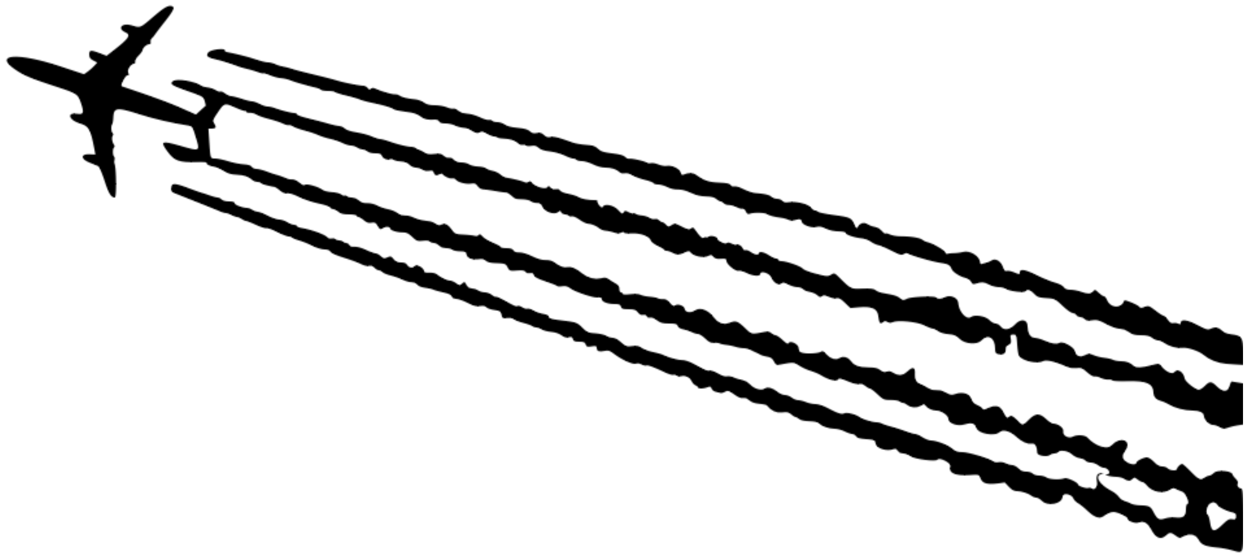
django-chemtrails Documentation

Release 0.0.19

Inonit AS

Sep 04, 2017

Table of Contents:



CHAPTER 1

Configuration

Add `chemtrails` to `INSTALLED_APPS` in your `settings.py` file.

```
#
# settings.py
#

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    ...
    'chemtrails',                          # Core functionality
    'chemtrails.contrib.permissions'   # If you want to use the permission system␣
↪(requires django-rest-framework)
]
```

