BUILDING INTEGRATIONS FOR MALTEGO

This guide aims to walk you through the process of designing, planning, developing, and launching your own Maltego integration—keeping it as simple as possible to help you get started quickly.

Whether you're an amateur OSINT enthusiast, a professional Maltego user who needs to access internal data and systems within Maltego, or a data provider looking to build a commercial integration for the Transform Hub, this guide is here to help you get started.

Integrations are what make Maltego so useful. Maltego is a highly flexible tool and can be customized to display and interface with almost any API-driven system you can think of. Doing so is often easier than people think. Writing Transforms is quite simple—the difficulties usually lie in understanding how a Maltego integration works and is set up, and then in designing useful and reusable Transforms. This guide aims to provide brief, effective, and easy-to-follow guidance, especially in these areas.

Why a Maltego Integration?

If you're here, chances are you've already decided you want to write an integration for Maltego. Whether that's the case or not, here are some of the main reasons and benefits of Maltego integrations:

- **See and traverse your data.** Maltego can be thought of as a generic frontend: it adds an extra interface for data or tools you already work with and helps you visually explore them.
- **De-silo your data.** A good Maltego integration makes disparate systems seamlessly compatible in one user interface.
- **Streamline investigations.** Jump between data points, spot patterns, and connect the dots in ways that a spreadsheet and text document just don't allow.

It's easy. Don't be intimidated—this guide is lengthy because we spend a fair amount of time talking about processes and best practices. Writing the integration code is quite straightforward.

Cheatsheet

This guide will teach you what you need to know, but if you want a quick reference (and checklist) to refer back to later or throughout the process, **use the cheat sheet here**.

Table of Content

Overview

High-level process

To help you navigate the process at a high level, here's a breakdown of what is involved in creating a typical Maltego integration:

- **Understand the system or API you want to integrate.** This is usually done by sketching out the types of objects the system represents, as well as their relationships.
- **Plan your Maltego Transforms and Entities.** This consists of translating the conceptual sketch from step 1 into a concrete list of Entities to represent data and Transforms to query the data and traverse relationships.
- **Start developing Transforms.** You can start with local Transforms, but if you're planning on distributing your Transforms later, it may be useful to begin with remote (TDS-based) development right away.
- **Deploy your Transforms and test them in the Maltego ecosystem.** Once you start using your Transforms, you may notice some incompatibilities or design issues that need to be addressed before you can launch.
- **Launch your integration.** Once your Transforms are live, there are a number of ways you can distribute them to other users and considerations to keep in mind during that process.

Terminology

Just the basics to start with, we'll explain details as we go along.

Entity: A node on a Maltego graph. Represents an object in the world (an IP address, a domain, a person, a company, etc.). Entities consist of a type, a value, and one or more properties.

Transform: A small script/function that takes one Maltego Entity as input and returns zero or more Entities as outputs.

Running a Transform: a single Entity is sent to the Transform server, and multiple Entities are returned and added to the graph.

TRX Library: Maltego's open-source Python Transform development library, which is what you will be writing your integration in.

Transform Server / TRX Server: A server that can run Transforms (usually written using the TRX library).

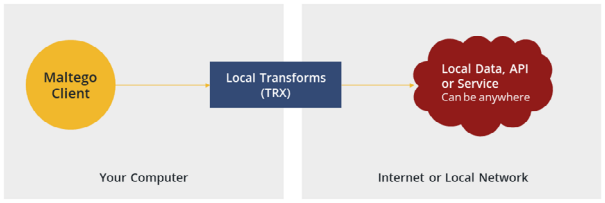
Transform Distribution Server (TDS): A server that connects Maltego Desktop Clients to Transform servers. The Maltego Client connects to the TDS to discover Transforms, and when it runs a Transform, that request is forwarded and executed by the Transform server.

Architecture Basics

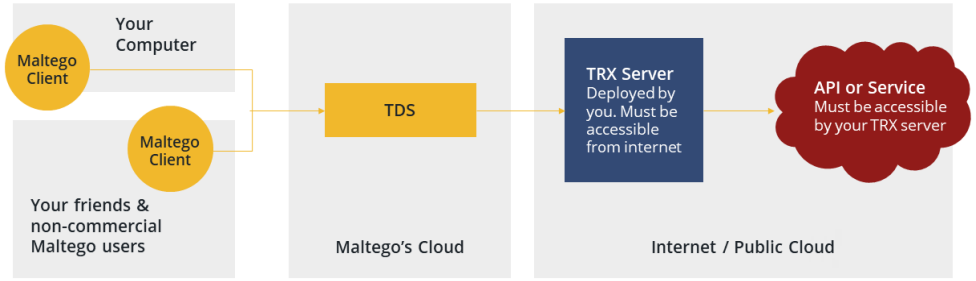
> You can skip this section if you just want to get started, but it will help you orient yourself and understand the full picture.

We'll start with two important pictures that show where your integration will live and how data ends up in a Maltego Client.

There are many ways to deploy and operate a Maltego integration, however, in this guide, we will focus on the two basic types of setup that apply to most projects.



(1) Local Transform Integration



(2) TDS Integration

In the diagrams above, **blue boxes represent your integration, i.e. the Transforms** you will be writing.

Local Transforms

The simplest case is local Transform development. In this setup, your Maltego Client will simply call a local Python script to run your Transform. Local development is quick to get started with, requires no servers, but can be a bit cumbersome to operate, and makes it difficult to share Transforms with others. Luckily, even if you begin by writing local Transforms, you can migrate to a server-based setup quite easily later.

"Public TDS" Integration

In this setup, your Transforms run on a server that you will set up yourself and Transform discovery for Maltego Clients is handled by Paterva's Public Transform Distribution Server (pTDS).

This is a good option if you are aiming for:

- Private projects for you (and your friends)
- Community and free OSINT integrations
- Getting started with development (even if you are a commercial user or data provider - eventually, you may have to switch to an on-premise or commercial integration option, but this is simple)

The role of the TDS is to provide a so-called "Seed URL" that can be plugged into Maltego Clients, which allows the remote calling of your Transforms. The TDS will proxy those requests to your TRX server, which handles the execution of the Transform.

iTDS (On-Premise) Integration

If you are integrating some local database or restricted API, you'll need to make sure that your TRX server is accessible from the internet (or at least that the IP address block of the Public TDS is whitelisted).

The "public" in public TDS **does not** mean your data will be public—it just means that anyone with whom you share the Seed URL will be able to install those Transforms. All traffic is still TLS-encrypted, and your Transforms can be set to require authorization either by adding "Transform Settings" for credentials or setting up OAuth for your Transforms.

If you require fully on-premise data integration, you'll need to deploy your own internal TDS (iTDS), which allows you to set up Transforms from your own team completely on-premise or in whatever private cloud environment you prefer, with no Maltego/Paterva infrastructure in the loop.

This is a good option for:

- Enterprise users with sensitive data and/or users that want to keep all traffic within their own networks

- Enterprise users with data or systems that are purely available on-premise and should not be made available off-premise
- Commercial data providers that prefer to be fully in control of their Transform infrastructure (note that providing Transforms to commercial, non-internal users is not covered under the standard iTDS license. Reach out to the Maltego Transform Hub team for more details). This is a good option for:

In terms of setup, the architecture will look essentially the same as above, except that the TDS is within your own infrastructure and not in Maltego's cloud. This only complicates setup in the sense that your organization will also have to deploy the iTDS; from a development perspective, there is essentially no change.

Let's get started!

This guide will follow the process outlined in the overview. Each section will start by explaining the main steps covered and then expand the details through the provision of concrete examples.

Specifically, we'll use the following example

Example Scenario: OpenStreetMaps Nominatim API
We want to make use of Maltego for analyzing some of the location-related aspects of the data in our investigations. That means searching for places by name, converting addresses to coordinates and vice versa, and doing other simple operations on location-related data. After doing some research, we settle on OpenStreetMap's Nominatim API as a solid, free, and easy-to-use data source, also because some of our team members have used it in the past and said it's been a useful tool.

