MISP

INTELLIGENCE SHARING

MISP modules

Enrichment, data exchange and workflow actions

CIRCL / MISP Project

https://www.misp-project.org/

Reference snapshot: 29 October 2026

What you will be able to do

1. Select and discover modules for an analyst's task.
2. Connect MISP to a working module service.
3. Operate configuration, caching and query diagnostics.
4. Generate exports and distinguish them from workflow actions.
5. Use the standalone website and develop a local module.

Hands-on route

The accompanying lab uses synthetic data, local processing and an explicit dry run. No provider account or running MISP instance is needed.

A dated snapshot, not a promise about every install

- Reviewed against upstream `main`, commit `3a903305cf37`, on 29 October 2026.
- Package metadata declares version `3.1.1`. Features on this branch can postdate a published package.
- Source inventory: **270 candidate module files** across four families.
- Installed dependencies, credentials and service reachability determine which integrations can actually run.

Inspect the running service

Check `/version`, `/modules` and `/openapi.json`. A source file count is not the number of operational integrations.

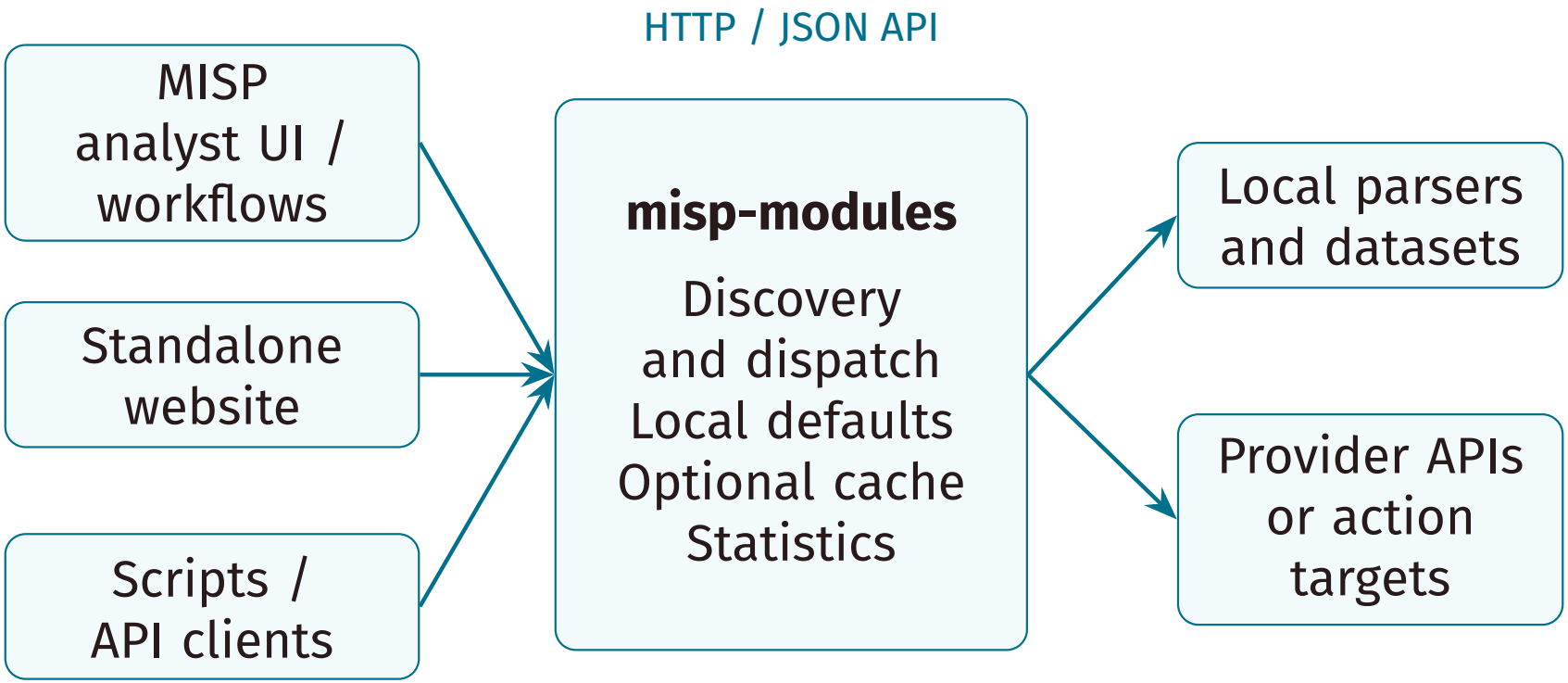
Pinned source links and the inventory method are in this deck's README.

