Custom extension development for EclecticIQ Platform

Build custom extensions for tailored integrations with EclecticIQ Platform

Last generated: October 20, 2017

Table of contents

Create custom enrichers

Implement custom extensions to integrate EclecticIQ Platform with external intel providers and data sources through incoming feeds and enrichers, as well as to publish platform intel downstream in your prevention and detection toolchain.

About extensions

EclecticIQ Platform integrates with many external prevention/detection solutions and intel providers. It can exchange information through feeds, retrieve data through enrichers, and it can communicate with third-party systems through its API and ad-hoc apps that implement interoperability with specific products such as Splunk and IBM QRadar.

EclecticIQ Platform ships with out-of-the-box, ready-to-use enrichers to augment cyber threat intel with observables providing additional context. It also includes a web-based UI to create incoming and outgoing feeds, as needed.

Besides the default feeds and enrichers, you can create and implement your own. Custom feeds and enrichers implemented by third-parties other than EclecticIQ are called extensions, since they extend the platform native feature set. You can create extensions to implement additional transport types or content types for incoming or outgoing feeds, as well as new enrichers to poll data from specific intel providers.

Create enricher extensions

Before getting your hands dirty, have a look at the main steps to create an enricher extension from a boilerplate:

- Download, clone, or copy the ***eclecticiq-extension-example*** `(https://github.com/eclecticiq/platform-extensions/tree/master/eclecticiq-extension-example)` extension.
- Import dependencies.
- Create a JSON schema for the UI, if your enricher extension features a UI.
- Set a schema definition for validation.
- Define the enricher type and its instantiation through decorator functions.
- Include the necessary logic to configure the enricher behavior:
  - Define the tasks the enricher should execute;
  - Define the extract types you want the enricher to look for and retrieve.
  - Configure appropriate transport and content types to handle data formats and retrieval.
- Restart Supervisor, so that all managed processes can configure the newly added extension in **Data configuration > Enrichers**.
- Enable the extension.
- Initialize the extension by running the fixtures (applies only to enricher extensions).

Prepare the boilerplate

To make it easier to create custom enricher extensions, you can use our boilerplate enricher: *eclecticiq-extension-example*.
It is a sample enricher that augments entities with social URI observables polled from Twitter and/or Facebook.
Use it as a scaffold you can rework and customize into the desired enricher extension.

- Download, clone or copy the ***eclecticiq-extension-example*** `(https://github.com/eclecticiq/platform-extensions/tree/master/eclecticiq-extension-example)` extension, save it locally, and decompress it, if necessary.
- Rename the directories as needed.
- In the root directory, open ***setup.py*** `(https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/setup.py)`:

```
from setuptools import setup, find_packages

setup(
    name='eclecticiq-extension-example',
    version="1.0",
    description="Example extension for EclecticIQ Platform",

    # Look for packages to build the extension
    packages=find_packages(),

    # List dependencies the extension requires
    install_requires=[
        'eiq-platform'
    ],

    # Look for and include data files, if applicable
    include_package_data=True,

    # Set an entry point for the extension
    # so that it can initialize
    entry_points={
        'eiq.extensions': [
            'example = eiq.extensions.example:prepare_extension'
        ],
    }
)
```

Edit the setup file

These are the ***setup.py*** `(https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/setup.py)` parts you can, and should, edit as applicable:

`name`

In the extension `name`, change `example` to a more meaningful name for your enricher extension, but leave the `eclecticiq-extension-` prefix as is.
Example:

```
name='eclecticiq-extension-fraud-ip-observables',
```

`version`

Change the `version` number as appropriate.

Make sure you implement versioning for your enrichers.
For example, increment the version number after adding or removing arguments/parameters.
The platform checks `version`, and it won't load the enricher if it detects parameter changes without a corresponding version number increment.
Example:

```
version="1.1",
```

`description`

Change the `description` value, so that it provides basic details about the enricher extension.
Example:

```
description="Custom extension to retrieve and save fraudulent IPs as observables",
```

Include dependencies

Add to the `install_requires` list the Python libraries and modules your enricher extension needs to access for it to work as expected.
`eiq-platform` is a mandatory dependency, it needs to always be included in the list.
Other Python libraries and modules you need to import and include in this list vary, depending on the specific enricher extension you are building.

Example:

```
install_requires=[
    'eiq-platform',
    'requests',
    'cabby',
    'furl'
],
```

Set the entry point

`eiq.extensions` is the designated entry point referring to the extension definitions.
The platform needs this pointer to recognize, load, and register extensions. Do not remove it.
Change `example` to a more meaningful name for your enricher extension, but leave the `eclecticiq.extensions.` prefix as is.
Example: `eclecticiq.extensions.fraud-ip-observables`

Example:

```
'eiq.extensions': [
    'fraud-ip-observables = eclecticiq.extensions.fraud-ip-observables:prepare_extension'
],
```

Define the enricher initialization

This part of the procedure customizes the fixtures you will need to run later to initialize the enricher, after enabling it.

`prepare_extension` defines how to initialize the enricher extension you are building.
In `prepare_extension` you specify what the extension should get ready before you execute it for the first time.

Change the `Extension` return values as applicable. Use the actual, correct names, descriptions, and values you define, set, and plan to use in your enricher extension.

Open ***init.py*** (`https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/__init__.py`).

Boilerplate:

```
from eiq.extension_api import Extension

from .enrichers import enrich_from_social_network
from .transformers import transform_malware_domainlist_csv
from .provider import oauth_http_download

def prepare_extension():
    return Extension(
        name=__name__,
        description='Example extension for EclecticIQ Platform',
        enrichers=[enrich_from_social_network],
        transformers=[transform_malware_domainlist_csv],
        transports=[oauth_http_download]
    )
```

Example:

```
# Import the 'Extension' class from the extension API
from eiq.extension_api import Extension

# Import your custom enricher
from .enrichers import enrich_fraud_ip_observables
# Import the data formats the enricher needs to handle
from .transformers import <content_type>
# Import the data carrier the enricher uses to exchange data
from .provider import <transport_type>

# Define how to initialize the custom enricher
def prepare_extension():
    return Extension(
        name=__name__,
        description='Custom extension to retrieve and save fraudulent IPs as observables',
        enrichers=[enrich_fraud_ip_observables],
        transformers=[<content_type>],
        transports=[<transport_type>]
    )
```

Start by importing the platform class defining extensions, as well as your custom extension, so that we can initialize it. You import the standard extension class for the platform through the `extension_api`. This is the dedicated API to use when developing custom extensions.

Import the enricher from ***enrichers.py*** (https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/enrichers.py).
This module defines, among others, enricher extension instance and type, as well as any functions containing the logic driving the enricher behavior, and the UI schema, if applicable.

Import one or more content types from ***transformers.py*** (https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/transformers.py), so that the enricher can recognize and handle the expected data formats.
This module defines the available content types for the data the enricher extension handles: this is where you import the allowed data formats for the enricher from.

Import a transport type from ***provider.py*** (https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/provider.py), so that the enricher can send and receive the expected data.
This module defines the available transport types for the data the enricher extension handles.

Change the `description`, `enrichers`, `transformers`, and `transports` metadata values `Extension` returns, so that they reflect the actual name, description, enricher type, data format, and data carrier to use in the enricher extension.

Build the enricher

Enricher extension logic and any necessary functions that drive the enricher behavior live in the ***enrichers.py*** (https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/enrichers.py) module.