We want our integration to be useful in everyday investigations to allow us to quickly pull location-related pivots from a range of data we have in Maltego:

this may include IP geolocations, addresses listed for companies or individuals, or simply GPS coordinates.

Luckily for us, the tool also comes with a simple and well-documented API: **https://nominatim.org/release-docs/develop/api/Overview/**

How to Read This Guide
You can follow the guide itself to see the steps you should take to design and build your integration. In each section, we will show examples of executing these steps for our example integration.

Integration Design

This section describes best practices for steps 1 and 2 in the high-level overview outlines above.

In a nutshell, designing an integration means to:

- Model the domain of the service you want to integrate
- Describe the relationships of the different objects in that domain
- Translate these into specific Entities and Transforms
- Account for "real-world" caveats that result from the structure of your API, rate-limits, query performance, etc.

(The last step often happens later and iteratively, while you're already developing Transforms.)

Cheatsheet

Step 1: Understand and Model the Domain

Write Down Nouns

In our experience, the best way to start is by writing down a list of nouns: What are all the types of things that are represented in the service you want to integrate with? You don't have to think on the level of Maltego Entities for this step yet, but if you're already familiar with them, it helps to keep them in mind.

You can already note down typical properties of these types of objects if you want, but you can also just think about those later on.

Side note: If you're used to object-oriented programming, this exercise is very similar to specifying your schema of classes or similar to designing a database schema.

Level of abstraction. Avoid thinking of the system itself in this step; instead think of the data it represents. If say you're integrating data from a local MySQL database (e.g. for performing fraud-checks), then "Database" and "tables" are not great objects to represent in Maltego, however "Customer", "Order", "Email Address", "IP address" are great examples. "Order from January" or "Active User", on the other hand, are probably a bit too specific, even if your underlying data might be modelled in such a way for whatever reason.

You don't have to create a structured table here, but sketching this out on a piece of paper or writing it down is usually a good idea.

It's also a good idea to begin thinking about inheritance at this stage: is one of your types actually a subtype of another type you listed?

This step is worth spending some time on, even if it seems trivial. The clearer you are on the object types in your domain, the better your integrations are going to be (and you'll save lots of time during the implementation phase).

Example - Modeling Domain Objects

For our example integration, the following list of nouns seems like a reasonable fit:

Types for OpenStreetMaps Nominatim API

Names	Properties	Notes
Address	Street, Country, ZIP Code	Just a plain old address
Named Location / Place	Name, Country, ZIP Code, Wikipedia URL	Similar to address, but describes a place with a name, or even a large place like an Airport or a Shopping Mall that may not just have one address
Coordinates	Latitude, Longitude	
Phrase	Text	We'll need this as a search input
Company	Lots of properties...	We don't care about the properties, but it could be useful to search places by company name
URL	URL	Only needed because we listed "Wikipedia URL" above, so we should make note of it

We'll leave it at that - there are certainly more things we could try to model (coordinate polygons, buildings, etc.), but the above seems to cover the main object types we'll be interested in.

As for inheritance: it seems like "Place" might actually be a subtype of "Address" - it might be up to interpretation in some cases, but looking at the way Nominatim modeled their API, it might be a reasonable way to go. (Spoiler alert: in Maltego, we'll actually model this using a maltego.Location Entity, so the ambiguity resolves itself somewhat, and the inheritance makes a lot of sense).

Sketch the Relationships

Next, it's helpful to complete the picture of the domain by sketching out how the different types are connected. The relationships we identify in this step will be turned into Transforms later on.

Avoiding spaghetti models. While in principle, the fact that all your data is in one system means it is "connected", try to limit yourself to direct and meaningful "one-hop" connections that are easy to traverse in practice. For example, if you were modelling user activity logs with a corresponding user database, you might be tempted to connect the server type to the user type directly and label the relationship something like "was accessed by". However, it may be better to explicitly model the connecting event (e.g. "Login Event")

as well, since it also carries important metadata that will be useful for link analysis, and it lets you design more fine-grained Transforms so that your Maltego graphs don't become crowded too quickly.

As with Entities, modelling relationships effectively comes down to choosing a level of abstraction and can also depend on personal taste. Think about how the data is connected "step by step" and avoid being too abstract or too specific in your choice of relationships. In some (rare) cases, you may also decide that a "relationship" should actually be modelled as a full Entity (for example, when it carries important metadata which is useful in itself for link analysis). Examples of this in Maltego can be found in integrations like OCCRP Aleph ("Directorship", "Membership", etc., are all proper Entities). In general, though, it's best to keep things simple and just model any relationships in the data using Transforms.

Back-and-forth. One of the most important things to look out for when modelling relationships is to cover both possible "directions" of relationships wherever possible. If a user can pivot from a Domain to the associated IP address, that is great, but if they can also pivot from IP address to associated Domains, then suddenly they have a powerful way of mapping related infrastructure. The same applies in almost every domain in one form or another. Here are a few examples:

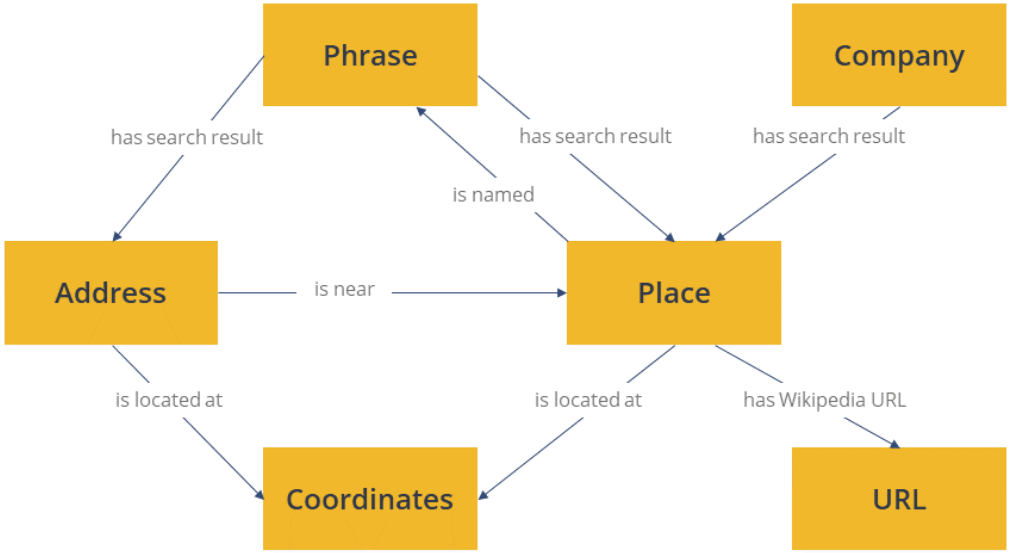
- Company ⟶ Director ⟶ other Companies, ...
- IP ⟶ Vulnerabilities ⟶ other Vulnerable IPs, ...
- Symptom ⟶ Possible Illness ⟶ Other Symptoms, ...

This pattern should almost certainly be found somewhere in your integration.

Leaf nodes. On a related note, you should also avoid leaf-node objects, i.e. there should be very few (if any) relationships that result in the kind of object which does not have any outgoing relationships. An important takeaway from this is that at least your custom, new objects that are not present elsewhere in Maltego should have outgoing relationships (otherwise, consider just leaving them as properties or other visual effects on another Entity type).

