,
        "description4": None,
        "notes": None,
        "active_alert": None,
        "alert": None,
    },

    # ── Timestamp ──────────────────────────────────────────────────────────
    "epoch": 1710000000,              # UTC timestamp (integer) at context generation time
}
```

Flow Context Addition: `context['flow']`

When a task runs **inside a flow** (via `run_task_wrapper`), the following key is automatically added before the `TaskContext` is built:

```
context['flow'] = {
    'user': {
        'id': user.id,
        'username': user.username,
        'full_name': user.get_full_name(),
        'email': user.email,
    }
    # Returns empty dictionary {} if no user is associated with the flow
```

```
}
```

When a user runs the flow, user info is also loaded as real runtime variables:

```
flow_variables = {'user': None}
flow_variables['user'] = {
    'id': user.id,
    'username': user.username,
    'full_name': user.get_full_name(),
    'email': user.email,
}
```

Task Results Addition: `context['tasks']`

At the beginning of each task execution within a flow, the task results accumulated so far for the *current element* are merged into the context:

```
if self.ctx.flow and self.ctx.flow_context:
    self.ctx.context['tasks'] = self.ctx.flow_context[-1]['tasks']
```

This allows scripts to read results from previously-executed tasks of the same element within the current flow run.

TaskContext Class

The `TaskContext` groups all execution-time variables for a single task run on a single element.

Source Location: `apps/automation/task_utils.py → TaskContext`

TaskContext Properties

```
class TaskContext:
    element        # Element ORM instance
    flow           # Flow ORM instance (or None if standalone task)
    flow_context   # list[dict] — the live flow context list (see Flow Context section)
    context        # dict — base element context (element_context_generate), modified in-
flight
    user           # User ORM instance (or None)
```

Important: The `context` property is **mutable throughout the pipeline**. Management scripts, processing scripts, and the task runner itself add keys to it as the task progresses.

Flow Context (`flow_context`)

A **Flow** runs over one or more elements (devices). The `flow_context` is a **list** built progressively as the flow processes elements — one entry per element.

Source Location: `apps/automation/flow_utils.py → FlowManager`

Per-Element Entry Structure

For each element, the flow creates this structure:

```
{
    "id": element.id,                  # element.id
    "context": element_context_generate(element),  # full element_context_generate() result
    "tasks": {}                        # populated after each task finishes
}
```

Each element in the flow generates an entry with the following structure:

```
{
    "id": 42,                          # element.id
    "context": {                       # full element_context_generate() result
        ...                            # all keys from Base Element Context section
    },
    "tasks": {                         # populated after each task finishes
        "<task_pk_as_str>": {
            # Keys depend on which steps ran:
            "management_script": {
                "status": "OK",
                "result": {...}
            },
            "process_model": {
                "status": "OK",
                "result": "<rendered model string>"
            },
            "model_commands": {
                "status": "OK",
```

```
                "result": "<raw command output>"
            },
            "processing_model_out": {
                "status": "OK",
                "result": {...}
            },
            "element_data_mapped": {
                "status": "OK",
                "result": {...}
            },
            "execute_service": {
                "status": "OK",
                "result": {}
            },
        },
        "<another_task_pk>": { ... },
    }
}
```

The list grows as each element is processed by the flow:

- **Serial flows:** Process one element at a time
- **Parallel flows:** Launch all elements concurrently, but the list is still populated per-element

How Flow Context is Used

- The flow knows which element is being processed (`id` + `context`).
- The flow keeps task results in `tasks` as execution moves forward.
- Next flow steps can make decisions using those accumulated results.

Script Contexts

FLUX automation provides five distinct types of scripts, each receiving a different set of variables and serving specific purposes in the automation pipeline.

Flow Pre-Execution Script

Purpose: Runs **once before any element loop**, used to sort or filter the element list.

Script Type: `processing_script` on Flow
Source Location: `FlowManager.flow_execute_processing_script()`

Context Variables

```
{
    "elements": list[Element],         # full list of elements that will run the flow
    "order_flow_elements": Callable,   # sort_flow_elements(elements, sort_fn, reverse=False)
    "filter_flow_elements": Callable,  # filter_flow_elements(elements, filter_fn,
inplace=True)
    "settings": {},                    # unused output placeholder
}
```

Output Expectations

- No explicit output expected
- The `elements` list is modified in place

Available Modules

`math`, `json`, `random`, `naturaltime`, `re`, `ssh_key_generate`, `time`

Flow Execution Script

Purpose: The main flow script that calls `run_task()` / `run_flow()` to orchestrate execution.

Script Type: `execution_script` on Flow

Serial Flow Context

Execution: One invocation **per element**
Source Location: `SerialFlowManager.run_script()`

