Model Data Explorer Prototype Development

Release 0.0.1dev0

Philipp S. Sommer **Linda Baldewein** **Hatef Takyar**
Housam Dibeh **Rehan Chaudhary** **Marie Ryan**
Max Böcke **Andrea Pörsch** **Andreas Lehmann**
Annika Örtel **Beate Geyer** **Carsten Lemmen**
Elke Meyer **Emanuel Söding** **Julian Quinting**
Klaus Getzlaff **Lars Buntemeyer**
Ludwig Lierhammer **Marcus Lange**
Matthias Gröger **Nikolaus Groll** **Sascha Hokamp**
Tilman Dinter

Mar 02, 2022

RESOURCES

This document describes the first phase of the development of the model data explorer. We try to be as inclusive as possible so please let us know if you encounter any issues or have any questions on the process.

Within the *prototype* development, we discuss the workflows and develop a prototype to get a feeling how the model data explorer will work.

The development team comes up with suggested workflows and discusses this with the scientific collaborators.

CHAPTER

ONE

RESOURCES

We have three important resources for during this phase of development:

1. The prototype, see *Using the prototype*.
2. The Epics described in this document, see *Epics in the prototype development*.
3. The issues in the gitlab repository, see *Usage and navigation on Gitlab*.

CHAPTER

TWO

CONTACT

Please do not hesitate to contact the development team directly if you have any questions or points for discussion. We are reacheable via

- comments in the prototype
- the Model Data Explorer Mattermost channel,
- the mde-dev mailing list
- or via email at hcdc_support@hereon.de.

2.1 Using the prototype

You can access the prototype at https://www.figma.com/proto/FoCucLiWsUTfEQ6Fnob19s/Model-Data-Explorer-Portal. This prototype is interactive but it's just a preview on how the model data explorer might look and feel. Everything you see that is not really automated.

Here are some hints on the usage of this prototype:

- Click anywhere on the prototype and you'll see the clickable elements highlighted in blue. We recomment to use these elements for navigation rather than the slide show buttons in the bottom center.
- We are happy to receive your comments in the prototype. For this, use the little speech bubble in the upper left corner. Please note that you have to register to leave comments (it's free).
- The prototype is not made for mobile devices. But the model data explorer will be designed mobile responsive.
- Unless you are experienced with Figma, we recommend that you stay in the prototype mode and to not click on *Open in editor* (which is what you could do from the dropdown menu in the upper center).

2.2 Epics in the prototype development

We use SCRUM project management for the model data explorer. As such, we divided the tasks into multiple so-called epics. Each epic corresponds to one topic that is addressed by the model data explorer.

Each epic contains a list of issues that we want to solve (in SCRUM, you'd call them *User stories*). You'll find all these issues and their discussions in the gitlab repository and you're kindly invited to contribute your thoughts (see *Usage and navigation on Gitlab*).

Each issue has a label that describes the epic. You can use this label to filter for the issues that corresponds to a specific epic. Or you can use the links on this website.

2.2.1 Metadata

Browse issues

Within this EPIC, we want to clarify the work flows for the handling of metadata within the Model Data Explorer framework.

Central questions are

1. What is the necessary metadata for model runs?
2. How to populate the metadata of a model run?
3. How to search and filter based on the metadata?
4. What are the possible relations and what do they imply between
    1. groups
    2. users and groups?
    3. model runs?
    4. groups and model runs?
    5. users and model runs?

The outcome of this epic will be a document Metadata handling in the Model Data Explorer that describes these relations and serves as a basis for the final documentation and SOPs within the the Model Data Explorer.

2.2.1.1 Terminology

Relation

Relations consist of three parts. Let's look at the following example:

> *coastDat-3_COSMO-CLM_ERAi has been forced with Era Interim*.

The three parts that describe this relation are:

1. The left object (here *coastDat-3_COSMO-CLM-ERAi*)
2. the right object (here *Era Interim*)
3. The description of the relation (here *has been forced with*)

Each relation has an inverse, in this case the inverse is

Era Interim forces coastDat-3_COSMO-CLM-ERAi

Permissions

Sometimes, a *relation* between two items can also be interpreted as a *permission*. If *CoastDat is created by Hereon*, the dataset owner of *Coastdat* implicitly grants the *Hereon* group the permission to list this dataset as one of theirs.

Permissions apply between *datagroups <relation-datagroup> datasets and datagroups*, between *datasets and users*, and between *users and datagroups*.

Roles

Roles within the model data explorer are purely considered from the metadata perspective (see *Authors and Contact Persons*) and describes how an *author* or *datagroup* has been involved in the creation of a dataset. When interpreting metadata of a dataset, e.g. by reading the ISO INSPIRE metadata (see *Metadata of datasets*), the model data might suggest to grant *permissions* to users based on the role that the user had in the dataset.

Todo: Link to interpretation of roles and permissions

Author

Todo: Add Django graphs for author model

An author in the MDE framework is an individual that participated in the generation of a dataset. Uniquely identified by his or her OrcID. Each author will also get a unique handle in the Model Data Explorer.

An author can have multiple aliases (e.g. *Philipp Sommer*, *Philipp S. Sommer*, etc.)

Note: Not all authors will have OrcIDs. To decrease the amount of duplicates, we can ask them, and we can ask their co-authors that have an *user* account (as researchgate does). This is something that still needs to be clarified.

User

Todo: Add Django graphs for user model

A user is a person with a login for the Model Data Explorer. A user can be uniquely identified from the email address. Every user is related to one specific *author*.

Dataset

Todo: Add Django graphs for dataset model

A **dataset** (formerly known as *model run*) is a collection of variables with a temporal, spatial and optional vertical dimension. It can be the output of a single run of your model, or other raster data, e.g. derived from satellite measurements. A dataset might comprise multiple parameters. All data that is represented by one single *dataset* has been generated with one specific methodology and one specific set of input parameters.

In the language of CERA and CMIP6, one *dataset* in the Model Data Explorer corresponds to one *experiment*, e.g. coastDat-3_COSMO-CLM_ERAi in CERA or rcp45 in CMIP6.

Each dataset in the model data explorer will have a unique persistent identifier (PID) and a unique page where it's possibilities (visualization on the map, statistical analysis, download, metadata are accessible).

Examples

One *dataset* in the model data explorer might correspond to - the output that you generated with one runscript of your model - an ensemble mean of multiple models and/or experiments - multiple realizations (ensemble members) produced with variations of the same scenario (in CMIP6: all data for a given model (e.g. *MPI-ESM-LR*) in a given experiment (e.g. *rcp85*)) - a single ensemble member with multiple variables for one CMIP6 scenario of one model with a specific forcing - data from a specific satellite derived with a specific method

Examples that would **not** be a *dataset* are: - a subset data, e.g. 1 out of 99 years of a model run (unless you want to throw away the other 98 years) - Multiple experiments (e.g. a combination of *rcp26*, *rcp45*, *rcp60* and *rcp85*) of one model participating in CMIP6 (unless you only refer to ensemble statistics) - Multiple runs of the same model (unless they are follow-ups of each other, e.g. via restart files) - Data from multiple models (unless you only refer to the combined ensemble statistics)

Data Group

Todo: Add Django graphs for datagroup model

A **data group** is a group of *datasets* (see above) that are managed by a certain set of users.

Each *data group* will have an own webpage and a unique handle where all the datasets are accessible and the metadata of the group is shown.

A *data group* is owned by a certain set of users (see below) and can have regular members.

Datasets can be associated with groups and provide edit and access rights.

Examples

- a research center, e.g. *Hereon*
- a department or institute, e.g. the Institute of Coastal Systems - Analysis and Modeling
- a unit, e.g. Regional Land and Atmosphere Modeling
- a project, e.g. CMIP6, MOSES, or MuSSEL

Tags

A standard way to describe a dataset or a data group would be to assign so-called *tags* or *keywords*. We follow this approach in the model data explorer. A tag in the model data explorer might have different aliases (e.g. *coastDat3* and *coastDat-3*), and we implement them as *hierarchical tags* (e.g. *coastDat* > *coastDat-3*).

Examples

- *coastDat-3_COSMO-CLM_ERAi* has the *coastDat3* tag
- the *CMIP6* has a *rcp45* child tag

2.2.1.2 Relations between Users, Groups and datasets

Related Issues	Document Status
#17	In Progress
#21	In Progress

Section authors: *Philipp S. Sommer* , *Linda Baldewein* , *Hatef Takyar, Andrea Pörsch, Emanuel Söding* , *Housam Dibeh, Carsten Lemmen* , *Elke Meyer, Marcus Lange, Sascha Hokamp, Ute Daewel, Julian Quinting, Annika Oertel, Andreas Lehmann, Klaus Getzlaff*

Datasets, groups, users and authors are related to each other in the model data explorer framework. The kind of relations between the objects

1. describes how the relation is displayed on the frontend
2. has implications about who can edit what
3. categorizes the content and makes it findeable

This is necessary to generate pages where all datasets of a data group are listed, or all datasets that an author participated in. With this strategy we honor active users and groups that make their data available to the public, and it helps to browse the available resources.

In the following, we describe the relations that are possible in the model data explorer, namely between

- users and authors
- authors and datasets
- users and data groups
- data groups
- datasets
- datasets and data groups

See the *Terminology* section if you want to know more about the different terms.

Relation-based permission system

From a database perspective, a relation is a many-to-many relationship with a certain access right. Consider the following model:

```
class DatasetUserRelation:

    description: string  # Human-readable description of the relation, optional.
    left: Dataset
    right: User
    permissions: List[Permission]
    relation_type: List  # e.g. "originated by", "distributed by", ...
```

Here we have a relation between a dataset and a user. This relation is described by a certain `relation_type` coming from a controlled vocabulary (see the roles for *Authors and Contact Persons* for instance) and involves two parties: a left party (the dataset) and a right party (the user). Our aim within the model data explorer is to get this information (`relation_type`, `left` and `right` party) from the metadata in the original dataset, e.g. the netCDF-Header or INSPIRE ISO (see *Metadata of datasets*).

Such a relation may, however, also imply some priviliges. If a `user` is the distributor of a dataset, he or she should also have the possiblity to edit the dataset. Therefore the creator of a dataset can equip a relation with *permissions*, e.g. the permission to edit a dataset, or to view a dataset, or to list a dataset on the users personal page in the model data explorer.

These *permissions* must be confirmed by both parties, the granting party (the creator of the *left* dataset) and the granted party (the *right* user in the relation above).

As such, our Database looks like this:

Graph

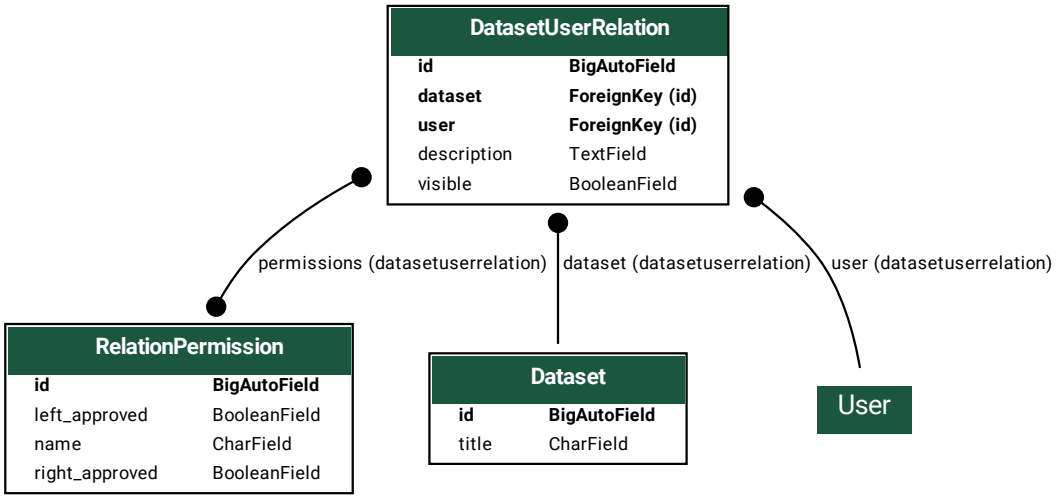


Fig. 1: Representation of the Dataset - User relation.

