wagtail-graphql-api

Jul 10, 2019

Contents

CHAPTER 1

Index

1.1 Getting started

1.1.1 Requirements

- Python 3[1]
- A Wagtail project[2]

If you do not have a Wagtail project set up, please follow the guide[3] to create one.

1.1.2 Download

This package can be installed from the PyPI[4] via pip.

```
pip install wagtail-graphql-api
```

This package should be installed as a dependency of an existing Wagtail project.

1.1.3 Configuration

The package is a Django application. It needs to be added to your Wagtail project's setting file. Also `graphene_django` is a package used by `wagtail-graphql-api`, therefore it needs to be enabled as well.

```
# settings.py

INSTALLED_APPS = [
```

(continues on next page)

[1] https://www.python.org/downloads/
[2] https://wagtail.io/developers/
[3] http://docs.wagtail.io/en/stable/getting_started/tutorial.html
[4] https://pypi.org/project/wagtail-graphql-api

(continued from previous page)

```
    # The rest of your apps...
    'graphene_django',
    'wagtail_graphql',
]
```

Next step is to set up Graphene[5] to use the schema provided by `wagtail-graphql-api`.

```
# settings.py

GRAPHENE = {
    'SCHEMA': 'wagtail_graphql.schema.schema'
}
```

After that is done, the GraphQL endpoint has to be exposed in the URL dispatcher. To do that you need to add a path in the configuration, usually it is a *urls.py* file.

```
# urls.py

from django.urls import path

from graphene_django.views import GraphQLView

urlpatterns = [
    # Other URL paths...
    path('graphql/', GraphQLView.as_view(graphiql=True, pretty=True)),
    # Other URL paths...
]
```

Then after the development server is started (`./manage.py runserver`), the GraphQL endpoint should be accessible via http://localhost:8000/graphql/.

Integrate models

By default the library will only add a GraphQL pages endpoint for the wagtail's core Page model[6]. It can be queried via the GraphQL endpoint with the following query:

```
query {
  pages {
    wagtailcore {
       page {
         id
         title
         url
       }
    }
  }
}
```

Enabling page model to be accessible via GraphQL endpoint

To query any specific page model fields, it needs to first be registered. To do that the page model has to inherit *`wagtail_graphql.models.GraphQLEnabledModel`* (page 25).

[5] https://graphene-python.org/
[6] https://docs.wagtail.io/en/stable/reference/pages/model_reference.html#page

```
# blog/models.py
from wagtail.core.fields import StreamField
from wagtail.core.models import Page

from wagtail_graphql.models import GraphQLEnabledModel


class BlogPage(GraphQLEnabledModel, Page):
    introduction = models.TextField(help_text='Text to describe the page',
                                    blank=True)
    body = StreamField(BaseStreamBlock(), verbose_name="Page body", blank=True)
```

Assuming that the model exists under the `blog` app, it should be possible to query it with the following query:

```
query {
  pages {
    blog {
      blogPage {
        id
        title
        url
      }
    }
  }
}
```

Specifying GraphQL fields

The fields exposed in the endpoint will also have to be explicitly defined. It requires adding `graphql_fields` list with *wagtail_graphql.models.GraphQLField* (page 25) instances to the model definition, e.g.

```
# blog/models.py
from wagtail.core.fields import StreamField
from wagtail.core.models import Page

from wagtail_graphql.models import GraphQLEnabledModel, GraphQLField


class BlogPage(GraphQLEnabledModel, Page):
    introduction = models.TextField(help_text='Text to describe the page',
                                    blank=True)
    body = StreamField(BaseStreamBlock(), verbose_name="Page body", blank=True)

    graphql_fields = [
        GraphQLField('introduction'),
        GraphQLField('body'),
    ]
```

Now those fields should be accessible via the endpoint in the following way:

```
query {
  pages {
    blog {
      blogPage {
        introduction
```

(continues on next page)

(continued from previous page)

```
        body
      }
    }
  }
}
```

Snippets

Snippets that inherit *wagtail_graphql.models.GraphQLEnabledModel* (page 25) will be accessible via the GraphQL endpoint. The query structure is as follows:

Custom models

Custom models object types can also be added to the GraphQL schema with this library in the same way as page models or snippets. The only difference to the snippets and pages is that that it will not be query-able. The sole point will be to register the object type in the schema so it can be used to resolve related objects or can be used as a custom field types without having to manually specify the Graphene type.

Fields customisation

To add a non-database field to the GraphQL object representation of a model, a special arguments have to be specified on a *wagtail_graphql.models.GraphQLField* (page 25) instance.

class wagtail_graphql.models.**GraphQLField**(*name*, *resolve_func=None*, *graphql_type=None*)

Bases: object

Specify metadata about a model field that is to be registered at a GraphQL object type.

Parameters

- **name** – Name of the field.
- **resolve_func** (*callable*) – A custom resolve function that will be used to resolve data for this field.
- **graphql_type** – Graphene type that will be used by that field.