Four families, different contracts

Family	Directory	Files	Purpose
Expansion	`expansion`	203	Enrich an observable; some modules also support hover.
Import	`import_mod`	22	Parse uploaded data or retrieve a report into MISP results.
Export	`export_mod`	32	Transform supplied MISP data into a downloadable format.
Action	`action_mod`	13	Run an operation from a workflow, such as a notification or publication.

Directory names are loader families. Metadata uses `expansion/hover`, `import`, `export` and `action`.

One service, several clients



The Python service can run without a MISP server. The website is a separate application.

Use and understand modules

Choose a module by the question you need to answer

Analyst question	Examples in the current source
What does this URL contain?	`extract_url_components`: local structured parsing.
What infrastructure is related?	`dns`, `reversedns`, `circl_passivedns`: different DNS evidence.
Is this observable known or noisy?	`hygiene`, `hashlookup`, `greynoise`.
What vulnerability context exists?	`cve`, `gcve_cpe`, `rapid7_attackerkb`.
What does analysis tooling know?	`mwdb_query`, `assemblyline_query`, sandbox query/submit integrations.

Read the module's input types, configuration, dependencies and side effects. A lookup and a submission are different operational choices.

Discover what is really loaded

```
curl --fail --silent --show-error \
  http://127.0.0.1:6666/modules |
  jq '.[] | {name, type, mispattributes, meta}'
```

- `name`: API identifier, usually the Python filename.
- `meta.name`: human-readable display name, when supplied.
- `mispattributes`: input/output capabilities and format.
- `meta.config`: expected settings, not configured secret values.
- `meta.module-type`: capabilities used by MISP.

Two expansion formats still coexist

Simplified / legacy

```
{"module": "training_uppercase",
 "text": "misp training"}

{"results": [
  {"types": ["text"],
   "values": ["LOCAL: MISP
    TRAINING"]}
]}
```

MISP standard

```
{"module": "...",
 "attribute": {
   "type": "url",
   "value": "...",
   "uuid": "..."
 }}

{"results": {
  "Attribute": [], "Object": []
}}
```

The module's introspection selects the contract. `format: misp_standard` enables structured results and references.

A complete local enrichment request

Save or use the included `examples/url-request.json`:

```
{
  "module": "extract_url_components",
  "attribute": {
    "type": "url",
    "value": "https://portal.example.org/a?item=1",
    "uuid": "13d8d18b-784a-42a0-9a58-a9cddbfda561"
  }
}
```

```
curl --fail --silent --show-error \
  http://127.0.0.1:6666/query \
  -H 'Content-Type: application/json' \
  --data @examples/url-request.json
```

Read the result as evidence

- URL parsing produces a `url` object, its component attributes, and a reference to the source attribute.
- Current DNS enrichment can express a `dns-record` object and related records, rather than flattening a lookup to one IP.
- Keep source identifiers and object references when pivoting.
- Review provenance, time, relevance and sharing constraints before adding or using results.

Enrichment is not a detection verdict

Context, reputation, warninglist matches and generated rules still need an analyst's interpretation.

Connect MISP: service discovery and user access

In **Administration** → **Server Settings & Maintenance** → **Plugin settings**:

1. Configure the relevant service family: `Enrichment`, `Import`, `Export` or `Action`.
2. Set its `services_enable`, `services_url` and `services_port` settings.
3. Reload settings, enable the desired module and set its credentials.
4. Check organisation restrictions and supported input types.