Django source

```
from django.contrib.auth.models import User


class RelationPermission(models.Model):

    name = models.CharField(max_length=20)
    left_approved = models.BooleanField(default=False)
```

(continues on next page)

(continued from previous page)

```
    right_approved = models.BooleanField(default=False)


class DatasetUserRelation(models.Model):

    description = models.TextField(
        help_text="Human-readable description of the relation."
    )
    dataset = models.ForeignKey("Dataset", on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    permissions = models.ManyToManyField(RelationPermission)
    visible = models.BooleanField(default=True)


class Dataset(models.Model):

    title = models.CharField(max_length=50)
    users = models.ManyToManyField(
        User, through=DatasetUserRelation
    )
```

This methods allows us to implement a fine-grained permission system where the dataset owner can tightly control who can do what with his or her data. But to simplify this, we also define *roles* within the model-data explorer that are preconfigured permission sets. If a user has an *Owner* role for a dataset, for instance, the corresponding `DatasetUserRelation` has the `can_edit`, `can_view`, `can_delete` and `can_list` permission (or even more fine-grained, see *Users and datasets*).

Users and Authors

A user actively with a login for the the Model Data Explorer, is always associated with one specific author.

Not every author has a user, but every author is related to one or more datasets and is listed on the detail page of the dataset.

Authors and datasets

An author can be a contributor to a dataset, meaning that the author participated in the generation of a dataset.

This has no further implications, but all authors are listed when displaying the metadata of a dataset.

Users and datasets

A user can be linked to a dataset and can be equipped with a combination of the following roles:

Owner The user has full control over the dataset. He or she can

- delete the dataset
- change the metadata
- change the groups
- register new services (for visualization, analysis, etc.)

- remove services

Data Manager The user can register and remove services

Editor The user can change the metadata of a dataset. He or she can

- change the metadata
- change the groups

Viewer The user can see the metadata and services of the dataset.

Authors and data groups

There are no planned relations between authors and data groups. Mainly because there is no way to validate whether an author is really a member of a data group, as the author does not have the possibility to verify unless he or she has an associated user account.

Users and data groups

A data group is a collection of users and datasets. Each related user of a data group can have multiple of the following roles:

Owner The user has full control over the data group and the associated contents. He or she can

- delete the group
- change the metadata of the group
- change the group-to-group relations
- remove or add linked datasets
- add new users to the group
- approve roles of users
- remove (disable) users from the group

User manager A user manager can control who is in the data group and approve roles

Data manager The user has *Data manager* priviliges on all datasets that are owned by the group (see *Users and datasets* above, and *Datasets and data groups* below)

Data editor The user has *Editor* priviliges on all datasets that are owned by the group (see *Users and datasets* above, and *Datasets and data groups* below)

Editor The user can change the metadata, e.g.

- title and group description
- group-to-group relations

Member The user can view all datasets that the group has view permissions on. Members of a group can also grant the following permissions to the group:

1. edit permissions: all datasets of the user can be edited by the data managers/editors of the group
2. view permissions: all datasets of the user can be viewed by members of this group

These permissions can also be granted on a per-dataset basis (see *Datasets and data groups* below).

He or she can furthermore specify if all datasets of the user are automatically marked as products of the group (see below).

Data groups

Description

Data groups can also be related to each other. This way, we can visualize the network of a group, and we can make sure that the content is maintained, even if a group ends.

Relations between two groups must be confirmed by the owners of both groups (i.e. users with *can_edit* permission).

Two groups can be related in the following ways:

Parental data groups

Group A is parent of Group B

Members of a data group might form a subgroup of a larger group to visualize the content of this group on a dedicated side.

Implications

- All datasets related to the child group are also related to the parent group
- Owners of the parent groups have the same rights as the owners of the child group
- Members of the child group have view permissions on datasets of the parent group
- Members of the parent group do **not** automatically have view permissions on the items of the child group (unless explicitly configured)

Example

Helmholtz-Zentrum Hereon is parent of the Institute of Coastal Systems - Analysis and Modeling

Collaborating data groups

Group A is collaborating with Group B

A permanent collaboration between is a visual implication and acknowledges the partner group for their contribution.

Implications

No implications on permissions, just to display the network on the webpage.

Adds datasets of Group B as *dataset through collaboration* with Group A (see below)

Example

- MuSSeL is collaborating with Hereon
- Hereon is collaborating with AWI

Graph

Graph

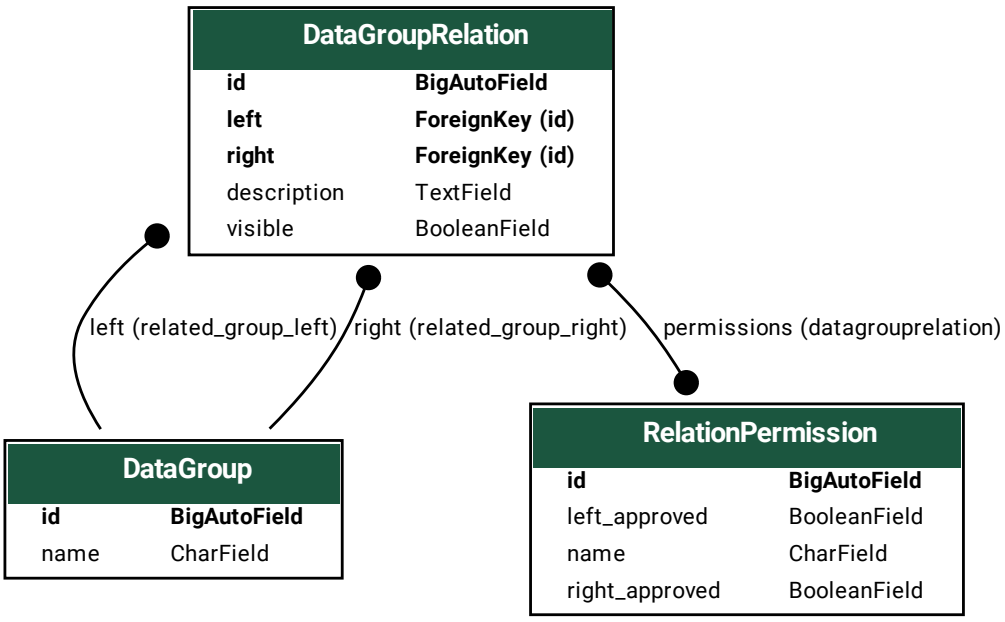


Fig. 2: Relations between data groups

Django source

```
class DataGroup(models.Model):

    name = models.CharField(max_length=100)
    related_groups = models.ManyToManyField(
        "self",
        through="DataGroupRelation",
        through_fields=("left", "right"),
    )
```

(continues on next page)

(continued from previous page)

```
class RelationPermission(models.Model):

    name = models.CharField(max_length=20)
    left_approved = models.BooleanField(default=False)
    right_approved = models.BooleanField(default=False)


class DataGroupRelation(models.Model):

    description = models.TextField(
        help_text="Human-readable description of the relation."
    )
    left = models.ForeignKey(
        DataGroup,
        on_delete=models.CASCADE,
        related_name="related_group_left",
    )
    right = models.ForeignKey(
        DataGroup,
        on_delete=models.CASCADE,
        related_name="related_group_right",
    )
    permissions = models.ManyToManyField(RelationPermission)
    visible = models.BooleanField(default=True)
```

Datasets and data groups

The link between datasets and data groups is important in multiple aspects:

1. datasets should still be editable, even if the creator of the data left the institute/project/science
2. data groups should be able to list the datasets that their members have made public.

To view and manage datasets of a specific data group, one can set the same roles as for *Users and datasets*. Depending on the role in the data group, the users will then get the appropriate rights (see *Users and data groups*).

Datasets

Datasets can also be related to each other. These relations would not have any implications on the permissions who can see or edit datasets. They are of informative nature only, e.g. to create a machine-readable representation what dataset has been forced etc. by what other dataset.

The relations that we can think of are

- Dataset A is forced by Dataset B (i.e. dataset A uses dataset B as (local) boundary conditions)
- Dataset A supplements Dataset B
- Dataset A references Dataset B
- Dataset A is new version of Dataset B **what does this mean?**
- Dataset A continues Dataset B

- Dataset A has Dataset B as its initial conditions
- Dataset A requires Dataset B
- Dataset A replaces Dataset B

Datasets should also have a status, e.g. inactive, active, deprecated.

Metadata through Tags

When using metadata tags (see *Tags*), one can add another relation, namely *Dataset A is described by tag B*. For instance *MPI-ESM-LR-rcp45* is described by the *rcp45* tag. And since *rcp45* is a child tag of *CMIP*, it can be automatically found under the *CMIP* tag.

This gives us a possibility to generate a *controlled vocabulary* for metadata with unique handles per metadata items.

2.2.1.3 Metadata of datasets

Related Issues	Document Status
#18	In Progress

Section authors: *Philipp S. Sommer* , *Marie Ryan, Linda Baldewein* , *Hatef Takyar, Andrea Pörsch, Beate Geyer* , *Lars Buntemeyer* , *Emanuel Söding* , *Nikolaus Groll, Ludwid Lierhammer, Klaus Getzlaff* , *Tilman Dinter*

A central aspect in the model data explorer is the metadata of the datasets. We do not want to mimic a metadata portal such as geonetwork, but nevertheless, we need information to

1. let the user know what he or she is looking at
2. link datasets to groups
3. link datasets to people

Each author and group has a dedicated site where all the related datasets are listed. So we need to find a way to uniquely identify authors and associate the datasets with them.

We will implement a manual way where you can select the authors and edit the metadata through a web interface, but it should be possible to automatically interprete metadata standards.

In the following sections, we will describe what metadata we implement in the, and how.

Note: This document only covers global metadata of a dataset. Variable related metadata (units, standard name, etc.) shall be handled in a different document.

Todo: Document variable related metadata

Required metadata

Datasets in the model data explorer must define the following metadata attributes:

title A one-line description of the dataset

Optional metadata

Optional but recommended metadata attributes are:

contacts A list of authors that have some role related to the dataset. They participated in the generation, are responsible for providing the data, etc.

institutions The institutions that are responsible for the dataset

projects related projects that provided funding for the generation of the dataset

bbox The bounding box of the geographic region of the data

abstract A short description of the dataset

data_relations datacite relation types (see *Relations between Users, Groups and datasets* and https://support.datacite.org/docs/relationtype_for_citation)

temporal_extent The temporal window that is covered by the dataset

---

Todo: document geographic and temporal resolution as well?

Todo: Add descriptive spatial extent (such as global, continental, etc.)

Todo: add creation, publication and revision date

Interpretation of standards

The items mentioned in the previous sections are encoded in the metadata standards that we support, namely the netCDF header and the INSPIRE ISO- Standard. Our aim is to develop readers for each standard that transform the corresponding conventions into the metadata scheme of the model data explorer (see next section, *Implementation details*).

The exact database structure that allows this interpretation is however part of a different user story, namely #20.

