Enin Datasets API Tutorial

PDF Version (datasets_tutorial.pdf)

The Datasets API focuses on delivering large amount of data for statistical analysis, machine learning modeling and data warehousing. It primarily aims for daily data loads, but there is no technical limitation regarding this. Depending on the source, some data will be updated on varying schedules.

The Datasets API uses HTTPS requests like most RESTful APIs, however, the underlying infrastructure is a little bit more flexible, and some particular optimizations have been done to make things faster and to minimize data size. In the future, alternative interfaces will be developed, e.g., FTP transfer or e-mail (for smaller datasets).

Available API Endpoints

The Datasets API consists of multiple endpoints which are divided into logical groupings. As of May 2020, the following groupings of Datasets API endpoints are available:

- Datasets: Accounts
- Datasets: Company
- Datasets: Company Flag
- Datasets: Company Event (ready summer 2020)

Each of these differ of course in what data is available in them, but also for what granularity the data is delivered.

There is also an “Experimental” datasets endpoint group which contain legacy endpoints, or brand new API endpoints not ready for production. We discourage the use of these in production, since they may be changed or removed at any moment. When API endpoints in the this group are ready for production, they will be moved to an appropriate Datasets API endpoint group.

In addition to the datasets endpoint groups there are some system specific endpoint groups:

- Miscellaneous
- API Client
- Batch

These are there to test your connection, get information about your client, or to set up custom collections of companies, which we call company batches.

The following section describes how to use the API. It will guide you through the implementation of a basic script for downloading datasets to files on disk.

Requirements

At Enin we use python as our primary backend programming language, so examples will be in that language. If you’d like to get help with integration with your own tool or language, we are more than happy to guide you through the process. If we are familiar with your tool or programming language

we can even help port the examples. That said, all operations here are in essence just HTTPS requests, so if your tool or programming language supports this you should be able to consume the data with little effort.

To be precise, code in this tutorial was writen using the PyCharm 2019.3 IDE using Python 3.6.9 running on Ubuntu 18 in a Vagrant/VirtualBox hosted on a Windows 10 machine.

Only the requests library (https://requests.readthedocs.io/) will be an external dependency. It can be installed via `pip` :

```
pip3 install requests
```

Now that our prerequisits are in order we will start our python script. Create a file named `datasets_script.py` , and include the following imports:

```
import requests
import json
from datetime import datetime, timedelta
```

These imports will be used later. Continue to build on this script as we move along. Also we'll be using a custom helper function to print some objects.

```
def print_obj(obj):
    print(json.dumps(obj, indent=4, ensure_ascii=False))
```

You can alternatively just use the built in `print()` function or the `pprint` module.

Note that we are using basic python scripting here without bells-and-whistles like a main entry point, or classes, or methods, or functions. This is to keep the code clear for users who aren't familiar with python's idioms. You should of course use a appropriate coding structure for your purposes. At some later time, we will include a properly designed python client, which can be used directly.

Authentication

Now that we have a script file, lets focus on authentication, i.e., the process of us knowing that you are really who you are saying you are.

Authentication is done with expirable Access Tokens (https://auth0.com/docs/tokens/concepts/access-tokens) or Basic HTTP Authentication (https://en.wikipedia.org/wiki/Basic_access_authentication) . We recommend using tokens for production systems. In the future we will require IP whitelisting if you use basic authentication.

> **Note** : This tutorial is self contained, so you should be able to just follow along, including basic authentication process. But if you need extra insight, or just need some help, you can follow our getting started documentation (https://api.enin.ai/) (pdf (https://api.enin.ai/getting_started.pdf)), or just contact us on team@enin.ai (mailto:team@enin.ai) or the Enin External Slack (https://enin-external.slack.com/) if we have set an account up for you.

Now, time to make our first request. Let's check that you can reach the Datasets API at all - without authentication:

```
system_status = requests.get("https://api.enin.ai/datasets/v1/system-status").json()
print_obj(system_status)
```

This should print something like the following:

```
{
    "message": "Enin Dataset API is operational.",
    "python_version": "3.6.9 (default, Nov  7 2019, 10:44:02)  [GCC 8.3.0]"
}
```

Next, let's include authentication. For this tutorial we will be using basic authentication, i.e., just a password and username. These are called `Basic Auth Client ID` and `Basic Auth Client Secret` in your provided credentials file. If you haven't received a credentials file yet, then either contact us at team@enin.ai (mailto:team@enin.ai) or speak to someone at your organization which has gotten the credentials file.

Please take care to save the credentials file properly, i.e., encrypted and inaccessible to other parties. When using the credentials in your software remember to always use best practice secret handling. In our example we will be storing the credentials as a read restricted json file named `.auth.json` located next to the python script `datasets_script.py` . You *can* technically hard code the client id and secret in your source code, however, we highly discourage it.

The `.auth.json` file should contain the following json data where `YOUR_CLIENT_ID` and `YOUR_CLIENT_SECRET` are replaced appropriately:

```
{
  "client_id": "YOUR_CLIENT_ID",
  "client_secret": "YOUR_CLIENT_SECRET"
}
```

If you are running on Linux/Ubuntu you can restrict this file to your user by running:

```
sudo chmod 600 .auth.json
```

If you are using a source control system like Git (https://git-scm.com/) , make sure to not add this file as source code. In git you can do this by adding the following line to the end of your `.ignorefile` :

```
.auth.json
```

Now, lets load the authentication information from file.

```
with open('.auth.json') as file:
    auth_json = json.load(file)
client_id = auth_json['client_id']
client_secret = auth_json['client_secret']
auth = (client_id, client_secret)
```

Next, we will try to access an authentication guarded endpoint.

```
auth_status = requests.get(
    "https://api.enin.ai/datasets/v1/auth-status",
    auth=auth
).json()
print_obj(auth_status)
```

If all goes well, then this should print:

```
{'message': 'You are authenticated.'}
```

Let’s have a look at our API client identity:

```
api_client_identity = requests.get(
    "https://api.enin.ai/datasets/v1/api-client-identity",
    auth=auth,
).json()
print_obj(api_client_identity)
```

This will give summary of some basic identity information about your API Client.

An expanded version of this endpoint is the “composite” version of this endpoint, `api-client-identity-composite` . Let’s have a look.

```
api_client_identity_composite = requests.get(
    "https://api.enin.ai/datasets/v1/api-client-identity-composite",
    auth=auth,
).json()
print_obj(api_client_identity_composite)
```

This should print out something like the following:

```
[
    {
        "api_client": {
            "uuid": ...,
            "company_uuid: "b1ba011f-6350-45b3-9b0a-f02c07c0cff5,
            ...
        },
        "api_client_identity": {
            "uuid": ...,
            "company_uuid: "b1ba011f-6350-45b3-9b0a-f02c07c0cff5",
            ...
        },
        "app_customer": {
            "app_url": "https://app.enin.ai/settings/customer/b1ba011f-6350-45b3-9b0a-
f02c07c0cff5",
            "app_user_email": ...,
            "app_user_uuid": ...,
            "desensitize_flag": 0,
            "long_name": "Bank of Testing",
            "name": "Test Bank",
            "picture": "https://i.imgur.com/oXbiJYz.png",
            "slug": "test",
            "uuid": "b1ba011f-6350-45b3-9b0a-f02c07c0cff5"
        },
        "app_url": "https://app.enin.ai/settings/customer/b1ba011f-6350-45b3-9b0a-f02c07c0cff5",
        "uuid": ...
    }
]
```


Notice that we still get the `api_client_identity` entity, but now also `api_client` and `app_customer` entities. This is a general pattern you will find in the Enin APIs. Any time you see `-composite` in an endpoint. Think of it as a version of an existing “base” entity which has been expanded to include more information, yet still retains the granularity of the base entity.