Boilerplate:

```
import json

import requests
from marshmallow import Schema, fields

# Import the enricher extension dependencies
from eiq.extensions_api import (
    EnrichmentResult,
    enricher_instance,
    enricher_type,
)

# Define the Marshmallow schema for the enricher params
# The UI schema is validated against the params defined here
class SocialURIEnricherSchema(Schema):
    check_twitter = fields.Boolean(required=True)
    check_facebook = fields.Boolean(required=True)

# Define the UI schema
ui_form_schema = [
    {
        "label": "Check Twitter",
        "name": "check_twitter",
        "required": True,
        "type": "checkbox",
        "format": "bool",
        "hint": "Should the enricher query Twitter"
    },

    {
        "label": "Check Facebook",
```

```
        "name": "check_facebook",
        "required": True,
        "type": "checkbox",
        "format": "bool",
        "hint": "Should the enricher query Facebook"
    },
]

@enricher_instance(
    # Increment the version number if defaults change after
    # deploying the enricher.
    # For example, if it gets new/additional parameters
    version=1,

    # Enricher name and description displayed to users
    name='Example Social URI Enricher',
    description=('Query for registered Twitter/Facebook '
                 'accounts with provided handle/name'),

    # Boolean switch to enable or disable the enricher by default
    is_active=True,

    # If the enricher creates entities, this is the default reliability
    # value assigned to them
    source_reliability='C',

    # Additional enricher parameters.
    # They must match the name and type configured in
    # 'ui_form_schema' and in the Marshmallow schema
    check_twitter=True,
    check_facebook=True,
)

@enricher_type(
    # Define the observable types the enricher supports
    input_extract_types=['handle', 'name', 'person'],

    # Assign a UI schema to the enricher editor page in the web UI
    parameter_ui_form_schema=ui_form_schema,

    # Define the validation and deserialization UI schema
    parameter_serialization_schema=SocialURIEnricherSchema,

    # URL templates link users to the enricher source
    source_urls={
        'handle': 'https://twitter.com/${input}',
        'name': 'https://twitter.com/${input}',
        'person': 'https://twitter.com/${input}',
    },
)

# Define the logic driving the enricher behavior
def enrich_from_social_network(
        # Request arguments
        extract_type,
        extract_value,
        check_twitter,
        check_facebook):
    # The response should return:
    # - A list with matching observables
    # - A list with the raw responses, for reference
    enrichment_extracts = []
    raw_responses = []  # Keep raw response headers for user reference

    if check_twitter:
        # Send an HTTP HEAD request to Twitter
```

```
        # to see if it takes the handle
        twitter_url = 'https://twitter.com/{}'.format(extract_value)
        response = requests.head(twitter_url)

        if response.ok:
            # Return and append the raw response(s)
            raw_responses.append(dict(response.headers))
            # Return and append supported observable types and values
            enrichment_extracts.append({
                'kind': 'uri',
                'value': twitter_url
            })

    if check_facebook:
        # Send an HTTP HEAD request to Facebook
        # to see if it takes the handle
        facebook_url = 'https://facebook.com/{}'.format(extract_value)
        response = requests.head(facebook_url)

        if response.ok:
            raw_responses.append(dict(response.headers))
            enrichment_extracts.append({
                'kind': 'uri',
                'value': facebook_url
            })

    return EnrichmentResult(
        # Return the raw responses as UTF-8 JSON
        raw_data=json.dumps(raw_responses).encode('utf-8'),
        # Return a key/value pair JSON list with the retrieved observables
        extracts=enrichment_extracts,
        # Return a list with entities, if applicable
        entities=[])
```

Let's break it down.

Import dependencies

Make sure you include the necessary Python libraries and modules in ***enrichers.py*** (`https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/enrichers.py`), so that the custom enricher extension can access the functionality required to work as expected.

The Python libraries and modules you may need to make available to your custom extension vary, depending on the extension design, scope, and purpose.

For example:

Dependency	Description
`import requests`	Adds handy automation to **HTTP requests** (`http://docs.python-requests.org/en/master/`).
`from marshmallow import Schema, fields`	Marshmallow schemas validate UI schemas and form input.

Dependency	Description
`from eiq.extensions_api import (EnrichmentResult, enricher_instance, enricher_type,)`	The foundations to build your enricher on.

```
import json

import requests
from marshmallow import Schema, fields

# Import the enricher extension dependencies
from eiq.extensions_api import (
    EnrichmentResult,
    enricher_instance,
    enricher_type,
)
```

Set the UI schema definition

If your enricher extension features a UI, you need to include a UI JSON schema, which you need to validate.
The schema definition to validate UI schema and form input is based on a Marshmallow schema definition.

The **Marshmallow schema** `(https://marshmallow.readthedocs.io/)` defines the behavior of the controls and the components on the UI form, and the `ui_form_schema` JSON schema needs to match it to pass validation.

First, import the following classes from Marshmallow:

```
from marshmallow import Schema, fields
```

Then, set the Marshmallow schema definition to validate the `ui_form_schema` JSON schema against.
You can customize the Marshmallow schema definition as needed.

Example:

```
# Define the Marshmallow schema for the enricher params
# The UI schema is validated against the params defined here
class SocialURIEnricherSchema(Schema):
    check_twitter = fields.Boolean(required=True)
    check_facebook = fields.Boolean(required=True)
```

Lastly, include `parameter_serialization_schema` in the `enricher_type` decorator in ***enrichers.py*** `(https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/enrichers.py)`, and set it so that it points to the appropriate schema definition name for the UI schema validation.

Example:

```
# Define the validation and deserialization UI schema
parameter_serialization_schema=SocialURIEnricherSchema,
```

Define the UI schema

If your enricher extension requires a UI frontend where users can make selections and set specific options, you need to include a UI schema in JSON format.
Each JSON field in the schema defines a UI component to implement in the extension. For example, an input field, or a checkbox.

Include the UI schema as a JSON array inside ***enrichers.py*** `(https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/enrichers.py)`.
You can customize the UI schema as needed.

Example:

```
    # Definition of the UI form rendered in the platform UI
    # Data configuration > Enrichers > <enricher_name> > Edit enricher task
    # 'ui_form_schema' is the UI schema name; do not change it.
    ui_form_schema = [
        {
            "label": "Check Twitter",
            "name": "check_twitter",
            "required": True,
            "type": "checkbox",
            "format": "bool",
            "hint": "Should the enricher query Twitter"
        },

        {
            "label": "Check Facebook",
            "name": "check_facebook",
            "required": True,
            "type": "checkbox",
            "format": "bool",
            "hint": "Should the enricher query Facebook"
        },
    ]
```

Then, include `parameter_ui_form_schema` in the `enricher_type` decorator in ***enrichers.py*** `(https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/enrichers.py)`, and set it so that it points to `ui_form_schema`.

Example:

```
# Assign the UI schema to the enricher editor page in the web UI
parameter_ui_form_schema=ui_form_schema,
```

You can define any UI schema that satisfies your requirements, provided it complies with the following guidelines:

- The UI schema format must be valid JSON.
- A UI schema for a form is a user-defined list of fields.

- Define each field with key/value pairs.
- Each key/value pair describes an attribute of the field.

Field attributes

You are free to define the naming convention and the terminology for the field names. However, field attributes are constrained and predefined. Each field takes at least two or more attributes.

`name` and `type` are required attributes, and you always need to include them in a field description.
All other attributes are optional.

`name`

The name identifying the field in the JSON object.
This name is usually not displayed to users. It is included in the JSON object containing the field, the UI schema, and the extension schema that is returned when sending an API request to the `/api/extensions/` endpoint.
Example: *includeWhois*

`label`

The name of the field as displayed as a label on the resulting object in the UI form.
Example: *Include whois information*

`type`

It defines the type of field, that is, the object it represents on the UI form:

- `text`: a one-row text input field.
  It can take the following sub-attributes:
  - `format`: it defines and restricts the allowed input format for the field, which needs to be in the specified format value.
    Allowed values:
    - `datetime`
    - `host`
    - `url`
    - `email`
    - `regex`
    - `path`
    - `text`
    - `int`
    - `float`
    - `bool`
- `textarea`: a multiple row alphanumeric text input field.
- `password`: an alphanumeric input field that accepts a user password.
- `select`: a list with multiple options. Users can select one or more options.
  It can take the following sub-attributes:
  - `options`: a JSON array with key/value pairs. Each key/value pair defines one option.
    Format: `[{"name": "...", "value" :"..."},...]`
  - `multiple`: Boolean, either `true` or `false`. It defines whether users are allowed to make multiple selections.