Example - Modeling Domain Relationships

For our sample integration, here are the relationships we may come up with:



You can also sketch these in the form of a table or matrix `(read {left header} - {relationship} -> {top header}`, e.g. `Address - is located at -> Coordinates)`:

Relationships for OpenStreetMaps Nominatim API

From → To	Address	Place	Coordinates	Phrase	Company	URL
Address		Resolves to / is near	is located at			
Place	has / includes		is located at	is named		has Wikipedia URL
Coordinates	Resolves to / is near	Resolves to / is near				
Phrase	has search result	has search result				
Company	has search result	has search result				
URL						

We could probably make some other pivots work as well, but this seems like a good start. We can move from addresses and places to coordinates, as well as back, and we can perform searches based on a few inputs. We can also connect a Place to a URL that will tell us more about the place.

We also see that none of our relationships results in a company—this may be fine for now, but on closer inspection, it turns out that some "Places" in Nominatim are actually business locations! We could correct this oversight, but for now, we'll leave it be as it was not originally in the scope of our requirements for this integration. One easy way to add this functionality later would be to add a special kind of "Place" Entity in Maltego that gets returned for "Place" results that were tagged as offices or other kinds of companies by Nominatim for each of the searches.

Back-and-forth: In our example, Addresses, Places and Coordinates follow this pattern, as well as Phrases (albeit in a less intentional way).

Leaf nodes: We can identify these by checking for empty rows. URL is a leaf node, however, the effect is not that bad since there are dozens of other Maltego integrations that allow us to do interesting things using a URL.

Step 1: Plan your Maltego Entities and Transforms

> **"Entity type" and "Entity" are sometimes used interchangeably in the following sections.** A good rule of thumb to disambiguate is this: when we're talking generally about kinds of things or types, "Entity" is usually short for "Entity type". When talking about specific objects, such as returning an Entity from a Transform, that means "Entity" in the conventional sense of "a single specific thing".

Now that you have a model of the system you're integrating, it's time to start planning the details of the Maltego integration.

In short, you'll translate the **nouns** and **relationships** into Entity types and Transforms, respectively. This is a relatively simple process, but some care has to be taken to avoid common design pitfalls.

Choosing Maltego Entity Types

The main thing to optimize for in this step is a good tradeoff of re-used/reusable Entity types and specificity of Entities to your domain.

If you're not already familiar, start by taking some time to **browse through the Entity types already present in Maltego**. This is a good idea even if you have used Maltego before; even our own developers often discover useful Entity types that they weren't aware of. Alternatively, you can open up Maltego, install the Standard Transforms and CaseFile Entities, and browse them in the Entity Manager.

When you find an Entity type that closely corresponds to one of the domain objects you identified, write down its ID (usually "<Org>.<TypeName>", but make sure to check). The output of this step should be a list of more or less the same size as your domain objects from before, but you may decide to fold together or further separate some of your types on the Maltego level.

> **Subtypes and Entity Merging.** When an Entity returned from a Transform is already on the graph, then we do not want to add a duplicate of it, so the results should merge instead. This happens automatically, but only under certain conditions:
>
> - **The two Entities must have the same type**
> - The two Entities must have the same main value, also called "edit value" (more on that later)
> - All "strict" properties must be equal (more on that later)
>
> We highlight the first point here for an important reason: if you were to subclass e.g. maltego.IPv4Address, even when the value of one of your returned IP addresses is equal to an IP already present on the graph, these Entities **would not merge**.

This is currently a limitation in Maltego and an important consideration to make before creating new types.

Our recommendation: only create subclasses where absolutely necessary and avoid subclassing any "cyber-domain" Entities like IP addresses, domains, URLs, etc. further, unless you are sure that they do not need to be merged with standard Entities.

"Real-world" Entities like companies and persons are always hard to merge anyway, so subclassing them is less risky, it also tends to be a more frequent requirement.

Note for Cyber Threat Intelligence: Maltego is in the process of officially adopting **STIX 2.1** Entity types. If you can't wait for us to release and announce these Entity types, you can preview them here: https://github.com/amr-cossi/maltego-stix2

Once Maltego officially adopts these Entities, you may have to delete and reinstall them from your Maltego Client and possibly make minor adjustments to your Transforms since any further changes to the Entities from our side would not automatically propagate to your Maltego environment.

Example - Choosing Entity Types

Below is the set of Maltego Entity types that we'll use for the Nominatim API.

Planned Transforms for OSM Nominatim API

Relationships	Transforms	Notes
Address → is located at →Coordinates	To Coordinates [Nominatim]	Perform Geocoding.
Address → resolves to / is near → Place	Find nearby Places [Nominatim] Normalize to Place [Nominatim]	Search for Places by address. The "Normalize to Place" Transform tries to return a single best-guess Place that corresponds to the input location, rather than return more search results.
Place → has / includes → Address		Not needed, since a Place *is* now an address
Place → is located at → Coordinates		Not needed, can reuse the maltego.Location → "To Coordinates" Transform because of inheritance
Place → is named → Phrase	To Name [Nominatim]	Just returns the name property as a phrase. Seems relatively uninteresting, maybe we'll skip implementing this one until we need it. A better version would resolve it to a Company Entity, if the place represents e.g. an office, and other types accordingly.
Place → has Wikipedia URL → URL	To Wikipedia URL [Nominatim]	Just resolve the wiki URL property and add to graph
Coordinates → resolves to → Place	Find nearby Places [Nominatim]	Search by coordinates (reverse geocoding)
Coordinates → resolves to → Address		Not needed because of inheritance.
Phrase → has search result → Place	Find Places by name [Nominatim]	Could consider more variants for this, or different filter Transform Settings
Phrase → has search result → Address		Not needed because of inheritance.
Company → has search result → Place	Find Places by name [Nominatim]	Search Places by company name. We could try and add some filters here ("find McDonald's restaurants in New York"). Logic-wise, basically almost the same as a phrase search.
Company → has search result → Address		Not needed because of inheritance.

"Address" turns out to be best represented using `maltego.Location`. `nominatim.Place` is a new type that we will create, and we will inherit maltego. Location in this type. Everything else is unsurprising, and we made sure to reuse standard Maltego Entities to maximize compatibility with other integrations.

Planning your Transforms Based On the Relationships

In this step, you'll translate the set of relationships into a set of Transforms. Note that this correspondence is not necessarily 1-to-1: at this step, you'll want to take into account practical concerns, such as:

- **Will users frequently want to filter the results of a Transform in a certain way?** If there's a particular kind of filter that is tedious to set but frequently useful, it may make sense to add a dedicated Transform for it.
  - For example, take Shodan: in Shodan, you can search for subdomains of a domain, and you can optionally include historical subdomains in that search. That could be modelled as one Transform ("To Subdomains", DNS Name □ DNS Name) with a boolean setting, but because the use-cases for the latter option are different from the first, we decided to explicitly model it as two Transforms ("To Subdomains" and "To Subdomains (+historical)").
- **Does the API let me traverse the relationship in a straightforward way?** In Maltego's Pipl integration, we decided to separately model a "Pipl Person" and a "Pipl Possible Person" because of the way Pipl internally represents and serves data. A direct connection like (maltego.Person ⟶ pipl.Person ⟶ Details) would have been nice, but in practice it didn't work out well to write the Transforms this way.

Transform Names and Descriptions

Transforms have an internal name and a display name. The internal name is usually not important to the user, but you should take care to create useful and precise display names. Don't cryptically name your Transform like the functions in your code, and don't be overly generic either.

Our own best practices are as follows:

- Transforms that search for things are roughly named according to this pattern:
  - `Search for {output type name}s [{data source}]`
  - or, if you need to differentiate variants:
    - Search for `{output type name}s ({variant description}) [{data source}]`
    - (or something like `Find {output type name}s by {input type or criterion} [{data source}]`)
- For example:
  - `Search Companies [OpenCorporates]` and `Search Companies at this Address [OpenCorporates]`
  - See more examples for Nominatim in the table below.
  - Also, check out naming patterns used by official integrations by Maltego Technologies integrations in our docs for more examples.
- If your Transform won't trigger a search but follow a link on the object or make a direct API lookup of a connected object:
  - `To {connected entity} [{data source}]` (again, you can add more text to clarify variants, but keep the name short)
  - Example: `To Wikipedia URL [Nominatim]` (see example later in this section)
  - Example: `To DNS Names [Shodan]`
- Other names are also fine; Transforms don't need to start with "Search" or "To", but make the name brief, descriptive and easy to grasp even for users that are not intimately familiar with the service you're integrating. Usually, a Transforms name should at least include a verb since Transforms generally trigger or perform some sort of action.
- **We strongly encourage you to write good Transform descriptions.** Transform names are very important, but descriptions can help you make your integration almost fully self-documenting.
  - The same goes for Entity types: if you add custom Entity types, take the extra 20 seconds to add a good Entity description, as well as a description on the different properties.