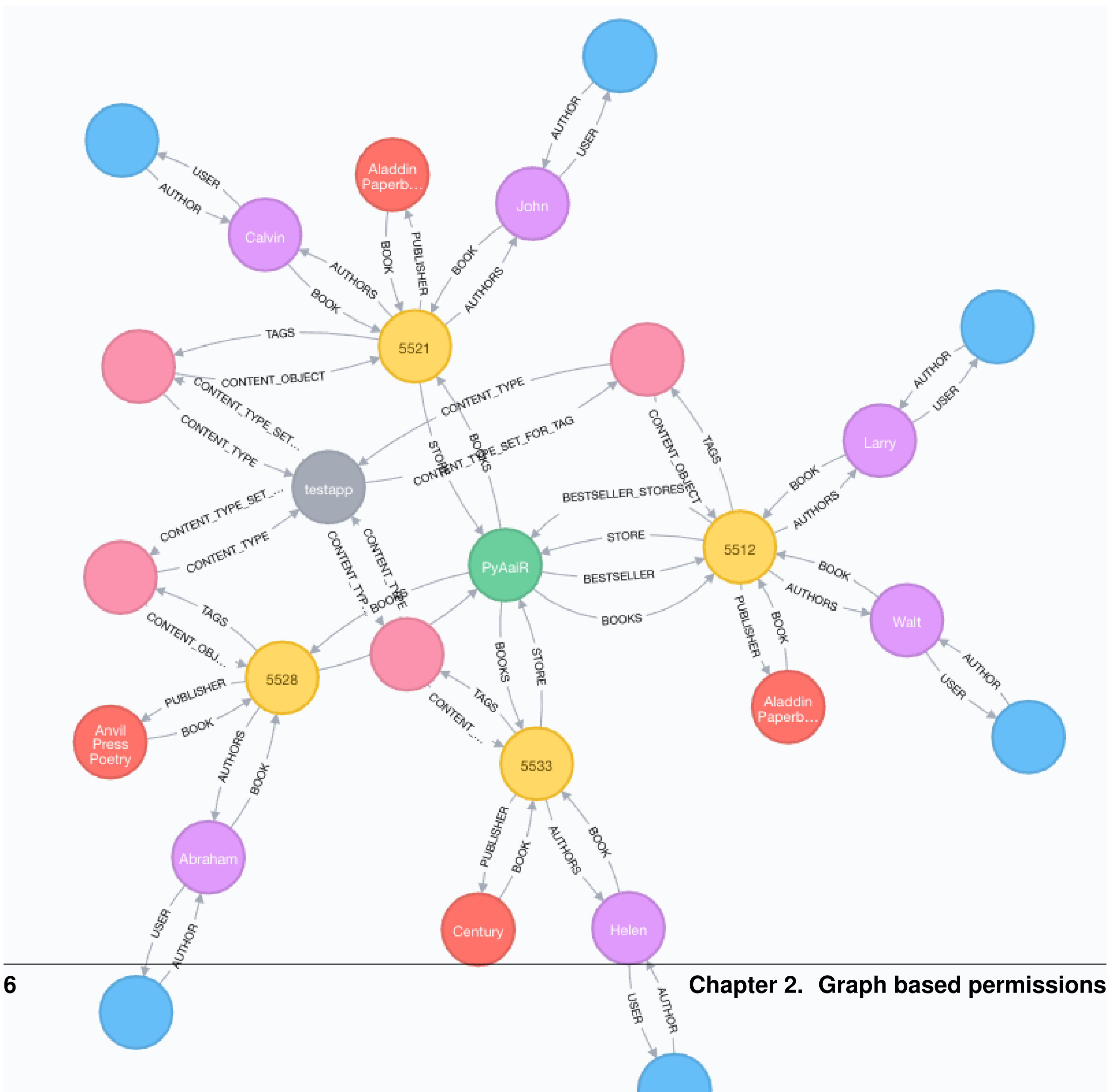
Chemtrails settings

Settings for `chemtrails` are all namespaced in the `CHEMTRAILS` setting dictionary.

```
#
# settings.py
#

CHEMTRAILS = {
    # Flip the chemtrails-switch. Boolean value to indicate that mind-control fluid␣
↪should
    # be released and all the worlds knowledge written to the Neo4j database.
    # Defaults to True.
    'ENABLED': True,

    # Maximum depth of recursive connections to be made when synchronizing a node.
    # Defaults to 1, which means that the node will recursively connect to other␣
↪nodes,
    # which has a direct connection to the source node. Setting a value of 2 will␣
↪cause
    # each connected node to recursively connect their directly connected nodes and␣
↪so on.
    # Setting to 0 will disable connecting relationships.
    'MAX_CONNECTION_DEPTH': 1,

    # If True, relationships will be named (loosely) after the attribute name
    # on the Django model. If False, relationships will have a generic name of
    # either 'RELATES_TO', 'RELATES_FROM' or 'MUTUAL_RELATION' based on the␣
↪relationship type.
    # Defaults to True.
    'NAMED_RELATIONSHIPS': True,

    # If True, make a META relation between the meta-node instance and the node
    # instances for this type.
    # Defaults to False.
    'CONNECT_META_NODES': False,

    # A list of models that should be excluded from mirroring.
    # Defaults to the example shown below.
    'IGNORE_MODELS': [
        'migrations.migration'
    ],
}
```

CHAPTER 2

Graph based permissions



Once we have the database mapped and synchronized we have the ability to read relationship data in a brand new way. We now can generate paths, and inspect the way our dataset is related in a much broader way than we can with a pure relational database.