```
{
    "context": dict,            # flatten_dictionary(flow_context) — flattened snapshot
    "flow_variables": {
        "user": {
            "id": ...,
            "username": ...,
            "full_name": ...,
            "email": ...
        }                       # or None if no user
    },
```

```
    "element": Element,         # current element ORM instance
    "elements": list[Element],  # parent flow's elements (if nested), else None
    "current": int,             # index of current element (0-based)
    "previous": int,            # current - 1
    "last": int,                # len(elements) - 1
    "first": 0,
    "settings": {},             # unused
    "run_task": Callable,       # run_task(id=None, short_name=None, retry_attempts=1,
retry_delay=0)
    "run_flow": Callable,       # run_flow(id=None, short_name=None, elements=None, ...)
}
```

Key Points:

- The `run_task` callable executes the task for the **current element** only
- After script execution, `action_status` and `action_info` are read from `script_context` and stored in the flow log

Magic Replacements: Applied to the script string before execution:

Placeholder	Replaced With
`current`	`str(current_order)`
`previous`	`str(current_order - 1)`
`last`	`str(flow_elements_count - 1)`
`first`	`"0"`

Parallel Flow Context

Execution: One invocation for **all elements at once**
Source Location: `ParallelFlowManager.run_script()`

```
{
    "context": dict,                # flatten_dictionary(flow_context)
    "flow_variables": dict,         # same as serial (see above)
    "elements": list[Element],      # all elements of this flow
    "settings": {},
    "run_task": Callable,           # run_task_in_elements(id=, short_name=, elements=None,
...)
    "run_flow": Callable,           # run_flow_in_elements(id=, short_name=, elements=None,
...)
    "filter_flow_elements": Callable,
```

```
}
```

Key Points:

- The `run_task` callable spawns one thread per element

Available Modules

`math`, `json`, `random`, `naturaltime`, `re`, `ssh_key_generate`, `time`, `pause`

TaskFlow Condition Script

Purpose: Executed before deciding whether to execute a task step for a given element.

Script Type: `condition_value` on TaskFlow
Source Location: `task_flow_execute_condition_script()`

Context Variables

```
{
    "flow_context": list[dict],   # full flow_context list (see Flow Context section)
    "current": int,               # current element order
    "previous": int,              # current - 1
    "last": int,                  # len(elements) - 1
    "first": 0,
    "settings": {},
}
```

Output Requirements

The script **must** assign: `condition = True` or `condition = False`

Available Modules

`math`, `json`, `random`, `naturaltime`, `re`, `ssh_key_generate`, `time`

Task Management Script

Purpose: Runs as **Step 1** of the task pipeline, before any connection to elements. Used to enrich the context (e.g., resolve credentials, build payloads, call APIs, wait for element availability).

Script Type: `management_script` on Task
Source Location: `task_execute_script(..., klass='mgmt')`

A **Task** receives element context and can enrich it while it runs. When a task starts inside a flow:

```
element_context = element_context_generate(element)
element_context['flow'] = flow_variables if flow_variables else {}
```

Then, right before task execution, previous task results for that same element are injected:

```
self.ctx.context['tasks'] = self.ctx.flow_context[-1]['tasks']
```

Context Variables

```
{
    "element": PublicElement,         # wraps the Element (restricted write interface)
                                      # only if 'element.' appears in the script text
    "config": PublicElementConfig,    # wraps ElementConfig (read/save last config)
                                      # only if 'config.' appears in script text
    "data": {},                       # always empty for management scripts
    "context": dict,                  # full element context dict (mutable — write here)
    "flow_context": None,             # NOT passed from run_management_script
    "settings": {},                   # not used as output for mgmt scripts
    "pause": Callable,                # pause(sleep=30, user_msg=None)
    "wait_for_connect": Callable,     # wait_for_connect(timeout=10, retry=1, sleep=30,
user_msg=None)
    "log_message": Callable,          # log_message(level, msg)
}
```

Writing Output

```
# Add/update keys in element context
context['my_new_variable'] = 'value'
context['credentials'] = {'user': 'admin', 'pw': 'secret'}
```

Output Processing

- **Extraction Method:** `result.pop('context', {})`
- **Integration:** The entire `context` dict after execution is merged into `TaskContext.context` via `.update()`

If a management script returns new context data, it is merged directly:

```
self.ctx.context.update(script_out)
```

Available Modules

`math`, `json`, `random`, `naturaltime`, `re`, `ssh_key_generate`, `time`

Task Processing Script

Purpose: Runs as **Step 7**, after model commands have been sent and parsed. Used to transform, validate, or enrich the data extracted from the element before it is stored in the database.

Script Type: `processing_script` on Task
Source Location: `task_execute_script(..., klass='proc')`

Context Variables

```
{
    "element": PublicElement,         # same as management script
    "config": PublicElementConfig,
    "data": dict,                     # current processing_out (command results, parsed data)
    "context": dict,                  # full element context dict (read + write)
    "flow_context": None,             # NOT currently passed from run_processing_script
    "settings": {},                   # output dict — write results here
    "pause": Callable,
    "wait_for_connect": Callable,
    "log_message": Callable,
}
```

Writing Output

```
settings['hostname'] = data.get('Hostname', '').strip()
settings['action_status'] = 'ok'       # optional — stored in task results
settings['action_info'] = 'Hostname updated'
```

Special Output Keys

Key	Purpose
`action_status`	Stored in `TaskResults.action_status` and flow log
`action_info`	Stored in `TaskResults.action_info` and flow log

Output Processing

- **Extraction Method:** `result.pop('settings', {})`
- **Integration:** Merged into `TaskManager.results.processing_out` via `.update()`

Available Modules

`math`, `json`, `random`, `naturaltime`, `re`, `ssh_key_generate`, `time`

How Task Context is Used

The task can:

- Read base element data (`element`, `epoch`, settings/group variables)
- Read flow user data in `context['flow']`
- Read previous task outputs in `context['tasks']`
- Add more data for later steps

Concrete runtime behavior already used in code:

```
if not self.results.commands_out and self.ctx.context.get('automation_custom_output'):
    self.results.commands_out = self.ctx.context['automation_custom_output']
```

Model Python Script Context

For models with `klass in ('pyms', 'pyps')`, a Python script runs inside the model data mapping process.

Source Location: `apps/automation/utils.py → model_data_map_python_script()`

Context Variables

```
{
    "model": Model,          # the Automation Model ORM instance
    "data": str,             # raw text data received from element (for pyps)
                             # empty string for pyms (no raw data)
    "context": dict,         # full element context dict (read-only recommendation)
    "output": {},            # write results here
    "settings": {},          # alternative output dict
}
```

Writing Output

```
output['parsed_value'] = some_function(data)
output['hostname'] = re.search(r'hostname (\S+)', data).group(1)
```

Output Processing

- **Extraction Method:** `result.pop('output', {})`
- **Special Handling:** If `context` was also modified, it is merged under `output['context']`

Additional Modules

`math`, `json`, `random`, `naturaltime`, `re`, `time`, `boto3`, `botocore`, `collections`, `datetime`, `timedelta`, `gcp_bigquery`, `gcp_service_account`, `timezone`, `pytz`

Service Execution Context (email-msg)

Purpose: Handles email service execution during task completion.

Source Location: `apps/automation/utils.py → task_execute_service()`
Called In: `TaskManager.execute_service()` (Step 9)

A **Service** (email service in this case) uses the context available at service execution time.

Email Context Assembly

At service step, the email context is assembled exactly like this:

```
email_context = {
    # ── Fixed Keys ──────────────────────────────────────────────────────
    'element': element,              # raw Element ORM instance
    'task': task,                    # Task ORM instance (name, short_name, id, ...)
    'body_custom_msg': notification_content,  # 'content' param from service settings

    # ── Task Context (spread) ──────────────────────────────────────────
    # Everything from TaskContext.context at the time execute_service runs:
    #   - all base element context keys (group variables, element settings,
context['element'], epoch)
    #   - context['flow'] = flow_variables (if running inside a flow)
    #   - context['tasks'] = previous tasks results for this element in this flow
    #   - any keys added by management_script or processing_script
```

```
    **task_context,

    # ── Flow Context Entry (spread) ──────────────────────────────────────
    # The flow_context[-1] dict for the current element (see Flow Context section), or {} if
standalone:
    #   - "id": element.id
    #   - "context": element_context_generate(element)  (original snapshot)
    #   - "tasks": { "<task_pk>": task_detail, ... }
    **flow_context_entry,
}
```

Service Settings Parameters

These parameters are configured on the `Service` instance of type `email-msg`:

Parameter	Usage
`from-email`	Sender address
`custom_to`	Comma-separated recipient list
`subject`	Email subject (Django template rendered)
`content`	`body_custom_msg` in the email context
`template`	Template ID (optional, selects HTML template)

Injecting Custom Variables

The `task_context` received by `task_execute_service()` is directly `self.ctx.context` — the same mutable dictionary that circulates through the entire task pipeline. Since the service runs in step 9, results from both the management script (step 1) and processing script (step 7) are available.

Management Script ? Official Channel

Script results are applied with `.update()` to `ctx.context`:

```
# In task management_script (klass='svc'):
context['recipient_email'] = element.get('email_contact', '')
context['alert_level'] = 'critical' if context.get('status') == 'down' else 'info'
context['custom_subject'] = f"[{element['name']}] Alert processed"
# All these keys will be available in the email template
```

Processing Script ? Direct Dictionary Mutation

The processing script receives the same `context` object by reference. Its official output (`settings`) goes to `processing_out` (and from there to the database), **not** to `ctx.context`. However, since the dictionary is passed by reference, writing directly to `context` within the processing script also persists and reaches the service:

```
# In task processing_script:
settings['sw_version'] = data.get('version', '')   # → processing_out → Database

# To pass something additional to the service:
context['parsed_hostname'] = data.get('hostname', '')  # → persists in ctx.context → reaches service
```

Data Flow Summary

Source	Destination	Available in email_context?
`context['x'] = val` in management script	`ctx.context` via `.update()`	✓ Yes
`context['x'] = val` in processing script	`ctx.context` direct (by reference)	✓ Yes
`settings['x'] = val` in processing script	`results.processing_out` → Database	✗ No (unless also copied to `context`)
`flow_variables` (user info from flow)	`context['flow']`	✓ Yes (as `context['flow']['user']`)

How Service Context is Used

Email subject/body and template can use:

- Task and element data
- Anything previously added to `task_context`
- If the flow already has task results, those are also available through the merged context

Context Lifecycle Within Task Execution

The following table summarizes which context is available at each step of `TaskManager.execute()`:

Step	Method	Context at Entrance	Context Modifications
—	`execute()` starts	Base element context	`context['tasks']` added if inside flow

Step	Method	Context at Entrance	Context Modifications
1	`run_management_script()`	Base context	Script can add/modify any key in `context`
2	`element_data_mapping()`	Context enriched by step 1	`results.model_out` set
3–6	`connect_with_element()` …	Same context	`results.commands_out`, `processing_out` built
7	`run_processing_script()`	Same context + `data=processing_out`	`results.processing_out` updated from `settings`; `action_status` / `action_info` stored
8	`mapping_result_to_database()`	Same context + processing_out	Element parameters saved to database
9	`execute_service()`	Same context	Email sent; service_out merged into processing_out
10	`finish_execution()`	Final context	Task log created; `flow_context[-1]['tasks'][task_pk]` updated

Context Lifecycle Within Serial Flow Execution

```
FlowManager.flow_execute()
│
├─ flow_execute_processing_script()  ← receives: elements list (can sort/filter)
│
└─ for each element:
       flow_context.append(get_flow_context_for_element(element))
       │
       └─ SerialFlowManager.run_script(element, order)
              │   receives: flattened flow_context, element, positions, run_task, run_flow
              │
              └─ run_task(id, short_name)
                     │
                     └─ run_task_wrapper()
                            │   element_context = element_context_generate(element)
                            │   element_context['flow'] = flow_variables
                            │
```

```
                    └─ TaskManager.execute()
                           ├─ context['tasks'] = flow_context[-1]['tasks']
                           ├─ run_management_script()   → context enriched
                           ├─ element_data_mapping()
                           ├─ connect_with_element()
                           ├─ model_commands_execution()
                           ├─ custom_model_commands_output_processing()
                           ├─ convert_command_results...()
                           ├─ run_processing_script()   → processing_out enriched
                           ├─ mapping_result_to_database()
                           ├─ execute_service()         → email sent with full context
                           └─ finish_execution()
                                  └─ flow_context[-1]['tasks'][task_pk] = task_detail
```

Context Lifecycle Within Parallel Flow Execution