CF-Conventions

For netCDF Headers (and NcML, a special markup language used by THREDDS) we want to develop guidelines based on the Binding Regulations for Storing Data as netCDF Files. For this purpose, we will transform the guidelines into a web-based format and enhance it with templates to make them easier to apply.

The guidelines are based on the CF-Conventions and extend by further attributes that are mainly motivated by the Conversion methodology to INSPIRE developed at the Geomar.

Todo: The `UnidataDD2MI.xsl` methodology needs to be elaborated further.

INSPIRE ISO

ISO-conform XML files will be read using the owslib python library. We will orient the format on the *UnidataDD2MI.xsl* file that has been developed by Franziska Weng (Geomar) and Andrea Pörsch (GFZ) (currently still work in progress).

Implementation

Note:

Graph

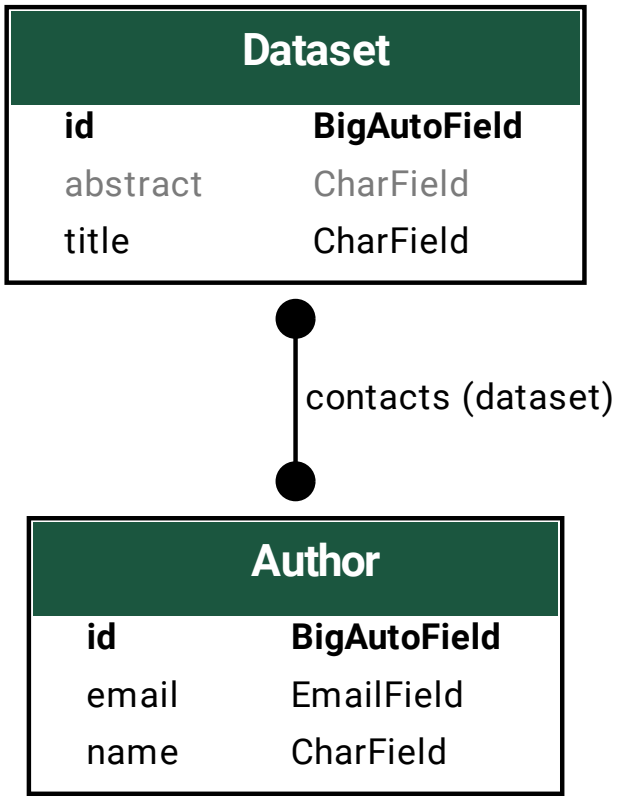


Fig. 3: Object attribute vs. object relations

Django source

```
class Author(models.Model):

    name = models.CharField(max_length=100)
    email = models.EmailField(max_length=100)

class Dataset(models.Model):

    title = models.CharField(max_length=50)
```

(continues on next page)

(continued from previous page)

```
    abstract = models.CharField(max_length=50, null=True, blank=True)
    contacts = models.ManyToManyField(Author)
```

Metadata items described above are represented in the model data explorer as properties of Django objects that in turn translates into connections and attributes in a relational database. But for this document we will keep it simple and distinguish two metadata types: attributes and relations.

Attributes are simple string properties of a dataset. A *title* for instance. Relations describe how the dataset is connected to other items in the database. A dataset won't have an *authors* string property, for instance, but it will define a connection to author objects, where one author holds a first name and last name attribute (for instance).

An example is shown in the graph on the right, *Object attribute vs. object relations*.

Attributes

A *Dataset* defines three simple attributes, *title*, *abstract* and bounding box (*bbox*), see the graph about *Attributes of a dataset*.

Graph

Django source

```
class Dataset(models.Model):

    title = models.CharField(max_length=50)
    abstract = models.TextField(max_length=10000, null=True, blank=True)
    bbox = models.JSONField(null=True, blank=True)
    start = models.DateTimeField(null=True, blank=True)
    end = models.DateTimeField(null=True, blank=True)

    start_s = models.CharField(max_length=50, null=True, blank=True)
    end_s = models.CharField(max_length=50, null=True, blank=True)
```

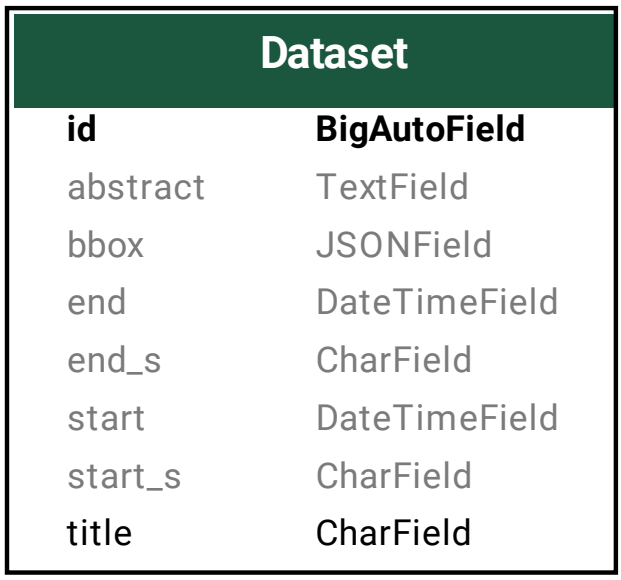


Fig. 4: Attributes of a dataset

Title

The *title* is a short human-readable description as string of the dataset and should describe the purpose of the data in one sentence.

Interpretation of the title

CF-Conventions

The CF-Conventions define a `title` netCDF attribute that will be used

INSPIRE

We are using the <`gmd:title`> tag of the `CI_Citation` element.

Abstract

The *abstract* is a longer human-readable description of the dataset that describes the content, purpose and methodology in a bit more details.

Interpretation of the abstract

CF-Conventions

We wiill look for global *summary* or *abstract* attribute.

INSPIRE

We are using the <`gmd:abstract`> tag.

Bounding box

The *bbox* is a JSONField (or optionally we can also make it a georeferenced polygon) that defines the region where this dataset can be applied.

Interpretation of the bounding Box

CF-Conventions

We wiill look for the global *geospatial_lon_min*, *geospatial_lat_min*, *geospatial_lon_max* and *geospatial_lat_max* attributes, as well as a *Bbox* attribute.

INSPIRE

We are using the `EX_GeographicBoundingBox` element in the `<gmd:geographicElement>` tag, namely `westBoundLongitude`, `eastBoundLongitude`, `southBoundLatitude` and `northBoundLatitude`

Temporal extent

The temporal extent is a `DatetimeField` that defines the start and end of a time window. We will expect two ISO-FORMATTED TIMESTAMPS (e.g. 2007-03-13T07:35:10Z) here, one for the start and one for the end of the coverage.

This might not always be possible, as python does not support paleo dates. So we will also add a attributes `start_s` and `end_s` that accept plain text fields.

Interpretation of the temporal extent

CF-Conventions

We wiill look for the global *StartTime*, *StopTime*, *time_coverage_start* and *time_coverage_end* attributes.

INSPIRE

We are using the `EX_TemporalExtent` element in the `<gmd:temporalElement>` tag, namely `beginPosition`, `endPosition`

Authors and Contact Persons

Todo: Add OrcID

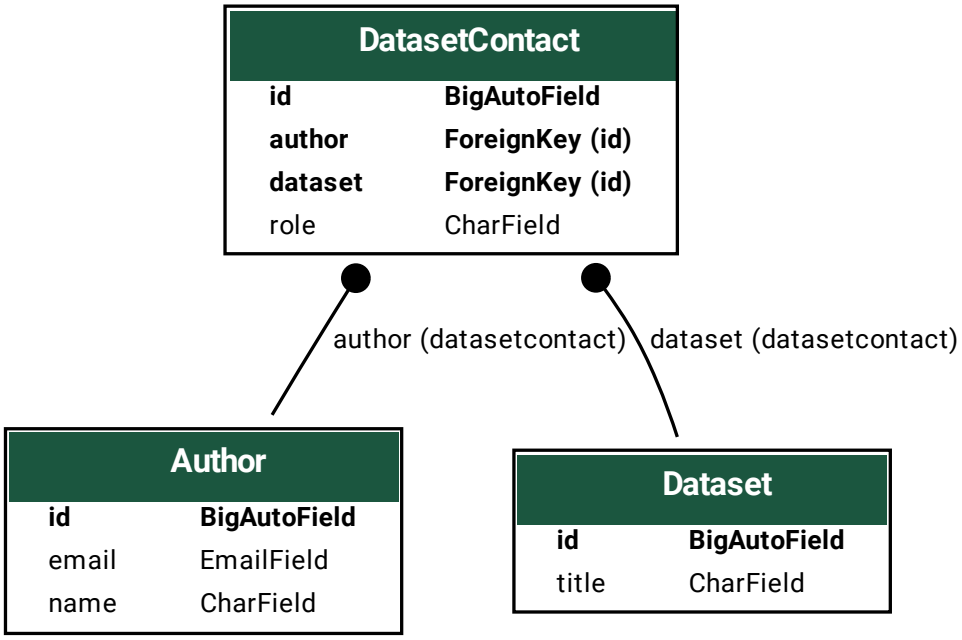
Description

Authors can have a dedicated role related when being related to a dataset. This roles describe how the authors have been involved in the generation of the dataset (motivated by the available roles for the `CI_RoleCode` tag in INSPIRE, see the *Roles* tab).

In the metadata display on the frontend, we will then group the contributors based on their role such that is clearly visible who is the responsible contact person.

Graph

Graph



Django source

```
class Author(models.Model):

    name = models.CharField(max_length=100)
    email = models.EmailField(max_length=100)

class DatasetContact(models.Model):

    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    dataset = models.ForeignKey("Dataset", on_delete=models.CASCADE)
    role = models.CharField(max_length=30)

class Dataset(models.Model):

    title = models.CharField(max_length=50)
    contacts = models.ManyToManyField(Author, through=DatasetContact)
```

Roles

Roles are taken from the INSPIRE ISO standard, https://inspire.ec.europa.eu/metadata-codelist/ResponsiblePartyRole. The *role* of an *Author* in a *Dataset* can be one of the following:

Code	Description
`resourceProvider`	Party that supplies the resource.
`custodian`	Party that accepts accountability and responsibility for the data and ensures appropriate care and maintenance of the resource.
`owner`	Party that owns the resource.
`user`	Party who uses the resource.
`distributor`	Party who distributes the resource.
`originator`	Party who created the resource
`pointOfContact`	Party who can be contacted for acquiring knowledge about or acquisition of the resource.
`principalInvestigator`	Key party responsible for gathering information and conducting research.
`processor`	Party who has processed the data in a manner such that the resource has been modified.
`publisher`	Party who published the resource.
`author`	Party who authored the resource.

Interpretation of Authors and Contact Persons

CF-Conventions

Although the CF-Conventions define an *originator* attribute, the information is rather limited. Therefore we aim to follow the suggestions by the *UnidataDD2MI.xsl* file of Franziska Weng (see *INSPIRE ISO*), and introduce further attributes such as *creator_email*, *originator_email*, *contact_email*, *pi_email*, *contributor_role*, etc.

INSPIRE

The implementation is pretty straight-forward and will be taken from the *CI_ResponsibleParty* tags.

Projects

Description

Projects are data groups within the Model Data Explorer Framework (see *Data Group*). As such, a *project* is also a relation between two objects in the database (see the *Graph* tab).

This relation can also be equipped with permissions, namely *can_edit*, *can_view* and *can_list* (see *Datasets and data groups*). These permissions need to be approved by both, the data group (project) owner and the dataset.

A relation can also be made visible or invisible, which will determine whether the group is listed explicitly on the detail page of the dataset or not.