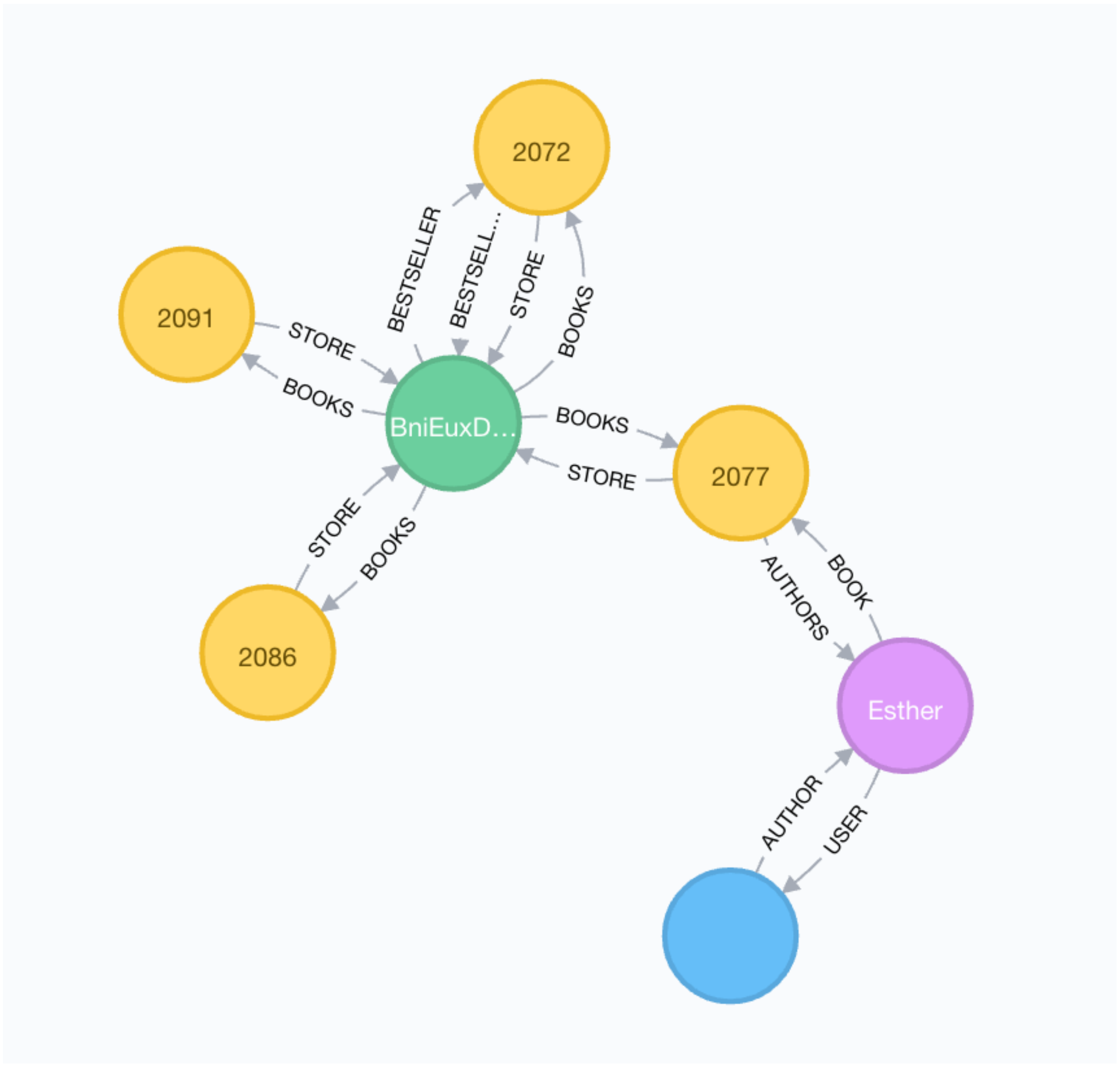
One thing we can do is to exploit this and build a permission system on top of it. Let's say we have a user Esther in the picture above (I know she's not there, it's an old image) which authenticates into the book store application. We want her to access all books she has written. This can be achieved by the executing the following Cypher statement.

```
MATCH (u:UserNode {pk:69})-[AUTHOR]->(a:AuthorNode)-[BOOK]-(b:BookNode) RETURN b
```

Further, if we want to return all books which is in the same store as one or more books Esther has written, we could easily extend the above cypher query. Let's store the results in a `path` variable as well and see what happens.

```
MATCH path = (u:UserNode {pk:69})-[*..3]->(s:StoreNode)-[BOOKS]-(b:BookNode) RETURN␣
↪path
```

The above query first finds a `UserNode` with property `pk:69`, then looks at all relationships 3 steps out, looking for a `StoreNode` which it will follow through a relationship called `BOOKS` to all nodes of type `BookNode`. Finally return the `path` variable.



As you can see, Cypher is a quite expressive query language.

The idea

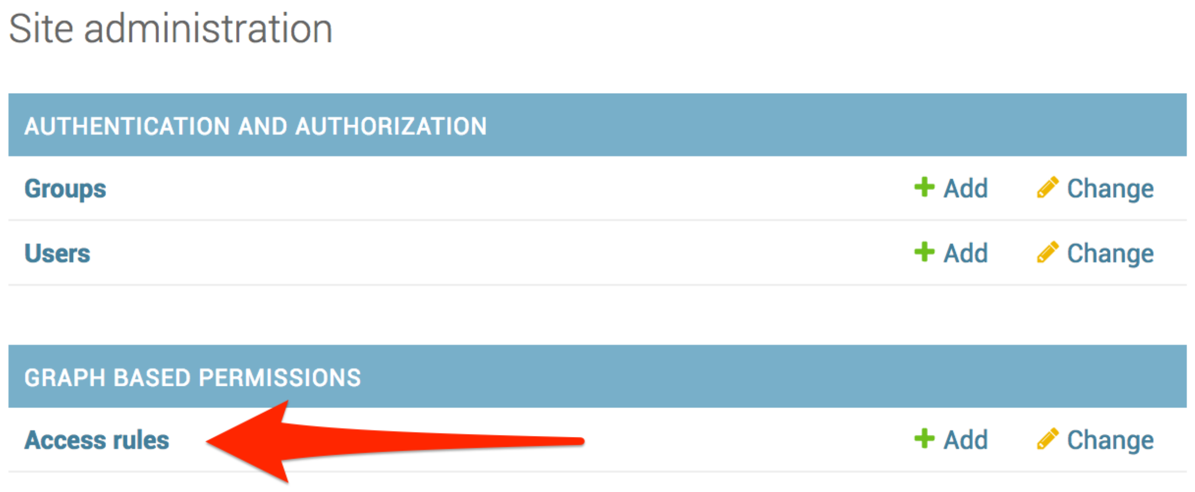
So, the general idea is to store these queries in a database. When a requests come in and want to access some object, we'll look up the authenticated user in the graph. Next we'll see if we have the requested object represented in the graph. Finally we'll check if we have any stored query which can generate a path from the user node to the object node. If so, return the object.