The name of the field can be custom as long as it does not interfere with other field names on the object.

The custom GraphQL type returned by the field can be specified using `graphql_type` parameter, e.g.

```
from wagtail_graphql.models import GraphQLField

GraphQLField('settings', graphql_type=graphene.JSONString)
```

However if there is no corresponding database field of that name, the field will not be accessible. To allow that `resolve_func` must be specified. The argument must be a Graphene-compatible resolver[7].

```
import json

import graphene
from wagtail_graphql.models import GraphQLField
```

(continues on next page)

[7] https://docs.graphene-python.org/en/latest/types/objecttypes/#resolvers

(continued from previous page)

```
GraphQLField('settings', graphql_type=graphene.Field(graphene.JSONString)),
             resolve_func=lambda self, info: json.loads(
                 self.settings
             ))
```

Model object types

Sometimes it may be necessary to use a Django model as an object type for a custom non-database field. To refer to an automatically generated object a special utility function has to be used that will resolve the object type lazily - `wagtail_graphql.lazy_model_type()`.

```
# locations/models.py
from django.db import models

from wagtail_graphql import lazy_model_type
from wagtail_graphql.models import GraphQLEnabledModel, GraphQLField


class Country(GraphQLEnabledModel, models.Model):
    # Fields about a country


class LocationPage(GraphQLEnabledModel, Page):
    lat_long = models.CharField()

    graphql_fields = [
        GraphQLField('country',
                     graphql_type=graphene.Field(
                         lazy_model_type('locations.Country')
                     ),
                     resolve_func=lambda self, info: self.get_country()),
    ]

    def get_country(self):
        # Logic to get a country object based on latitude and longitude.
        return country
```

QuerySetList

wagtail_graphql.types.structures.QuerySetList (page 21) is a custom list type that adds Django's QuerySet arguments like filtering or ordering. However to specify it on the model classes it is necessary to import it lazily using `wagtail_graphql.lazy_queryset_list()`. To benefit from the arguments built-in in the `QuerySetList`, the queryset has to be filtered through *wagtail_graphql.utils.get_base_queryset_for_model_or_qs()* (page 27) or if it is a page *wagtail_graphql.utils.get_base_queryset_for_page_model_or_qs()* (page 27) must be used.

```
# locations/models.py
from django.db import models

from wagtail_graphql import lazy_queryset_list
from wagtail_graphql.models import GraphQLEnabledModel, GraphQLField
```

(continues on next page)

(continued from previous page)

```
def resolve_locations(self, info, **kwargs):
    from wagtail_graphql.utils import get_base_queryset_for_page_model_or_qs

    return get_base_queryset_for_page_model_or_qs(
        self.get_location_pages(), info, **kwargs
    )


class Country(GraphQLEnabledModel, models.Model):
    # Fields about a country

    graphql_fields = [
        GraphQLField('locations', graphql_type=graphene.Field(
            lazy_queryset_list('locations.LocationPage')
        ), resolve_func=resolve_locations)
    ]

    def get_location_pages(self):
        location_pages_queryset = LocationPage.objects.all()
        # Filter the queryset
        return location_pages_queryset


class LocationPage(GraphQLEnabledModel, Page):
    # Fields about a location
```

Querying

Example queries facilitating `QuerySetList` parameters may be:

QuerySetList

Searching

If the model is enabled with the Wagtail Search[8], `searchQuery` parameter can be used to pass a search query as an argument:

```
query {
  pages {
    locations {
      locationPage(searchQuery:"test") {
        id
        title
      }
    }
  }
}
```

[8] https://docs.wagtail.io/en/stable/topics/search/

Get a specific object

To get an object of a specific ID, the ID can be passed as an argument to the `id` parameter.

```
query($id: ID) {
  pages {
    locations {
      locationPage(id: $id) {
        id
        title
      }
    }
  }
}
```

Limit and offset

```
query {
  pages {
    locations {
      locationPage(limit: 5, offset: 2) {
        id
        title
      }
    }
  }
}
```

Order by

Order by will feed the string into the QuerySet's order_by[9] method. Multiple fields can be specified with a comma as a delimiter.

```
query {
  pages {
    locations {
      locationPageByTitleAscending: locationPage(order:"title") {
        id
        title
      }

      locationPageByTitleDescending: locationPage(order:"-title") {
        id
        title
      }

      locationPageByTitleAndID: locationPage(order:"title,-id") {
        id
        title
      }
    }
```

(continues on next page)

[9] https://docs.djangoproject.com/en/stable/ref/models/querysets/#django.db.models.query.QuerySet.order_by

(continued from previous page)

```
  }
}
```

Page Interface

wagtail_graphql.types.pages.PageInterface (page 19) defines base model pages and methods that can be used on any page.