- `radio`: a control element that allows users to select one option in a set of options.
  It can take the following sub-attributes:
  - `options`: a JSON array with key/value pairs. Each key/value pair defines one option.
    Format: `[{"name": "...", "value" :"..."},...]`
- `checkbox`: a control element that allows users to select/deselect, enable/disable an item or a feature.
  It can take the following sub-attributes:
  - `options`: a JSON array with key/value pairs. Each key/value pair defines one option.
    Format: `[{"name": "...", "value" :"..."},...]`
  - If you do not include the `options` sub-attribute, the `checkbox` type defaults to a single component accepting Boolean values, either `true` or `false`.
- `extra`: include this type if you want to include in your UI form any additional free-form parameters, for example HTTP headers.
  It can take the following sub-attributes:
  - `names`: a JSON array holding the name values of the extra free-form parameters. For example, the specific HTTP header names you want to add.
    Format: `["name1", "name2", ...]`
  - `allow_new`: Boolean, either `true` or `false`. It allows/denies adding new keys to the extra parameter list.

`required`

Boolean, either `true` or `false`.
It flags the field as either mandatory, that is, users must specify a value for the field, or optional.

`default`

Any value you specify for this attribute corresponds to the default value the field is pre-populated with (autofill).

`hint`

A tooltip text to give a short explanation of the field and the action the user should carry out.
Example: *Enter a numeric value between 1 and 10.*

`when`

It defines a conditional flow to show or hide the component when the specified criteria are met or not met.
Format: `{"component_x": "value_y"}`, that is, when `component_x` is set to `value_y`, the component the fields belongs to is displayed on the UI.

Define the enricher behavior

The foundations of your enricher extension are the `enricher_instance` and `enricher_type` decorators.

`enricher_instance`

This decorator defines the default values the enricher takes when it is instantiated in the platform.

The following arguments are mandatory:

- `version`: make sure you implement versioning for your enrichers.
  For example, increment the version number after adding or removing arguments/parameters.
  The platform checks `version`, and it won't load the enricher if it detects parameter changes without a corresponding version number increment.
- `name`: a human-readable, user-friendly name that helps user understand what the enricher does.
  The `name` value is displayed on the UI.

- `description`: a human-readable, user-friendly, short free-text description that helps user understand what the enricher does.
  The `description` value is displayed on the UI.
- `is_active`: the `True` or `False` Boolean value controls the state of the enricher **Enabled** checkbox on the UI: either selected or deselected, respectively.

The following arguments are optional:

- `source_reliability`: if your enricher extension is designed to create entities as an output, this parameter sets a default reliability value for the entity data source.
- Any custom parameters defining the Marshmallow schema definition to validate the UI schema.

Example:

```
@enricher_instance(
    # Bounce the version number if you
    # add, change or remove parameters
    version=1,

    name='Example Social URI Enricher',
    description=('Query for registered Twitter/Facebook '
                 'accounts with provided handle/name'),

    is_active=True,
    # Optional, if your enricher returns entities
    source_reliability='C',

    # Custom Marshmallow UI schema definition values
    # to validate the enricher UI, if applicable
    check_twitter=True,
    check_facebook=True,
)
```

`enricher_type`

This decorator defines the enricher extension behavior. It contains the specific logic and the functionality powering your enricher.

The following arguments are mandatory:

- `input_extract_types`: a list of observable types.
  The observable types in this list are the input data the enricher takes.
  When it runs, the enricher searches for enrichment data to augment the observable types in the list.
- `parameter_ui_form_schema`: it needs to refer to `ui_form_schema`. Do not change the parameter name.
- `parameter_serialization_schema`: it needs to refer to the Marshmallow schema definition. Do not change the parameter name.
- `source_urls`: a key/value pair dictionary that produces clickable links to external data sources related to the retrieved observables. Do not change the parameter name.
  - The key name needs to be an observable type included in `input_extract_types`.
  - The value is a URL with the following format: `https://<data_source.com>/${input}`.
    The `${input}` URL variable takes the `extract_value` value.

    The resulting URL includes the original query as query URL params.
    The resulting clickable link points to the original web site the enrichment data was obtained from.

Example:

```
@enricher_type(
    input_extract_types=['handle', 'name', 'person'],

    parameter_ui_form_schema=ui_form_schema,
    parameter_serialization_schema=SocialURIEnricherSchema,

    source_urls={
        'handle': 'https://twitter.com/${input}',
        'name': 'https://twitter.com/${input}',
        'person': 'https://twitter.com/${input}',
    },
)
```

def magic

Now it's time for the magic to happen. Define a workhorse function to do the grunt work.

The function you pass with the decorator should contain the necessary logic to search for and retrieve the desired input data, any conditional flow and error handling, and to output the input data as valid observables for the platform.

The following arguments are mandatory:

- `extract_type`: include this parameter, so that your function can return observable types matching `input_extract_types`.
- `extract_value`: include this parameter, so that your function can return values associated with the observable types defined in `input_extract_types`.
- Any custom-defined Marshmallow schema definition variables, if your enricher features a UI component.

The function should also create two variables to hold the returned data:

- `enrichment_extracts`: an empty list that will be populated with any matching observables included in the response.
- `raw_responses`: an empty list that will be populated with the raw responses, for reference.

Example:

```
def my_custom_enricher(
        # Mandatory arguments:
        extract_type,
        extract_value,
        ...,
        # Pass as arguments also the Marshmallow schema definition vars:
        check_twitter,
        check_facebook,
        ...
        ):

    # The function should return 2 lists:
    # - Any retrieved enrichment observables,
    # - The raw responses, for reference
    enrichment_extracts = []
    raw_responses = []  # Keep raw response headers for user reference

    ...
    # Your black magic happens here
    ...
```

return EnrichmentResult

Complete the function by defining the objects it should return in the response.

The `EnrichmentResult` class helps store and handle output data. It returns the following output:

- Raw response, that is, enrichment raw data (as UTF-8 JSON),
- Observables (as a list),
- Entities (as a list of entity IDs).

When building an enricher extension, it is a good idea to always use these data types to return, even when no data may be returned for observables or entities. The raw data response should always be included in the return arguments.

Example:

```
return EnrichmentResult(
        raw_data=json.dumps(raw_responses).encode('utf-8'),
        extracts=enrichment_extracts,
        entities=[])
```

Define transport and content handlers

Your enricher extension needs to include two more components:

- ***provider.py*** (`https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/provider.py`): defines a mechanism to transport data.
  Typically, a provider implements a data transport protocol, such as HTTP, FTP, TAXII, and so on.
  Enrichers require a provider that fetches data from a source, and then passes it on to the platform for further processing.
- ***transformers.py*** (`https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/transformers.py`): defines one or more mechanisms to convert retrieved raw data to the JSON format the platform can accept.

Define the data provider

To define the data provider for the enricher, you can use the ***provider.py*** (`https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/provider.py`) as a scaffold you can rework and customize into the desired data transport provider for your enricher extension.