What's nice about an approach like this, is that we have a truly dynamic rule set for our application. We can choose to deactivate some rules for a limited time, or we can add or remove rules on the fly in order to let users either access new kind of data, or restrict their access to already available data.

In order to do that, we'll use *Access rules*.

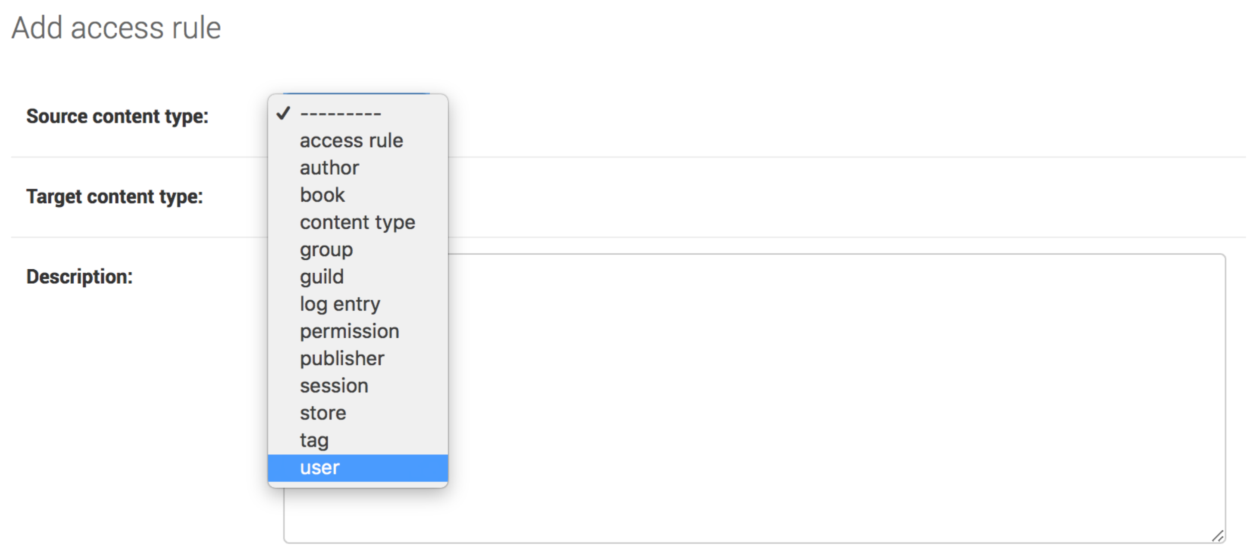
Access rules

If you have enabled both `chemtrails` and `chemtrails.contrib.permissions` in the Django settings file under `INSTALLED_APPS`, you should have a similar entry in the Django admin interface.



When evaluating access rules, the permission system will dynamically identify what kind of objects we're dealing with using the `ContentType` framework which ships with Django.

Content Types


Add access rule

Source content type:

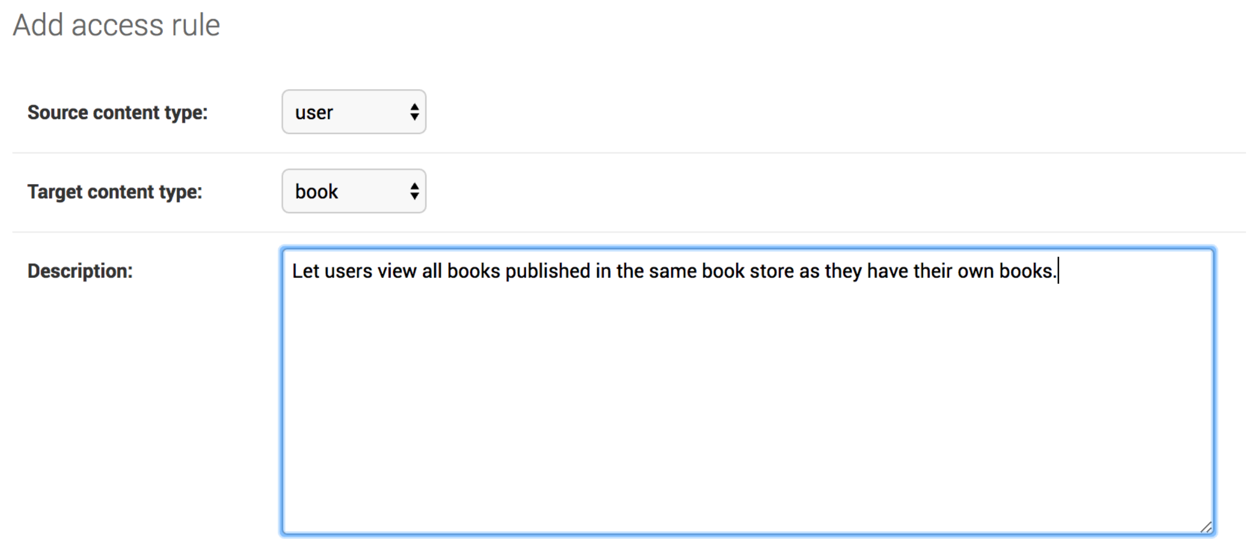
✓ ---------
access rule
author
book
content type
group
guild
log entry
permission
publisher
session
store
tag
user

