# How to ingest and process data using Deep Lynx

April 2020

*Note: This guide assumes you know how to create a container and upload or create your own ontology using Metatypes, Metatype Keys, and Metatype Relationships. This guide also assumes basic knowledge of HTTP and REST APIs and that the user has access to the schema of the Deep Lynx database.*

## Create a Data Source

In order to process data, you must first tell Deep Lynx where and how to retrieve that data. Creating a Data Source instructs Deep Lynx where and how to retrieve the data, keeps data separate from other ingestion sources, and allows for mapping specific to that Data Source. Data Sources are kept in the “data\_sources” table.

In order to create a Data Source, you must POST information to the containers/:id/import/datasources endpoint. Each method below will contain a sample payload. API documentation included in the project also contains sample payloads.

There are only two Data Sources available at time of writing: Manual and HTTP.

**HTTP:** This Data Source allows the user to specify an HTTP endpoint and any authentication method needed to access it (currently Basic Authentication and Bearer Token methods supported). The HTTP data source, once created, will poll the provided HTTP endpoint at a configured interval, with each poll including a query parameter stating the last time a successful poll was executed. HTTP expects that the response payload be an array of JSON objects.

Here is an example:

{

    "name": "Test HTTP Source",

    "adapter\_type":"http",

    "active": true,

    "data\_format": "json",

    "config": {

        "endpoint":"",

        "auth\_method":"basic",

        "username": "test",

        "password": "test"

    }

}

**Manual**: This data source allows a user to upload JSON metadata directly to Deep Lynx. After data source creation you can use the endpoint containers/:id/import/datasources/:id/imports to upload data. This adapter expects data to be an array of JSON objects.

Here is an example:

{

    "adapter\_type":"manual",

    "name": "Testing",

    "active": true,

}

## Allow Data Source to import data or upload manually.

Now that a Data Source has been created, you can upload data into Deep Lynx. In the case of the HTTP Data Source, you need only wait for the system to automatically poll the configured endpoint and load the data. In the case of a manual Data Source, you must use the appropriate endpoint to upload your data into Deep Lynx.

Once data has been uploaded it is stored as an Import in the “imports” table. Each Import contains the data, status, and data source that data belongs to. Data is stored in a column prefixed by the data type – e.g. json\_data or csv\_data.

Once an Import has been created and the data stored, automated processes will parse the original data payload into individual data chunks. Each chunk is stored in the “data\_staging” table as a JSON object. In order to process non-JSON data, a processor should be created that is called when an Import is converted from single record to data chunks and inserted into “data\_staging”.

Currently, only the JSON processor is active. It will automatically, using database triggers and cron jobs, take and chunk the data from an Import into data\_staging.

## Create Type Mappings

For the chunked data to be inserted into Deep Lynx, the system must know **how** to store it. Because Deep Lynx stores data in a structured ontology, you must tell Deep Lynx how your data maps to its known Metatypes, their keys, and Metatype Relationships along with their keys. **Think of Type Mapping as teaching Deep Lynx how to interpret and store your data using its known types.**

Given the following payload from your Data Source (parsed from an import and stored in data\_staging):

{

    "RAD": 0.1,

    "COLOR": "blue",

    "TYPE": "EQUIP",

    "TEST": "TEST",

    "ITEM\_ID": "123"

}

You might create a Type Mapping (by POSTing this information to the proper endpoint) that looks like this:

{

    type\_key: "TYPE", // indicates what key contains the type name

    type\_value: "EQUIP", // the type name to match

    unique\_identifier\_key: "ITEM\_ID", // the payloads unique id

    metatype\_id: // known type ID

    keys: [

        {

            key: "RAD",

            metatype\_key\_id: // known type key id

        },

        {

            key: "COLOR",

            metatype\_key\_id: // known type key id

        }

    ],

    example\_payload: // the payload above

    ignored keys: []

}

Relationships between data might look like the following.

This payload indicates that this is both data and relationship – with PARENT\_ID indicating a unique identifier for the parent piece of data.

{

    "RAD": 0.1,

    "COLOR": "blue",

    "TYPE": "CONNECTION",

    "TEST": "TEST",

    "ITEM\_ID": "12346",

    "PARENT\_ID": "123",

}

A type mapping for this might look like:

{

    type\_key: "TYPE",

    type\_value: "PIPE",

    relationship\_type\_key: "TYPE",

    relationship\_type\_value: "CONNECTION",

    metatype\_relationship\_id: // Metatype Relationship ID

    origin\_key: "PARENT\_ID",

    destination\_key: // would be this items's id since this is a Child/Parent relationship

    unique\_identifier\_key: "ITEM\_ID",

    metatype\_id: metatype2ID,

    keys: [

        {

            key: "RAD",

            metatype\_key\_id: // type key id

        },

        {

            key: "COLOR",

            metatype\_key\_id: // type key id

        }

    ],

    example\_payload: // payload above

    ignored\_keys: ["ITEM\_ID", "TYPE"]

}

A Type Mapping accomplishes its goal by giving Deep Lynx a way to match the payload to an existing type, and to match that type’s properties to the payload properties. **Type Mappings are complex and should be created with care**

## Data is transformed and inserted into Deep Lynx

For an Import’s data to be inserted into Deep Lynx, each data chunk must be associated with a Type Mapping created above. Type Mappings are constantly being applied to matching data automatically, and requires no user intervention apart from teaching the system through creation of new Type Mappings. Once an Import’s data is completely mapped, the transformation and insertion process will begin automatically.