Using Maltego to design for Maltego. We actually find it useful to just use Maltego to create a schematic graph of Entity types and the Transforms that will be written between them—you can just drag and drop Phrase Entities to the graph, manually set their value to the type name they could represent, and manually draw links between the types, adding link labels to represent Transform names.An example of this is given below—don't be confused by this, we're not using Maltego to run any Transforms yet, we are just using its mindmapping-like capabilities to manually sketch out a graph that visualizes our planned Transform structure.

Example - Planning Transforms

Going off of the relationships identified above, here's an initial list of Transforms we might come up with (we can always add more later!):

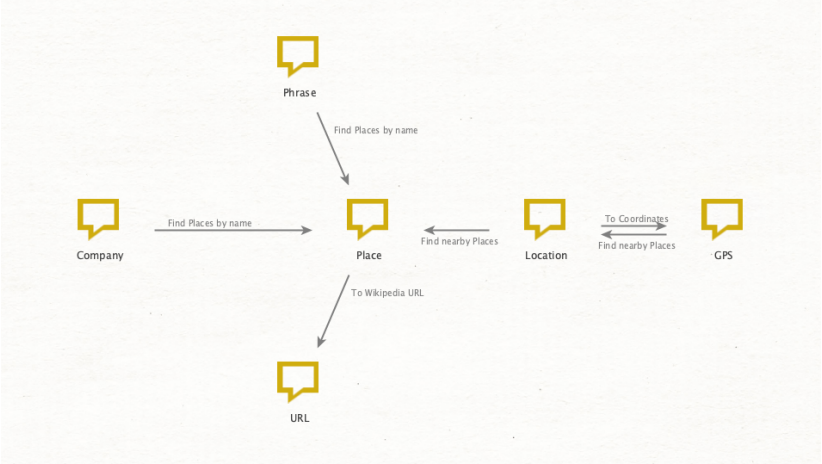
Planned Transforms for OSM Nominatim API

Relationships	Transforms	Notes
Address → is located at →Coordinates	To Coordinates [Nominatim]	Perform Geocoding.
Address → resolves to / is near → Place	Find nearby Places [Nominatim] Normalize to Place [Nominatim]	Search for Places by address. The "Normalize to Place" Transform tries to return a single best-guess Place that corresponds to the input location, rather than return more search results.
Place → has / includes → Address		Not needed, since a Place *is* now an address
Place → is located at → Coordinates		Not needed, can reuse the maltego.Location → "To Coordinates" Transform because of inheritance
Place → is named → Phrase	To Name [Nominatim]	Just returns the name property as a phrase. Seems relatively uninteresting, maybe we'll skip implementing this one until we need it. A better version would resolve it to a Company Entity, if the place represents e.g. an office, and other types accordingly.
Place → has Wikipedia URL → URL	To Wikipedia URL [Nominatim]	Just resolve the wiki URL property and add to graph
Coordinates → resolves to → Place	Find nearby Places [Nominatim]	Search by coordinates (reverse geocoding)
Coordinates → resolves to → Address		Not needed because of inheritance.
Phrase → has search result → Place	Find Places by name [Nominatim]	Could consider more variants for this, or different filter Transform Settings
Phrase → has search result → Address		Not needed because of inheritance.
Company → has search result → Place	Find Places by name [Nominatim]	Search Places by company name. We could try and add some filters here ("find McDonald's restaurants in New York"). Logic-wise, basically almost the same as a phrase search.
Company → has search result → Address		Not needed because of inheritance.

We can see that from 13 relationships, we ended up with only 8 Transforms to fulfil all of our main use-cases (and then some). Because of the inheritance between `maltego.Location` and `nominatim.Place`, we will be able to skip implementing a fair number of redundant Transforms.

One concern about this plan is that we subclassed the `maltego.Location` Entity, and our new `nominatim.Place`Entity will not automatically merge with existing Entities. This is not ideal, but in this case, still a good way to go since we are also adding a sort of "converter" Transform `Normalize to Place [Nominatim]` that allows users to turn any `maltego.Location` Entity into a `nominatim.Place` Entity (which have the added benefit of being easier to merge since they are normalized by the API).

Optionally, we can also represent our planned "schema" of Transforms as a manually sketched-out Maltego graph:



(If you're just skimming this: note that this is not a graph from our integration, it's just a schema of the Entity types and Transforms we drew manually to plan our integration)

Integration Development

You've modelled the domain; you have a plan for which Entity types you'll use (or create), and even for the Transforms you want to write. Let's get to work and start writing and running our first Transforms!

We'll be using **Maltego TRX** to write our Transforms. Maltego TRX is an open source (MIT licensed) microframework for Python that lets your write both local and hosted Transforms in one codebase.

Step 3 – Implement Your Integration

First Things First - Setup

You'll have to start by doing a little bit of setup. If you're new, please follow the guide linked below closely to get set up and run your first custom Transform. If you've developed Transforms in the past, we still recommend at least reading this guide and making sure you've understood it (and any potential differences to how you're used to developing for Maltego).

- **Learn more in the Maltego technical documentation**

Optional next step (you'll have to do this later anyway if you plan on using a TDS to distribute your Transforms):

- Learn more about setting up remote Transforms on a TDS in the **Maltego technical documentation**

After you're set up with Python 3 and have installed the Maltego TRX library, go ahead and start a new project with the name of your integration.

Transform Implementation - Advice on Technical Setup

- **Code structure.** It's a good idea to separate your Maltego-specific Transform code and your API-specific code. Typically, your Transforms should only be a few lines of code that read in the Entity value and/or and Entity properties and Transform settings you need, and a single call to your own API-specific "library" to actually execute the logic of your Transform.
  - A good structure is to have only Transforms in the /transforms module of your project and to create another package /api (or a module) next to it for your library code (you can of course choose a different name, for instance, if you're not integrating an API but a local database or something else).
  - If you find yourself making API requests in the body of a Transform function, that's usually a sign that you should consider refactoring your code.
- **Create Entities in one place.** A common source of problems is when different functions or Transforms in your integration each create their output Entities in different ways. A good way to avoid this is to create a module where you create each type of Entity using only one (or if needed more) function respectively (i.e. one function per entity type), and to use those Entity constructors from the rest of your codebase whenever you need to create a Maltego Entity.
- **Avoid calling local executables unless you have to.** If you write a Transform to open some file on your machine, or e.g. a browser window, that Transform cannot be deployed to a server and run remotely since it has local dependencies. Sometimes this is intended, but in most cases, it can be avoided and makes for more flexible Transforms. If you want to add convenient helpers to open e.g. a web link or make an image visible, have your Transform add Display Information instead (see below).
- **Be aware of the limits of local Transforms.**
  - When developing local Transforms, take note of the CLI/terminal character limit. Local Transforms will fail to run if the TransformMessage received from the client exceeds the CLI character limit.
  - Local Transforms will also set a default value of 100 on the slider instead of reflecting the true slider value of the Maltego client.
  - Printing any non-XML formatted information to stdout (e.g. debug logging or print statements) will also typically break your Transforms locally.