Target content type:

Description:

When dealing with authenticated web requests, we'll almost always use `user` as our *Source content type*. This is the kind of object we want to calculate the Cypher path *from*. As for `Target content type`, choose whatever object type you want the current access rule to evaluate. Let's follow the example from before (*Graph based permissions*) and create a rule which lets Esther look at all the books in the same book store she has her books.

So, proceed setting the `Target content type` to `book`. Next write a *good* description for what the rule is doing. This is very important as you might end up with hundreds of rules in a large application.


Add access rule

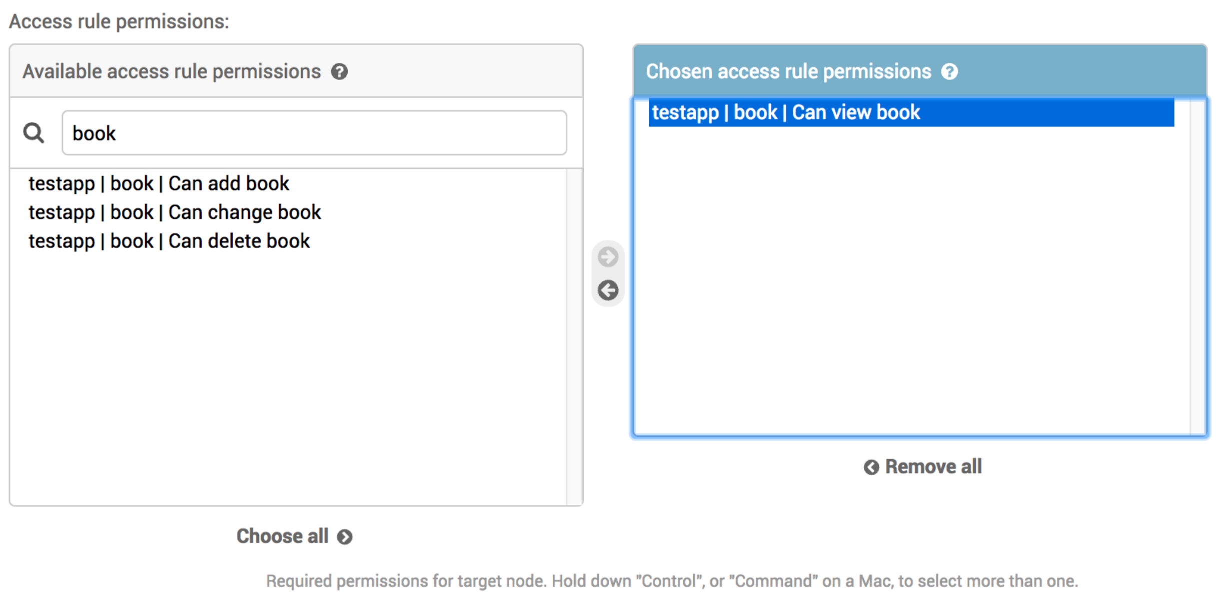
Source content type: user

Target content type: book

Description: Let users view all books published in the same book store as they have their own books.

Permissions

Next up is permissions. What kind of permissions should the new rule grant? We don't necessarily want all users that can view an object to also delete them. Let's grant `view` permission for our new access rule.


Access rule permissions:

Available access rule permissions

book

testapp | book | Can add book
testapp | book | Can change book
testapp | book | Can delete book

Choose all

Chosen access rule permissions

testapp | book | Can view book

Remove all

Required permissions for target node. Hold down "Control", or "Command" on a Mac, to select more than one.

Note: `view` is **not** a standard Django permission. It can be added relatively easily by using custom permissions.

Relation types

Next up is the `Relation types` field. This field contains a JSON-ish string which will be parsed and converted to Cypher whenever the access rule engine parses the rule.

Todo

Create a GUI widget for creating relation types so end-users don't have to understand the weird and confusing syntax.