The commonly used Wagtail methods available on any page type are:

- **Returning one `PageInterface` object:**
  - `parent`
  - `specific`
- **Returning list of `PageInterface` instances:**
  - `children`
  - `siblings`
  - `nextSiblings`
  - `previousSiblings`
  - `descendants`
  - `ancestors`

Pages query mixin

The pages query mixin adds two parameters to the standard `QuerySetList` set:

- `depth`
- `showInMenus`

This allows to filter pages by depth or whether they are supposed to be shown in the menu. For example, to get a potential set of pages to be used in the header navigation, the following query may be used:

```
query {
  pages {
    wagtailcore {
      page(depth: 3, showInMenus: true) {
        id
        title
        pageType
      }
    }
  }
}
```

Images

Querying all the images

There is an additional field added to allow querying all the available images. An example query to get all the images may be:

```
query {
  images {
    id
    title
    rendition {
      url
      alt
    }
  }
}
```

This feature can be disabled with a Django setting `WAGTAIL_GRAPHQL_ENABLE_IMAGES`.

```
# settings.py
WAGTAIL_GRAPHQL_ENABLE_IMAGES = False
```

Renditions

The image object type allows to resolve Wagtail image renditions with different filters.

Note: Different filters are described in the Wagtail documentation[10].

To specify a desired rendition filter, a *filter* parameter can be used on the rendition field, e.g.

```
query {
  images {
    id
    title
    rendition(filter: "width-1200") {
      url
      alt
    }
  }
}
```

The rendition filters allowed to be used have to be specified with a Django setting, `WAGTAIL_GRAPHQL_ALLOWED_RENDITION_FILTERS`.

```
# settings.py
WAGTAIL_GRAPHQL_ALLOWED_RENDITION_FILTERS = ['original', 'width-1200']
```

Warning: `['*']` value can be used for the `WAGTAIL_GRAPHQL_ALLOWED_RENDITION_FILTERS` setting to whitelist all valid rendition filter specifications. However it is discouraged because an attacker may send malicious requests to generate a lot of unnecessary renditions that may have serious consequences for the server's performance or storage space taken.

[10] https://docs.wagtail.io/en/stable/topics/images.html

The default value if the `filter` argument is not specified can be set using the `WAGTAIL_GRAPHQL_DEFAULT_RENDITION_FILTER` setting.

```
# settings.py
WAGTAIL_GRAPHQL_DEFAULT_RENDITION_FILTER = 'original'
```

1.2 Static sites generation with GatsbyJS

This guide will guide developers towards developing a statically-generated site with GatsbyJS.

Note: This section requires the reader to be familiar with technologies such as:

- React[11]
- GatsbyJS[12]
- JavaScript language (EcmaScript 6)

GatsbyJS is a static site generator that can be used together with an external GraphQL source, such as a Wagtail GraphQL API generated with this library.

1.2.1 Requirements

- Node.js[13] (recommended newest LTS version)
- Gatsby CLI installed (NPM[14])
- A Wagtail project with the GraphQL API enabled using this library.

1.2.2 New GatsbyJS project

Note: This guide will use `npm` commands, but equivalent `yarn` commands can be used as well.

To aid the basic set-up, a template for a new Gatsby project to use with this library can be used to bootstrap a new project using the following command:

```
gatsby new your-project-name https://github.com/tm-kn/wagtail-graphql-api-gatsby-
↪starter
cd your-project-name
```

Note: The template assumes that you use a default Wagtail start project. If you do not, please comment out `'home.HomePage': path.resolve('src', 'pages', 'home-page.js')` in `gatsby-node.js`.

Before generating the site, the Django server must be started (`./manage.py runserver` in the Wagtail's project directory). Next step is to point at the server location using an environment variable or `.env` file. It should be

[11] https://reactjs.org/
[12] https://www.gatsbyjs.org/
[13] https://nodejs.org/en/download/
[14] https://www.npmjs.com/package/gatsby-cli

sufficient to copy `.env.example` to `.env` (`cp .env.example .env`). The contents of `.env` need to show the path to the GraphQL endpoint, e.g.

```
WAGTAIL_GRAPHQL_ENDPOINT=http://localhost:8000/graphql/
```

Then the Gatsby development server can be started by executing `npm start`. After the server started, the command line should output a link which can be used to access the website (by default http://localhost:9000/).

Media & documents

If the CMS website is supposed to be hidden from the public, there are two topics that need to be covered:

- Media files such as images
- Documents

The media files have to be served from a third-party service or via a proxy. The API will return absolute links to images by default. If they are server from the same web server as CMS, the proxy needs to be set up. A third-party storage service like AWS S3[15] can be used as well.

Wagtail documents rely on privacy-checks carried out in a Python code. The proxy should be used to route to the documents if back-end needs to be disguised. If all documents are deemed to be public, they can also be served from a third-party service such as S3.