Example - Setting up and Writing Transforms for OSM Nominatim

After setting up a project using maltego-trx and adding our first bits of implementation, we have the following code structure:

```
/osm_nominatim
    /api
        /__init__.py
        /nominatim.py  # our Nominatim API wrapper code
        /util.py       # Entity constructors are here
    /transforms        # the actual Transforms implementation
        /__init__.py
        /LocationToCoordinates.py
        /LocationToNearbyPlaces.py
        /NormalizeLocationToPlace.py
        /PlaceToPhrase.py
        /PlaceToWikiUrl.py
        /CoordinatesToPlaces.py
        /FindPlaceByName.py
        /CompanyToPlace.py
    /project.py        # automatically generated project.py
```

Every Transform imports the relevant API helper functions and Entity constructor functions, and therefore fairly little logic is needed in the Transforms themselves.

For example here's our maltego.Location ⟶ **`Find nearby Places [Nominatim]`** Transform (**`transforms/LocationToNearbyPlaces.py`**):

```
from maltego_trx.maltego import MaltegoMsg, MaltegoTransform
from maltego_trx.transform import DiscoverableTransform
from api.nominatim import find_places_using_location_details
from api.util import create_place_from_nominatim_search_json


class LocationToNearbyPlaces(DiscoverableTransform):
    """
    Query Nominatim API for nearby places from a maltego.Location Entity.
    """

    @classmethod
    def create_entities(cls, request: MaltegoMsg, response:
MaltegoTransform):
        name_of_location = request.Value  # not needed here, but this is
how you access the main Value
        location_details = request.Properties
        results = find_places_using_location_details(location_details)
        for res_json in results:
            entity = create_place_from_nominatim_search_json(res_json)
            response.entities.append(entity)
```

We also have the following **`api/nominatim.py`** module, which mostly consists of logic for constructing an OpenStreetMaps query from Maltego input properties of a Location Entity, and simple logic for making the API calls:

```
import requests

API_BASE = "https://nominatim.openstreetmap.org"


def find_places_using_location_details(location_properties):
    name = location_properties["location.name"]
    country = location_properties["country"]
    city = location_properties["city"]
    street_address = location_properties["streetaddress"]
    area_or_state = location_properties["location.area"]
    zip_code = location_properties["location.areacode"]
    country_code = location_properties["countrycode"]
    address_parts = [
        street_address or name,  # prefer street_address over name
        city, zip_code, area_or_state,
        country or country_code  # prefer country over country code
    ]
    address_combined = ", ".join([part.strip() for part in address_parts
if part.strip()])
    if not address_combined:  # not enough info to search
        return []

    return find_places_by_free_form_query(address_combined)  # reuse our
standard search


def find_places_by_free_form_query(name):
    params = {"q": name, "addressdetails": 1, "extratags": 1}
    results_json = make_api_call("/search", params)
    return results_json


def make_api_call(route, params):
    params = params or {}
    params["format"] = "json"  # we always want JSON returned
    res = requests.get(f"{API_BASE}{route}", params=params)
    if res.status_code != 200:
        return None
    return res.json()
```

The last missing piece is some logic for turning the returned JSON objects into instances of the new nominatim.Location Entity (more on how we created this Entity in a later section). Here is our Entity creation utility module **`api/util.py`**:

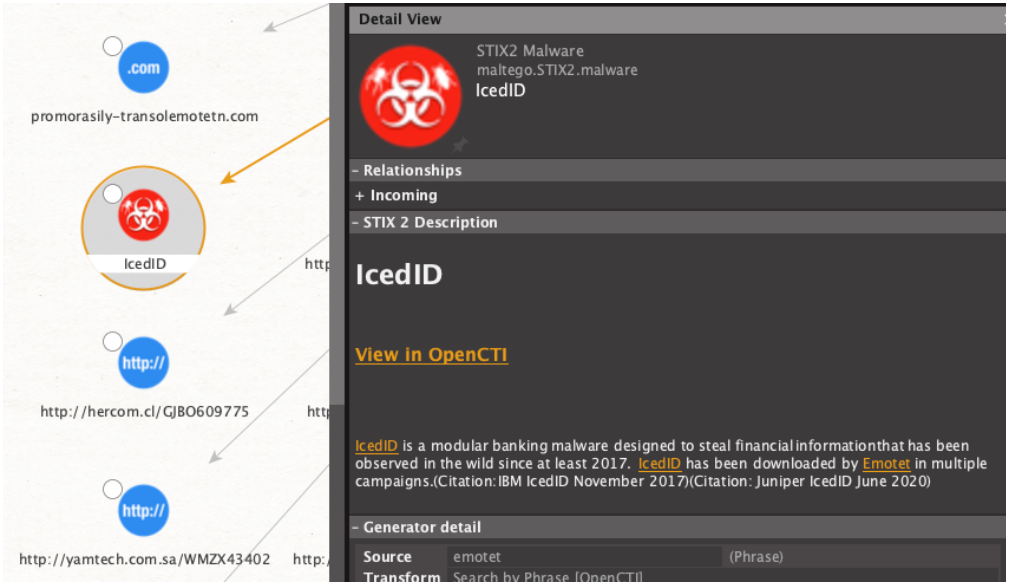
```
from maltego_trx.maltego import MaltegoEntity


def create_place_from_nominatim_search_json(json_result):
    place_id = json_result["place_id"]
    entity = MaltegoEntity("nominatim.Place", place_id)  # value is the
ID to facilitate precise merging
    display_name = json_result["display_name"]
    address_data = json_result["address"]

    municipality = address_data.get("municipality")
    city = address_data.get("city")
    town = address_data.get("town")
    village = address_data.get("village")


    region = address_data.get("region")
    state = address_data.get("state")
    state_district = address_data.get("state_district")
                state_and_district = f", ".join(filter(None, [state_
district, state]))

    county = address_data.get("county")
    county_code = address_data.get("county_code")

                street = address_data.get("road")
                house_number = address_data.get("house_number")
                street_address = f" ".join(filter(None, [house_number,
street]))

                nominatim_class = address_data.get("class")

                postcode = address_data.get("postcode")

    entity.addProperty("location.name", display_name)
    entity.addProperty("country", value=country)
    entity.addProperty("city", value=city)
    entity.addProperty("location.area", value=state_and_district)
    entity.addProperty("streetaddress", value=street_address)
    entity.addProperty("location.areacode", value=postcode)
    entity.addProperty("countrycode", value=county_code)
    entity.addProperty("nominatim_class", value=nominatim_class)

    return entity
```

We won't go through all the code here, but the rest is actually quite simple: all the Transform files are already listed in our project structure above, you'll basically just have to fill in the rest of these in a very similar fashion to the first code sample above, and add more helpers for querying the API and creating response Entities according to the other two examples above.

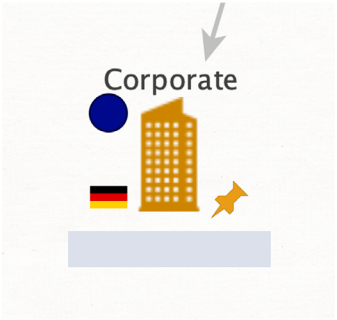
Once we're ready, we can this code either as a local Transform or as a deployable Flask Transform server (using `python3 project.py runserver`). If we deploy it to an internet-accessible machine, or at least one that whitelists the TDS (either the Paterva pTDS or your own iTDS), we can configure these Transforms to be accessible from any Maltego Client that is able to talk to the TDS. For details on this, check out our docs: **Setting up Transforms on an iTDS**.

Transform Implementation - Making Use of Maltego Features

- **Use the Slider value.** The incoming Transform request has a `maltego_msg.Slider` value that you should use to both limit the amount of data you fetch from an API as well as the amount of data you send back to the client.
  - You can also use it for pagination: add a Transform setting for the page number and assume the slider value is the page size.
- **Display Information.** If your Entities have long-form text, images, useful outgoing web links or even basic tabular content you would like to make visible to the user, use Display Information. In TRX, this can be added using the `entity.addDisplayInformation(...)` method. The display information can include basic HTML formatting to provide rich content (images, tables, links, text formatting, etc.), but no CSS or Javascript.