Note: A dataset can also be related to other types of data groups, such as *institutions* and this will be using the same methodology as this. As such, we will distinguish projects from platforms, etc. based on the `DS_InitiativeTypeCode` identifier (see *Data Group* and #20).

Graph

Graph

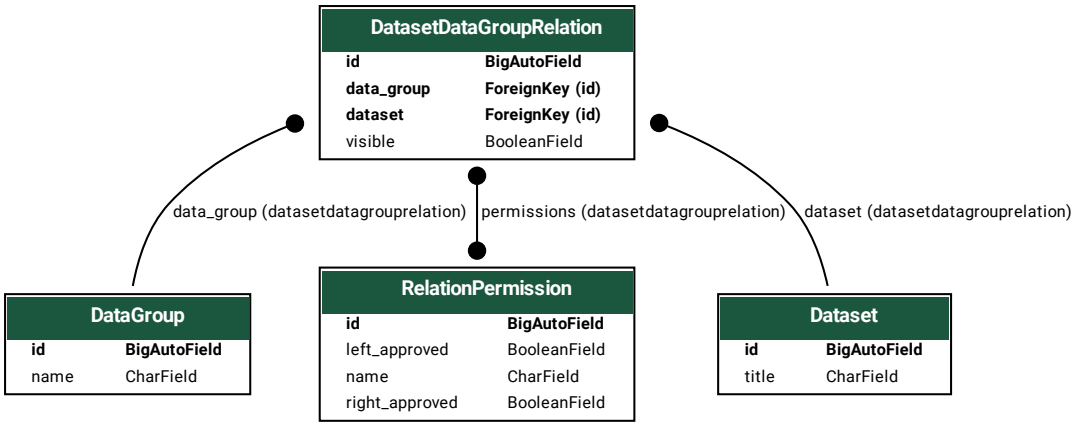


Fig. 5: Representation of the Dataset - Project relation.

Django source

```
class DataGroup(models.Model):
    name = models.CharField(max_length=100)


class RelationPermission(models.Model):

    name = models.CharField(max_length=20)
    left_approved = models.BooleanField(default=False)
    right_approved = models.BooleanField(default=False)


class DatasetDataGroupRelation(models.Model):

    data_group = models.ForeignKey(DataGroup, on_delete=models.CASCADE)
    dataset = models.ForeignKey("Dataset", on_delete=models.CASCADE)
    permissions = models.ManyToManyField(RelationPermission)
    visible = models.BooleanField(default=True)


class Dataset(models.Model):

    title = models.CharField(max_length=50)
    data_groups = models.ManyToManyField(
```

(continues on next page)

(continued from previous page)

```
        DataGroup, through=DatasetDataGroupRelation
    )
```

Interpretation of Projects

CF-Conventions

netCDF files can define a *project*, *program*, *projects* or *project_name* attribute. We will then search for matching names in the data groups that define a `DS_InitiativeTypeCode` kind of project and suggest them to the data submitter. This will also be documented in the netCDF guidelines (see *CF-Conventions*).

INSPIRE

We will look for `MD_AggregateInformation` entries that define a `DS_AssociationTypeCode` of `largerWorkCitation` and match the `MD_Identifier` against the available data groups.

Institutions

Todo: Add ROR ID

Institutions are handled the same internally as *projects* as both are represented as *data groups* in the model data explorer. Just the interpretation of the metadata standards differ.

Interpretation of Institutions

CF-Conventions

netCDF files can define an *institution* or *creator_institution* attribute, together with a corresponding *institution_references* attribute. They will then be matched against available names of institutions in the database to make suggestions to the data submitter.

INSPIRE

Institutions will be identified from the `organisationName` in a `CI_ResponsibleParty` (see *Authors and Contact Persons* above).

Other relations

Description

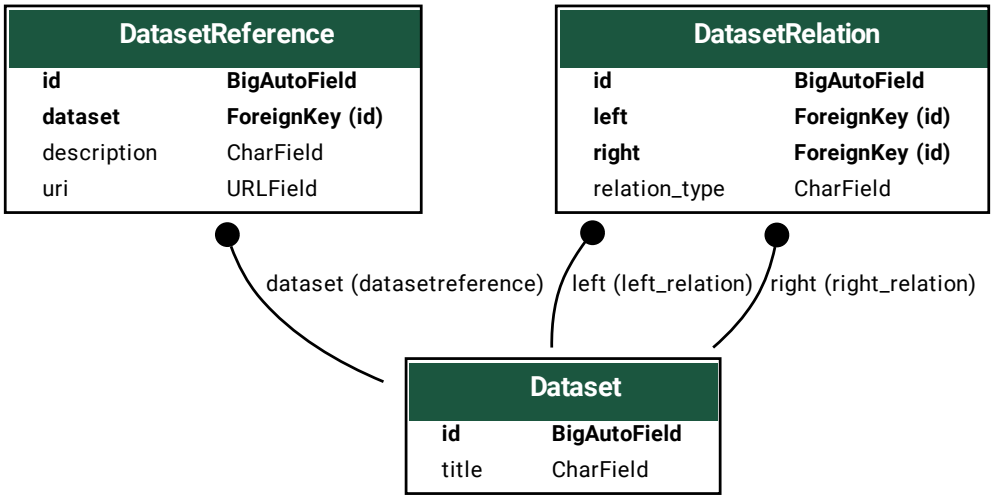
Other relations are references to internal or external resources, such as related studies or datasets. They are commonly described by datacite related identifiers, see https://support.datacite.org/docs/relationtype_for_citation.

However, neither the CF-Conventions nor INSPIRE define such a relation type. But both give the possibilities to add supplementary studies, (see below) and we'll just add these informations as a *DatasetReference* object (see the *Graph* tab).

If the URI however corresponds to a handle in the model data explorer, we can also directly transfer this into a relation between datasets (see *Graph* tab) and suggest that *Dataset A is supplement to Dataset B* (see *Datasets*).

Graph

Graph



Django source

```
class Dataset(models.Model):

    title = models.CharField(max_length=50)

class DatasetReference(models.Model):

    dataset = models.ForeignKey(Dataset, on_delete=models.CASCADE)
    description = models.CharField(max_length=400)
    uri = models.URLField(max_length=300)

class DatasetRelation(models.Model):

    left = models.ForeignKey(Dataset, on_delete=models.CASCADE, related_name="left_
↪relation")
    right = models.ForeignKey(Dataset, on_delete=models.CASCADE, related_name="right_
↪relation")
    relation_type = models.CharField(max_length=30)
```

Interpretation of relations

CF-Conventions

netCDF files can define global *references* and *doi* attributes. We will check here for common DOI patterns and use this to extract the uri for the `DatasetReference` (see the *Graph* tab above).

INSPIRE

INSPIRE encodes references as `MD_AggregateInformation` with a specific `DS_AssociationTypeCode`, namely `crossReference`. So we will just use the `MD_Identifier` of these tags. If the `MD_Identifier` is listed as `gmd:code`, we will assume it's a DOI and transform it to the corresponding URL, otherwise we take it as the description of the `DatasetReference` and try if we find a URL in it.

2.2.2 Map Frontend

Browse issues

In this epic we want to shape the central element of the model data explorer, the Map interface. Model runs can define services such as WMS, RestService or WFS as a visual representation of the model run. This visualization is based on standardized interfaces for which we want to develop work flows that describe how the scientists can generate and add them to the model data explorer themselves.

The outcome of this epic is the prototype UI that defines

- how to find data in the Model Data Explorer
- and how to select what to visualize

We are also talking in this epic about the work flows how to automatize the generation of services using the metadata of the OGC-standards. If one has a working WMS on a THREDDS-Server for instance, it should be straight-forward to generate the visualization in the model data explorer.

2.2.3 Statistical analysis

Browse issues

The model data explorer will be based upon the Digital Earth Messaging Framework, a remote procedure call framework for distributed data analysis. We want to offer the user the possibility to analyze and compute on the data without downloading it, through the web and through scripts (e.g. Python).

Within this epic, we want to discuss the implementation of this distributed analysis work flow. How it can be configured, secured and designed for a maximum of flexibility and usability.

We want to generate work flow descriptions and documentations about:

- the basic API for a backend module
- how to create a backend module from templates
- how to register a backend module
- how to secure the backend module
- how to visualize the features of a backend module in the Model Data Explorer Web UI
- how to access the features of a backend module from script

2.2.4 THREDDS Server configuration and implementation

Browse issues

THREDDS server are very rich in functionalities and provides many services that can be used within the model data explorer. And it integrates well with the data upload functionalities of the SFTP server (see *Data upload*).

However, the standard setup has some disadvantages:

1. the catalog entries can only be edited by admins of the THREDDS-server
2. catalogues are written with and XML syntax that is unknown to most scientists
3. the handling and overview of multiple catalogue files can be quite a challenge
4. one cannot give access to all catalogue files to individual scientists in order to prevent unauthorized access

This is why we intend to develop a django app that generates the catalogue entries using the power of the Django templating system. Another advantage of using Django is that we can integrate it with the permission and relation system of the Model Data Explorer (see #21 and #17). This management app lets the scientists configure their own content on the THREDDS and how it is displayed.

Within this epic, we want to learn more about the functionalities of the THREDDS-Server through discussions with more experienced THREDDS-Server-Managers, and we want to define the necessary features and workflows that need to be made available. In the end, we want an implementation in the prototype to demonstrate these workflows. We will, however, for now only schedule one user story to discuss the features of the THREDDS-Server. Within this user story, we define how we want to proceed.

2.2.5 Download of raw data

Browse issues

We need to implement a method to download the data that is displayed on the map, such that the user can access and analyze the raw data with their own scripts. Obviously, the download should then not only be restricted to the visible data, but should provide the option for multiple layers, times and parameters.

The idea here is to have three possibilities for the implementation:

1. As we plan for the visualization via WMS (see #25), we want to find a way how to populate download parameters from a WCS. The selection of parameters and times is then transformed into a request to the WCS service
2. describe where (and when) the data can be found on the SFTP-Server
3. generate a download url via a backend module
4. define the size of the data download (data quality in addition to extent and resolution) according to user needs
5. Generate a python code block that allows to download the data directly via the OpenDAP interface of the underlying THREDDS server and then save it

2.2.6 Data upload

Browse issues

In order to enable the upload of files for the THREDDS-Server (see *THREDDS Server configuration and implementation*), we want to setup an SFTP-Server.

Users can ask for folders and a quota for a model run, which is then granted by the admins. This folder is tight to the user groups in the shared LDAP-authentication to configure the read-/write access from the Django application.

Within this EPIC we want to evaluate SFTP vs. the webDAV protocol and MAMS (as suggested by the IT). If we decide to go for SFTP, we want to continue with defining and demonstrating the work flows to

- request for space on the data volume
- tie this folder to the permission system
- link the folders with related users, groups and model runs

2.2.7 Stakeholders

Browse issues

Todo: still have to write down some description on the *Stakeholders* epic

2.2.8 Federation and Harvesting of Model Data Explorers

Browse issues

The Model Data Explorer should be as FAIR as possible. There are many reasons, why one would want to run his or her own Model Data Explorer.

One example might be to manage your own THREDDS-Server via the Django-solutions that we develop here, or to add other Django apps on top for the evaluation of model data.

Than we should implement the possibility to harvest other instances of the Model Data Explorer. Within this epic, we want to define

1. how to ensure a safe mapping from model run at Model Data Explorer 1 to Model Data Explorer 2
2. how to ensure the correct mapping of relations between model runs, users and groups between the two model data explorer instances

2.3 Implementation details

In this part, we want to become a bit more concrete on the exact implementation in the model data explorer. This is therefore getting a bit more technical.