Example

`Plugin.Enrichment_services_enable`
Enable the enrichment family, then enable/configure its individual modules.

Verify connectivity from the MISP host/container; its localhost may not be the module server.

Deploy and operate the service

Choose the dependency footprint

Install	Python command	What to expect
Base	`pip install .`	Core service, workflow support and modules whose imports are satisfied.
Common extras	`pip install '.[minimal]'`	A broader set of Python dependencies, intended without extra system packages.
Broad extras	`pip install '.[all]'`	Wider coverage; some native tools/libraries are also required.

The snapshot requires **Python 3.10 or newer, below 4.0**. The lab uses Python 3.12 and the base installation.

Use a virtual environment and a supervised process

```
git clone https://github.com/MISP/misp-modules.git
cd misp-modules
python3 -m venv .venv
. .venv/bin/activate
python -m pip install .
misp-modules -l 127.0.0.1 -p 6666
```

- Pin a reviewed revision for a reproducible workshop.
- For development, use an editable installation or Poetry.
- In production, use a service manager with the actual venv executable and service account.

No global sudo pip installation or rc.local startup is needed. See the README for the pinned lab setup.

The service API is discoverable

Endpoint	What it tells you
`GET /healthcheck`	The HTTP service is responding.
`GET /version`	Installed package version.
`GET /modules`	Loaded module metadata and capabilities.
`GET /openapi.json`	Startup-generated API specification.
`GET /openapi`	Interactive Swagger UI using that specification.
`POST /query`	Dispatch a request to one selected module.
`GET /statistics`	Query counts, outcomes, latency and optional cache metrics.

A healthy service does not prove that a provider query will succeed.

Configure the right layer

- **MISP settings**: enablement, organisation restrictions and configuration sent in a query's `config`.
- **Module-server defaults**: optional `--module-config` JSON for credentials, URLs and other settings.
- **Service environment**: used by integrations that explicitly read environment variables, including some publication actions.
- **Website settings**: a separate application and stored module settings; not the MISP database.

Local defaults: explicit precedence

```
{
  "version": 1,
  "modules": {
    "training_uppercase": {
      "config": {"prefix": "LOCAL: "},
      "fallback_when_empty": ["prefix"]
    }
  }
}
```

Start with `–module-config examples/module-defaults.json`.

- Missing settings use defaults; explicit API values win.
- Empty strings/null use defaults only for named `fallback_when_empty` keys.
- Read once at startup; restart after edits or credential rotation.

Optional caching: choose read-only expansions

- Enable with `--cache-config`; disabled by default.
- Opt in each expansion module, with a TTL, entry limit and response-byte limit.
- Match the **whole effective request**, after local defaults: UUIDs, context, credentials and timeout matter.
- Identical concurrent requests can share one execution.
- Cache entries and counters are process-local and reset on restart.

Choose by behavior, not just family

Some expansion modules submit data or cause side effects. Caching one can suppress a repeated operation.

Observe queries, failures and cache savings

```
curl --fail --silent --show-error \
  http://127.0.0.1:6666/statistics | jq .
```

- Per-module and aggregate counts include loaded, unused modules.
- Distinguish module errors, exceptions and timeouts.
- Compare success/error rates, latency and in-flight requests.
- Optional cache metrics include hits, misses and coalesced queries.
- Query counts measure service requests, not provider HTTP calls.

Operate the service within its trust boundary

- Keep the API reachable only by intended clients; it has no built-in authentication layer in the reviewed dispatcher.
- Review which modules contact providers or submit samples.
- Keep local credentials outside the checkout and readable only by the service account.
- Use normal debug logs first. `--debug-extended` logs outbound `requests` exchanges, including credentials and full payloads.

Troubleshoot the failing layer

Symptom	First checks
Cannot reach service	Process, listen address, container network, host/port.
Module absent	Correct venv/package, optional dependencies, loader logs, restart.
Present, absent in MISP	Correct family settings, metadata type, reload settings.
Visible, unavailable to user	Module enablement, organisation restriction, supported input.
HTTP 200, failed query	Inspect `error`/`errors`; config, quotas, provider response.
Unexpected stale result	Effective request, TTL, correct process and cache configuration.