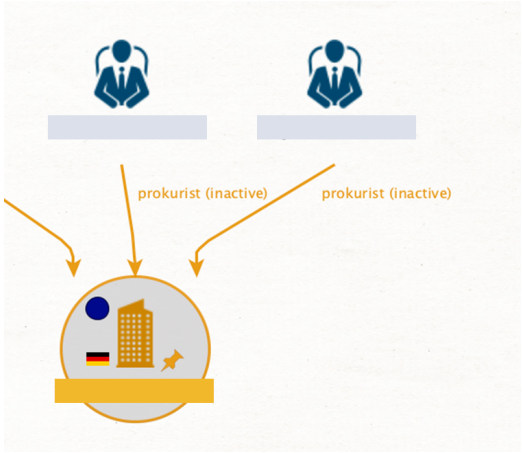


- **Overlays.** You can use overlays to provide simple visual indicators via `entity.addOverlay(...)`. In the OpenCTI Screenshot above, the white dot on the Entities is used to indicate that they are TLP-white and can be publicly disclosed. The screenshot of Maltego's Orbis integration below shows three overlays: the "Corporate" text on top of the Entity indicates additional information on what type of company or organization the user is seeing, the blue dot indicates that the data is fully enriched (partially complete Entities have no dot or a different color dot, this pattern is also used in OpenCorporates and Pipl), and the flag is used to specify the country the company is based in. See also: https://www.maltego.com/blog/customize-entity-overlay-icons/



Overlay example from BvD Orbis

- **Link directions.** When returning an Entity from a Transform, you can reverse the default "outgoing" link to instead point from output to input using `entity.reverseLink()`. You should make sure to have your links reflect meaningful directions when the information has some implicit hierarchy, or directed links are in any way meaningful. For example: if you pivot from a Company to its owners (Person and/or Companies), then the resulting links should likely be reversed so that in a hierarchical graph layout, these Entities will appear on top of the input Entity.



Overlay example from BvD Orbis

Similarly, going from IP address to "Domains hosted on IP" could also benefit from reversing the link direction, to give another example.

- **Link labels.** In addition, it can be very helpful to make use of link labels to specify additional information about the relationship implied by a link.

- **Icons.** You can customize Entity icons from the server using `entity.setIconURL(...)`. The provided URL will be rendered as the Entity's icon by the client. We recommend only using this in particular scenarios where the icon provides helpful context (like e.g. profile pictures on social media account Entities). Please do not replace standard Maltego icons with icons of the underlying service you are integrating - Maltego Entity icons should help you identify Entities, not where the data came from.

- **Transform settings.** Transform settings are often important and can be used to customize the input and/or behavior of Transforms in many different ways. See: **https://docs.maltego.com/support/solutions/articles/15000019338-transform-settings**

  - **Transform settings** (or of course OAuth) are also how your Transforms should perform authentication to the underlying API or service you are integrating. If the service requires an API key, users should supply this key as a Transform setting.

- **Don't assume the presence of any dynamic properties.** In principle, you can add any property to any kind of Entity, and if it is not defined on that Entity it will become a so-called "dynamic property". This can be a useful feature and may make it easier to re-use existing Maltego Entities, however, you should never assume that a dynamic property exists on an Entity just because your Transforms tend to add it. In some cases it may make sense to have your Transform check for the presence of a dynamic property and use the information contained in it, but you must always assume that input Entities might have originated from another integration and therefore will not have the properties you expect.

  - **If you absolutely need a dynamic property for a Transform to run, you should likely create a new subclassed Entity type and instead define your Transform only for this type, not the parent type.**

- **Exception handling and output window messages.** Your Transforms can (and should) write info, warning and error messages to Maltego's output window to provide additional context to the user. If a Transform failed, it is best if you communicate to the user why it failed. Even if your Transform does return data, it can be helpful to provide explanations about anything the user should additionally be aware of. You can return messages to the Maltego graph in the TRX library by using the `addUIMessage` method (for an example, **see here**)

- **Strict/loose property matching.** When you add a property to an Entity in a Transform, you have to specify either strict or loose as its matching rule. Remember that Entities will only automatically merge in a Maltego graph if the following conditions hold:

  1. They have the same Entity type
  2. Their main (edit) values are equal
  3. All strict matching properties are equal

Takeaways

- Make sure to keep code structure clean and simple, keeping Transform code short and re-using API wrapper code & entity constructors
- Avoid calling local code, keep your Transforms self-sufficient
- Make use of Maltego's more advanced display features to make your integration more useful and intuitive

Creating Custom Entities (and other Configurations)

If you've decided that your integration will need custom Entity types to work properly, you'll need to create these in the Maltego client and export them to a so-called MTZ file. This file can then be served by a TDS (or manually installed, for local Transforms) to distribute the Entities.

The same goes for any other settings you create: Transform sets, custom icons, even Machines can be exported, and if they are an important part of your integration, you need to include all of them in the MTZ file you will be uploading and providing from the TDS.

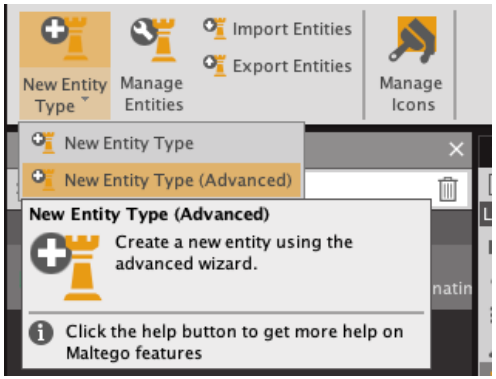
To learn how to create custom Entities, please refer to the following guides:

- Blog: https://www.maltego.com/blog/create-your-own-custom-entities-in-maltego/
- Docs: https://docs.maltego.com/support/solutions/articles/15000010462-create-new-entity
- Notes on best-practices: https://docs.maltego.com/support/solutions/articles/15000019245-custom-entity-guidelines

In a nutshell, here are some key rules summarized:

- Choose useful inheritance to existing standard Entities, and make sure to also reference the parent Entity properties correctly in your Transforms later on
- Keep properties simple—Entities can have many properties, but this isn't usually necessary
- Avoid dead-end Entities that have no outgoing Transforms (just like you would avoid leaf-nodes during relationship design)
- If possible, use the same display value and main (edit) value. If your main value is a database ID (common pattern when integrating systems that may return different/partial information for the same ID), it usually makes sense to break this rule and use a different display value.
  - For example: Depending on which API route is used, BvD Orbis returns simplified or full names for a person, but the underlying ID will be the same. We therefore made this ID the main value (so that merging works as expected) but used the person full name as the display value (because IDs make it impossible to quickly tell who a person is by looking at the graph).
  - (If your integrated system has similar behaviour, but you could, for the example above, expect the person's name to stay identical, you can use that name as both edit and display value, and simply make the ID a strict-matching property)

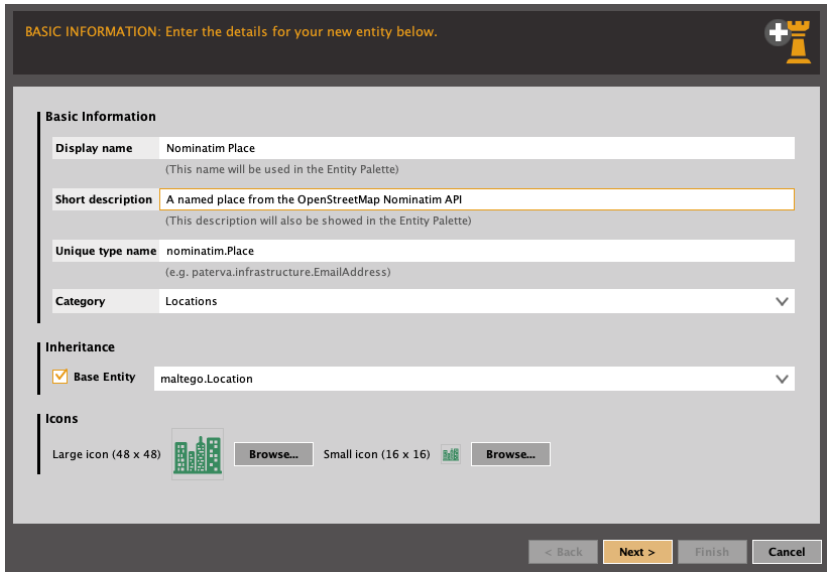
Changing Entities. Take special care to get your Entities exactly right before going into production! Once an Entity (or any other configuration) is published and installed by Maltego users, it will, in most cases, not automatically be updated again. If you forgot to add an inheritance, that is a very difficult problem to fix after launch. Users would have to manually delete the affected Entity and then reinstall your integration.