We describe here the basic implementation with Django and the workflows.

2.3.1 Models

Related Issues	Document Status
#17	In Progress
#21	In Progress

A prototypical implementation of the django based model system is available on https://gitlab.hzdr.de/model-data-explorer/django-psql-dag-test.

As you see from the Graph, the exact implementation is quite complicated due to the many relations between the different objects. But we'll break it down in the following paragraphs.

2.3.1.1 DataGroup Relations

To efficiently mimic the relations between multiple DataGroups, we make use of so-called *directed acyclic graphs* implemented by the django-postgresql-dag library.

Description

There are four main classes that we need to represant the relations between data groups:

`DataGroup` The data group as a central object.

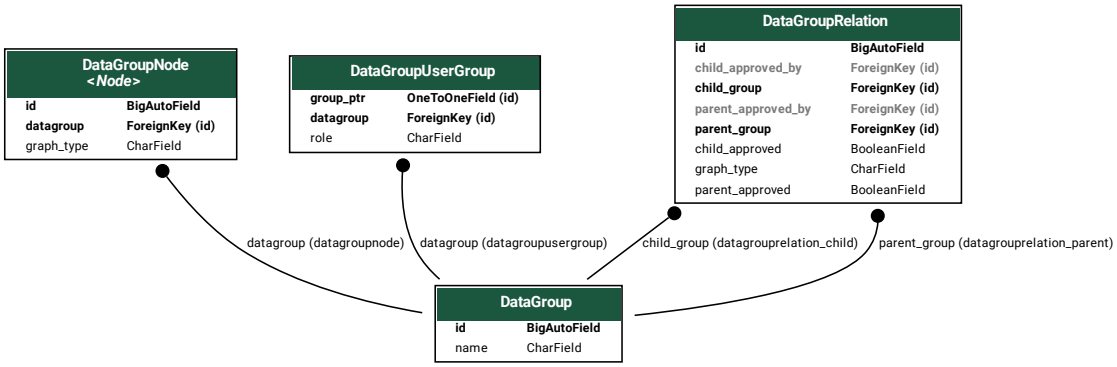
`DataGroupNode` Each `DataGroup` holds a reference to multiple nodes, and each node represents one specific *graph type* that represents the permission system. There is a *parent* graph, a *member* graph and a *list* graph. See *DataGroup Permission Graphs* for more explanation.

`DataGroupUserGroup` Each `DataGroup` holds reference to multiple *user* groups, one group per role that a user can have in terms of a data group (see *Users and data groups*). This is mainly relevant for the *parent* and *members* graph, see *below*.

`DataGroupRelation` A relation between two data groups. Each `DataGroupRelation` is for one specific graph system (*parent*, *member* or *list*, see *below*). If an owner of the parent or child group requests a relation, such a `DataGroupRelation` object is created. The owners of the other groups are then asked for approval. As soon as they approve, the connections between the two *DataGroupNode*s in the corresponding graph are made.

Graph

Graph



Django source

```
from django.db import models
from django_postgresql_dag.models import node_factory, edge_factory
from django.contrib.auth.models import User, Group


class DataGroupEdge(edge_factory("templateapp.DataGroupNode", concrete=False)):

    def __str__(self):
        return (
            f"{self.parent.graph_type}: {self.parent.datagroup} "
            "→ {self.child.datagroup}"
        )


class DataGroupNode(node_factory(DataGroupEdge)):
    """A node to display relations between data groups."""

    class Meta:

        unique_together = ("graph_type", "datagroup")

    class GraphType(models.TextChoices):
        """Type of the graph for a datagroup."""

        parent_graph = "PARENT", "Parent-Child graph"
        member_graph = "MEMBER", "Member inheritance graph"
        list_graph = "LIST", "Dataset listing graph"

    graph_type = models.CharField(max_length=10, choices=GraphType.choices)
```

(continues on next page)

(continued from previous page)

```
    datagroup = models.ForeignKey("DataGroup", on_delete=models.CASCADE)


class DataGroupUserGroup(Group):
    """A group with a certain role in a :class:`DataGroup`"""

    class Roles(models.TextChoices):
        owner = "OWNER", "owner priviliges"
        user_manager = "USERMANAGER", "user manager privileges"
        data_manager = "DATAMANAGER", "data manager privileges"
        data_editor = "DATAEDITOR", "data editor privileges"
        editor = "EDITOR", "editor privileges"
        member = "MEMBER", "view permissions"

    datagroup = models.ForeignKey("DataGroup", on_delete=models.CASCADE)
    role = models.CharField(max_length=20, choices=Roles.choices)


class DataGroupRelation(models.Model):
    """An relation between two datagroups that awaits approval."""

    class Meta:

        unique_together = ("child_group", "parent_group", "graph_type")

    child_group = models.ForeignKey(
        "DataGroup", on_delete=models.CASCADE, related_name="%(class)s_child"
    )
    parent_group = models.ForeignKey(
        "DataGroup", on_delete=models.CASCADE, related_name="%(class)s_parent"
    )

    child_approved = models.BooleanField(default=False)
    parent_approved = models.BooleanField(default=False)

    child_approved_by = models.ForeignKey(
        User,
        blank=True,
        null=True,
        on_delete=models.SET_NULL,
        related_name="%(class)s_child_approval",
    )
    parent_approved_by = models.ForeignKey(
        User,
        blank=True,
        null=True,
        on_delete=models.SET_NULL,
        related_name="%(class)s_parent_approval",
    )

    graph_type = models.CharField(
        max_length=10, choices=DataGroupNode.GraphType.choices
```

(continues on next page)

(continued from previous page)

```
    )


class DataGroup(models.Model):
    """A DataGroup in the Model Data Explorer"""

    name = models.CharField(max_length=100)
```

DataGroup Permission Graphs

The *parent* graph

Nodes in the *parent* graph inherit the owners, data managers, etc. from the parents. As an example, consider *Hereon* and the *Institute for Coastal Systems*.

Hereon has a node in the parent graph, and the *institute* does. To represent the relation between the two, we set the `DataGroupNode` in the *parent* graph of *Hereon* as a parent for *institutes* `DataGroupNode` in the *parent* graph. As a consequence, each owner, data manager, etc. of *Hereon* will become an owner, data manager, etc. of the *institute*.

The *members* graph

Nodes in the *parent* graph inherit members. But in contrast to the *parent* graph above, members of the child node are added as members of the parent node. As an example: If there is a relation in the *members* graph like *Hereon* → *Institute for Coastal Systems*, then each member of the *Institute for Coastal Systems* will be automatically added as a member of *Hereon*.

The *list* graph

The *list* graph implements the permission to list the contents of a data group as the result of another data group. If there is a relation *Hereon* → *Institute for Coastal Systems* between the two `DataGroupNodes` in the *list* graph, this means that

1. the *institute* is listed as a child on the *Hereon* page
2. each *Dataset* that grants *list* permissions to the *institute* is also listed on the *Hereon* page

2.3.1.2 Dataset relations

Relations between `Datasets` and `DataGroups` and between `Datasets` and `Users` are implemented more or less the same, as they both share the same roles (see *Users and datasets* and *Datasets and data groups*). Each relation needs to be approved by the dataset owner and the related party (user or data group). Once the relation is approved, the owner or corresponding `DataGroupUserGroups` get the relevant permissions.

Description

Four models are important here:

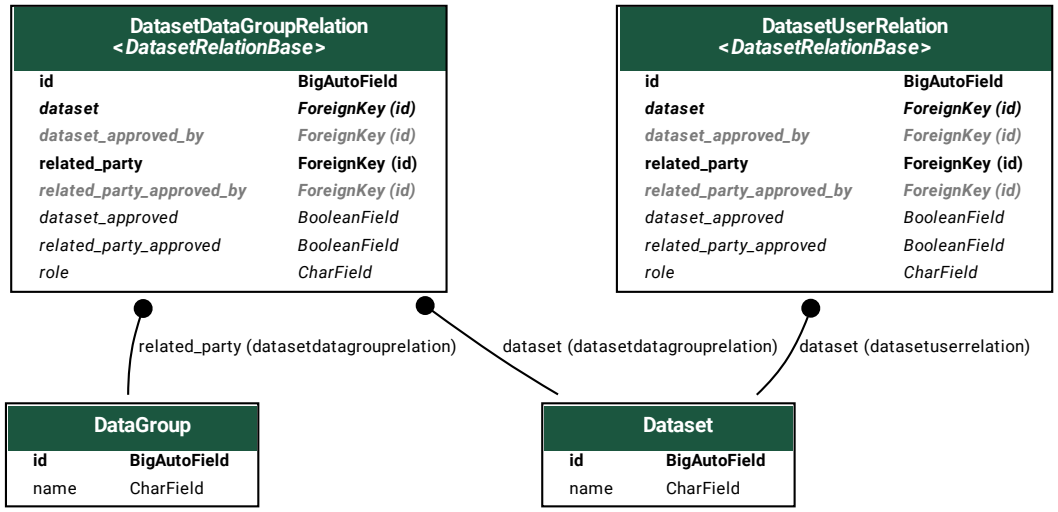
Dataset The dataset that is related to the user or data group

DatasetUserRelation A relation between a dataset and a user

DatasetDataGroupRelation A relation between a dataset and a data group

Graph

Graph



Django source

```
from django.contrib.auth.models import User, Group

class DataGroup(models.Model):
    """A DataGroup in the Model Data Explorer"""

    name = models.CharField(max_length=100)


class Dataset(models.Model):
    """A dataset in the model data explorer."""

    name = models.CharField(max_length=100)
```

(continues on next page)

(continued from previous page)

```
class DatasetRelationBase(models.Model):
    """Abstract base for a relation between dataset and user or group."""

    class Meta:

        unique_together = ("dataset", "related_party", "role")
        abstract = True

    class Roles(models.TextChoices):
        owner = "OWNER", "owner priviliges"
        data_manager = "DATAMANAGER", "data manager privileges"
        data_editor = "DATAEDITOR", "data editor privileges"
        editor = "EDITOR", "editor privileges"
        member = "MEMBER", "view permissions"


    dataset = models.ForeignKey(
        "Dataset", on_delete=models.CASCADE, related_name="%(class)s"
    )
    role = models.CharField(
        max_length=20, choices=Roles.choices
    )

    dataset_approved = models.BooleanField(default=False)
    related_party_approved = models.BooleanField(default=False)

    dataset_approved_by = models.ForeignKey(
        User,
        blank=True,
        null=True,
        on_delete=models.SET_NULL,
        related_name="%(class)s_dataset_approval",
    )
    related_party_approved_by = models.ForeignKey(
        User,
        blank=True,
        null=True,
        on_delete=models.SET_NULL,
        related_name="%(class)s_related_party_approval",
    )


class DatasetDataGroupRelation(DatasetRelationBase):
    """A permission for a relation."""

    related_party = models.ForeignKey(DataGroup, on_delete=models.CASCADE)


class DatasetUserRelation(DatasetRelationBase):
    """A permission for a relation."""

    related_party = models.ForeignKey(
```

(continues on next page)

(continued from previous page)

```
        User, on_delete=models.CASCADE, related_name="%(class)s_related_party"
    )
```

2.3.2 Workflow descriptions

Related Issues	Document Status
#17	In Progress
#21	In Progress

Here we describe the different workflows in the model data explorer on a more technical basis.

Our relation model includes several permission levels and roles as decribed in *Datasets and data groups* and *Data groups*. In this section, we describe how the permissions are implemented and what happens if we change relations between users, datagroups and datasets.