- Begin by importing the necessary Python modules and dependencies to handle the desired source data content type. In the example, the transport type is HTTP using OAuth authentication.
- `extension_api` is the dedicated API to use when developing custom extensions.
  From `extension_api`, import the following functions:
  - `options`: the `provider_options` passes it as an argument to control and validate UI options.
    Marshmallow uses it to set and validate dynamic form options in the UI.
  - `provider`: decorator function that marks a function as a provider type.
  - `provider_options`: decorator function to define a set of configuration options for a data provider.

```
from requests_oauthlib import OAuth1Session

from eiq.extensions_api import options, provider, provider_options
```

- The `provider` decorator flags a function as a data transport type/data provider in the platform.
  The decorator takes the following arguments:
  - `name`: an alphanumeric string holding the internal name identifying the data provider.
  - `title`: an alphanumeric string an alphanumeric string displayed in the UI. It corresponds to the transport type name for an enricher or an incoming feed.
  - `description`: an alphanumeric string displayed in the UI. It corresponds to a short description about the data provider/transport type purpose.

```
# The ``provider`` decorator will mark a certain function as a type
# of provider, along with some metadata primarily to inform users.
@provider(
    name='eiq.providers.oauth_http_download',
    title='OAuth HTTP Download',
    description=('Download content the configured URL using OAuth 1 '
                 'authentication')
)
```

- The `provider_options` decorator enables you to define any custom or additional data provider settings that platform users can select and configure in the UI through the enricher form.

  The decorator can take any custom argument key names, depending on how the source data provider interface works. Argument keys can take the following values:
  - `options.boolean`: it works like a standard Python `bool` object; it takes either `True` or `False`.
    On the UI it displays a selection checkbox form element.
  - `options.datetime`: it works like a standard Python `datetime.datetime` object.
    On the UI it displays a date-time calendar picker form element.
  - `options.integer`: it works like a standard Python `int` object.
    On the UI it displays a text input form element.
  - `options.password`: it works like a standard Python `str` object.
    On the UI it displays a password input form element.
  - `options.string`: it works like a standard Python `str` object.
    On the UI it displays a text input form element.
  - `options.url`: it works like a standard Python `str` object.
    On the UI it displays a text input form element that validates the input as a valid URL.

Each of these parameters can take additional *kwargs* to include extra information displayed to users on the UI:

- `label`: an alphanumeric string displayed in the UI. It corresponds to a UI option caption.
- `hint`: an alphanumeric string displayed in the UI. It corresponds to a pop-up tooltip triggered by a mouseover event.
- `required`: flags a UI option as required or optional. It takes either `True` or `False`, respectively.
- `default`: depending on the option type, it pre-populates the UI element with a default value.
- `validate`: it takes a function to validate user input with.
  Design the validation function so that it takes *one* value, and it returns `False` if the value is not valid.

```
# With ``provider_options`` additional configuration for a provider
# can be specified. These options will show up to users in the UI in
# an HTML form.
@provider_options(
    url=options.url(label="URL", hint="URL to download content from"),
    client_key=options.string(label="Client key"),
    client_secret=options.password(label="Client secret"),
    resource_owner_key=options.string(label="Resource owner key"),
    resource_owner_secret=options.password(label="Resource owner secret"),
)
```

Time for more magic: define a data provider to do the grunt work. The function you pass with the decorator should contain the necessary logic to handle data transport for the designated data format, and it should return/submit the data for further processing in the platform.

The function logic varies, based on your specific needs, and on the source data provider transport and content types. Make sure the function can `ctx.submit` or `return` the data for further processing, depending on how you design it.

The data provider function takes the following arguments:

- `ctx`: this must always be the first argument.
- Any arguments specified in `provider_options`.

```
# The provider function must accept a ``ctx`` object as its first
# argument. Other optional arguments are:
# :param content_type: The name of the related content_type for the incoming
#                      feed this function runs for.
# :param log: A logger instance
def oauth_http_download(
        ctx,
        url,
        client_key,
        client_secret,
        resource_owner_key,
        resource_owner_secret,
        log):

    oauth1_session = OAuth1Session(
        client_key,
        client_secret=client_secret,
        resource_owner_key=resource_owner_key,
        resource_owner_secret=resource_owner_secret)

    rs = oauth1_session.get(url)
    rs.raise_for_status()
    log.info("Fetched {} bytes of data".format(len(rs.content)))
    ctx.submit(rs.content)
```

Define the data transformer

To define the data format conversion mechanism for the enricher, you can use the ***transformers.py*** (https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/eclecticiq/extensions/example/transformers.py) as a scaffold you can rework and customize into the desired input-data-format-to-JSON converter for your enricher extension.

- Begin by importing the necessary Python modules and dependencies to handle the desired source data content type. In the example, the input data is in *.csv* format.

```
import csv
import io
```

- `extension_api` is the dedicated API to use when developing custom extensions.
  From `extension_api`, import the following functions:
  - `declare_content_type`: it enables you to set the input data format your enricher expects to retrieve.
  - `transformer`: it enables you to configure the mechanism to convert the expected data input format to platform-friendly JSON.
    It returns a dictionary containing `"type": "package"` similar to a STIX package, or a dictionary containing `"type": "entities"`, and `"entities": [<dicts with valid entity data>]`.

```
from eiq.extension_api import (
    declare_content_type,
    transformer,
)
```

- Define the input data format your enricher expects to retrieve by assigning it `declare_content_type` as a value.
  In the example, the input data is in *.csv* format.
  Use `declare_content_type` to define the data format by setting the following parameters:
  - `id`: the identifier value is a **URN** `(https://tools.ietf.org/html/rfc2141)`.
    Format: `urn:<namespace_identifier>:<namespace_specific_string>:<version>`
    Example: `urn:malwaredomainlist:csv:1.0`
  - `name`: an alphanumeric string displayed in the UI. It corresponds to the enricher name.
  - `description`: an alphanumeric string displayed in the UI. It corresponds to a short description about the enricher purpose.
  - `file_extension`: sets the expected file extension type for the input data.
    It needs to correspond to the MIME type.
    Example: `csv`
  - `mime_type`: sets the expected **MIME type** `(https://www.iana.org/assignments/media-types/media-types.xhtml)` for the input data.
    It needs to correspond to the file extension.
    Example: `text/csv`

```
malware_domainlist_csv = declare_content_type(
    id="urn:malwaredomainlist:csv:1.0",
    name="Malware Domain List CSV",
    description="",
    file_extension="csv",
    mime_type="text/csv",
)
```

- The `transformer` decorator passes the function you define to handle data conversion from the configured input data format to platform-friendly JSON.
  It takes an `incoming_content_types` list type argument, whose values are the defined input data format names.

  Your custom function should always pass `blob` as an argument.
  The function logic varies, based on your specific needs, and on the source data provider transport and content types.
  The generic steps you should consider when building such a function are:
  - Set the expected data format and data encoding.
  - Define the logic to iterate through the input data, and to extract the desired data.
  - Create a variable to act as a container to hold the input raw data in a list.
  - Return a JSON dictionary with the desired data.
- You may want to delegate entity or observable object creation to a separate function. In the example, it is the `make_entity` function.
  - Since the input data format in the example is *.csv*, the function takes `row` as a parameter to extract the desired data, which is then stored in two JSON objects:
    - `data`: a JSON array containing the extracted entity or observable object data.
    - `meta`: a JSON array containing the extracted entity or observable object metadata.
  - Return the `data` and `meta` JSON objects.
- The function you pass with the `transformer` decorator uses the entity or observable object creation function to return a JSON array with the extracted (linked) entities.

```
@transformer(incoming_content_types=[malware_domainlist_csv])
def transform_malware_domainlist_csv(blob):
    # The blob object is of type bytes.
    # The expected encoding from malwaredomainlist.com
    # is utf-8
    csv_data = blob.decode('utf-8')

    # DictReader expects to iterate over lines of text, so we wrap the data:
    csv_data = io.StringIO(csv_data)
    csv_reader = csv.DictReader(
        csv_data,
        fieldnames=[
            'date_utc',
            'domain',
            'ip',
            'reverse_lookup',
            'description',
            'registrant',
            'asn'
        ]
    )

    # List of entities generated form each extracted csv row
    entities = [make_entity(row) for row in csv_reader]

    # Return (linked) entities
    return {
        'linked-entities': {
            'entities': entities,
        },
    }

def make_entity(row):
    data = {
        'type': 'indicator'
```

```
        'type': 'indicator',
        'title': row['domain'] + ' MalwareDomainList',
        'description': (
            'domain:         {}\n'
            'ip:             {}\n'
            'reverse lookup: {}\n'
            'description:    {}\n'
            'registrant:     {}\n'
            'asn:            {}'.format(
                row['domain'],
                row['ip'],
                row['reverse_lookup'],
                row['description'],
                row['registrant'],
                row['asn'])),
        'timestamp': row['date_utc'],
        'producer': {
            'description': 'Malware Domain List',
            'type': 'information-source',
        },
    }

    meta = {
        'manual_extracts': [
            {'kind': 'uri', 'value': row['domain'], 'level': 1},
            {'kind': 'ipv4', 'value': row['ip'], 'level': 1},
            {'kind': 'domain', 'value': row['reverse_lookup'], 'level': 1},
            {'kind': 'registrar', 'value': row['registrant'], 'level': 1},
            {'kind': 'asn', 'value': row['asn'], 'level': 1},
        ],
    }

    return {'data': data, 'meta': meta}
```