Based on the information in the `app_customer` entity it seems we are part of the `Bank of Testing` customer organization. Let’s see if there are more API clients registered for this customer. The `uuid` ref (https://en.wikipedia.org/wiki/Universally_unique_identifier) of the `app_customer` entity is `b1ba011f-6350-45b3-9b0a-f02c07c0cff5`, and the customer `slug` is `test`. We could use either to query for all of this customer’s `api_client_identity` entities. Let’s use the `uuid`:

```
app_customer_api_client_identity_composite = requests.get(
    "https://api.enin.ai/datasets/v1/app-customer/b1ba011f-6350-45b3-9b0a-f02c07c0cff5/api-client-
identity-composite",
    auth=auth,
).json()
print_obj(app_customer_api_client_identity_composite)
```

This returned just the same object as before. That means this is the only API Client for this customer organization. But if there was more they would be listed.

> **Note** : There are few to no secrets internally to one customer organization. Any application user or API client can see most if not all information about their customer organization. If you need chinese walls, or otherwise separate information between multiple members of a customer organization, then we will help you set up a sister customer organization.

Enough about metadata. Let's get som real datasets out of the API.

Datasets

First, lets talk about the dataset endpoints in general.

Things you can typically do with each endpoint includes:

- Fetch simple data objects, called entities, or larger composites of entities.
- Select fields to include in the dataset by:
  - … keeping only chosen fields or entity types.
  - … ignoring chosen fields or entity types.
- Apply arbitrary filters by:
  - … using most possible filtering operators on any available field.
  - … manually creating a batch of companies to query from.
- Choose desired file type: `csv` , `json` , or `jsonl`
- Order by fields.
- Limit/offset the number of entries for testing and pagination.

Each of these operations are enabled through query strings (https://en.wikipedia.org/wiki/Query_string) .

Entities and their composites

Before we get started with examples, it might be useful to understand our API meta data design principles. One integral API design philosophy is that the API itself seldom uses individual fields of data, but rather full objects we call Entities. This means you will seldom se custom entities which merge fields from multiple sources, instead you'll find entities of one type or larger composites of entities.

Take accounts data. The most basic Entity for accounts is the `AccountsEntity` , it hardly has any information. It just says whether there is an account at all. It is perfectly possible for us to *know* there are accounts for a company without us yet having fetched the associated data. As of May 2020, the the `AccountsEntity` is defined with the following fields/meta data:

```
{
    "uuid": String,
    "company_uuid": String,
    "accounting_year": Integer,
    "accounting_announcement_date": Date,
    "accounting_schema": String,
    "accounting_from_date": Date,
    "accounting_to_date": Date,
    "accounts_type_uuid": String,
    "app_url": String,
}
```

Using the Datasets API, this data can simply be fetched from `https://api.enin.ai/datasets/v1/accounts` using an HTTP `GET` request. Without any extra parameters this will give you all the entities available of this kind. Of course you might want to filter this, and we will get to that in the next sections. But another glaring issue is that the data you get from that endpoint is far from enough to do anything useful with. So, we of course have other associated entities like `AccountsBalanceSheetEntity` and `AccountsIncomeStatementEntity` with their respective fields.

It would be very cumbersome if you had to fetch each of these individually. This is where composites enter the scene. Composites are simply multiple entities merged into one larger entity at the granularity of the entity in focus, i.e., the *subject* of the composite. Let's look at what an `AccountsCompositeEntity` would look like:

```
{
    "accounts": AccountsEntity,
    "accounts_highlights": AccountsHighlightsEntity,
    "accounts_type": AccountsTypeEntity,
    "accounts_income_statement": AccountsIncomeStatementEntity,
    "accounts_balance_sheet": AccountsBalanceSheetEntity,
    "company": CompanyEntity,
}
```

Here you see each element is it's own entity. These have their own fields/metadata, so let's expand the first two to see what is under the hood, i.e., let's expand `AccountsEntity` and `AccountsHighlightsEntity` .

```
{
    "accounts": {
        "uuid": String,
        "company_uuid": String,
        "accounting_year": Integer,
        "accounting_announcement_date": Date,
        "accounting_schema": String,
        "accounting_from_date": Date,
        "accounting_to_date": Date,
        "accounts_type_uuid": String,
        "app_url": String,
    },
    "accounts_highlights": {
        "uuid", String,
        "accounts_uuid", String,
        "income_statement__operating_revenue", Float,
        "income_statement__ebitda", Float,
        "income_statement__ebit", Float,
        "income_statement__ordinary_result_after_taxes", Float,
        "income_statement__net_income", Float,
        "balance__intangible_assets", Float,
        "balance__fixed_assets", Float,
        "balance__cash_and_deposits", Float,
        "balance__total_current_assets", Float,
        "balance__total_assets", Float,
        "balance__equity", Float,
        "balance__liabilities", Float,
        "current_ratio", Float,
        "quick_ratio", Float,
        "return_on_assets", Float,
        "profit_margin", Float,
        "equity_profitability", Float,
        "equity_ratio", Float,
        "debt_ratio", Float,
        "income_statement__ebt", Float,
        "income_statement__currency_code", String,
        "accounting_year", Integer,
        "app_url", String,
    },
    "accounts_type": AccountsTypeEntity,
    "accounts_income_statement": AccountsIncomeStatementEntity,
    "accounts_balance_sheet": AccountsBalanceSheetEntity,
    "company": CompanyEntity,
}
```

To download this object you'd do a HTTP `GET` request to `https://api.enin.ai/datasets/v1/accounts-composite`.

As you can see, there is a lot of fields. The sheer volume of meta data in our system is why we seldom operate and manage individual fields. We define the meta data once, centrally, and reuse that definition across our platform. If we had to manage them individually all the time, Enin would not be doing much more than meta data curation. That said, as a consumer of our data you do want to manage the fields you are interested in, and not ALL data ALWAYS. Because of this we have created some tools to be able to select fields and filter entries to your needs. This will be the focus in the next sections.

Fetching basic entity data

While discussing entities, we've already looked at some basic API endpoints, now let's be explicit about how to use them. We will be fetching entities of type `CompanyEntity` and `CompanyCompositeEntity`, from the endpoints `https://api.enin.ai/datasets/v1/company` and `https://api.enin.ai/datasets/v1/company-composite`, respectively.