We document this with so-called *sequence diagrams*. If you are not familiar with this concept of diagrams, please read the following short description.

Explanation on Sequence Diagrams

Sequence diagrams, as their name suggests, describe a sequence of actions between actors (such as a *DataGroup Owner*) and objects (such as a *DataGroup*).

Each actor and owner has a so-called *lifeline*. If action *A* is below action *B* on a *lifeline*, then *A* happens before *B*.

An *action* in that sense is visualized by an arrow. A solid arrow means that the object/actor requests something, a dashed arrow means that the object/actor returns something.

Furthermore we have so-called interaction fragments, denoted by a gray rectangle. Here we use the following fragments:

loop A *loop* fragment implies a loop on the arguments that are given in square brackets (e.g. `[DataGroupNode]`)

ref A *ref* fragment (also called *interaction use*) points to a different sequence diagram (e.g. the *Link DataGroups* fragment refers to *Sequence Diagram to link two DataGroups in the parent graph*).

alt An *alt* fragment refers to alternative decisions, like an *if-else*-case. We usually use these to document something like *if approved by DataGroup Owner, do something, else, revert*.

If you want to know more about sequence diagrams, please have a look at https://www.uml-diagrams.org/sequence-diagrams.html.

2.3.2.1 Create DataGroup workflow

Related Issues	Document Status
#21	In Progress

Every registered user in the model data explorer can create a `DataGroup` and become an owner it. Once a `DataGroup` object has been created, several `DataGroupUserGroup` objects are created, one for each role that a user can have in a datagroup (see *Users and data groups* and *DataGroup Relations*). Furthermore, we create one `DataGroupNode` per graph type, see *Data groups* and *DataGroup Permission Graphs*.

The user who created the `DataGroup` is then added as an owner for this data group and automatically added to the corresponding `DataGroupUserGroup`.

This workflow with `DataGroupUserGroups` provides an efficient workflow as permissions to datasets, etc. are granted to the user group and not to the individual owner. As such, if a user does stops being an owner, data editor, etc. we just remove him or her from the corresponding `DataGroupUserGroups` and he or she looses all the permissions.

A more schematic illustration of this workflow is displayed in the figure *Sequence Diagram to create a DataGroup*.

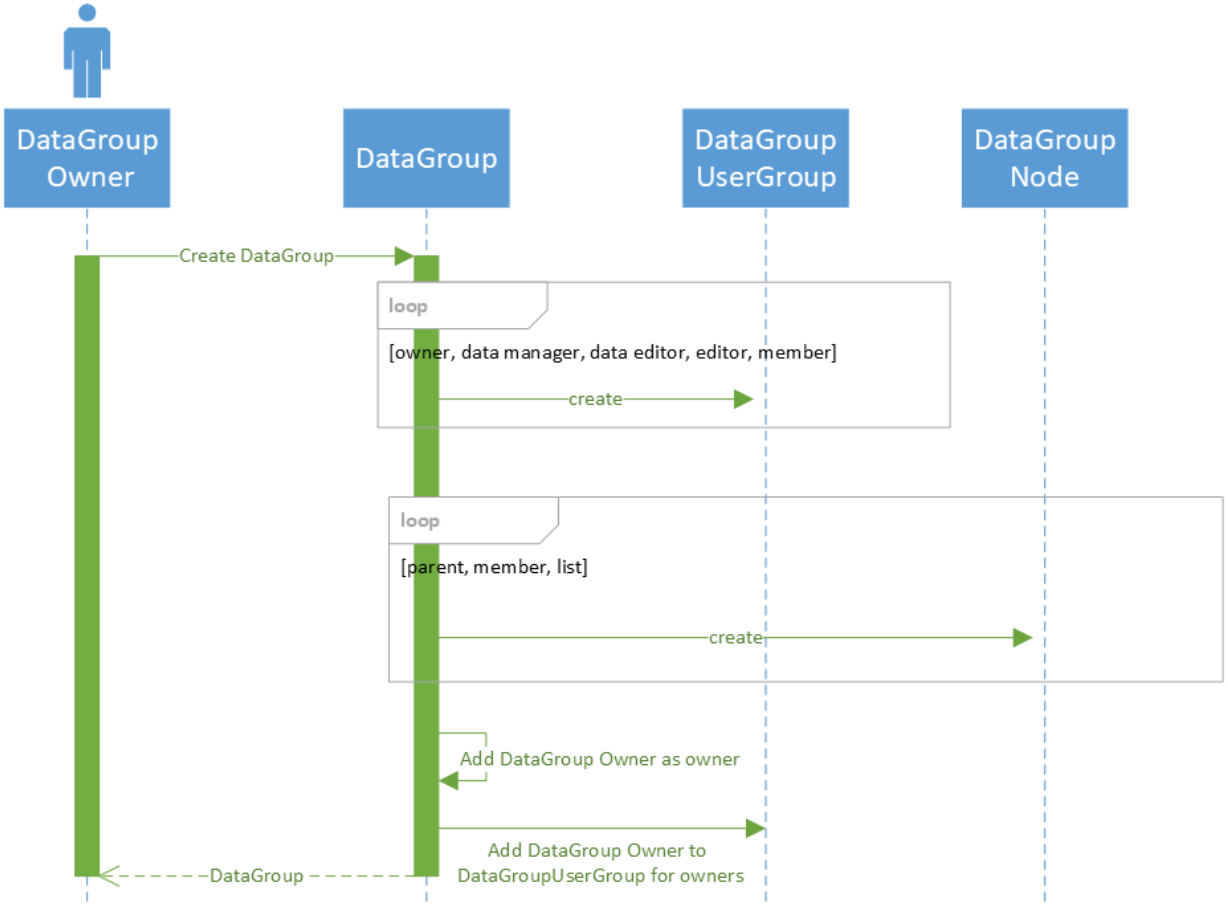


Fig. 6: Sequence Diagram to create a `DataGroup`

2.3.2.2 Link `DataGroups` workflow

Related Issues	Document Status
#17	In Progress
#21	In Progress

Each owner of a `DataGroup` can link the group to another one. He or she can then select whether the group should be a child or a parent of the other group. In any case, an owner of the other group has to confirm the membership.

The figure *Sequence Diagram to link two DataGroups in the parent graph* illustrates this workflow for the case that the owner of the *child* `DataGroup` requests the link. A `DataGroupRelation` object is created (see also *DataGroup Relations*) and the owners of the other (*parent*) `DataGroup` get an email with a link to approve or reject the request. If rejected, the `DataGroupRelation` object is deleted again, otherwise the users of `DataGroupUserGroups` of the *parent* `DataGroup` are added to the corresponding `DataGroupUserGroups` of the child.

In the future now, all users from the parents `DataGroupUserGroup`s are automatically added or removed from the childs `DataGroupUserGroup`s (see *Add an owner for a parent DataGroup workflow* and *Add an owner for a parent DataGroup workflow*).

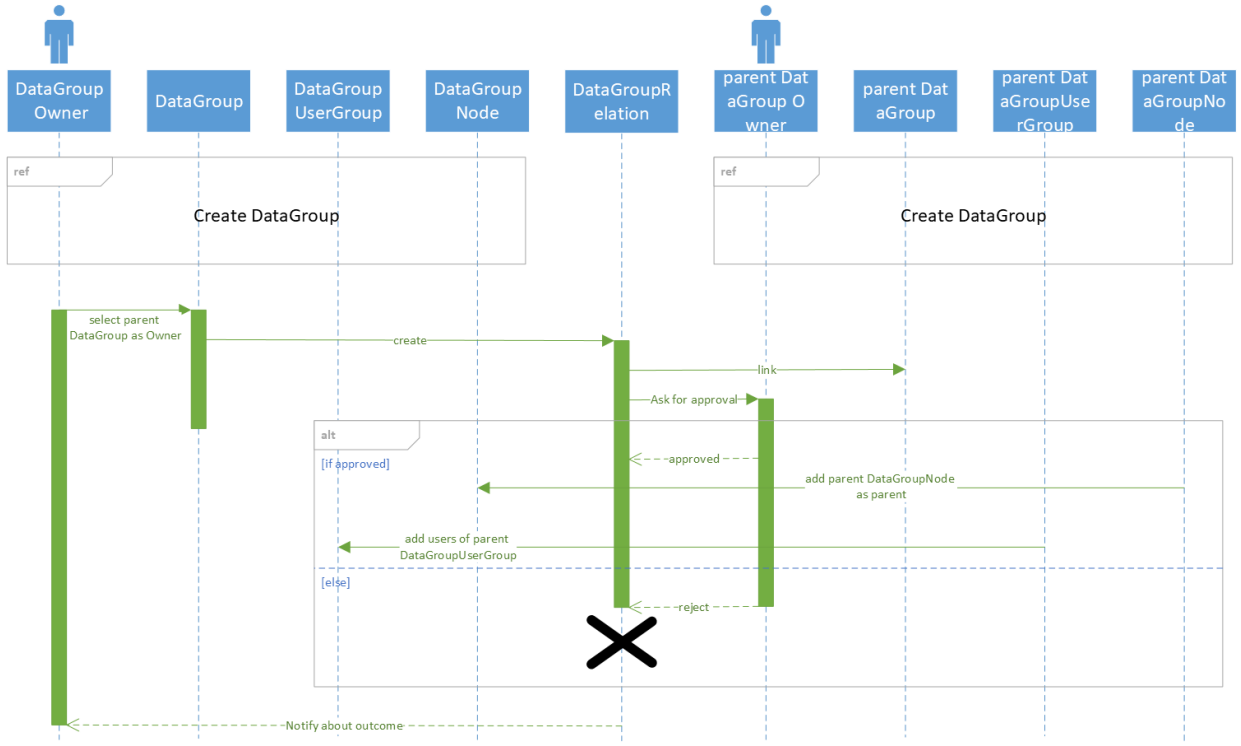


Fig. 7: Sequence Diagram to link two `DataGroup`s in the *parent* graph

2.3.2.3 Refresh a `DataGroupUserGroup` workflow

Related Issues	Document Status
#17	In Progress
#21	In Progress

There are multiple scenario, where the `DataGroupUserGroup` of a data group needs to be updated. E.g. when a user has been removed from the list of owners of the data group itself, or from one of it's parent `DataGroup`s (see *Remove an owner from a parent DataGroup workflow* for instance).

To efficiently refresh the owners of the data group, we need to query the `DataGroupUserGroup`s of all it's parents and add the resulting users to the `DataGroupUserGroup` of the `DataGroup` that is about to be refreshed.

Using the `DataGroupNode` relations implemented with django-postgresql-dag, we can get all owners of the parents with one single database transaction and synchronize the `DataGroupUserGroup` with this list.

The figure *Sequence Diagram to refresh the owners of a DataGroup* illustrates this workflow.

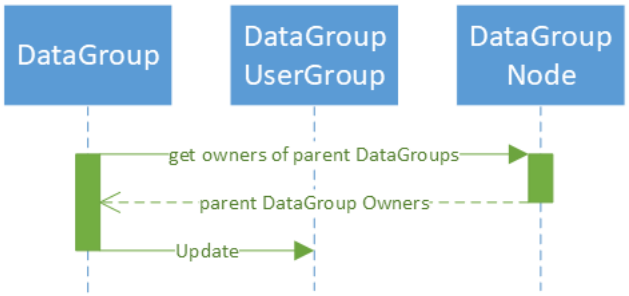


Fig. 8: Sequence Diagram to refresh the owners of a `DataGroup`

2.3.2.4 Add an owner for a parent `DataGroup` workflow

Related Issues	Document Status
#17	In Progress
#21	In Progress

When a new owner (or any other role) is added to a `DataGroup`, we need to make sure that this owner is also added to the `DataGroupUserGroup`s of the `DataGroup`s children. Here we can use the `DataGroupNode` in the *parent* graph to query the children and add the new user to the corresponding user groups (see *Sequence Diagram to add a new owner to a parent DataGroup* for an illustration).