Import, export and act

Import: turn external material into MISP data

- Import sources vary: file upload, pasted text, or remote report retrieval.
- Check `inputSource`, `userConfig` and module settings.
- File-oriented API requests commonly carry base64 `data`.
- Review returned attributes, objects and references before ingestion.
- Examples include email, sandbox reports, PCAP and TAXII integrations.

Use the module's actual contract

A remote-report importer may need configuration instead of an uploaded file. File and report inputs are not interchangeable.

Export modules are an active part of the platform

Target	Examples / output
Detection and SIEM	`zeek_intel_export`, `elastic_ecs_export`, `splunk_export`, `wazuh_cdb_export`.
Host investigation	`osqueryexport`: query pack by default, optional raw SQL.
Network policy	PAN-OS, Junos, FortiOS, ASA, RouterOS, PF and nftables exporters.
Data interchange	CSV and other module-specific formats.

Native MISP exports also remain useful. Choose by target format and supported data, not an old assumption that module exports will disappear.

Lab: generate a Zeek Intel file

Use the included `examples/export-request.json`.

```
curl --fail --silent --show-error \
  http://127.0.0.1:6666/query \
  -H 'Content-Type: application/json' \
  --data @examples/export-request.json > /tmp/zeek-response.json

python3 -c 'import base64,json; r=json.load(open("/tmp/zeek-
    response.json")); print(base64.b64decode(r["data"]).decode("
    ascii"))'
```

Expected: `example.org` and `192.0.2.10`, with native Zeek types. The non-IDS attribute is excluded by default.

Exporting a file and publishing an action differ

	Export	Workflow action
Main result	A file or generated representation.	An operation and its status/result.
Examples	Zeek feed, osquery pack, firewall fragment.	Notification, local pack write, SIEM/EDR publication.
Credentials	Often unnecessary for local generation.	Often needed for the target; some support credential-free dry runs.
Completion	Output generated.	Check acknowledgement, partial failures and activation requirements.

Workflow actions connect intelligence to operations

- Notifications: Slack, Matrix, IRC, Signal and Nextcloud Talk.
- Target publication: CrowdStrike IOC, SentinelOne IOC, QRadar reference sets and Wazuh CDB lists.
- Local artifact publication: `osquery_pack_publish`.
- Inspect action metadata: parameters, blocking behavior, filters and expected MISP-core format.
- Supply the intended data selection and explicit target scope.