Let's start by requesting 3 entities of type `CompanyEntity`. You can limit the number of entries returned to 3 by adding the query string `?limit=3`:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company?limit=3",
    auth=auth,
).json()
print_obj(companies)
```

By default this could print something like the following:


```
[
    {
        "insert_timestamp": "2019-08-05T00:18:08.349297+00:00",
        "name": "EIENDOMSMEGLER 1 HEDMARK EIENDOM AS AVD GJØVIK",
        "org_nr": "972150797",
        "org_nr_schema": "NO",
        "update_timestamp": "2019-08-05T00:18:08.349297+00:00",
        "uuid": "14d39ef2-d1a1-46b9-8571-440d23dbb6d0"
    },
    {
        "insert_timestamp": "2019-07-24T07:34:44.004922+00:00",
        "name": "SPAREBANKEN SOGN OG FJORDANE AVD FLORØ",
        "org_nr": "973167529",
        "org_nr_schema": "NO",
        "update_timestamp": "2019-07-24T07:34:44.004922+00:00",
        "uuid": "b5b222fa-a9fe-4c04-bc9f-c9a9f9c32f44"
    },
    {
        "insert_timestamp": "2018-10-16T12:33:39.528266+00:00",
        "name": "Unknown Name",
        "org_nr": "927822334",
        "org_nr_schema": "NO",
        "update_timestamp": "2018-10-16T12:33:39.528266+00:00",
        "uuid": "59866d11-405e-44f2-9c70-f6345b2c9e4f"
    }
]
```


Notice that this returns valid JSON. If you are downloading few entries and the data fits into memory, then this is a good format to use, and there are plenty of libraries which handle it nicely. However, JSON is a little annoying to work with if it gets large. The fact that there are start and end square brackets, and the fact that there are N-1 number of commas delimiting each entity makes it hard to stream. There are of course tools which handle this, but to make things easier we also support
the JSONL format, which has one JSON object (entity) per line, and CSV which is has the same feature and is a lot more compact (doesn't repeat the field names all the time).

Let's try JSONL:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company?limit=3&file_type=jsonl",
    auth=auth,
).content.decode()
print(companies)
```

This outputs:

```
{"name": "EIENDOMSMEGLER 1 HEDMARK EIENDOM AS AVD GJØVIK", "uuid": "14d39ef2-d1a1-46b9-8571 ... }
{"name": "SPAREBANKEN SOGN OG FJORDANE AVD FLORØ", "uuid": "b5b222fa-a9fe-4c04-bc9f-c9a9f9c ... }
{"name": "Unknown Name", "uuid": "59866d11-405e-44f2-9c70-f6345b2c9e4f", "org_nr": "9278223 ... }
```

If you would like to stream this using python you could do:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company?limit=3&file_type=jsonl",
    auth=auth,
    stream=True
)
for line in companies.iter_lines():
    payload = json.loads(line.decode())
    print_obj(payload)
```

A similar strategy can work with other tools as well.

Fetching the same file as CSV can be done as follows:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company?limit=3&file_type=csv",
    auth=auth,
).content.decode()
print(companies)
```

… which prints:

```
uuid,name,org_nr,update_timestamp,insert_timestamp,org_nr_schema
14d39ef2-d1a1-46b9-8571-440d23dbb6d0,EIENDOMSMEGLER 1 HEDMARK EIENDOM AS AVD GJØVIK,9721507 ...
b5b222fa-a9fe-4c04-bc9f-c9a9f9c32f44,SPAREBANKEN SOGN OG FJORDANE AVD FLORØ,973167529,2019- ...
59866d11-405e-44f2-9c70-f6345b2c9e4f,Unknown Name,927822334,2018-10-16 12:33:39.528266+00,2 ...
```

Fetching composite entity data

When fetching composites things change slightly. With a composite you will notice that the entries are nested one level deeper. The JSON objects are nested one more level, and the CSV field names uses dot notation the indicate deeper structures.

Let’s fetch the JSON version of `CompanyCompositeEntity` :

```
company_composites = requests.get(
    "https://api.enin.ai/datasets/v1/company-composite?limit=3",
    auth=auth,
).json()
print_obj(company_composites)
```

This will give quite a huge object:

```
[
    {
        "company": {
            "insert_timestamp": "2018-04-27T16:33:37.237567+00:00",
            "name": "AMUND BROMSTAD",
            "org_nr": "991665595",
            "org_nr_schema": "NO",
            "update_timestamp": "2019-04-23T00:37:00.033835+00:00",
            "uuid": "000004ff-c89f-405d-b8dc-fcc1bc293019"
        },
        "organization_type": {
            "insert_timestamp": "2018-05-16T13:36:29.145604+00:00",
            "org_nr_schema": "NO",
            "organization_type_code": "FLI",
            "organization_type_description": "Forening/lag/innretning",
            "update_timestamp": "2020-05-11T05:18:02.127477+00:00",
            "uuid": "dbccefba-f227-568d-9933-7084395aba98"
        },
        "company_affiliation_accountant": { ...},
        "company_affiliation_auditor": { ... },
        "company_affiliation_type_accountant": { ... },
        "company_affiliation_type_auditor": { ... },
        "company_details": { ... },
        "company_location_business_address": { ... },
        "company_nace_code_primary": { ... },
        "company_nace_code_secondary": { ... },
        "company_nace_code_tertiary": { ... },
        "geo_location_business_address": { ... },
        "nace_code_primary": { ... },
        "nace_code_secondary": { ... },
        "nace_code_tertiary": { ... }
    {
        "company": {
            "name": "SYKKYLVEN GATEBILKLUBB",
            ...
```


The same query with `file_type=csv` as follows:

```
company_composites = requests.get(
    "https://api.enin.ai/datasets/v1/company-composite?limit=3&file_type=csv",
    auth=auth,
).content.decode()
print(company_composites)
```

… you will get:


```
company.uuid,company.name,company.org_nr,company.update_timestamp,company.insert_timestamp, ...
000004ff-c89f-405d-b8dc-fcc1bc293019,AMUND BROMSTAD,991665595,2019-04-23 00:37:00.033835+00 ...
0000075d-9a33-4d17-be6f-3adda98ebcae,SYKKYLVEN GATEBILKLUBB,912801535,2020-05-11 05:16:45.9 ...
00000d51-acaf-4db6-be73-13c727a57f9d,MOLIN MASKIN & TRANSPORT MARIUS TOBIASSEN,913966082,20 ...
```


Notice how `company.uuid` is nested using dot notation. This is will be relevant later when we select fields, and filter on field data.

Selecting fields

There is a wealth of fields available, but you probably don't want all of them in one go. There are two ways to filter down what fields you get in return. One is using the `keep_only_fields` query parameter, the other is to user `ignore_fields` . These can be used on individual fields like

`company.uuid` or `company.org_nr_schema`, or you can apply them to all fields of an entity by omitting the last part of the dot notation, e.g., you can use just `company` as a parameter value and it will apply to all fields of that entity.