Custom page types

Background

The GraphQL query used to generate a website structure is placed in the `gatsby-node.js` file within the `createPages` function:

```
const path = require('path');

const PAGE_TYPES = {
   'home.HomePage': path.resolve('src', 'pages', 'home-page.js')
};

function getComponentPathForType (pageType) {
   return PAGE_TYPES[pageType] || path.resolve('src', 'pages', 'base-page.js');
}

exports.createPages = ({ graphql, actions }) => {
   const { createPage } = actions;

   return graphql(` {
         wagtail {
            pages {
               wagtailcore {
                  page {
                     id
                     url
                     pageType
                  }
               }
```

(continues on next page)

[15] https://aws.amazon.com/s3/

(continued from previous page)

```
            }
         }
      }
   `).then(({ data, errors }) => {
      if (errors) {
         throw errors;
      }

      data.wagtail.pages.wagtailcore.page.forEach(({ url, id, pageType }) => {
         createPage({
            path: url,
            component: getComponentPathForType(pageType),
            context: {
               pageID: id
            }
         });
      });
   });
};
```

It uses the core Wagtail's Page model to find all the pages and its paths. Using the `getComponentPathForType` function it determines what React component to use for the given page type. The available types are defined in the `PAGE_TYPES` object. By default the `base-page.js` component will be used for a page type without a specific component mapped to it.

Defining a custom type

In this example a `locations.LocationPage` model is used.

To define a custom type you need to create a new React component.

Then in `gatsby-node.js` the page type has to be linked with that component.

```
// gatsby-node.js
const PAGE_TYPES = {
   // Other possible page types
   'locations.LocationPage': path.resolve(
      'src', 'pages', 'location-page.js'
   )
}
```

Make sure that the key of the object matches `pageType` value of the GraphQL page object (it is case sensitive).

After that the Gatsby server has to be restarted and the new component should be used for instances of the new page type.

Rich text

The template comes with a pre-defined component to be used as RichText container, e.g.

```
// src/pages/home-page.js
import RichText from '../components/rich-text';

const HomePage = ({ data }) => {
```

(continues on next page)

(continued from previous page)

```
   const page = data.pages.home.homePage[0];

   return (
      <div>
         {/* Any other components */}
         <h1>{page.title}</h1>
         <RichText>{page.promoText}</RichText>
         {/* Any other components */}
      </div>
   )
}

export default HomePage;
```

Streamfields

Each Wagtail project will have its own definition of stream field blocks. `wagtail-graphql-api` does a job of serialising them. However each of the custom blocks has to be defined in the front-end.

The template comes with a StreamField component included however it needs configuration before it can be used.

For any new block type a switch case has to be added, e.g. for an `ImageChooserBlock` it could be:

Then any page that has a StreamField, it can use that component, e.g.

Deployment

The app can be deployed in a plethora of way. Consult GatsbyJS' guide[16] for more information.

Index

Netlify

Netlify[17] is a platform that allows deployment of static sites.

1. Import your website from Git.
2. In the app settings, go to *Buid & Deploy* and *Environment*.
3. Add an environment variable `WAGTAIL_GRAPHQL_ENDPOINT` pointing at your website's GraphQL endpoint.
4. Trigger the build.

For more information consult GatsbyJS's guide[18].

Automatic deployments from Wagtail

Netlify allows creating build hooks. They are URLs that can be used to trigger a new deployment.

To set up a deployment hook on page publish in Wagtail, please add a new signal handler in the project.

[16] https://www.gatsbyjs.org/docs/deploying-and-hosting/
[17] https://www.netlify.com/
[18] https://www.gatsbyjs.org/docs/hosting-on-netlify/

```
# models.py
from django.conf import settings

from wagtail.core.signals import page_published, page_unpublished

import requests


def deploy_to_netlify_on_change(**kwargs):
    try:
       netlify_deploy_hook_url = getattr(settings, 'NETLIFY_DEPLOY_HOOK_URL')
    except KeyError:
       return
    if not netlify_deploy_hook_url:
       return
    r = requests.post(netlify_deploy_hook_url)
    r.raise_for_status()


page_published.connect(deploy_to_netlify_on_change)
page_unpublished.connect(deploy_to_netlify_on_change)
```

Then add the Netlify deploy hook to your settings.

```
# settings.py
import os

if 'NETLIFY_DEPLOY_HOOK_URL' in os.environ:
    NETLIFY_DEPLOY_HOOK_URL = os.environ['NETLIFY_DEPLOY_HOOK_URL']
```

1. Go to the Netlify app settings, *Build & Deploy* and *Build Hooks*. Add a new build hook for the Wagtail CMS.
2. On the back-end server set the environment variable `NETLIFY_DEPLOY_HOOK_URL` to the generated hook URL.