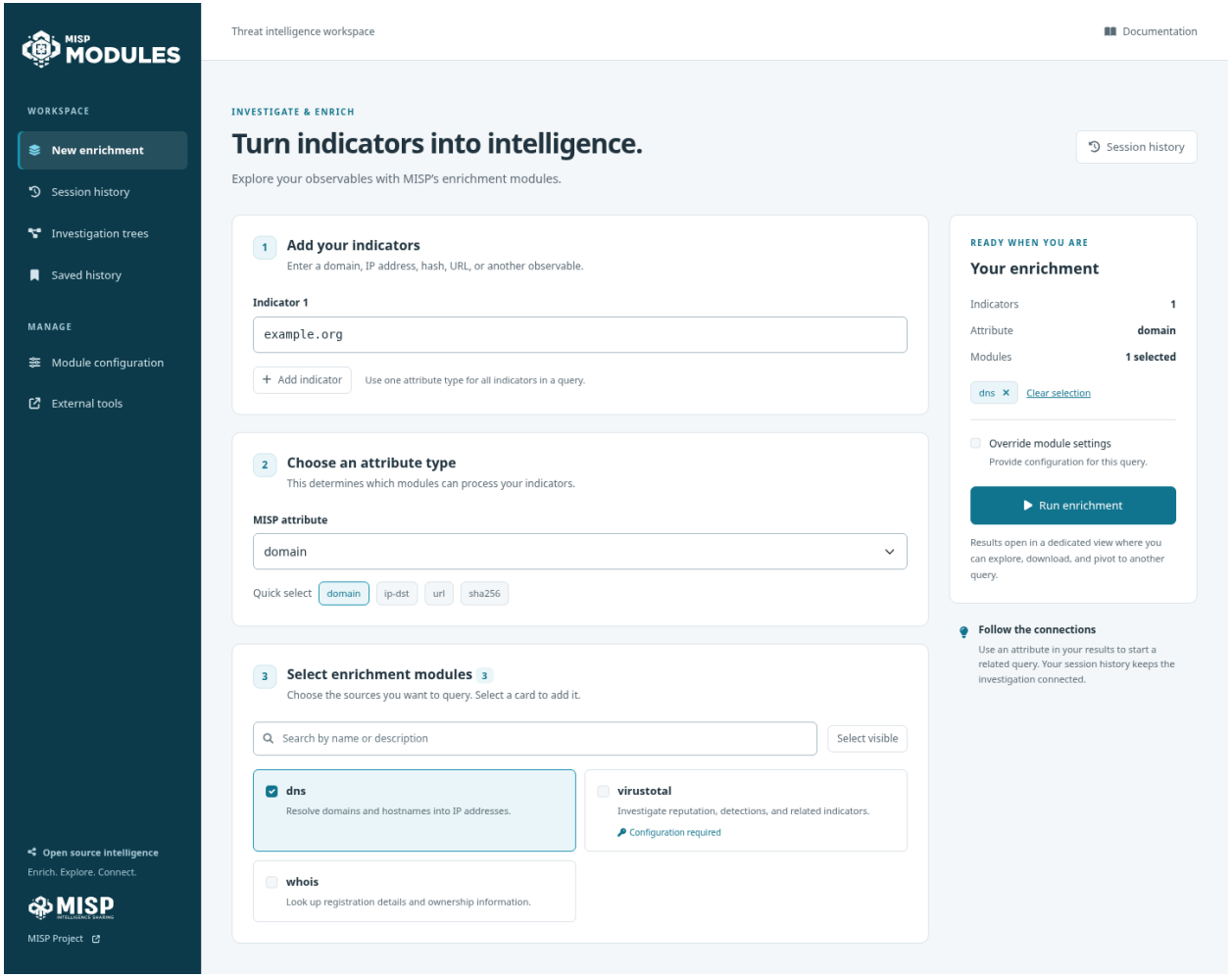
Lab: plan an osquery publication

```
{
  "module": "osquery_pack_publish",
  "params": {
    "pack_name": "training",
    "dry_run": "true",
    "overwrite": "false"
  },
  "data": [{"Event": {"Attribute": [
    {"type": "ip-dst", "value": "192.0.2.10", "to_ids": true}
  ]}}]
}
```

Expected: `data.status = planned`, a pack and provenance. This dry run writes no pack and executes no endpoint queries.