Ignoring particular fields or entities

If you would like to ignore some fields or entities, then `ignore_fields` is your go to parameter. Say, you don't want the UUID of the `CompanyEntity` then you could write the query parameter as `ignore_fields=company.uuid`. You can also specify a list of fields or entities you would like to ignored by delimiting them by a comma (`,`). Say, you would like to also ignore the organization number schema (typically the country code), then you'd write the query parameter as `ignore_fields=company.uuid,company.org_nr_schema`. Or, say you would like to ignore the company entity all together, then you could write the query parameter as just `ignore_fields=company`.

Let's try this out on the `CompanyEntity` endpoint:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company?
limit=3&ignore_fields=company.uuid,company.org_nr_schema",
    auth=auth,
).json()
print_obj(companies)
```

Because the argument list is getting large let's use the `requests` library's query parameter functionality. The above query is identical to:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company",
    params={
        "limit": 3,
        "ignore_fields": "company.uuid,company.org_nr_schema",
    },
    auth=auth,
).json()
print_obj(companies)
```

And, we can also split the `ignore_fields` list using python's `str.join()` function. This means that the above is identical to the following:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company",
    params={
        "limit": 3,
        "ignore_fields": ','.join(
            [
                "company.uuid",
                "company.org_nr_schema",
            ]
        )
    },
    auth=auth,
).json()
print_obj(companies)
```

If you are making more advanced calls, as we are about to do, you should consider also parameterize like this in your tool or programming language.

Running the previous code prints the following:

```
[
    {
        "name": "EIENDOMSMEGLER 1 HEDMARK EIENDOM AS AVD GJØVIK",
        "org_nr": "972150797",
        "insert_timestamp": "2019-08-05T00:18:08.349297+00:00",
        "update_timestamp": "2019-08-05T00:18:08.349297+00:00"
    },
    {
        "name": "SPAREBANKEN SOGN OG FJORDANE AVD FLORØ",
        "org_nr": "973167529",
        "insert_timestamp": "2019-07-24T07:34:44.004922+00:00",
        "update_timestamp": "2019-07-24T07:34:44.004922+00:00"
    },
    {
        "name": "Unknown Name",
        "org_nr": "927822334",
        "insert_timestamp": "2018-10-16T12:33:39.528266+00:00",
        "update_timestamp": "2018-10-16T12:33:39.528266+00:00"
    }
]
```

Notice how `org_nr_schema` and `uuid` are missing.

Keeping only particular fields or entities

The previous section was all about ignoring fields you aren't interested in. That is nice to have, but what if you only need a few fields and entities? Listing almost all fields can get cumbersome fast. This is especially true if you are using composite API endpoints, which tend to have rather many fields and entities. Also, over time there are bound to be added more fields which would show up if you only relied on `ignore_fields` .

This is where the `keep_only_fields` query parameter comes in to play. It ignores all other fields than those you specified. Let's try it out by fetching `CompanyEntity` with it's primary `NaceCodeEntity` using the `https://api.enin.ai/datasets/v1/company-composite` API endpoint, and say, return it as a CSV file:

```
company_composites = requests.get(
    "https://api.enin.ai/datasets/v1/company-composite",
    params={
        "limit": 3,
        "file_type": "csv",
        "keep_only_fields": ','.join(
            [
                "company",
                "nace_code_primary",
            ]
        )
    },
    auth=auth,
).content.decode()
print(company_composites)
```

Notice that we just asked for the company and nace code entities, and not particular fields of those fields. This will return the following:

```
company.uuid,company.name,company.org_nr,company.update_timestamp,company.insert_timestamp, ...
000004ff-c89f-405d-b8dc-fcc1bc293019,AMUND BROMSTAD,991665595,2019-04-23 00:37:00.033835+00 ...
0000075d-9a33-4d17-be6f-3adda98ebcae,SYKKYLVEN GATEBILKLUBB,912801535,2020-05-14 01:03:19.4 ...
00000d51-acaf-4db6-be73-13c727a57f9d,MOLIN MASKIN & TRANSPORT MARIUS TOBIASSEN,913966082,20 ...
```

That's still a lot of data. We can't even fit it in the output box above. Let's use the ignore field functionality and remove the company UUID and timestamps:

```
company_composites = requests.get(
    "https://api.enin.ai/datasets/v1/company-composite",
    params={
        "limit": 3,
        "file_type": "csv",
        "ignore_fields": ','.join(
            [
                "company.uuid",
                "company.update_timestamp",
                "company.insert_timestamp",
            ]
        ),
        "keep_only_fields": ','.join(
            [
                "company",
                "nace_code_primary",
            ]
        )
    },
    auth=auth,
).content.decode()
print(company_composites)
```

At least we now can now fit some NACE code data into the output box:

```
company.name,company.org_nr,company.org_nr_schema,nace_code_primary.uuid,nace_code_primary.
AMUND BROMSTAD,991665595,NO,,,,,,,,
SYKKYLVEN GATEBILKLUBB,912801535,NO,840f6832-13de-43ab-9455-93822c3f5717,94.991,5,Aktivitet ...
MOLIN MASKIN & TRANSPORT MARIUS TOBIASSEN,913966082,NO,cc9b075e-ce7f-4255-b894-97ed6e9257b4 ...
```

Turns out `AMUND BROMSTAD` doesn't have a NACE code. You can see that by the empty data entries. Also we can barely see that `SYKKYLVEN GATEBILKLUBB` has the NACE code `94.991`.

Let's just keep only `nace_code_primary.nace_code` and `nace_code_primary.short_name`, and also ignore `company.org_nr_schema`:

```
company_composites = requests.get(
    "https://api.enin.ai/datasets/v1/company-composite",
    params={
        "limit": 3,
        "file_type": "csv",
        "ignore_fields": ','.join(
            [
                "company.uuid",
                "company.org_nr_schema",
                "company.update_timestamp",
                "company.insert_timestamp",
            ]
        ),
        "keep_only_fields": ','.join(
            [
                "company",
                "nace_code_primary.nace_code",
                "nace_code_primary.short_name",
            ]
        )
    },
    auth=auth,
).content.decode()
print(company_composites)
```

Finally, a nice short selection of fields we are interested in:

```
company.name,company.org_nr,nace_code_primary.nace_code,nace_code_primary.short_name
AMUND BROMSTAD,991665595,,
SYKKYLVEN GATEBILKLUBB,912801535,94.991,Interesseorganisasjoner ellers
MOLIN MASKIN & TRANSPORT MARIUS TOBIASSEN,913966082,43.120,Grunnarbeid
```

The take away is that you can combine `ignore_fields` and `keep_only_fields` on both fields and entities to select exactly the data you need.

Running this without the `limit` parameter took 3 minutes and returned 2165983 entries, which is exactly how many unique organization numbers we have in our system as of May 2020. This, is a lot. Let's start work on filtering.

Filtering entities

Some filters are special for particular endpoints, but in general the Datasets API have to main types of filters. Those which can be applied to data fields of the entities you are downloading, or based on what we call company batches.

Basic data field filtering

To spice things up, let's switch to accounting data using the `https://api.enin.ai/datasets/v1/accounts-composite` API endpoint. This endpoint is a composite endpoint with a wealth of information regarding company accounts and financial data of companies. The accounts entity composition looks as follows:

```
{
    "accounts": AccountsEntity,
    "accounts_type": AccountsTypeEntity,
    "company": CompanyEntity,
    "company_batch_mapping": CompanyBatchMappingEntity,
    "accounts_highlights": AccountsHighlightsEntity,
    "accounts_income_statement": AccountsIncomeStatementEntity,
    "accounts_balance_sheet": AccountsBalanceSheetEntity,
}
```

This has the granularity of each accounting report, typically one each year, and will have duplicate companies if there are multiple accounts for a company.

Let’s start by fetching info about the revenue of companies:

```
accounts_composites = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "limit": 3,
        "file_type": "csv",
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts.accounting_year",
                "accounts_income_statement.total_operating_income",
            ]
        )
    },
    auth=auth,
).content.decode()
print(accounts_composites)
```

This prints the following:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2000,Rana Maskinstasjon A/S,832071072,1687000.0
2018,OMIT AS,916648464,0
2018,HALSEN BORETTSLAG,952825240,998000
```

Those results seem very arbitrary. And if you removed the limit you’d get get 4814791 entries.

Filtering on entity fields is done using the previously introduced dot notation, in addition to an operator for more complex queries. For the most basic equality filtering you can simply use the dot notation as a query argument. The not notation itself indicates to the endpoint that you want to filter.

Let’s try it out by filtering on year:

```
accounts_composites = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "limit": 3,
        "file_type": "csv",
        "accounts.accounting_year": 2018,
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts.accounting_year",
                "accounts_income_statement.total_operating_income",
            ]
        )
    },
    auth=auth,
).content.decode()
print(accounts_composites)
```

In return we get:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2018,TRAIN ING. AS,997924231,
2018,HELLY INVEST AS,990759413,
2018,MOMENTUM RETAIL NORWAY AS,913104358,12385000
```

It seems to have worked. Adding more filters narrows down the dataset, as it assumes the filters are combined using the boolean `AND` operators. Downloading this without a limit gives 349125 entries. You can apply the filters on any connected entity.

Let's fetch data on a particular company, but this time also print out the generated URL just to highlight how the filters look when URL encoded, and not just when they are in object form.

```
accounts_composites_response = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "limit": 3,
        "file_type": "csv",
        "accounts.accounting_year": 2018,
        "company.org_nr_schema": 'NO',
        "company.org_nr": '812750062',
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts.accounting_year",
                "accounts_income_statement.total_operating_income",
            ]
        )
    },
    auth=auth,
)
accounts_composites = accounts_composites_response.content.decode()
print(accounts_composites_response.url)
print(accounts_composites)
```

The first print statement returns something like the following:

```
https://api.enin.ai/datasets/v1/accounts-composite?
limit=3&
file_type=csv&
accounts.accounting_year=2018&
company.org_nr_schema=NO&
company.org_nr=812750062&
keep_only_fields=
    company.name,
    company.org_nr,
    accounts.accounting_year,
    accounts_income_statement.total_operating_income
```

(I've taken the liberty to format the above URL for clarity, the real printout has all commas encoded as `%2C` and there are no white spaces.)

As you can see parameters which use the dot notation are filters.

The second print statement gives the following:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2018,Bergene Holm AS,812750062,1619605000
2018,Bergene Holm AS,812750062,1619310000
```

If you are acquainted with Norwegian accounting data, the fact that two entries were returned for one company on the same year might be confusing. Generally speaking company only returns one official accounts report to the authorities per year. The there are two here is because one is the corporate accounts and one is the company accounts for the same organization. Let's illustrate this by adding `accounts_type.accounts_type_key` to the `keep_only_fields` parameter. This gives:

```
accounts.accounting_year,accounts_type.accounts_type_key,company.name,company.org_nr,accoun ...
2018,annual_corporate_group_accounts,Bergene Holm AS,812750062,1619605000
2018,annual_company_accounts,Bergene Holm AS,812750062,1619310000
```

As we can see these are divided by `annual_corporate_group_accounts` and `annual_company_accounts`. Adding `accounts_type.accounts_type_key=annual_company_accounts` as a dot notation filter query parameter brings this down to only one entry, as expected:

```
accounts.accounting_year,accounts_type.accounts_type_key,company.name,company.org_nr,accoun ...
2018,annual_company_accounts,Bergene Holm AS,812750062,1619310000
```

Advanced data fields filtering

We have so far only handled equality filters. These are the default when using dot notation filters. In fact, the notation we have used so far is only the shorthand version of the equality filter.

The filter `accounts_type.accounts_type_key=annual_company_accounts` writen using an explicit operator is done as follows: `accounts_type.accounts_type_key=EQ:annual_company_accounts`. Both give identical results. But the latter can be changed to work with other operators, especially usefull when filtering numeric values.

The general syntax for dot notation filtering is as follows:

```
<entity>.<field>=<negation><operator>:<filter_value>
```

`<entity>` is as expected, a named collection of fields. The database equivalent is a table.

`<field>` is also as expected, a named value as part of an entity. The database equivalent is a column.

`<negation>` can either be tilde (`~`) or empty string. If used it negates the whole operation. It enables operations like “not in” or “is not null”.

`<operator>` can be one of the following:

Operator	Equivalent SQL	Description
`GTE`	`>=`	Greater than or equal.
`LTE`	`<=`	Less than or equal.
`GT`	`>`	Greater than.
`LT`	`<`	Less than.
`IN`	`in`	Checks equality on a comma separated list of values.
`IS`	`is`	Checks identity. Only works for empty string or `null` .
`ILIKE`	`ilike`	Case-insensitive partial text match.
`LIKE`	`like`	Case-sensitive partial text match.
`EQ`	`equal`	Equal.

The `ILIKE` and `LIKE` operators use `%` as a wild card of any length, and `_` as a single character wild card. You can escape these by using `\%` and `\_`

Finally, `<filter_value>` is the value you want the filtering to be done based on. Must be a supplied value, and can not reference other fields.

Let’s try this out by filtering some numeric value in the accounts data. Let’s find all accounts with more than 50 billion in revenue in 2018.