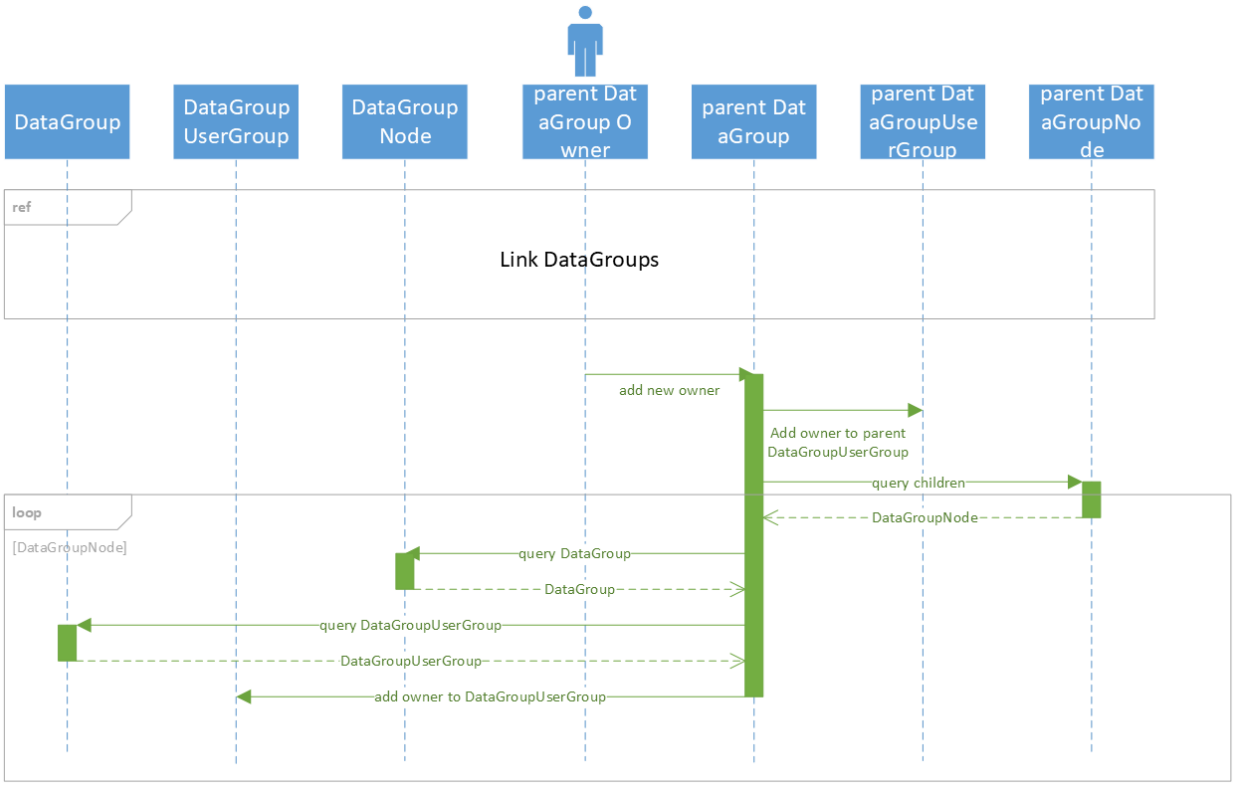


Fig. 9: Sequence Diagram to add a new owner to a parent `DataGroup`

2.3.2.5 Remove an owner from a parent `DataGroup` workflow

Related Issues	Document Status
#17	In Progress
#21	In Progress

Removing an owner (or any other role) from a `DataGroup` is a bit more tricky than adding one. As in the workflow to *add a new owner*, we use the `DataGroupNode` of the *parent* graph to query the children. However, now we cannot simply remove the user from the corresponding `DataGroup` as we need to check whether their might be any other parent `DataGroup` of the child that justifies the owners role for the child `DataGroup`. So we query the parents users or we *refresh the data group*.

This workflow is illustrated in *Sequence Diagram to remove an owner from a parent DataGroup*.

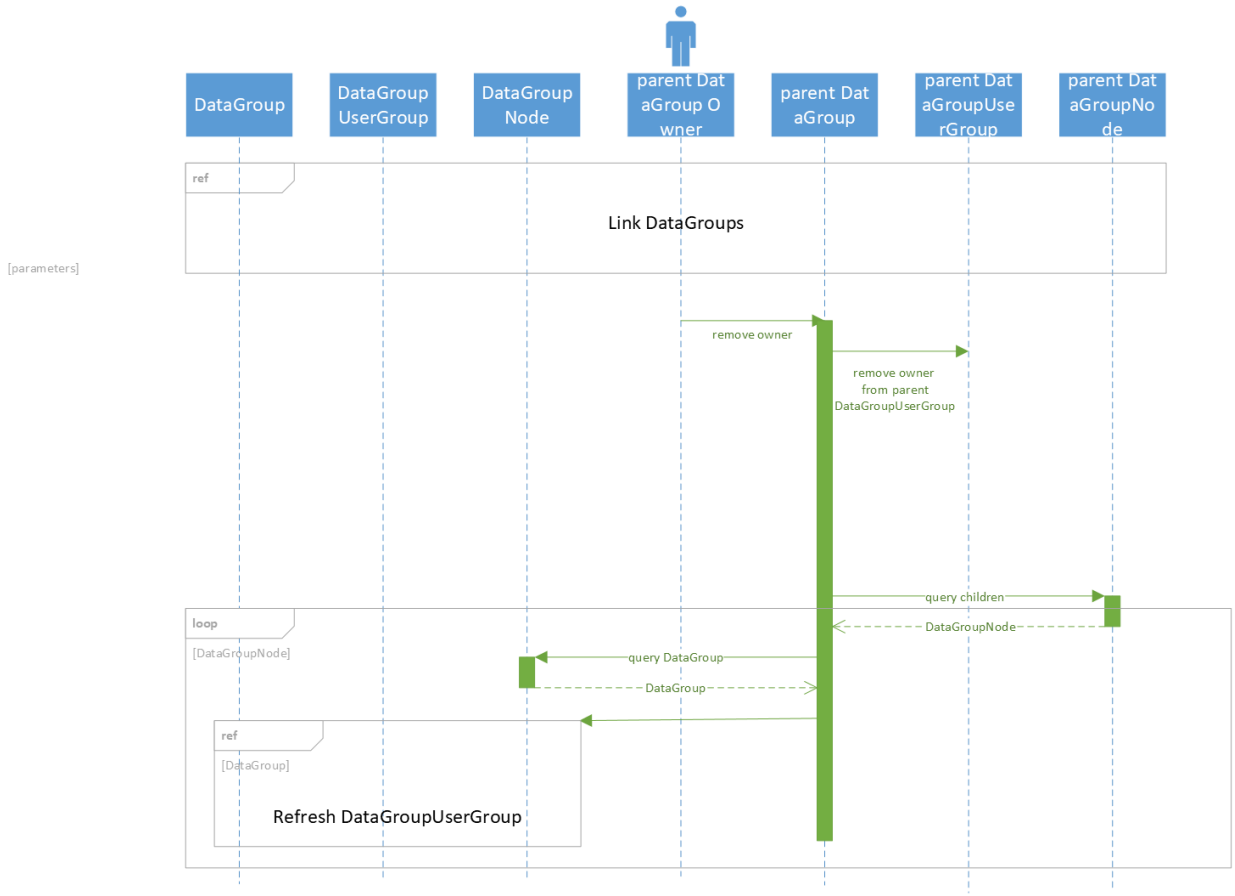


Fig. 10: Sequence Diagram to remove an owner from a parent `DataGroup`

2.3.2.6 Unlink DataGroups workflow

Related Issues	Document Status
#17	In Progress
#21	In Progress

Unlinking two `DataGroups` (i.e. removing the relation that has been added with *Link DataGroups workflow*) is the same as *removing a user from a parent DataGroup* and is illustrated in *Sequence Diagram to remove the link between two DataGroups*. We need to query all child `DataGroupNodes` and trigger a *refresh of the user groups*.

Afterwards, we remove the `DataGroupRelation` object.

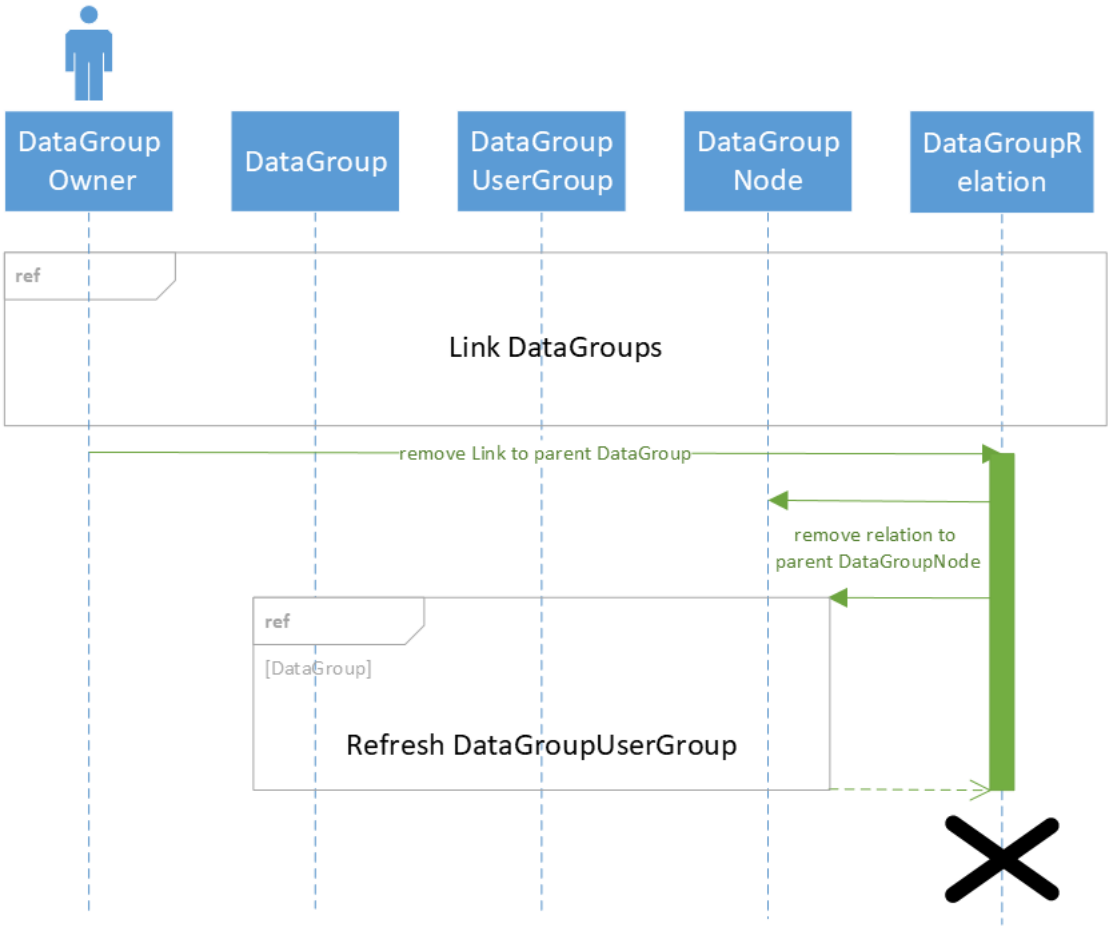


Fig. 11: Sequence Diagram to remove the link between two `DataGroups`

2.3.2.7 Create Dataset workflow

Related Issues	Document Status
#21	In Progress

Every registered user in the model data explorer can generate a `Dataset`. This then automatically creates a `DatasetUserRelation` that highlights him or her as an owner of the dataset.