1.3 API reference

1.3.1 Subpackages

wagtail_graphql.inventory package

Submodules

wagtail_graphql.inventory.base module

class wagtail_graphql.inventory.base.**BaseModelInventory**
Bases: `object`

Base class for an inventory of Django models.

create_model_graphql_type(*model*, *fields*)
Create a GraphQL object type for a model and fields specified.

get_model_fields_for(*model*)
Find all GraphQL field definitions set on the registered models.

graphql_types
List of GraphQL registered with this inventory.

models
List of models registered with this inventory.

resolve_graphql_types()
Convert models and field definitions into GraphQL types.

resolve_model_fields_for(*model*)
Discover GraphQL fields definition for a particular model.

resolve_models()
Discover the models that need registering with the inventory.

wagtail_graphql.inventory.models module

class wagtail_graphql.inventory.models.**ModelInventory**
Bases: *wagtail_graphql.inventory.base.BaseModelInventory* (page 14)

Inventory of models that are not pages nor snippets.

create_model_graphql_type(*model*, *fields*)
Create a GraphQL object type for a model and fields specified.

resolve_models()
Resolve registered Django models omitting pages and snippets. The models need to subclass *wagtail_graphql.models.GraphQLEnabledModel* (page 25).

wagtail_graphql.inventory.pages module

class wagtail_graphql.inventory.pages.**PageInventory**
Bases: *wagtail_graphql.inventory.base.BaseModelInventory* (page 14)

Store metadata about Wagtail page models exposed to GraphQL.

create_model_graphql_type(*model*, *fields*)
Create a GraphQL type for the specified page model.

resolve_models()
Find all Wagtail page models eligible to be in the GraphQL endpoint. They need to subclass *wagtail_graphql.models.GraphQLEnabledModel* (page 25).

wagtail_graphql.inventory.snippets module

class wagtail_graphql.inventory.snippets.**SnippetInventory**
Bases: *wagtail_graphql.inventory.base.BaseModelInventory* (page 14)

Inventory of snippet models.

create_model_graphql_type(*model*, *fields*)
Create a GraphQL object type for a model and fields specified.

resolve_models()
Discover the models that need registering with the inventory.

Module contents

class wagtail_graphql.inventory.**Inventory**
Bases: object

Store metadata about objects exposed to the GraphQL endpoints.

wagtail_graphql.query_mixins package

Submodules

wagtail_graphql.query_mixins.base module

wagtail_graphql.query_mixins.base.**get_app_query_attributes**(*by_app_attributes*, *prefix=''*)

wagtail_graphql.query_mixins.base.**get_model_query_attributes_by_app**(*graphql_types*, *resolve_objects_func*, *field_arguments=None*)

Segregate model object types by app and generate attributes for the query object.

wagtail_graphql.query_mixins.documents module

class wagtail_graphql.query_mixins.documents.**DocumentQueryMixin**
Bases: object

documents = <wagtail_graphql.types.structures.QuerySetList object>

resolve_documents(*info*, ***kwargs*)

wagtail_graphql.query_mixins.images module

class wagtail_graphql.query_mixins.images.**ImageQueryMixin**
Bases: object

images = <wagtail_graphql.types.structures.QuerySetList object>

resolve_images(*info*, ***kwargs*)

wagtail_graphql.query_mixins.pages module

class wagtail_graphql.query_mixins.pages.**PageQueryMixin**
Bases: object

Meta
alias of create_query_mixin.<locals>.PageQueryMixinMeta

pages = <graphene.types.field.Field object>

resolve_pages(***kwargs*)

wagtail_graphql.query_mixins.pages.**create_query_mixin**()
Create the page query mixin dynamically.

wagtail_graphql.query_mixins.pages.**get_page_attributes_by_app**()

wagtail_graphql.query_mixins.pages.**get_pages_type**()

wagtail_graphql.query_mixins.sites module

class wagtail_graphql.query_mixins.sites.**CurrentSiteMixin**
Bases: object

current_site = <graphene.types.field.Field object>

resolve_current_site(*info*)

wagtail_graphql.query_mixins.snippets module

wagtail_graphql.query_mixins.snippets.**SnippetQueryMixin**
alias of wagtail_graphql.query_mixins.snippets.EmptySnippetQueryMixin

wagtail_graphql.query_mixins.snippets.**create_query_mixin**()
Create a query mixin dynamically.

wagtail_graphql.query_mixins.snippets.**get_snippets_attributes_by_app**()

wagtail_graphql.query_mixins.snippets.**get_snippets_by_app_type**()

Module contents

class wagtail_graphql.query_mixins.**CurrentSiteMixin**
Bases: object

current_site = <graphene.types.field.Field object>

resolve_current_site(*info*)

class wagtail_graphql.query_mixins.**ImageQueryMixin**
Bases: object

images = <wagtail_graphql.types.structures.QuerySetList object>

resolve_images(*info*, ***kwargs*)

class wagtail_graphql.query_mixins.**PageQueryMixin**
Bases: object

Meta
alias of create_query_mixin.<locals>.PageQueryMixinMeta

pages = <graphene.types.field.Field object>

resolve_pages(***kwargs*)

wagtail_graphql.query_mixins.**SnippetQueryMixin**
alias of wagtail_graphql.query_mixins.snippets.EmptySnippetQueryMixin

class wagtail_graphql.query_mixins.**DocumentQueryMixin**
Bases: object

documents = <wagtail_graphql.types.structures.QuerySetList object>

resolve_documents(*info*, ***kwargs*)

wagtail_graphql.types package

Submodules

wagtail_graphql.types.base module

wagtail_graphql.types.base.**create_model_type**(*model*, *fields*, *meta_attrs=None*)
Create a generic GraphQL type for a Django model.