Package and deploy the enricher

Create a Python package for the extension you just built, and then use `pip install` to install it on the target system where the platform is running.

- Pack the extension to create a source distribution by running the following command(s):

```
$ python setup.py sdist
```

- Copy the packaged extension to the target location where you want to deploy it.
- Launch the platform Python virtual environment. To enable `venv`, run the following command(s):

```
$ source /opt/eclecticiq/platform/api/bin/activate
```

- In the virtual environment, install the extension by running the following command(s):

```
$ pip install /tmp/<your-custom-enricher-extension>.tar.gz
```

The `pip` example installs from `/tmp/` to avoid dealing with file access rights and permissions.

Restart processes

After completing the extension installation restart all *Supervisor* processes, so that all managed processes can configure the newly added extension in **Data configuration > Enrichers**.

- Reload Supervisor configurations and restart all Supervisor-managed processes by running the following command(s):

```
$ supervisorctl reload
```

Check that the enricher is registered

Make an API call with HTTPie

Verify that the extension is picked up and registered correctly.
To do so, save the following script to a *.sh* file, and then make it executable:

```
#!/bin/bash

#
# Helper for interacting with the platform API from the command line. This tool
# is a wrapper around httpie. It will take care of authentication and some
# other things. It takes at least three arguments:
#
# - host name
# - http method
# - relative url (without the /api/ part)
#
# Any additional arguments are propagated to httpie.
#
# Examples:
#
#   platform-api-http https://some.host/ GET /users/
#
#   platform-api-http localhost:8000 POST /some/endpoint/ @request-stored-in-a-file.json
#

set -e

readonly HTTPIE=http
readonly HTTPIE_ARGS="--check-status --verify=no"
readonly USERNAME=<valid_platform_signin_user_name>
readonly PASSWORD=<valid_platform_signin_password>

usage() {
    echo "Usage: $(basename $0) host method path [http-args]" > /dev/stderr
    exit 1
}

main () {
    local HOST="$1"
    local METHOD="$2"
    local API_PATH="$3"
    shift 3 || usage
    local TOKEN=$(${HTTPIE} ${HTTPIE_ARGS} POST "${HOST}/api/auth" username=${USERNAME}
password=${PASSWORD} | jq --raw-output '.token')
    local URL="${HOST}/api${API_PATH}"
    ${HTTPIE} ${HTTPIE_ARGS} ${METHOD} ${URL} Authorization:'"Bearer ${TOKEN}"' "$@"
}

main "$@"
```

To make the script executable, run the following command(s):

```
$ chmod +x ~/<filename>.sh
```

The script takes the following input parameters:

Parameter	Description
`https://<platform_host>/`	*Required* — The name of the host used to reach the API endpoint and to communicate with the API service.
`POST`, `GET`, `PUT`, `DELETE`	*Required* — A valid **HTTP method** (`http://www.restapitutorial.com/lessons/httpmethods.html`) to create, read, update, or delete a resource.

Parameter	Description
`/<API_endpoint>/`	*Required* — A relative URL pointing to the API endpoint that exposes the service you want to consume.
`?url=true&query=search-or-filter&params=4`	*Optional* — URL query parameters to send any additional search parameters and/or to filter the results returned in the response.

Besides appending URL query parameters, you can also send your request parameters as a JSON file. Example:

```
$ platform-api-http https://platform.host/ get /entities/ @request-parameters.json
```

To make a **HTTPie** (`https://httpie.org/`) call using the script, use the following format:

```
$ platform-api-http https://<host> <method> <api_path>
```

To check if the newly created enricher extension is correctly registered in the platform, make an API call to the `/extensions/` API endpoint:

```
$ platform-api-http https://platform.host.com get /extensions/
```

API call response

The call returns a JSON object containing all registered extensions.
Search for your enricher extension by name, description, or creation date.
If your enricher extension is included in the returned list, it is registered correctly.

Enable the enricher

- In the returned JSON object listing all registered extensions, search for your extension.
- In the extension JSON object, look for the following fields:
  - `id`: its value is a progressive integer that uniquely identifies the extension.
  - `is_active`: Boolean, either `True` or `False`. This flags the extension as either enabled or disabled, respectively.
- If `is_active` is set to `False`, the extension is currently disabled, and you need to enable it before you can use it.
  To enable the extension, make the following API call:

```
$ platform-api-http https://{platform_host} put /extensions/{id_number} data:='{ "data" : { "is_active" : true } }'
```

When you pass a JSON object with entity data in the body of your API request, you always need to wrap it in a data wrapper: `{ "data" : { ... } }`

- Enabled extension names should *not* be included in the `DISABLED_EXTENSIONS` list in the */opt/eclecticiq/etc/eclecticiq/platform_settings.py* configuration file.
  Any extensions on this list are automatically disabled.
  If an extension is on this list and you want to enable it, remove it from the list.

Initialize the enricher

The enricher extension is enabled, but not yet initialized. Platform enrichers though need to be initialized through fixtures before they become available.

Create and run the fixtures

- Log in to the system hosting the platform with either a user profile with admin rights, or with the `eclecticiq` user. You may need to grant the `eclecticiq` user admin privileges. If so, run the following command(s):
- Explicitly set the platform environment variable in the platform configuration file:

```
$ export EIQ_PLATFORM_SETTINGS=/opt/eclecticiq/etc/eclecticiq/platform_settings.py
```

- Launch the platform Python virtual environment. To enable `venv`, run the following command(s):

```
$ source /opt/eclecticiq/platform/api/bin/activate
```

- Start a Python shell:

```
$ /opt/eclecticiq/platform/api/bin/eiq-platform shell
```

- In the Python shell, create the fixtures for the extensions by running the following command(s):

```
>>> from eclecticiq.extensions.boilerplate import create_fixtures
>>> create_fixtures()
```

Test the enricher

“Nah, my code doesn’t need testing.”
(anonymous, booted)

Run extension validation

The `eiq-platform extensions validate` command validates the state of all registered and enabled extensions.

- If the validation completes successully, the command returns `Extensions look OK.`
- If the validation fails for one or more extensions, the command returns `Validation for {extension_name} failed:`, and it includes exception information for troubleshooting.

This command validates extensions based on the following criteria:

- The custom enricher extension name needs to match the `name` value defined in *setup.py*
- The extension name needs to be included in the platform `extension_registry` list, so that the platform can correctly recognize it and load it.
- The extension name should *not* be included in the `DISABLED_EXTENSIONS` list in the */opt/eclecticiq/etc/eclecticiq/platform_settings.py* configuration file.
- The extension needs to support the configuration parameters of any enricher tasks associated with it.