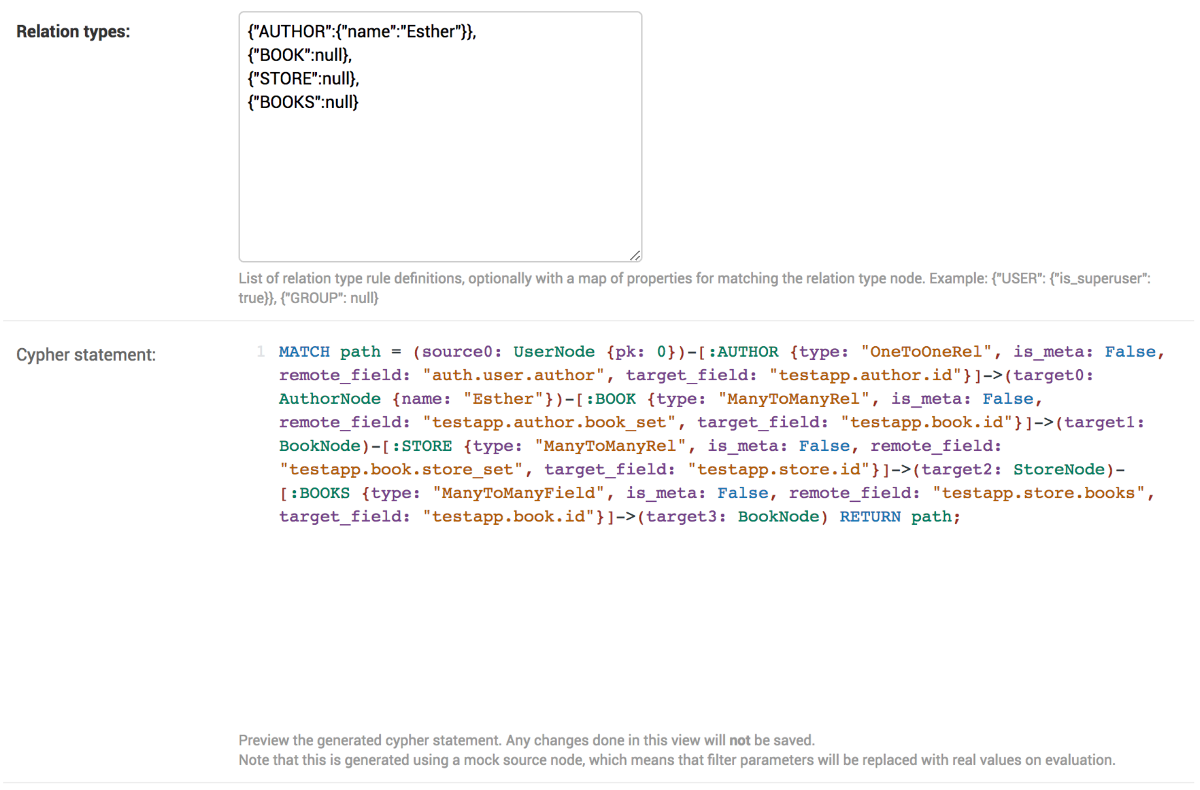
Unfortunately, this is rather confusing and difficult to work with at the moment, so plans exist in order to make a GUI widget which will take this fields place. The following piece of JSON should do:

```
{"AUTHOR":{"name":"Esther"}},
{"BOOK":null},
{"STORE":null},
{"BOOKS":null}
```

Important: What's important to notice here is that we **only specify the relationships**! The formatting in the input field must be a *comma separated list of JSON objects*, where each object has a *single key* being the relationship name, and either `null` or a *nested object* containing properties which should be matched for the *relationship target*.

Read the above message once more, and make sure you understand whats going on here!

Click the **"Save and continue editing"** button and you should get a nice preview of the generated Cypher statement right below the `Relation types` field.



Relation types:

```
{"AUTHOR":{"name":"Esther"}},
{"BOOK":null},
{"STORE":null},
{"BOOKS":null}
```

List of relation type rule definitions, optionally with a map of properties for matching the relation type node. Example: {"USER": {"is_superuser": true}}, {"GROUP": null}

Cypher statement:

```
MATCH path = (source0: UserNode {pk: 0})-[:AUTHOR {type: "OneToOneRel", is_meta: False,
remote_field: "auth.user.author", target_field: "testapp.author.id"}]->(target0:
AuthorNode {name: "Esther"})-[:BOOK {type: "ManyToManyRel", is_meta: False,
remote_field: "testapp.author.book_set", target_field: "testapp.book.id"}]->(target1:
BookNode)-[:STORE {type: "ManyToManyRel", is_meta: False, remote_field:
"testapp.book.store_set", target_field: "testapp.store.id"}]->(target2: StoreNode)-
[:BOOKS {type: "ManyToManyField", is_meta: False, remote_field: "testapp.store.books",
target_field: "testapp.book.id"}]->(target3: BookNode) RETURN path;
```