URL encoded you can write this as:

```
accounts.accounting_year=EQ:2018&accounts_income_statement.total_operating_income=GT:50000000000
```

Let’s do this in python:

```
accounts = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "file_type": "csv",
        "accounts_type.accounts_type_key": "EQ:annual_company_accounts",
        "accounts.accounting_year": "EQ:2018",
        "accounts_income_statement.total_operating_income": "GT:50000000000",
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts.accounting_year",
                "accounts_income_statement.total_operating_income",
            ]
        )
    },
    auth=auth,
).content.decode()
print(accounts)
```

This returns 4 accounts:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2018,HELSE SØR-ØST RHF,991324968,75744464000
2018,EQUINOR ENERGY AS,990888213,214951445000
2018,EQUINOR ASA,923609016,483083652000
2018,HYDRO ALUMINIUM AS,917537534,52570000000
```

If we would like another accounting year, say 2008, then then you could use the `IN` operator like this:

```
    accounts.accounting_year=IN:2008,2018
```

Using this returns 9 accounts including those previously fetched:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2018,EQUINOR ENERGY AS,990888213,214951445000
2008,EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS,914048990,67882000000
2008,ESSO NORGE AS,914803802,55560000000
2008,EQUINOR ENERGY AS,990888213,81474000000
2008,TOTAL E&P NORGE AS,927066440,57122000000
2018,HELSE SØR-ØST RHF,991324968,75744464000
2018,EQUINOR ASA,923609016,483083652000
2008,EQUINOR ASA,923609016,588422000000
2018,HYDRO ALUMINIUM AS,917537534,52570000000
```

This `IN` operator can be usefull for many sort of operations. You can for instance get the above list directly using their `org_nr` with the following query string:

```
company.org_nr=IN:
990888213,914048990,914803802,990888213,927066440,991324968,923609016,923609016,917537534
```

As a python script you can try like this:

```
accounts = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "file_type": "csv",
        "accounts_type.accounts_type_key": "EQ:annual_company_accounts",
        "accounts.accounting_year": "IN:2008,2018",
        "company.org_nr_schema": "EQ:NO",
        "company.org_nr": "IN:" + ','.join(
            [
                "914048990",
                "914803802",
                "917537534",
                "923609016",
                "927066440",
                "990888213",
                "991324968",
            ]
        ),
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts.accounting_year",
                "accounts_income_statement.total_operating_income",
            ]
        )
    },
    auth=auth,
).content.decode()
print(accounts)
```

This returns all of the previous entries, and a few more where some of the companies didn't have 50 billion NOK revenues:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2018,EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS,914048990,30239000000
2008,EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS,914048990,67882000000
2018,ESSO NORGE AS,914803802,31061000000
2008,ESSO NORGE AS,914803802,55560000000
2018,TOTAL E&P NORGE AS,927066440,35091000000
2008,TOTAL E&P NORGE AS,927066440,57122000000
2018,EQUINOR ASA,923609016,483083652000
2008,EQUINOR ASA,923609016,588422000000
2018,HELSE SØR-ØST RHF,991324968,75744464000
2008,HELSE SØR-ØST RHF,991324968,48986731000
2018,EQUINOR ENERGY AS,990888213,214951445000
2008,EQUINOR ENERGY AS,990888213,81474000000
2018,HYDRO ALUMINIUM AS,917537534,52570000000
2008,HYDRO ALUMINIUM AS,917537534,48952000000
```

This is all and good, but what happens if we want 10000 companies. In that case you will meet a technical limitation where the URL is capped to a fixed number of characters. To handle this we use company batches.

Company batches filtering

Company batches are simple collections of companies you can maintain temporarily or indefinitely. When created they are empty, but have an assigned UUID you can use to reference it, or you can supply your own "key" to identify your company batch with. The expiration date can be left unset, then a default will be used, or you can set it to whatever you like. If you don't want it to ever be deleted, you can assign it `null` which means it will last until manually deleted.

The neat thing about these batches is that the UUID or your assigned key they can be used as filters many places in the API.

Lets try to replicate the previous filter using a company batch.

First, we need to make a company batch using a HTTP `POST` request. These requests have a body we can supply data via, instead of a through the URL as a query parameter. In the `requests` library we can simply supply an object to the `post()` method as the `json` argument and it will automatically be interpreted as JSON HTTP `POST` request.

Alright, let's make our first company batch and name it `my_test_batch`, and set the `expiration_timestamp` to be one day from now. The format for the timestamp follows python's implementation of ISO8601 (https://en.wikipedia.org/wiki/ISO_8601) . This means you can specify it down to the millisecond, or just supply a date of the form `YYYY-MM-DD`.

```
expiration_timestamp = (datetime.now() + timedelta(days=1)).isoformat()
company_batch = requests.post(
    'https://api.enin.ai/analysis/v1/company-batch',
    auth=auth,
    json={
        'company_batch_key': 'my_test_batch',
        'expiration_timestamp': expiration_timestamp,
    }
).json()
print('expiration_timestamp:', expiration_timestamp)
print_obj(company_batch)
```

This first prints the supplied `expiration_timestamp`:

```
expiration_timestamp: 2020-05-15T18:42:49.668177
```

… then the server batch representation of the batch you supplied:

```
{
    "uuid": "65ed1de0-0b04-48be-abc1-d8099a0aaed9",
    "app_customer_uuid": "85e3e2f9-2114-4836-acda-6d1a30779c65",
    "expiration_timestamp": "2020-05-15T18:42:49.668177+00:00",
    "company_batch_key": "my_test_batch",
    "app_url": null
}
```

You can also find existing batches by listing all the available company batches for the customer you represent by doing a HTTP `GET` request to the company batch API endpoint:

```
company_batches = requests.get(
    'https://api.enin.ai/analysis/v1/company-batch',
    auth=auth,
).json()
print_obj(company_batches)
```

If you have only made the one batch in the above example it should list the previous batch as follows:

```
[
  {
    "uuid": "e11d6828-e3b1-4935-9dcc-4818c4ef5e4e",
    "app_customer_uuid": "85e3e2f9-2114-4836-acda-6d1a30779c65",
    "expiration_timestamp": "2020-05-15T18:42:49.668177+00:00",
    "company_batch_key": "my_test_batch",
    "app_url": null
  }
]
```

Next, we need to fill this batch with companies:

```
companies = requests.post(
    'https://api.enin.ai/analysis/v1/company-batch/{company_batch_identifier}/company'.format(
        company_batch_identifier=company_batch['uuid']
    ),
    auth=auth,
    json=[
        {'uuid': 'f0e01f1e-f977-429f-ac12-0f0418901bfe'},
    ]
).json()
print_obj(companies)
```

Here we are POSTing a list with only a single company (using its UUID) to the company batch we just created. Notice that we saved the new batch object as `company_batch` and we are using the generated company batch UUID as part of the request URL. In this case we are not using query parameters, but rather the URL itself. You can build URLs like this using string interpolation of some kind. In this case we are using python's `str.format(...)` function for this purpose.

We didn't have to use the UUIDs for POSTing here. Let's try using `company_batch_key` we supplied earlier and the organization numbers from those 50 billion NOK companies we queried earlier.