Parameters

- **model** – Django model.
- **fields** – A list of *wagtail_graphql.models.GraphQLField* (page 25) instances to be used on the type.
- **meta_attrs** – Additional meta attributes to be passed to the new GraphQL object type.

wagtail_graphql.types.collection module

class wagtail_graphql.types.collection.**CollectionObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of the Wagtail's Collection model.

images = <wagtail_graphql.types.structures.QuerySetList object>

resolve_images(*info*, ***kwargs*)
Resolve images belonging to a particular collection if privacy of the collection allows.

wagtail_graphql.types.documents module

class wagtail_graphql.types.documents.**DocumentObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

Represent the Wagtail's Document model as a GraphQL type.

resolve_url(*info*, *absolute*)

url = <graphene.types.scalars.String object>

wagtail_graphql.types.images module

class wagtail_graphql.types.images.**ImageInterface**(**args*, ***kwargs*)
Bases: graphene.types.interface.Interface

GraphQL interface for image object types.

focal_point_height = <graphene.types.scalars.Int object>

focal_point_width = <graphene.types.scalars.Int object>

focal_point_x = <graphene.types.scalars.Int object>

focal_point_y = <graphene.types.scalars.Int object>

height = <graphene.types.scalars.Int object>

id = <graphene.types.scalars.ID object>

rendition = <graphene.types.field.Field object>

resolve_id(*info*)

resolve_rendition(*info*, *rendition_filter*)
Resolve an image rendition with a specified Wagtail's image rendition filter.

Example:

```
query {
    images {
        rendition(filter: "fill-200x200") {
            url
        }
    }
}
```

title = <graphene.types.scalars.String object>

width = <graphene.types.scalars.Int object>

class wagtail_graphql.types.images.**ImageObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of Wagtail's image model.

class wagtail_graphql.types.images.**RenditionInterface**(**args*, ***kwargs*)
Bases: graphene.types.interface.Interface

GraphQL interface for rendition object types.

alt = <graphene.types.scalars.String object>

filter_spec = <graphene.types.scalars.String object>

height = <graphene.types.scalars.Int object>

id = <graphene.types.scalars.ID object>

resolve_id(*info*)

resolve_url(*info*, *absolute*)
Resolve to an absolute URL if necessary.

url = <graphene.types.scalars.String object>

width = <graphene.types.scalars.Int object>

class wagtail_graphql.types.images.**RenditionObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of the image rendition model.

wagtail_graphql.types.images.**get_allowed_rendition_filters**()

wagtail_graphql.types.images.**get_default_rendition_filter**()

wagtail_graphql.types.pages module

class wagtail_graphql.types.pages.**PageInterface**(**args*, ***kwargs*)
Bases: graphene.types.interface.Interface

Set basic fields exposed on every page object.

ancestors = <wagtail_graphql.types.structures.QuerySetList object>

children = <wagtail_graphql.types.structures.QuerySetList object>

depth = <graphene.types.scalars.Int object>

descendants = <wagtail_graphql.types.structures.QuerySetList object>

id = <graphene.types.scalars.ID object>

next_siblings = <wagtail_graphql.types.structures.QuerySetList object>

page_type = <graphene.types.scalars.String object>

parent = <graphene.types.field.Field object>

previous_siblings = <wagtail_graphql.types.structures.QuerySetList object>

resolve_ancestors(*info*, ***kwargs*)

resolve_children(*info*, ***kwargs*)

resolve_descendants(*info*, ***kwargs*)

resolve_next_siblings(*info*, ***kwargs*)

resolve_page_type(*info*)
Resolve a page type in a form of `app.ModelName`.

resolve_parent(*info*, ***kwargs*)

resolve_previous_siblings(*info*, ***kwargs*)

resolve_seo_title(*info*)
Get page's SEO title. Fallback to a normal page's title if absent.

resolve_siblings(*info*, ***kwargs*)

resolve_specific(*info*, ***kwargs*)

resolve_url(*info*)
Resolve a path to a page.

seo_description = <graphene.types.scalars.String object>

seo_title = <graphene.types.scalars.String object>

show_in_menus = <graphene.types.scalars.Boolean object>

siblings = <wagtail_graphql.types.structures.QuerySetList object>

specific = <graphene.types.field.Field object>

title = <graphene.types.scalars.String object>

url = <graphene.types.scalars.String object>

wagtail_graphql.types.pages.**create_page_type**(*model*, *fields*)
Generate a DjangoObjectType subclass for a Wagtail page.