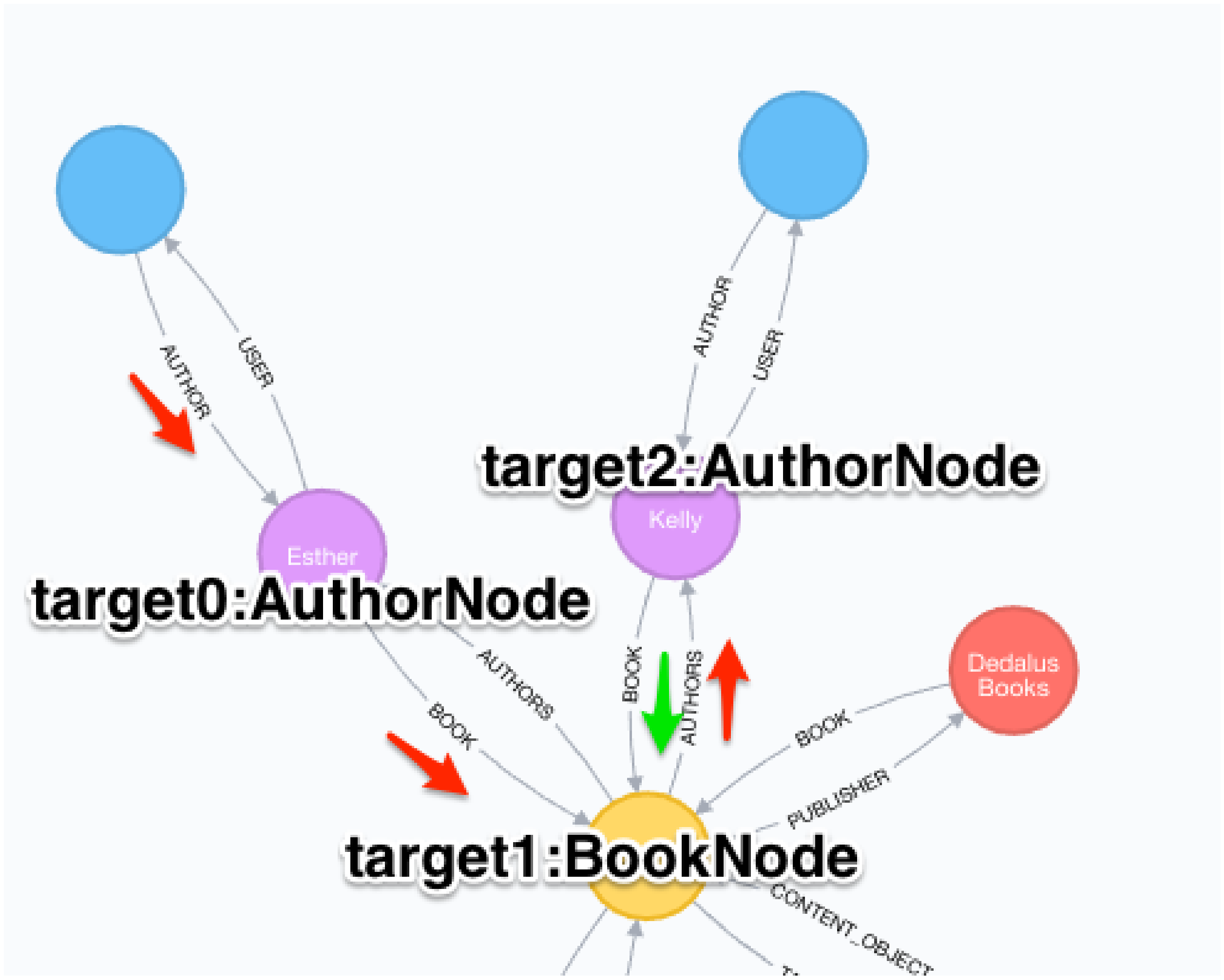
Preview the generated cypher statement. Any changes done in this view will **not** be saved.
Note that this is generated using a mock source node, which means that filter parameters will be replaced with real values on evaluation.

Take a minute to study the generated code and compare it to the JSON input. Upon evaluation the `pk` property in the `Source content type` node will be replaced by whatever primary key the authenticated user has, so we're sure the path is calculated from the correct node. Also, the generator algorithm will inspect the Django field type and make sure the path is calculated using the correct relationships.