To run `eiq-platform extension validate`, do the following:

- Explicitly set the platform environment variable in the platform configuration file;
- Run the command:

```
$ export EIQ_PLATFORM_SETTINGS=/opt/eclecticiq/etc/eclecticiq/platform_settings.py
$ /opt/eclecticiq/platform/api/bin/eiq-platform extensions validate
```

Error handling

If the validation detects issues that can prevent extensions from working correctly, it returns error messages with a short description of the problem.

Error message	Description
`Validation for {extension_name} failed:`	The enricher extension did not pass validation. Review the exception information included in the error message to start troubleshooting.
`Extension '{extension_name}' is in the list of disabled extensions.`	The extension name is included in the `DISABLED_EXTENSIONS` list in the */opt/eclecticiq/etc/eclecticiq/platform_settings.py* configuration file. The extension enricher is disabled. Enable it.
`Unknown extension '{extension_name}'`	The extension name is not included in `DISABLED_EXTENSIONS` list, or in the `extension_registry` list. Start troubleshooting by checking packaging and deployment, and that the enricher extension is registered.

Error message	Description
`Invalid parameters for enricher ({extension_name}). Errors: {returned_error_list}`	One or more parameters are invalid. For example, they may be assigned a wrong format, or the enricher extension parameter schema is not validated correctly.
`Task has parameters but enricher does not support any.`	The enricher extension does not support one or more parameters defined in the enricher extension parameter schema.

Test the enricher with a test file

You can test your code programmatically by creating a test file that provides a valid sample request and a valid sample response for the enricher extension you built. The ***test_socialurienricher.py*** `(https://github.com/eclecticiq/platform-extensions/blob/master/eclecticiq-extension-example/tests/test_socialurienricher.py)` file provides a boilerplate to build your customized enricher extension test file.

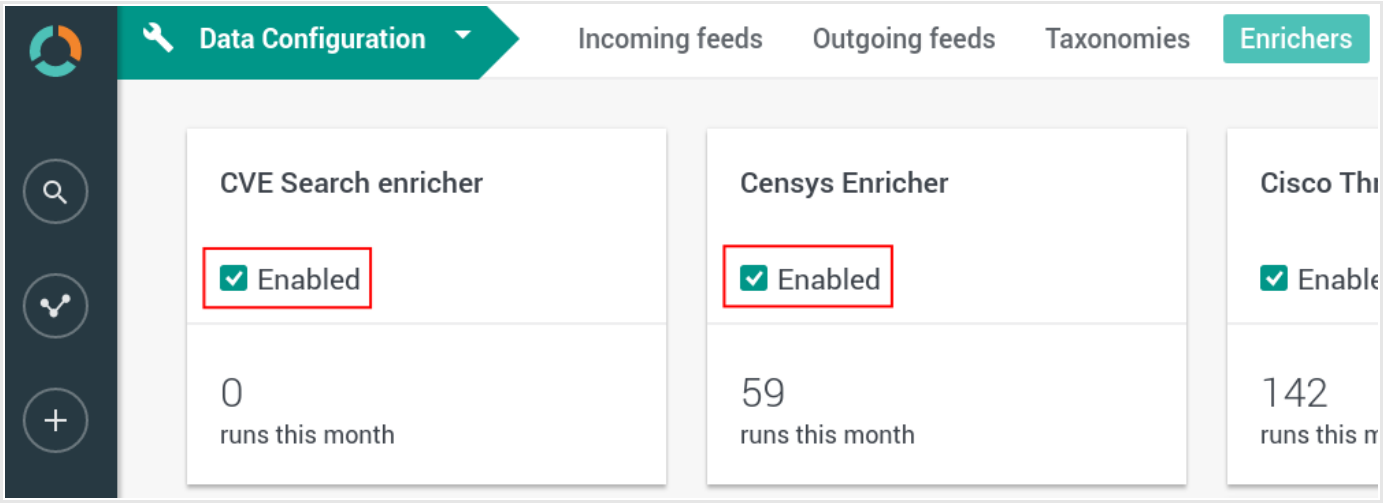
The test file uses **HTTPretty** `(https://httpretty.readthedocs.io/en/latest/index.html)` to mock HTTP responses, and it makes REST API testing easy and transparent.
Check the **HTTPretty GitHub repository** `(https://github.com/gabrielfalcao/httpretty)` for more details and usage examples.

Example:

```
# Import the libraries, modules and classes you need to test your extension
import httpretty

from <path.to.your.custom.enricher.extension> import (<CustomEnricherExtensionName>)

@httpretty.activate
# This example uses dummy names and values.
# Replace them with the appropriate ones for your extension.
# This function mocks a HTTP 200 response.
def test_socialurienricher_found():

    # Mock the API endpoint and any additional URL params
    # Mock the HTTP status code the response should return
    httpretty.register_uri(
        httpretty.HEAD, 'https://twitter.com/jhonny', status=200)

    httpretty.register_uri(
        httpretty.HEAD, 'https://facebook.com/jhonny', status=200)

    # Mock the observable types and values the response should return
    result = enrich_from_social_network(
        extract_type='name',
        extract_value='jhonny',

        # Mock any UI-configurable settings
        check_twitter=True,
        check_facebook=True,
    )

    # Verify that the response returns
    # the expected amount of observables
    # generated from the retrieved data
    assert len(result.extracts) == 2

    assert all(e['kind'] == 'uri' for e in result.extracts)

    values = [e['value'] for e in result.extracts]

    assert 'https://facebook.com/jhonny' in values
    assert 'https://twitter.com/jhonny' in values


@httpretty.activate
# This function mocks a HTTP 404 response.
def test_socialurienricher_not_found():
    httpretty.register_uri(
        httpretty.HEAD, 'https://twitter.com/jhonny', status=404)
    httpretty.register_uri(
        httpretty.HEAD, 'https://facebook.com/jhonny', status=404)

    result = enrich_from_social_network(
        extract_type='name',
        extract_value='jhonny',
        check_twitter=True,
        check_facebook=True,
    )

    assert len(result.extracts) == 0
```