```
companies = requests.post(
    'https://api.enin.ai/datasets/v1/company-batch/my_test_batch/company',
    auth=auth,
    json=[
        {"org_nr": "914048990", "org_nr_schema": "NO"},
        {"org_nr": "914803802", "org_nr_schema": "NO"},
        {"org_nr": "917537534", "org_nr_schema": "NO"},
        {"org_nr": "923609016", "org_nr_schema": "NO"},
        {"org_nr": "927066440", "org_nr_schema": "NO"},
        {"org_nr": "990888213", "org_nr_schema": "NO"},
        {"org_nr": "991324968", "org_nr_schema": "NO"},
    ]
).json()
print_obj(companies)
```

This time we didn't need to use string interpolation, and using the organization numbers makes things just that more easy to work with. Just remember that the `org_nr_schema` is required. The output from the above request is as follows:

```
[
    {
        "name": "HELSE SØR-ØST RHF",
        "uuid": "e7564cfd-1a26-464e-ac61-b8b5c9c18f74",
        "org_nr": "991324968",
        "org_nr_schema": "NO",
        "app_url": "https://app.enin.ai/company/e7564cfd-1a26-464e-ac61-b8b5c9c18f74"
    },
    { "name": "EQUINOR ENERGY AS", ... },
    { "name": "ESSO NORGE AS", ... },
    { "name": "TOTAL E&P NORGE AS", ... },
    { "name": "HYDRO ALUMINIUM AS", ... },
    { "name": "EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS", ... },
    { "name": "EQUINOR ASA", ... }
]
```

Notice that the post only returned the companies you just POSTed. Let's return them all:

```
companies = requests.get(
    'https://api.enin.ai/datasets/v1/company-batch/my_test_batch/company',
    auth=auth,
).json()
print_obj(companies)
```

This give all the entries in the batch:

```
[
    {
        "name": "ENIN AS",
        "uuid": "f0e01f1e-f977-429f-ac12-0f0418901bfe",
        "org_nr": "917540640",
        "org_nr_schema": "NO",
        "app_url": "https://app.enin.ai/company/f0e01f1e-f977-429f-ac12-0f0418901bfe"
    },
    {
        "name": "HELSE SØR-ØST RHF",
        "uuid": "e7564cfd-1a26-464e-ac61-b8b5c9c18f74",
        "org_nr": "991324968",
        "org_nr_schema": "NO",
        "app_url": "https://app.enin.ai/company/e7564cfd-1a26-464e-ac61-b8b5c9c18f74"
    },
    { "name": "EQUINOR ENERGY AS", ... },
    { "name": "ESSO NORGE AS", ... },
    { "name": "TOTAL E&P NORGE AS", ... },
    { "name": "HYDRO ALUMINIUM AS", ... },
    { "name": "EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS", ... },
    { "name": "EQUINOR ASA", ... }
]
```

Note that the company batch API endpoints are not special like the datasets API endpoints in that the special filtering and field selection features we've been talking about until now do not apply for the the content of company batches.

In the output above, we see that the first company we inserted (with an UUID of `f0e01f1e-f977-429f-ac12-0f0418901bfe`) is still there!

It seems a little *off* compared to the other companies in that list, so lets just delete it. This can be done with either the UUID or its organization number, both represent a “company identifier” and can be used interchangeably if you see the argument named `company_identifier` in the generated API documentation. Just make sure you prefix the organization number with the organization number schema (`NO` for Brønnøysundregistrene in Norway).

```
company = requests.delete(
    'https://api.enin.ai/datasets/v1/company-batch/my_test_batch/company/NO917540640',
    auth=auth,
).json()
print_obj(company)
```

Also deletes returns to you the object just deleted:

```
{
    "uuid": "f0e01f1e-f977-429f-ac12-0f0418901bfe",
    "name": "ENIN AS",
    "org_nr": "917540640",
    "org_nr_schema": "NO",
    "app_url": "https://app.enin.ai/company/f0e01f1e-f977-429f-ac12-0f0418901bfe"
}
```

Now doing a HTTP `GET` request to `https://api.enin.ai/datasets/v1/company-batch/my_test_batch/company` returns the expected 7 organizations:

```
[
    { "name": "HELSE SØR-ØST RHF", ... },
    { "name": "EQUINOR ENERGY AS", ... },
    { "name": "ESSO NORGE AS", ... },
    { "name": "TOTAL E&P NORGE AS", ... },
    { "name": "HYDRO ALUMINIUM AS", ... },
    { "name": "EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS", ... },
    { "name": "EQUINOR ASA", ... }
]
```

Now that we know how to create and manage company batches, let’s use one to filter datasets API endpoints. To do this we simply replace the `IN` dot notation filter we used earlier with simply a `company_batch_identifier` query parameter. Notice how it is not a dot notation filter, just a regular query parameter, and that it also is postfixed with `_identifier`, this indicates that it accepts multiple forms of keys. In this case both the UUID of the company batch and the `company_batch_key` we supplied earlier. Let’s use the latter:

```
accounts = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "file_type": "csv",
        "accounts_type.accounts_type_key": "EQ:annual_company_accounts",
        "accounts.accounting_year": "IN:2008,2018",
        "company.org_nr_schema": "EQ:NO",
        "company_batch_identifier": "my_test_batch",
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts.accounting_year",
                "accounts_income_statement.total_operating_income",
            ]
        )
    },
    auth=auth,
).content.decode()
print(accounts)
```

As we hoped for this returns the same results as previously:

```
accounts.accounting_year,company.name,company.org_nr,accounts_income_statement.total_operat ...
2018,EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS,914048990,30239000000
2008,EXXONMOBIL EXPLORATION AND PRODUCTION NORWAY AS,914048990,67882000000
2018,ESSO NORGE AS,914803802,31061000000
2008,ESSO NORGE AS,914803802,55560000000
2018,TOTAL E&P NORGE AS,927066440,35091000000
2008,TOTAL E&P NORGE AS,927066440,57122000000
2018,EQUINOR ASA,923609016,483083652000
2008,EQUINOR ASA,923609016,588422000000
2018,HELSE SØR-ØST RHF,991324968,75744464000
2008,HELSE SØR-ØST RHF,991324968,48986731000
2018,EQUINOR ENERGY AS,990888213,214951445000
2008,EQUINOR ENERGY AS,990888213,81474000000
2018,HYDRO ALUMINIUM AS,917537534,52570000000
2008,HYDRO ALUMINIUM AS,917537534,48952000000
```

The only difference is that there is no upper bound to how large the company batch can be!

Finally, let's clean up the test batch:

```
company_batch = requests.delete(
    'https://api.enin.ai/analysis/v1/company-batch/my_test_batch',
    auth=auth,
).json()
print_obj(company_batch)
```

We can also do this by using the company batch UUID.

Using order by with limit to get top lists

Sometimes it can be useful to order the dataset you are requesting. This can be done with the `order_by_fields` query parameter. It requires fully qualified field names using dot notation. By default the ordering is done ascending.

Let's start with just ordering companies by name:

```
companies = requests.get(
    "https://api.enin.ai/datasets/v1/company",
    params={
        "limit": 5,
        "file_type": "csv",
        "company.org_nr_schema": 'NO',
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
            ]
        ),
        "order_by_fields": 'company.name',
    },
    auth=auth,
).content.decode()
print(companies)
```

This returns:

```
name,org_nr
0001 INVEST AS,921826885
0001 PRODUCTION AS,919072083
001 KONSULENT1 Johansen,989258265
0047 IMPORT ROGER BERSVENDSEN,990149143
0047 OSLO AS,990893330
```

As you can see it sorts (lexicographically) in ascending order. You can set the ordering explicitly by the following syntax:

```
<field_name>:<asc|desc>
```

And, you can list multiple of these using comma separators (`,`).

To illustrate multiple orderings, let's order on name and revenue. And because there are very few companies with the same name, let's start from `MOTORSENTER AS` which we know there are multiple of.