wagtail_graphql.types.scalars module

class wagtail_graphql.types.scalars.**PositiveInt**(**args*, ***kwargs*)
Bases: graphene.types.scalars.Int

GraphQL type for an integer that must be equal or greater than zero.

static parse_literal(*node*)

wagtail_graphql.types.site module

class wagtail_graphql.types.site.**SiteObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of the Wagtail's Site model.

name = <graphene.types.scalars.String object>

resolve_name(*info*)
Map Site.site_name to *name* (page 21) for convenience.

wagtail_graphql.types.snippets module

wagtail_graphql.types.snippets.**create_snippet_type**(*model*, *fields*)
Generate a DjangoObjectType for a Wagtail page.

wagtail_graphql.types.streamfields module

class wagtail_graphql.types.streamfields.**StreamField**(**args*, ***kwargs*)
Bases: graphene.types.json.JSONString

Scalar used to represent a Wagtail's StreamField value.

static serialize(*value*)

class wagtail_graphql.types.streamfields.**StreamFieldSerializer**(*request=None*, *absolute_urls=None*, *rendition_filter='width-1200'*)
Bases: object

serialize(*block*)

serialize_block_value(*block*, *value*)

serialize_bound_block(*block*)

serialize_list_block(*block*, *value*)

serialize_stream_block(*stream_block*)

serialize_struct_block(*value*)

wagtail_graphql.types.streamfields.**convert_rich_text**(*source*, *request*, *absolute*)

wagtail_graphql.types.structures module

class wagtail_graphql.types.structures.**QuerySetList**(*of_type*, **args*, ***kwargs*)
Bases: graphene.types.structures.List

List type with arguments used by Django's query sets.

This list setts the following arguments on itself:

- id
- limit

- offset
- search_query
- order

Parameters

- **enable_limit** (*bool*) – Enable limit argument.
- **enable_offset** (*bool*) – Enable offset argument.
- **enable_search** (*bool*) – Enable search query argument.
- **enable_order** (*bool*) – Enable ordering via query argument.

class wagtail_graphql.types.structures.**TagList**(**args*, ***kwargs*)
Bases: graphene.types.json.JSONString

A tag list from the TaggableManager.

static serialize(*value*)

Module contents

class wagtail_graphql.types.**CollectionObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of the Wagtail's Collection model.

images = <wagtail_graphql.types.structures.QuerySetList object>

resolve_images(*info*, ***kwargs*)
Resolve images belonging to a particular collection if privacy of the collection allows.

class wagtail_graphql.types.**DocumentObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

Represent the Wagtail's Document model as a GraphQL type.

resolve_url(*info*, *absolute*)

url = <graphene.types.scalars.String object>

class wagtail_graphql.types.**ImageInterface**(**args*, ***kwargs*)
Bases: graphene.types.interface.Interface

GraphQL interface for image object types.

focal_point_height = <graphene.types.scalars.Int object>

focal_point_width = <graphene.types.scalars.Int object>

focal_point_x = <graphene.types.scalars.Int object>

focal_point_y = <graphene.types.scalars.Int object>

height = <graphene.types.scalars.Int object>

id = <graphene.types.scalars.ID object>

rendition = <graphene.types.field.Field object>

resolve_id(*info*)

resolve_rendition(*info*, *rendition_filter*)
Resolve an image rendition with a specified Wagtail's image rendition filter.

Example:

```
query {
    images {
        rendition(filter: "fill-200x200") {
            url
        }
    }
}
```

title = <graphene.types.scalars.String object>

width = <graphene.types.scalars.Int object>

class wagtail_graphql.types.**ImageObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of Wagtail's image model.

class wagtail_graphql.types.**PageInterface**(**args*, ***kwargs*)
Bases: graphene.types.interface.Interface

Set basic fields exposed on every page object.

ancestors = <wagtail_graphql.types.structures.QuerySetList object>

children = <wagtail_graphql.types.structures.QuerySetList object>

depth = <graphene.types.scalars.Int object>

descendants = <wagtail_graphql.types.structures.QuerySetList object>

id = <graphene.types.scalars.ID object>

next_siblings = <wagtail_graphql.types.structures.QuerySetList object>

page_type = <graphene.types.scalars.String object>

parent = <graphene.types.field.Field object>

previous_siblings = <wagtail_graphql.types.structures.QuerySetList object>

resolve_ancestors(*info*, ***kwargs*)

resolve_children(*info*, ***kwargs*)

resolve_descendants(*info*, ***kwargs*)

resolve_next_siblings(*info*, ***kwargs*)

resolve_page_type(*info*)
Resolve a page type in a form of app.ModelName.

resolve_parent(*info*, ***kwargs*)

resolve_previous_siblings(*info*, ***kwargs*)

resolve_seo_title(*info*)
Get page's SEO title. Fallback to a normal page's title if absent.