Test the enricher through the platfom UI

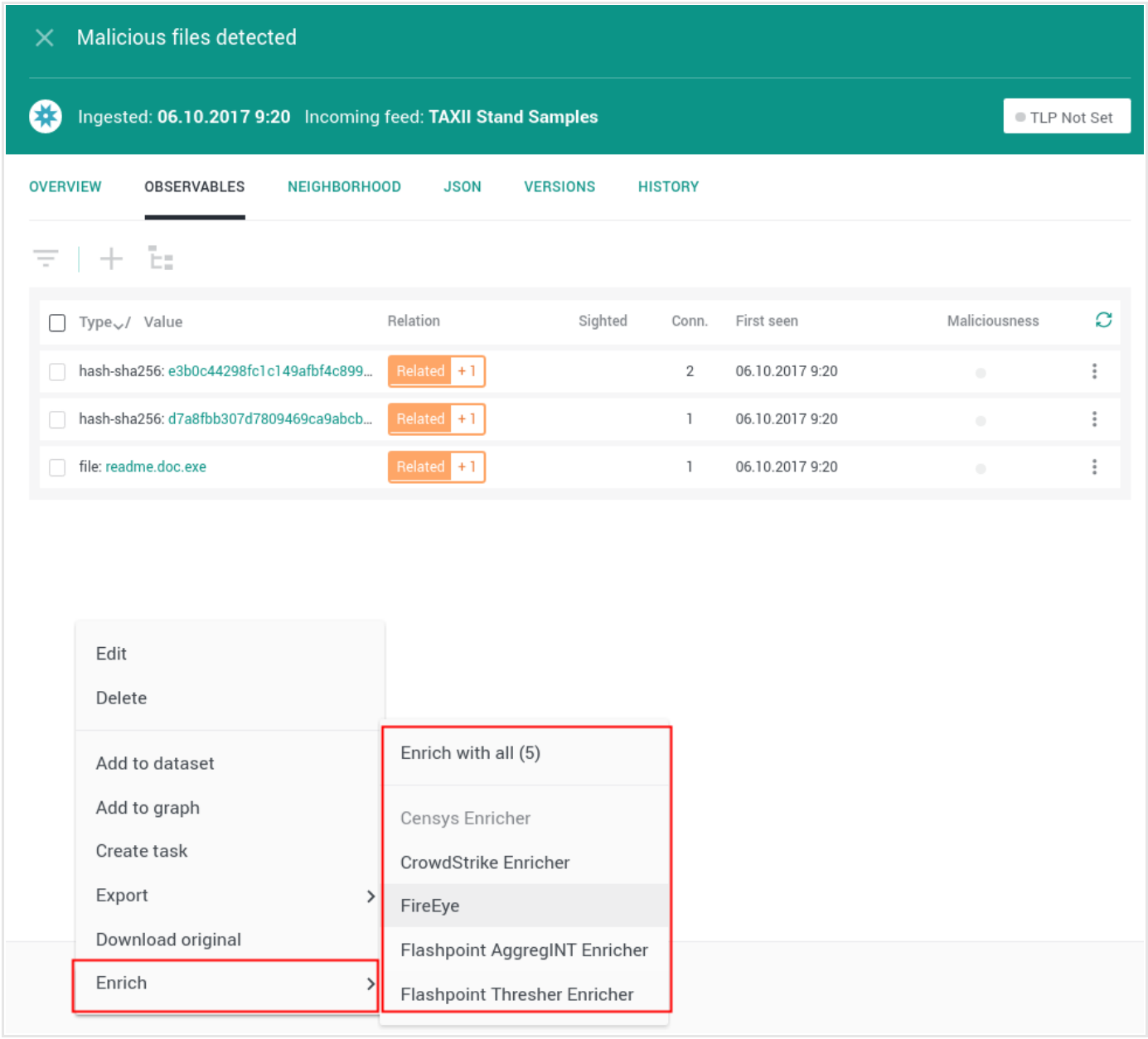
- To check if your enricher extension is available in the platform UI, go to **Data configuration > Enrichers**.
- Your enricher extension should be displayed in the tiled overview, and the corresponding **Enabled** checkbox should be selected to notify that it is enabled.



- To test if your enricher extension works as expected, look for an entity with observables that your enricher extension supports.
- Trigger a manual enrichment:
  - On the entity detail pane, click **Observables**.
  - The **Observables** tab shows an overview of the enrichment observables the entity has been augmented with.
  - To manually enrich the entity observables:
- Click the ↻ refresh icon to trigger a task run that polls all the enrichers configured for the entity.

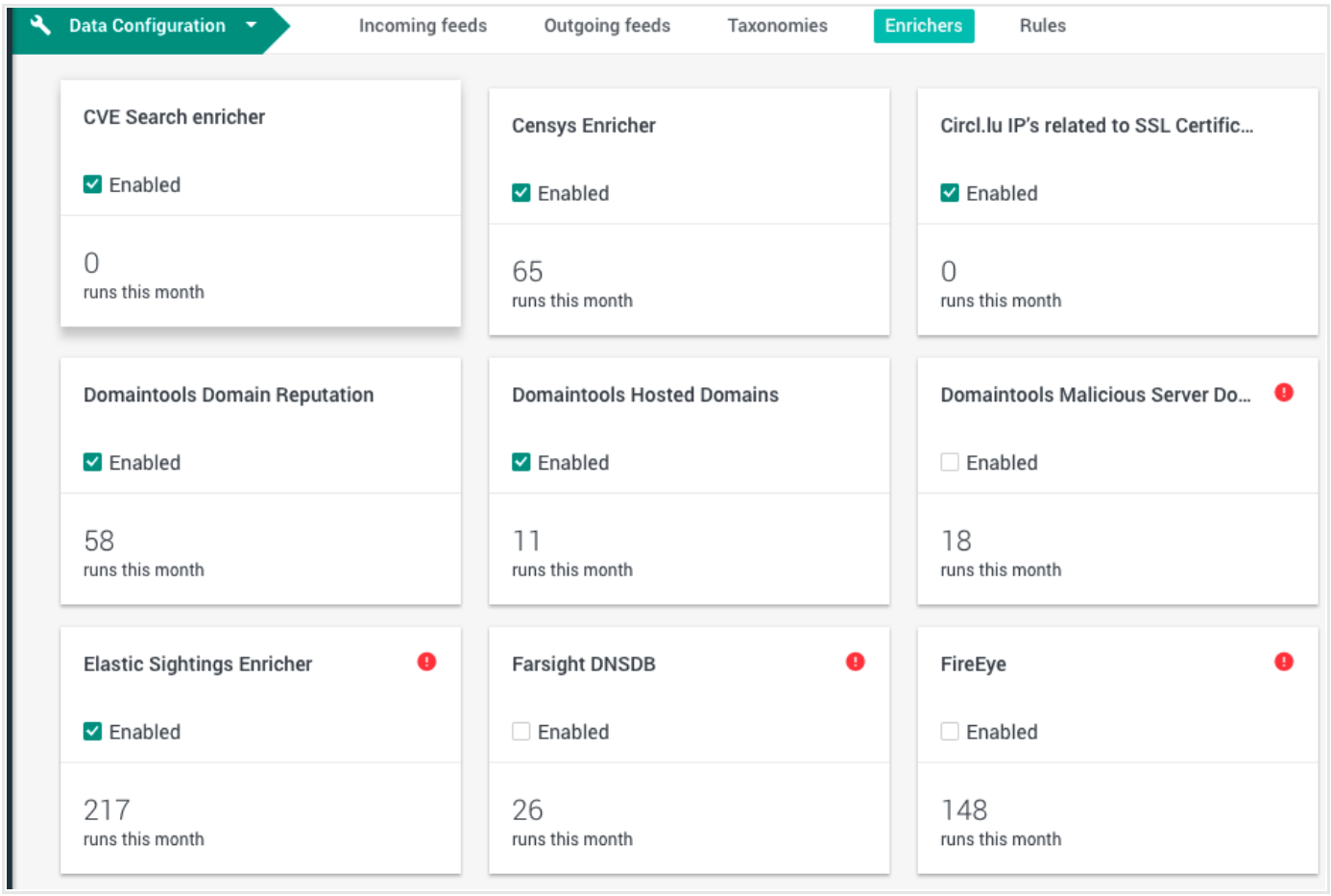
Alternatively:

- From the **Actions** pop-up menu, select **Enrich > Enrich with all**.
- The platform polls all applicable enrichers for the entity, and it enriches all the entity observables with the retrieved data.
  - To poll a specific enricher:
- From the **Actions** pop-up menu, select **Enrich**, and then click the specific enricher whose task run you want to trigger.
- The platform polls the specified enricher for the entity, and it enriches all supported entity observables with the retrieved data.