```
accounts = requests.get(
    "https://api.enin.ai/datasets/v1/accounts-composite",
    params={
        "file_type": "csv",
        "company.name": "GTE:MOTORSENTER AS",
        "accounts_type.accounts_type_key": "EQ:annual_company_accounts",
        "accounts.accounting_year": "EQ:2018",
        "keep_only_fields": ','.join(
            [
                "company.name",
                "company.org_nr",
                "accounts_income_statement.total_operating_income",
            ]
        ),
        "order_by_fields": ','.join(
            [
                "company.name:asc",
                "accounts_income_statement.total_operating_income:desc",
            ]
        ),
    },
    auth=auth,
).content.decode()
print(accounts)
```

This prints:

```
company.name,company.org_nr,accounts_income_statement.total_operating_income
MOTORSENTERET AS,966377607,41998000
MOTORSENTERET AS,971233974,15131000
MOTORSENTERET AS,976604628,5654000
MOTORSENTERET EIENDOM AS,987659327,410000
MOTORSENTERET HEIDAL AS,999301061,29965000
MOTORSENTERET-LIMO NORGE AS,911916797,6497000
MOTORSENTERET SELBU OG TYDAL AS,916983069,12266000
MOTORSERVICE ALTA AS,998539749,4061000
MOTOR SERVICE AS,817570852,7229000
MOTOR-SERVICE AS,914424011,819000
MOTOR-SERVICE BILSENTER AS,977199832,35394000
MOTORSERVICE DA,929710339,5176000
```

As you can see the names are sorted ascending, and because there are three “MOTORSENTERET AS” with their own organization numbers, they are sorted descending by `accounts_income_statement.total_operating_income`, same applies to `MOTOR SERVICE AS` further down the list.

Using limit and offset to paginate results

In general, we recommend fetching the full dataset you want in one go. Because the data is being streamed to you, you should be able to read the stream line-by-line, and that way be able to handle any memory issues. However, if for whatever reason this isn’t suitable for your use case it is possible to paginate results using the `limit` query parameter in conjunction with the `offset` query parameter. This will make it possible to fetch the whole dataset across multiple requests.

There is one inherent risk when using pagination: There is no guarantee that the dataset will remain unchanged while paginating through the results. However, if no data manipulation operations, i.e., inserts, updates, or deletes, are done while you paginate, the results should be consistent.

Furthermore, if you apply an `offset` as query parameter, the dataset must be sorted (https://www.postgresql.org/docs/current/queries-limit.html) in order to give consistent results. This will slow down your request even if you do not specify a sort order, (in which case sorting is done on the primary key of the dataset your requesting.)

> **Note** : Not all Datasets API endpoints support the `limit` and `offset` query parameter, or may support it in an unexpected way. As of May 2020, this limitation applies to the `company-flag` and `company-flag-composite` endpoints of the Datasets API. These two endpoints do not support the `offset` query parameter at all, and supports the `limit` query parameter only partially. In fact, the `limit` parameter is not even documented in the automatic API documentation, as it is not intended for production systems, only for testing purposes. It will limit the number returned entries, but may return up to a fixed multiple of requested limit due to technical limitations.

Alright, enough caveats. Let's paginate over all companies with an organization number starting with `9175406` . We can do this using the `LIKE` dot notation filter operator.

```
companies = None
offset = 0
limit = 6
while companies is None or companies:
    print("From entry", offset, "to entry", offset + limit)
    response = requests.get(
        "https://api.enin.ai/datasets/v1/company-composite",
        params={
            "file_type": "jsonl",
            "company.org_nr_schema": "EQ:NO",
            "company.org_nr": "LIKE:9175406%",
            "keep_only_fields": "company.uuid,company.name,company.org_nr",
            "order_by_fields": "company.name",
            'limit': limit,
            'offset': offset,
        },
        auth=auth,
    )
    companies = response.content.decode()
    print(companies)
    if response.status_code != 200:
        break

    offset += limit
```

Notice how we increment the offset by the limit size. This ensures you are getting disjoint sets of data. Also, notice that we break out of the loop if we get anything else than a 200 HTTP status code, i.e., there is probably a problem and we might have entered an infinite loop. Finally, notice how we sort by `company.name` , this isn't actually enough to ensure consistent ordering, so under-the-hood another ordering is added on `company.uuid` as it is the primary key of the subject entity, `company` .

The pagination loop runs three times, two times with data, and one where the result is empty:

```
From entry 0 to entry 6
{"company": {"name": "ANNE BIRGITTE NESSE", "uuid": "1ca93ab6-8507-488e-b903-888a4cf22b12", ...
{"company": {"name": "ÅRNES KORNSILO & MØLLE BA", "uuid": "10b32f80-c30f-443a-bc33-528b112e ...
{"company": {"name": "ENIN AS", "uuid": "f0e01f1e-f977-429f-ac12-0f0418901bfe", "org_nr": " ...
{"company": {"name": "HANNA BERGH", "uuid": "fcbde1ab-db34-4757-ac66-85c2ccd7db2b", "org_nr ...
{"company": {"name": "LINDA DAHL INVEST AS", "uuid": "7f71497e-e02c-459c-ae09-1463178bdfca" ...
{"company": {"name": "LOOMIAN ENTERTAINMENT BERG", "uuid": "5dc21c97-25ac-46a8-91d5-a9d9acc ...

From entry 6 to entry 12
{"company": {"name": "NP CONSULT SP.Z O.O", "uuid": "d3141183-eeff-4dbe-84af-25cc15876f6e", ...
{"company": {"name": "PEARL OF THE ORIENT ANABELLE DELOS SANTOS TVERÅ", "uuid": "defdb0b1-f ...
{"company": {"name": "TØRRES AVIATION", "uuid": "5faf1564-43d6-46ba-95d0-9989fa3c3df1", "or ...

From entry 12 to entry 18
```

When the data is empty it breaks out of the loop, and we are done.

Datasets API Metadata

For now we suggest requesting the endpoints using a limit of 1 to see what fields are available. We will expand with more documentation later, which will include dynamically updated metadata.

Business Relevant Examples

Accounts Examples

TODO : *Add even more business relevant examples.*

Company Examples

TODO : *Add even more business relevant examples.*

Company Flag Examples

TODO : *Add even more business relevant examples.*

Company Event Examples

TODO : *Add even more business relevant examples.*