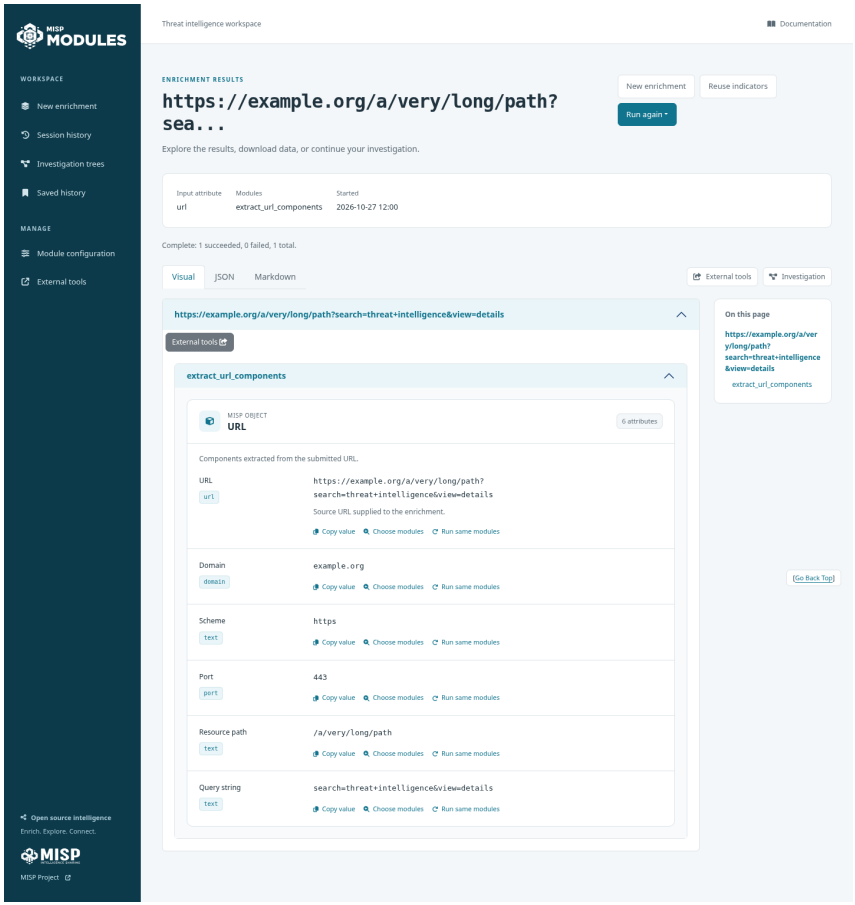
The standalone website

An enrichment workspace without a MISP instance



Current upstream documentation screenshot; sample catalogue from browser fixtures.

Structured results and investigation pivots



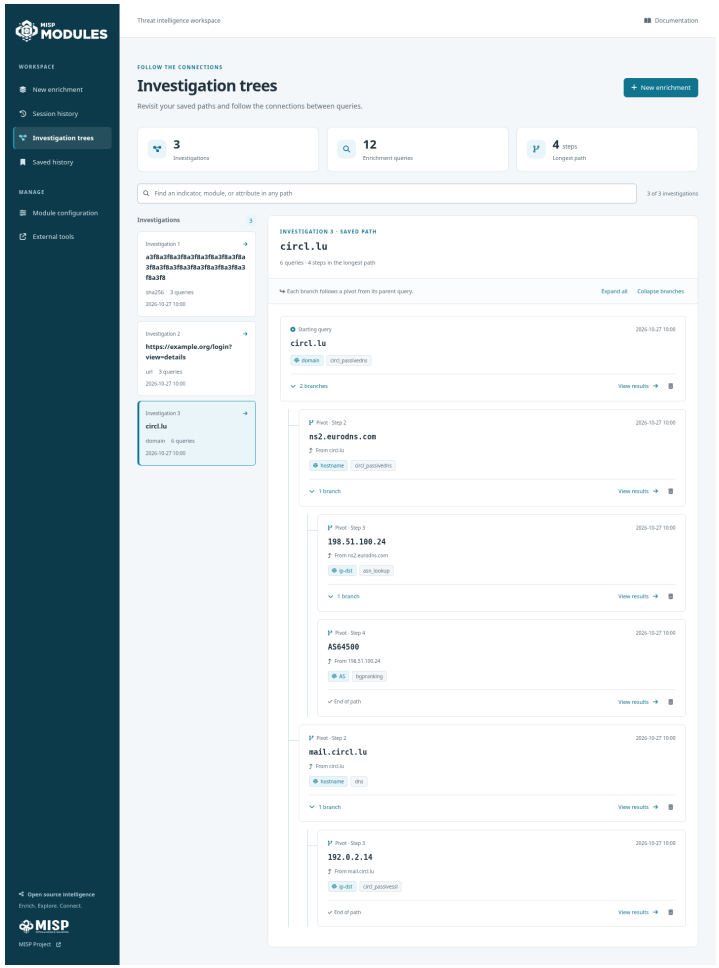
- Separate object cards and flat attributes.
- Visual, JSON and Markdown views.
- Copy complete values or start a related enrichment.
- Download results and use configured external tools.

Source: website/doc/interface.md and `object_result_cards.png` at the reviewed revision.