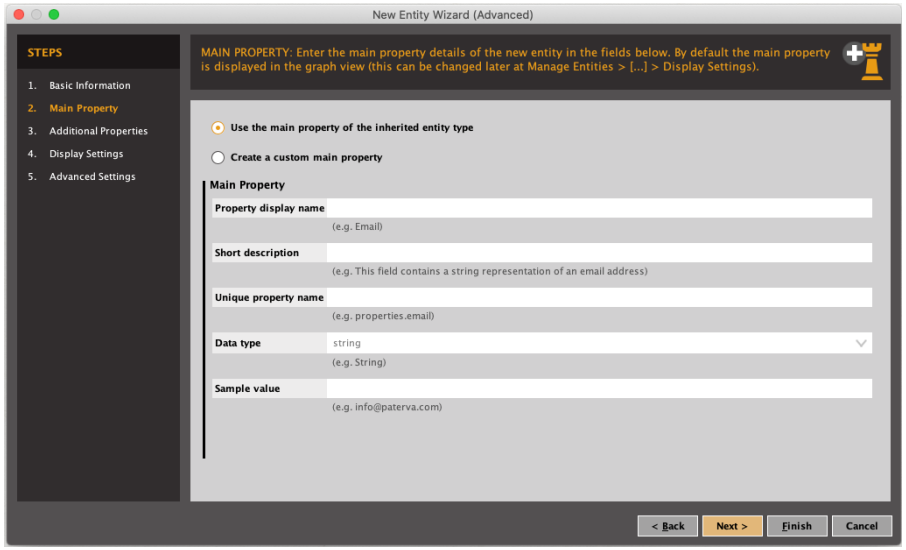
Example - Creating the custom nominatim.Place Entity

We create our new Nominatim Place Entity right in Maltego by selecting the "New Entity Type (advanced)" menu item.

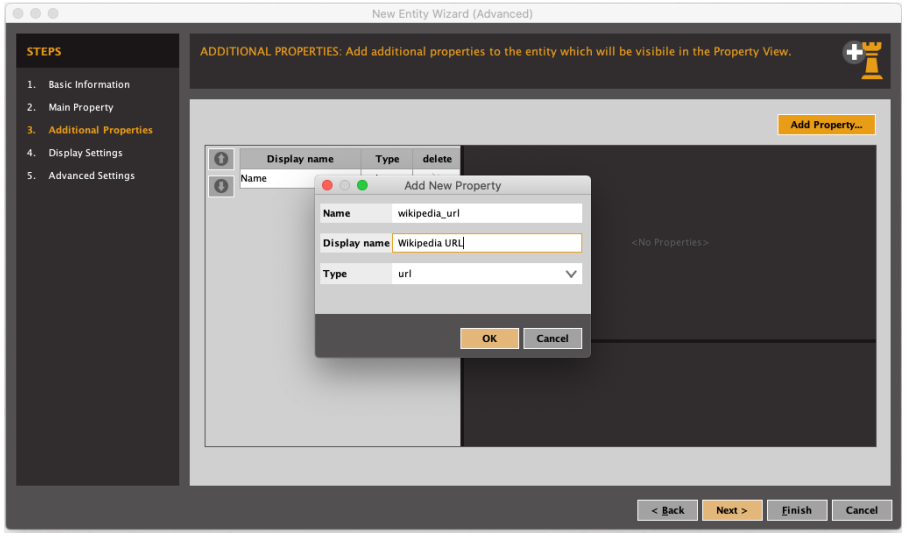
We start with the basic metadata and by selecting an Entity icon (you can also add your own).

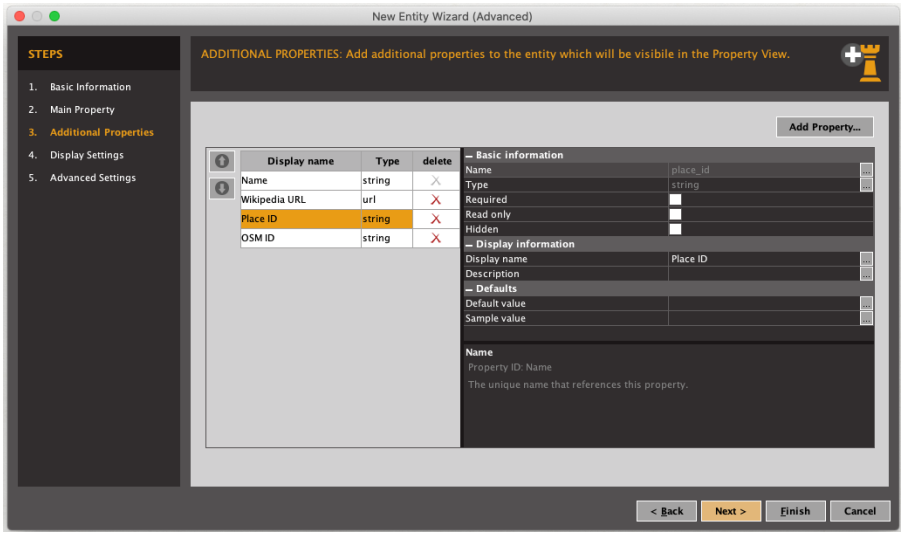


Next, since we still want the “Name” property that is inherited from maltego.Location as the main display value (i.e. Entity caption), we don’t really have to do anything in this step:



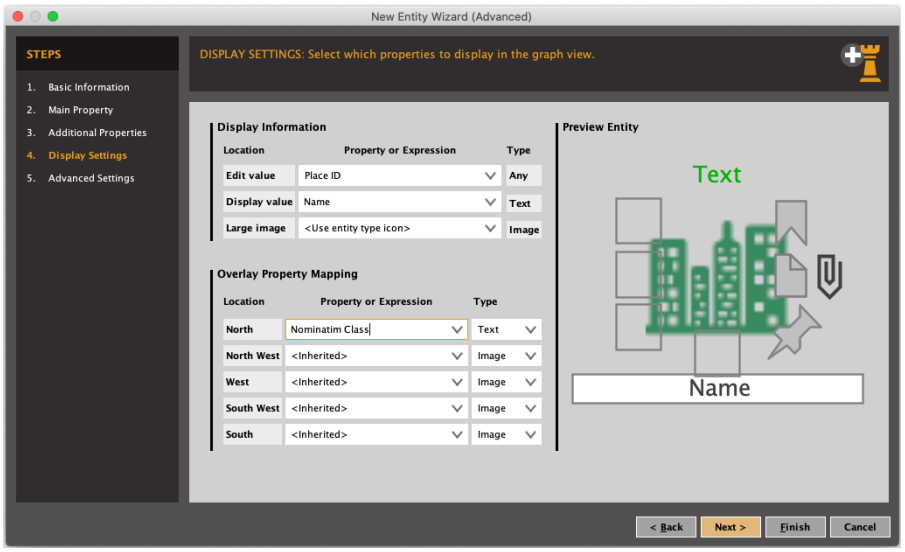
Now we create the properties we want to add to the Entity, making sure to select appropriate types:





(Note: we also added a nominatim_class property with type “string” that is not shown in the screenshot above!)