```
ParallelFlowManager
│
├─ for each element:  flow_context.append(get_flow_context_for_element(element))
│
└─ run_script()
       │   receives: flattened flow_context, all elements, run_task_in_elements
       │
       └─ run_task_in_elements(id, short_name, elements=None)
              │
              └─ spawns one thread per element → run_task_wrapper() (same as serial)
```

Key Difference: The parallel flow script sees all elements at once. The `context` variable in the script is the **flattened** snapshot of `flow_context` (all elements merged), not a per-element view.

Variable Availability Reference

This table shows which variables are available in each script type:

Variable	Flow Pre-exec	Flow Exec (Serial)	Flow Exec (Parallel)	Task Condition	Task Mgmt Script	Task Proc Script	Model pyms/pyps	Service (Email)
`element` (Element ORM)	✗	✓	✗	✗	✓ (PublicElement)	✓ (PublicElement)	✗	✓
`elements` (list)	✓	✓ (parent)	✓	✗	✗	✗	✗	✗
`context` (element ctx dict)	✗	✓ (flattened)	✓ (flattened)	✗	✓ (mutable)	✓ (read)	✓ (read)	✓ (spread)
`flow_context` (list)	✗	✗	✗	✓	✗	✗	✗	✓ (latest entry)
`flow_variables` (user info)	✗	✓	✓	✗	✗	✗	✗	✗
`data`	✗	✗	✗	✗	✓ (empty)	✓ (processing_out)	✓ (raw text)	✗
`settings` (output)	✗	✗	✗	✗	✗	✓ **write here**	✗	✗
`output` (output)	✗	✗	✗	✗	✗	✗	✓ **write here**	✗
`condition` (output)	✗	✗	✗	✓ **write here**	✗	✗	✗	✗
`run_task`	✗	✓	✓	✗	✗	✗	✗	✗
`run_flow`	✗	✓	✓	✗	✗	✗	✗	✗
`current` / `previous` / `last` / `first`	✗	✓	✗	✓	✗	✗	✗	✗
`config` (PublicElementConfig)	✗	✗	✗	✗	✓	✓	✗	✗
`pause(sleep)`	✓	✓	✓	✗	✓	✓	✗	✗
`wait_for_connect(...)`	✗	✗	✗	✗	✓	✓	✗	✗
`task` (Task ORM)	✗	✗	✗	✗	✗	✗	✗	✓

Variable	Flow Pre-exec	Flow Exec (Serial)	Flow Exec (Parallel)	Task Condition	Task Mgmt Script	Task Proc Script	Model pyms/pyps	Service (Email)
`body_custom_msg`	✗	✗	✗	✗	✗	✗	✗	✓

Writing to Context from Scripts

Management Script ? Context Enrichment

```
context['resolved_ip'] = '10.0.0.1'
context['credentials'] = {'username': 'admin', 'password': 'secret'}
# These are available to all subsequent steps in the same task execution
```

Processing Script ? Settings Population

```
settings['hostname'] = data.get('hostname', '').strip()
settings['sw_version'] = re.search(r'Version (\S+)', data.get('version', '')).group(1)
settings['action_status'] = 'ok'   # or 'warning', 'error'
settings['action_info'] = 'Parsed 3 parameters successfully'
# Keys in settings are merged into processing_out and saved to ElementSetting
```

Model pyms/pyps Script ? Output Writing

```
output['interface_count'] = len(re.findall(r'^interface', data, re.M))
output['mgmt_address'] = context.get('element', {}).get('management_address')
```

Flow Execution Script ? Results Reading

```
# After run_task runs, results are accumulated in flow_context and
# available via the flattened context dict in the next run_task call
# (for reading, use the full flow_context structure)
run_task(short_name='collect-hw')
run_task(short_name='send-report')  # can read results from collect-hw via context['tasks']
```

Summary: End-to-End Context Flow

Context is shared runtime data that follows this path:

Case	Real context source	Practical use
Flow	`{"id", "context", "tasks"}` per element + `flow_variables['user']`	Coordinate and track full flow execution
Task	`element_context_generate(...)` + `flow` + `tasks` + script updates	Run task logic with up-to-date shared data
Service	`email_context` = fixed keys + merged task/flow context	Send dynamic email content with real execution data

Real Path Through System

1. Flow creates per-element context: `id`, `context`, `tasks`.
2. Task starts and receives base context + `flow_variables`.
3. Task injects previous task results into `context['tasks']`.
4. Management/processing steps enrich available data.
5. Service runs and receives merged `email_context` (element + task + accumulated context).

Result: Services run with real, current data from the same execution — no manual copy/paste. Flows create and organize context, tasks enrich and consume it, and services use it to execute with full, real execution state.

Revision #2
Created 2026-03-13 03:10:49 UTC by mauro@zequenze.com
Updated 2026-03-17 03:10:56 UTC by mauro@zequenze.com