Import and export are website workflows too

- **Import data**: choose an active module and provide a supported upload, paste or remote-report configuration.
- **Export data**: provide MISP JSON, or choose a CIRCL OSINT feed event, then configure an active exporter.
- Results, text previews, decoded downloads and history use the shared interface.
- With `ADMIN_PASSWORD` configured, exports and their saved results require an administrator login.

Keep the investigation path



Session history, saved trees and related-query paths in the current upstream interface.

Develop and verify a module

A small interface, a real contract

- `introspection()`: supported input/output or import/export setup.
- `version()`: metadata, capabilities and configuration schema.
- `dict_handler(request=...)`: dictionary execution path supported by the current dispatcher.
- `handler(q=...)`: legacy JSON interface, still supported.
- Return useful `error` messages and valid result envelopes.

Choose the result format deliberately

The lab's simple text module uses legacy results. For structured enrichment, start from a current `misp_standard` module and retain source UUIDs and references.

Metadata is part of discovery

```
moduleinfo = {
    "version": "1.0",
    "author": "MISP Project training contributors",
    "name": "Training uppercase",
    "description": "Uppercase synthetic training text without
    network access.",
    "module-type": ["expansion", "hover"],
}
moduleconfig = ["prefix"]
mispattributes = {"input": ["text"], "output": ["text"]}
```

`training_uppercase.py` gives the API identifier. The display name is *Training uppercase*.

A complete handler, with validation

```
def dict_handler(request):
    if not isinstance(request, dict):
        return {"error": "Expected a request object."}
    value = request.get("text")
    if not isinstance(value, str) or not value.strip():
        return {"error": "A nonempty text value is required."}
    config = request.get("config", {})
    if not isinstance(config, dict):
        return {"error": "config must be an object."}
    prefix = config.get("prefix", "")
    if not isinstance(prefix, str):
        return {"error": "prefix must be a string."}
    return {"results": [{"types": ["text"], "values": [prefix +
    value.upper()]}]}
```

Load a local module without modifying the package

```
examples/custom/
  expansion/
    training_uppercase.py
```

```
misp-modules -l 127.0.0.1 \
  --custom "$PWD/examples/custom"
```

- Pass the custom *root*, not its `expansion` child.
- The loader discovers valid `.py` files; no `__all__` edit is needed.
- Restart, check logs and `/modules`, then enable/configure it in MISP.

Test behavior across the service boundary

- Compare dictionary and legacy handlers on valid and invalid input.
- Verify discovery and generated OpenAPI metadata.
- Check structured output and references using a local parser.
- Decode an export and verify indicator selection.
- Assert action status is `planned` in the dry run.
- Exercise defaults, API overrides, empty-value fallback, cache hits and error reporting.

The included `examples/verify_lab.py` performs 12 functional/contract tests.

Workshop: discover, transform, verify

1. **Discover**: find the lab modules and their formats in `/modules` and `/openapi.json`.
2. **Transform**: parse a URL, decode a Zeek file and inspect an osquery publication plan.
3. **Develop**: change the custom module's prefix and test a request override; inspect cache metrics and error responses.

```
python3 examples/verify_lab.py \
  --url http://127.0.0.1:6666 -v
```

Follow the README's full startup command before running the verifier.

What to take into the next integration

1. Discover the **running** module set and its contracts.
2. Separate service configuration, MISP enablement and provider readiness.
3. Preserve evidence and review enrichment before operational use.
4. Distinguish file generation, publication and activation.
5. Test through the dispatcher when relying on its configuration or cache.

Keep the material current

Recheck the pinned source audit, exercise results and module-specific documentation before teaching a newer revision.

References and next steps

- MISP/misp-modules: code and current catalogue
- Module documentation: install, contribute and module guides
- Standalone website and update instructions
- Standalone CLI client
- This deck's `README.md`: pinned sources, trainer notes, reproducible lab and validation scope.
- Continue with *Contributing MISP modules with an AI agent*; see the companion `a.14-ai-assisted-misp-modules` deck.