Special syntax

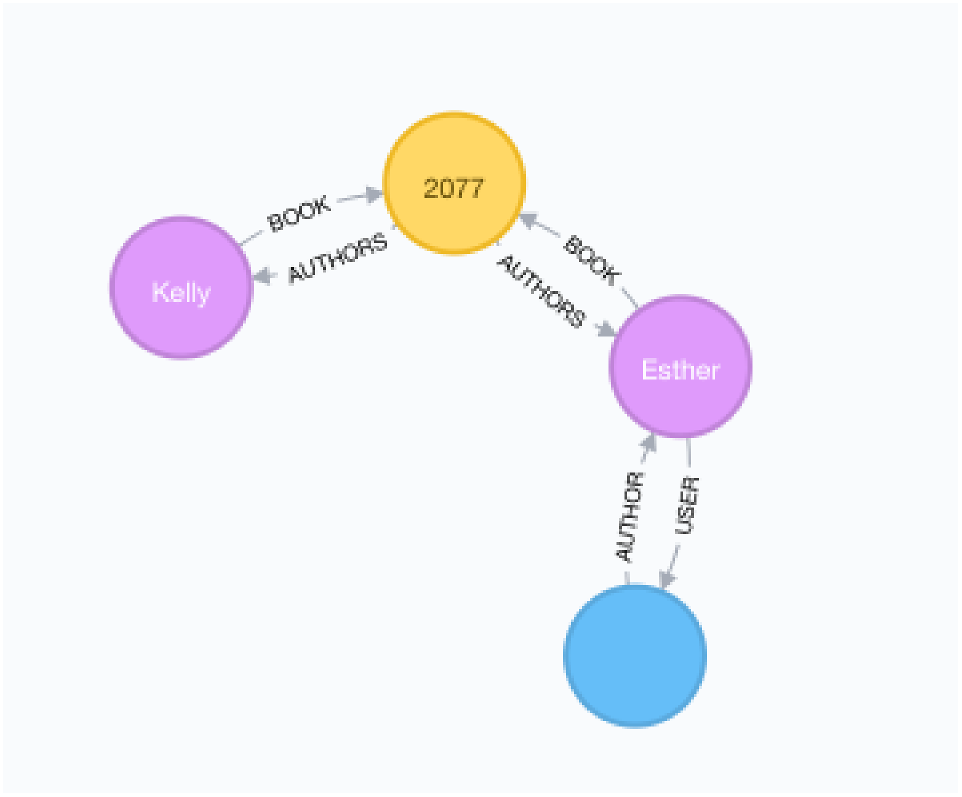
Back references

We have a few special syntax rules in order to do back references when generating the cypher statement. Imagine you want to create a path based on a condition in another node. If we look at the example from before, say we want to get *all books written by Esther which are co-authored by Kelly*. We want to *force* the cypher generator engine to traverse back to *exactly* the same node (BookNode) as we came from.



```
{"AUTHOR":{"name":"Esther"}},
{"BOOK":null},
{"AUTHORS":{"name":"Kelly"}},
{"{1:BOOK}":null}  // <- Note the "{1:BOOK}" syntax
```

By inserting `{"{1:BOOK}":null}` as the `relationship` name we make a *back reference* to `target1:BookNode` in the generated Cypher statement. This might not seem very intuitive (and it isn't) at first, so it might take some time in order to make it right.



Source reference

The other special reference we have is `{source}`. This can be used in order to reference properties on the source node. Using the example above we could do something like the below in order to match Esther's age to her primary key value (which is a rather absurd thing to do, but you get the idea).

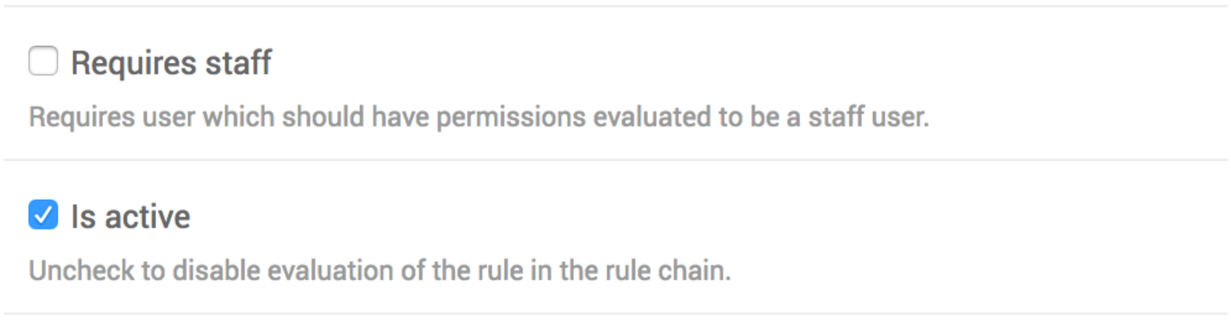
```
{"AUTHOR":{"name":"Esther"}},
{"BOOK":null},
{"AUTHORS":{"age":"{source}.pk"}}
```

Would generate the following Cypher statement:

```
MATCH path = (source0: UserNode {pk: 69})-[:AUTHOR {type: "OneToOneRel", is_meta:␣
↪False, remote_field: "auth.user.author", target_field: "testapp.author.id"}]->
↪(target0: AuthorNode {name: "Esther"})-[:BOOK {type: "ManyToManyRel", is_meta:␣
↪False, remote_field: "testapp.author.book_set", target_field: "testapp.book.id"}]->
↪(target1: BookNode)-[:AUTHORS {type: "ManyToManyField", is_meta: False, remote_
↪field: "testapp.book.authors", target_field: "testapp.author.id"}]->(target2:␣
↪AuthorNode {age: 69}) RETURN path;
```

Other security measures

Finally we have the `Requires staff` and the `Is active` fields. They should be pretty self-explanatory, where the first requires the authenticated user to be a *staff* user, and the latter can be used to deactivate a rule instead of deleting it. Inactive rules will not be evaluated.



☐ Requires staff

Requires user which should have permissions evaluated to be a staff user.

☑ Is active

Uncheck to disable evaluation of the rule in the rule chain.

CHAPTER 3

About

This project aims to solve complex object based permissions by utilizing the relationships between entities in a graph.

CHAPTER 4

Features

- Synchronize Django model instances to Neo4j
- Recursively connect related nodes

CHAPTER 5

Installation

```
$ pip install django-chemtrails
```

CHAPTER 6

Requirements

- A Django project
- Neo4j running and accepting connections using the bolt protocol

CHAPTER 7

Changelog

Some day...

CHAPTER 8

Indices and tables

- genindex
- modindex
- search