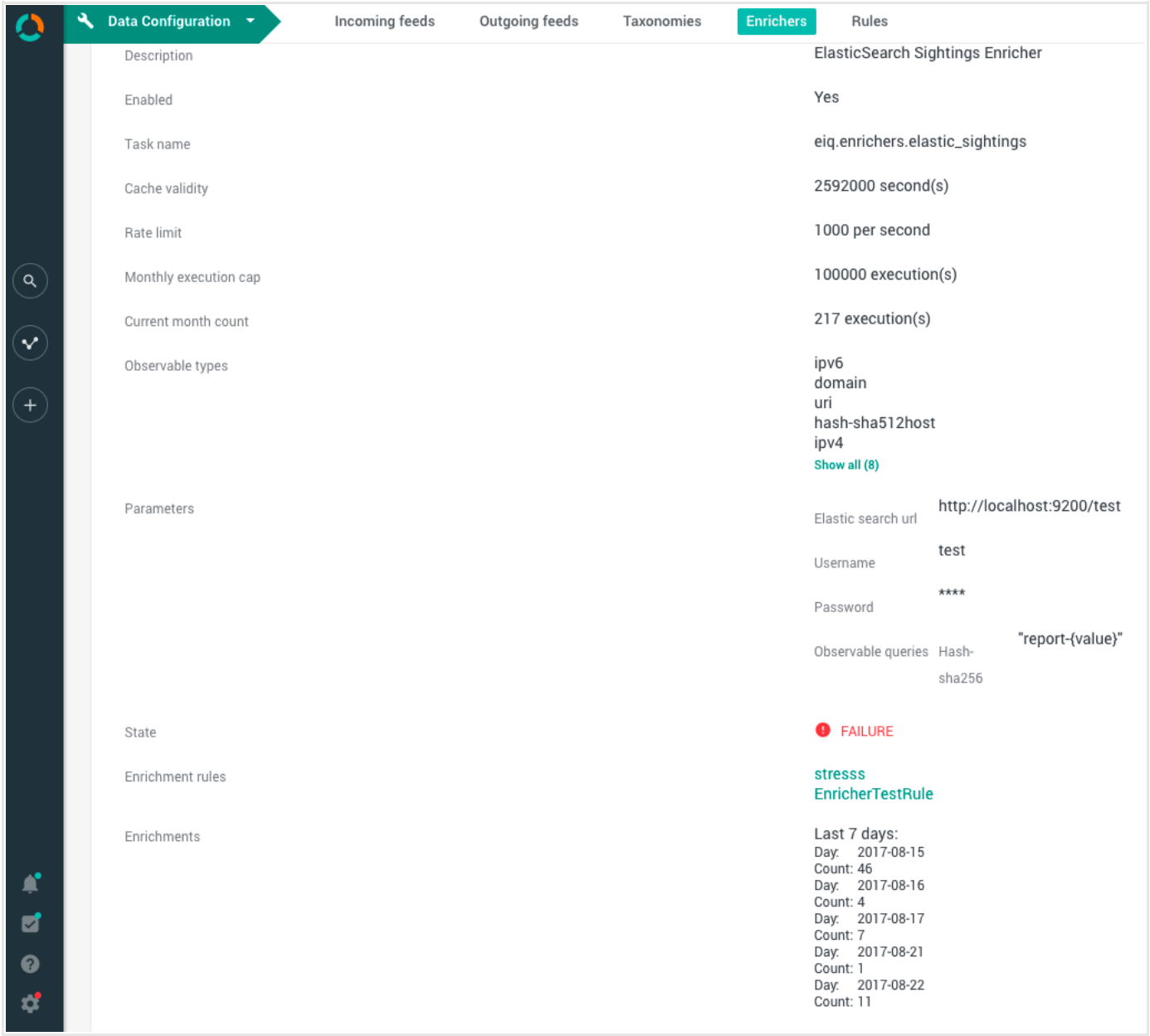


If you do not see any new observables after polling your enricher extension, check if the enricher crashed, and start investigating possible causes for the malfunction.

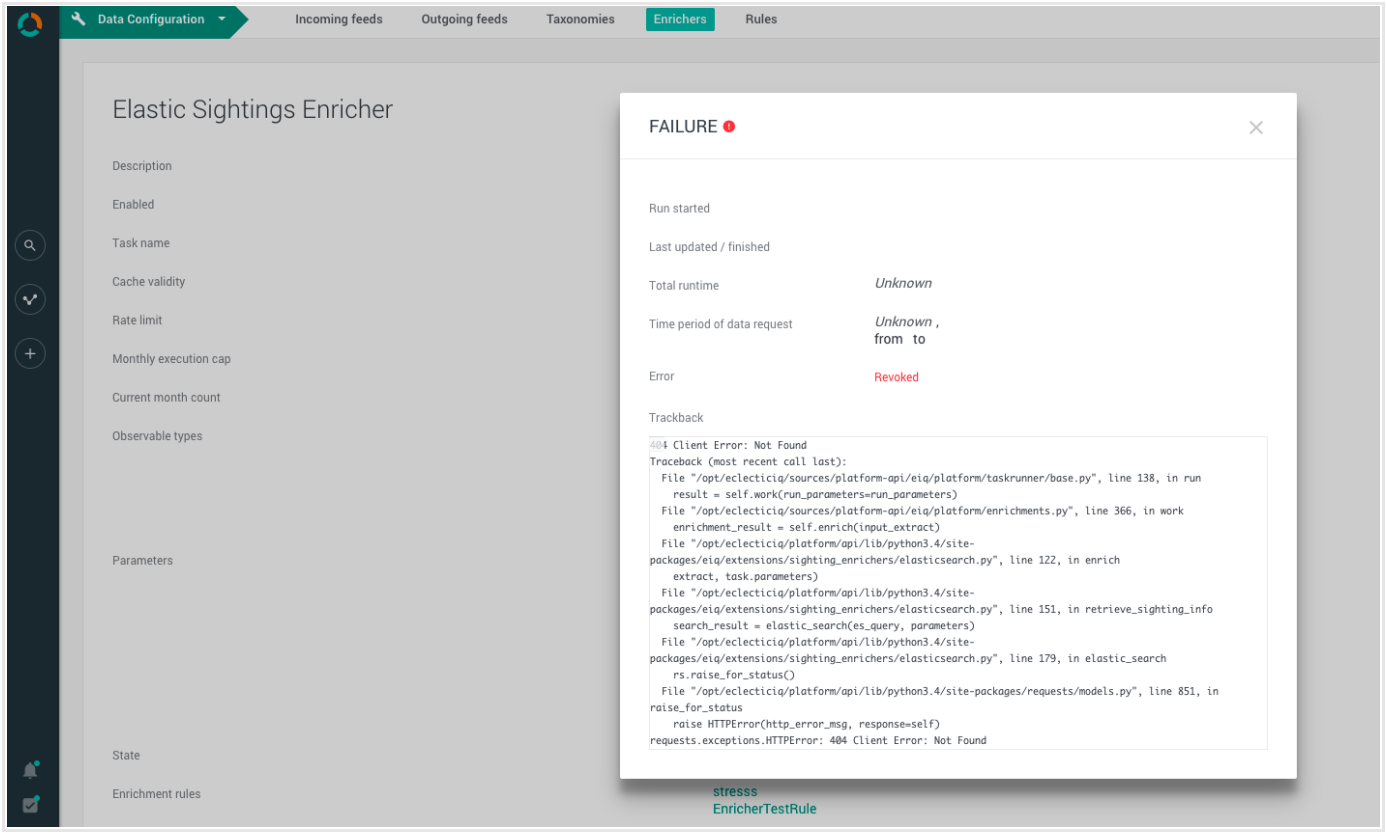
- In the platform UI, go to **Data configuration > Enrichers**.
  On the enricher catalog page you can see a tile overview of the configured enrichers for the platform.
- Look for your enricher extension. If a **(!)** icon is displayed, the enricher task failed to run correctly.



- Click the enricher tile.
- On the enricher detail page, click **(!) Failure**.



- An error dialog is displayed. The dialog title notifies the type of error, whereas the traceback area gives a detailed stack trace in reverse chronological order. The stack trace should give you at least some hints about the possible causes of the failure.



Disable extensions

To disable an enricher extension and prevent the platform from loading it, do the following:

- Open */opt/eclecticiq/etc/eclecticiq/platform_settings.py* in read-write mode.
- Browse to the `DISABLED_EXTENSIONS` parameter.
  If it is not there, add it to the settings file.
  This parameter is a list holding the `name` values of all disabled extensions, as defined in the setup file.
- Add your enricher extension `name` value to the list.
  Example:

```
DISABLED_EXTENSIONS =  [
    'eclecticiq-extension-fraud-ip-observables',
    ...,
]
```

- Save the settings file and exit.