See *Dataset relations* for a more detailed explanation of the models, and *Sequence Diagram to create a Dataset* for a schematic illustration of the workflow.

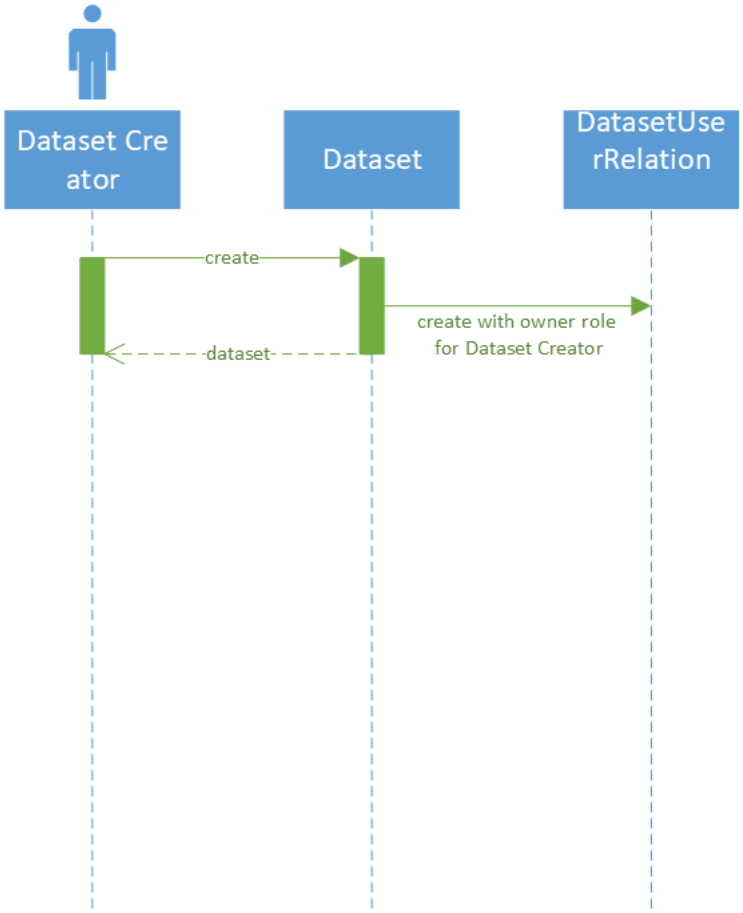


Fig. 12: Sequence Diagram to create a `Dataset`

2.3.2.8 Link `Dataset` and `DataGroup` workflow

Related Issues	Document Status
#17	In Progress
#21	In Progress

An owner of a dataset or and owner of a datagroup can request to link the two objects. He or she can then assign roles for this relation as described in *Datasets and data groups*.

The figure *Sequence Diagram to link a Dataset to a DataGroup* illustrates this workflow for the case that the dataset creator links the dataset to another datagroup.

Once such a link is requested, a `DatasetDataGroupRelation` object is created and the owners of the `DataGroup` are asked to approve the relation. If they approve, the corresponding permissions on the `Dataset` are granted to the `DataGroupUserGroup` of the `DataGroup`. If the `DataGroup` owner rejects, the `DatasetDataGroupRelation` is deleted.

As all owners of parent `DataGroup`s are contained in the `DataGroupUserGroup` (see link-datagroups), they also get the relevant permissions.

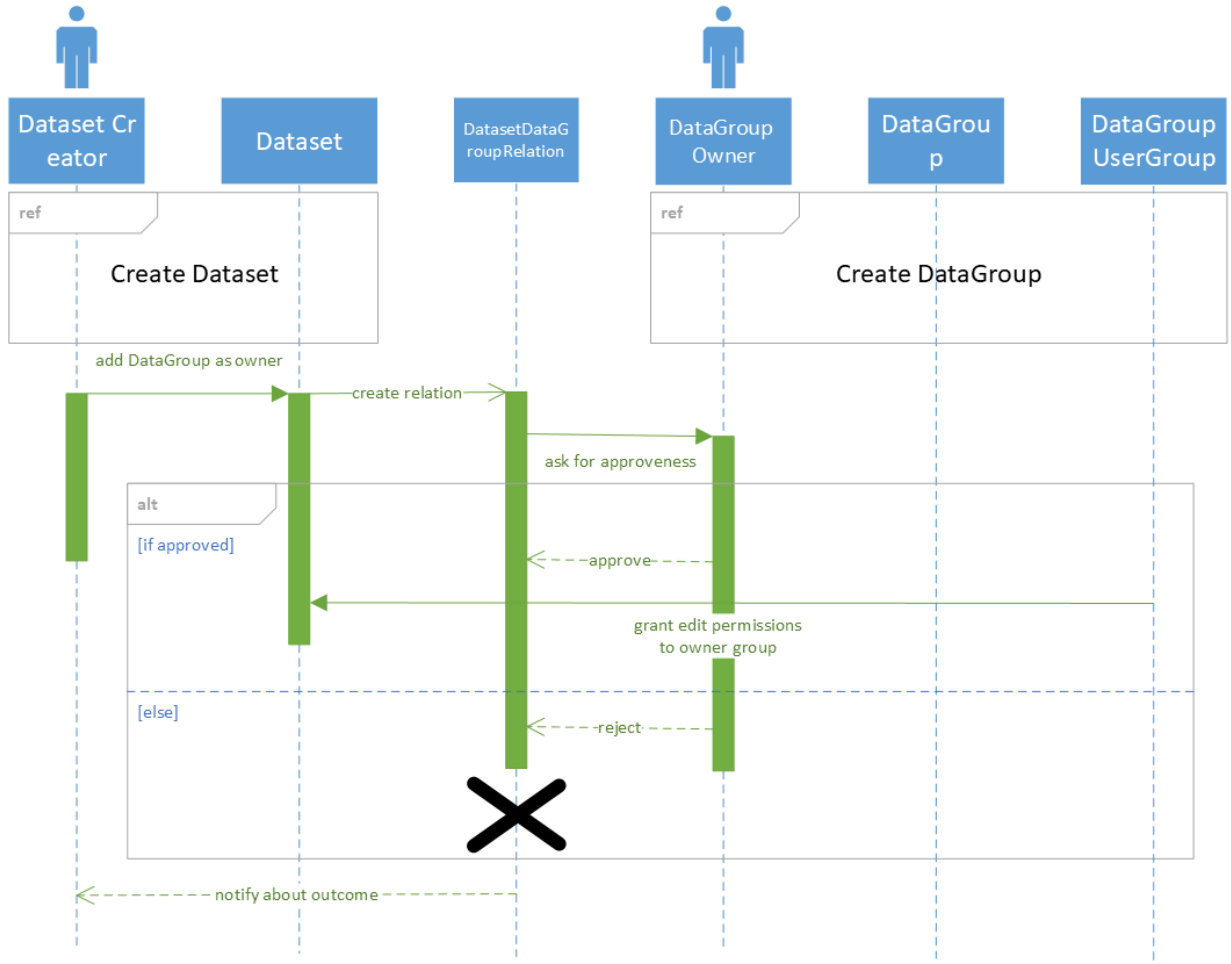


Fig. 13: Sequence Diagram to link a `Dataset` to a `DataGroup`

2.4 Usage and navigation on Gitlab

During the prototype development, we will make heavy use of the issues on Gitlab.

The model data explorer has a dedicated group at https://gitlab.hzdr.de/model-data-explorer and the prototype development takes place in the prototype repository on gitlab.

We created several issues there that we want to discuss during this first phase before we start with the development.

2.4.1 Email Notifications

The project is public, but as a member of the project team, you are automatically made a member of this repository. Note that you might want to set the notifications to *Participate*, otherwise you will get a mail for each and every comment on this repository. You can edit your notification settings with the bell symbol in the upper right corner of the repository website.

If you are then particularly interested in a specific issue, you should turn on the notifications and you'll be notified via mail when there is any progress. This can be achieved with the *Notifications* switch in the sidebar on the right of the detail view of an issue (see #5 for instance).

2.4.2 Navigation

Every issue on this repository is also assigned to an epic (see *Epics in the prototype development*), indicated by the purple issue label. You can click on that label to filter for all issues of this epic. As an alternative, every epic in this documentation also has a link to the corresponding issues, see *Metadata* for instance.

2.4.3 Participation

Please feel free to comment to the individual issues on gitlab or create new ones with questions or topics you want to discuss. If you never worked with gitlab, it's pretty simple. Just go to the detail page of the issue #5 for instance, and enter your comment in the text field at the bottom of the website.

2.5 Frequently Asked Questions

Here we collect questions related to the different aspects on the model data explorer.

2.5.1 Relations

Models are usually forced by multiple static (=initial) and temporally varying boundary conditions (=forcings). How can *right* or *left* be *lists*?

From *relation terminology*

You do not use a list, but one dataset can define the same relation type to multiple other datasets. E.g.

- coastDat-3_COSMO-CLM_ERAi has been forced with Era Interim
- coastDat-3_COSMO-CLM_ERAi has been forced with some other dataset

The inverse relationship does not make sense for boundary conditions, as typically a boundary conditions (say: bathymetry) has inifinite possibilities of forcing models ...

From *relation terminology*

Relations between datasets are optional. A dataset does not have to define a relation about the forcing, etc. Equally, one dataset might serve as forcing for an multiple datasets.

2.5.2 Datasets

How do you deal with unconvential (modular) modeling systems, such as MOSSCO, where the number of things *left* and *right* are on the order of 10-20? Theoretical example: `MuSSeL_SCHISM-physics-SED3D-sediment-ECOSMO-ecosystem-filtration-ICEALGAE-WW3-waves-CICE-ice-forced-by-5-HYCOM-ecosystem-boundaries-TOPX-tides-IOW-bathymetrie-version-30045-windfarm-parameterization-2035` Seems rather impractical.

From *dataset terminology*

We want to keep the name of a dataset informative. On WDCC, we often include the forcing etc. in the name, we do not want to rely on this in the Model Data Explorer.

2.6 ToDo

Todo: Link to interpretation of roles and permissions

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/terminology.rst, line 53.)

Todo: Add Django graphs for author model

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/terminology.rst, line 62.)

Todo: Add Django graphs for user model

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/terminology.rst, line 85.)

Todo: Add Django graphs for dataset model

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/terminology.rst, line 98.)

Todo: Add Django graphs for datagroup model

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/terminology.rst, line 148.)

Todo: still have to write down some description on the *Stakeholders* epic

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/stakeholders.rst, line 8.)

Todo: Document variable related metadata

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 57.)

Todo: document geographic and temporal resolution as well?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 94.)

Todo: Add descriptive spatial extent (such as global, continental, etc.)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 99.)

Todo: add creation, publication and revision date

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 104.)

Todo: The `UnidataDD2MI.xsl` methodology needs to be elaborated further.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 134.)

Todo: Add OrcID

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 322.)

Todo: Add ROR ID

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mde-prototype/checkouts/latest/source/epics/metadata/datasets.rst, line 539.)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search