Finally, we can customize our merging behavior and the overlay information Maltego will show for this Entity. Nominatim sometimes provides a more details “class” attribute that indicates tourist attractions, shops, etc. We can display this label on top of our entites automatically using settings like the ones below:



Finally, here's an instance of the finished Entity type on a Maltego graph:

Deployment, Quality Assurance and Launch

Step 4: Deploy and Test Your Transforms

Once you start using your Transforms, you may notice some incompatibilities or design issues that need to be addressed before launch.

Deployment to the TDS

To deploy your Transforms to a TDS, see **our technical documentation here**.

You'll also have to upload any custom Entities, Icons, Machines, and other configuration that your integration depends on. **See this guide in our documentation for step-by-step instructions.**

Running in production. If you're running your integration in production and expect a larger number of users may start using it, make sure to deploy your TRX server accordingly. Our example docker-compose setup runs Transforms in gunicorn by default, but you may want to consider adding e.g. an Nginx reverse proxy in front of this before going live. You'll also want to make sure gunicorn (or Apache, ...) is configured to spawn enough workers to handle the incoming requests: each Transform is one HTTP POST request that will take up one of your threads. Maltego integrations are usually strongly IO-bound (and your data source may take some time to return data); make sure to plan your worker configuration accordingly to avoid large request backlogs and potential outages.

Security. Your Transform server might be high-value targets for hackers and other interested parties. Make sure to configure proper SSL encryption on your server and to secure access to it. If you want to additionally prevent third parties from directly sending requests to your server, you may set up firewall rules such that only the TDS (whether internal or the Paterva Public TDS) can send HTTPS requests to your Transform server. If you need information on the outgoing IP addresses for pTDS traffic, please contact support@maltego.com.

Testing and Quality Assurance

This process is again very dependent on your specific integration, and of course, at a basic level, the most important thing is to just make sure that it works and there are no unexpected errors. You can of course, also consider unit-testing your Transforms, this helps detect changes to the underlying data source that may break your Transforms unexpectedly. In this guide though, we'll focus more on the qualitative aspects and acceptance testing of an integration.

Basically, you could expect that Maltego will evaluate the following when considering an integration for inclusion in the Transform Hub, and it's a good practice for you to hold your own integration to the same standards:

- **Typical use-cases of the integration are achievable:** Putting yourself in the shoes of a user (e.g. a cyber analyst), you're able to perform all the tasks that you would want to given the integrated service. This is by far the most important acceptance test, and you should spend time on it, requesting real user feedback if possible.

- **Compatibility and interoperability with other integrations:** At any point in time, if you see an Entity on your graph (that was returned by your integration) and you have an idea of what you would want to know about that Entity from another integration, the required Transform should be runnable on that Entity.

  - The same goes in the opposite direction: if you see an Entity from another integration and you instinctively want to run one of your own Transforms on it, make sure to make this Transform compatible with that Entity type (within reasonable bounds).

  - In practice, this sometimes cannot be achieved if the other integration is lacking certain Transforms (or "entry points" to its Transforms). However, you should do everything reasonably in your power to **reuse and return the right Entity types to maximize interoperability** from your integration.

  - If you notice a "missing link" to another integration, consider adding a missing Transform, changing your returned Entity type, changing your Entity inheritance

- **Unnecessary complexity:** Just because a feature can be built, it is not necessarily useful. If there are Transforms that are never used and are unlikely to be used, consider removing them. The same goes for superfluous properties, icons, overlays, display information and other elements. Your integration doesn't need to be minimalistic, but it should also not overwhelm the user.

- **Adherence to the design guidelines:** Hardly necessary to repeat, but all the best practices outlined above should be validated in practice. Are there any leaf node Entities? Do some Transforms "skip" conceptual links that should be explicitly modelled? Are any reverse pivots missing? Are link directions and link labels meaningful? Are slider value limits respected? Are namespaces chosen well?

Step 5: Launch Your Integration

Once your Transforms are live, there are a number of ways you can distribute them to other users and considerations to keep in mind in that process.

Distribution

Local Transforms:
If you developed local Transforms and have no plans to deploy and host them for your users, you could still share them by e.g. uploading the code to Github, along with any necessary configuration files and setup instructions. As with any open-source project, make sure not to accidentally check in any of your authentication credentials or other secret information that you may have needed during development!

Public TDS Seeds:
If your project is non-commercial, you can simply distribute a pTDS seed URL to your users and have them manually add the integration to Maltego. If your project is particularly cool, feel free to reach out to us and we'll be happy to consider giving you a shoutout on our blog, Twitter, or other media.

If you think the integration is mature enough (and you're willing to cover the hosting or work with us to set it up on Maltego's side), you can also contact us about an official Transform Hub membership! We're happy to provide third party integrations completely for free, as long as the underlying data source is also free, you're legally allowed to access it (and to integrate it into Maltego), and you do not ask any of your users to pay for using the integration. Of course, we do reserve the right to refuse hub membership even if these criteria are met, but please feel encouraged to reach out to us and discuss the possibility.

For more information on Hub membership, see below.

> Do not use the Public TDS for commercial deployments without prior discussion and consent from Paterva PTY.

Private iTDS Seeds

If you need to distribute Transforms inside your organization, **you should do so with an iTDS**.

Note that you can export the Maltego Client's Hub configuration to an MTZ file that you can tell your users to install into Maltego in order to set up the internal seed URL automatically for new users. You can also simply have the users add the Seed URL to their Maltego Client manually though. If you distribute a configuration file instead, you'll also be able to add a custom icon and description, which may make for a nicer user experience.

> If you're using an iTDS for non-internal users, you may be subject to different commercial terms than what a standard iTDS license entails. As a general rule, make sure to inform your sales contact at Maltego of any plans to provide Transforms as a (potentially paid) service for customers who are not part of your organization. Similarly, you may not use the pTDS for such activities without prior approval (see above).

Transform Hub

If you'd like to make your integration available to the whole (or potentially just the professional / enterprise segments of the) Maltego community, the Transform Hub is the best way to do so—whether you're a commercial data provider or just a private individual writing Transforms for fun.

> If your project is free (both the integration and underlying data), Maltego will not charge any fees for featuring it on the hub and making the integration accessible to users.
>
> If you have a commercial project or are integrating a commercial API, please reach out to the Transform hub team to discuss terms and conditions for being featured on the Transform hub.

> You can also make your integration available only to specific groups of users, including for example restricting access (as well as visibility) to vetted enterprise and/or government organizations. Reach out to the Transform Hub team for more details (using the form on the **Transform Hub website**).

> We make no guarantees that we will indeed feature a third-party integration on the Transform Hub. A common reason for this could be that we are already developing or are planning to develop an integration for a given service or product in-house. If you want to make sure you're not wasting efforts, reach out to us before starting your project to discuss likely scenarios for your planned integration.

Contact

You can reach out to Maltego's Transform Hub team using the form near the bottom of this page: **Data at your Fingertips**

About Maltego

Maltego is a comprehensive tool for graphical link analysis that offers real-time data mining and information gathering, as well as the representation of this information on a node-based graph, making patterns and multiple order connections between said information easily identifiable. With Maltego, you can easily mine data from disparate sources, automatically merge matching information in one graph, and visually map it to explore your data landscape. Maltego offers the ability to easily connect data and functionalities from diverse sources using Transforms, which are small pieces of code that automatically fetch data from different sources and return the results as visual entities in the Desktop Client. Via the Transform Hub, you can connect data from over 30 data partners, a variety of public sources (OSINT) as well as your own data and systems. The different Desktop Client editions, data sources and server solutions enable you to tailor Maltego to your specific needs in terms of data access, functionalities, and security requirements.

For more information, please visit: maltego.com