resolve_siblings(*info*, ***kwargs*)

resolve_specific(*info*, ***kwargs*)

resolve_url(*info*)
Resolve a path to a page.

seo_description = <graphene.types.scalars.String object>

seo_title = <graphene.types.scalars.String object>

show_in_menus = <graphene.types.scalars.Boolean object>

siblings = <wagtail_graphql.types.structures.QuerySetList object>

specific = <graphene.types.field.Field object>

title = <graphene.types.scalars.String object>

url = <graphene.types.scalars.String object>

class wagtail_graphql.types.**PositiveInt**(**args*, ***kwargs*)
Bases: graphene.types.scalars.Int

GraphQL type for an integer that must be equal or greater than zero.

static parse_literal(*node*)

class wagtail_graphql.types.**SiteObjectType**(**args*, ***kwargs*)
Bases: graphene_django.types.DjangoObjectType

GraphQL representation of the Wagtail's Site model.

name = <graphene.types.scalars.String object>

resolve_name(*info*)
Map Site.site_name to *name* (page 24) for convenience.

class wagtail_graphql.types.**StreamField**(**args*, ***kwargs*)
Bases: graphene.types.json.JSONString

Scalar used to represent a Wagtail's StreamField value.

static serialize(*value*)

class wagtail_graphql.types.**QuerySetList**(*of_type*, **args*, ***kwargs*)
Bases: graphene.types.structures.List

List type with arguments used by Django's query sets.

This list setts the following arguments on itself:

- id
- limit
- offset
- search_query
- order

Parameters

- **enable_limit** (*bool*) – Enable limit argument.
- **enable_offset** (*bool*) – Enable offset argument.
- **enable_search** (*bool*) – Enable search query argument.
- **enable_order** (*bool*) – Enable ordering via query argument.

wagtail_graphql.types.**create_model_type**(*model*, *fields*, *meta_attrs=None*)
Create a generic GraphQL type for a Django model.

Parameters

- **model** – Django model.
- **fields** – A list of *wagtail_graphql.models.GraphQLField* (page 25) instances to be used on the type.
- **meta_attrs** – Additional meta attributes to be passed to the new GraphQL object type.

wagtail_graphql.types.**create_page_type**(*model*, *fields*)
Generate a DjangoObjectType subclass for a Wagtail page.

wagtail_graphql.types.**create_snippet_type**(*model*, *fields*)
Generate a DjangoObjectType for a Wagtail page.

1.3.2 Submodules

1.3.3 wagtail_graphql.apps module

class wagtail_graphql.apps.**WagtailGraphQLConfig**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

name = 'wagtail_graphql'

ready()
Override this method in subclasses to run code when Django starts.

1.3.4 wagtail_graphql.checks module

wagtail_graphql.checks.**check_settings**(*app_configs*, ***kwargs*)

wagtail_graphql.checks.**register_checks**()

1.3.5 wagtail_graphql.converters module

wagtail_graphql.converters.**convert_stream_field**(*field*, *_registry=None*)
Register a GraphQL scalar for the Wagtail's StreamValue.

wagtail_graphql.converters.**convert_tags_to_list_of_strings**(*field*, *_registry=None*)
Register a GraphQL scalar for the TaggableManager used by Wagtail.

wagtail_graphql.converters.**register_converters**()
Register the custom converters in the graphene-django's registry.

1.3.6 wagtail_graphql.models module

class wagtail_graphql.models.**GraphQLEnabledModel**
Bases: object

Subclass used by all the models that are dynamically registered as a GraphQL object type.

class wagtail_graphql.models.**GraphQLField**(*name*, *resolve_func=None*, *graphql_type=None*)

Bases: object

Specify metadata about a model field that is to be registered at a GraphQL object type.

Parameters

- **name** – Name of the field.
- **resolve_func** (*callable*) – A custom resolve function that will be used to resolve data for this field.
- **graphql_type** – Graphene type that will be used by that field.

graphql_type

name

resolve_func

1.3.7 wagtail_graphql.schema module

class wagtail_graphql.schema.**WagtailQuery**(**args*, ***kwargs*)

Bases: graphene.types.objecttype.ObjectType, *wagtail_graphql.query_mixins.pages.PageQueryMixin* (page 16), wagtail_graphql.query_mixins.snippets.EmptySnippetQueryMixin, *wagtail_graphql.query_mixins.images.ImageQueryMixin* (page 16), *wagtail_graphql.query_mixins.sites.CurrentSiteMixin* (page 17), *wagtail_graphql.query_mixins.documents.DocumentQueryMixin* (page 16)

Main GraphQL query used directly by the endpoint.

1.3.8 wagtail_graphql.settings module

wagtail_graphql.settings.**WAGTAIL_GRAPHQL_ADD_SEARCH_HIT = False**

If search query is used in the API, a hit can be added to the Wagtail search Query object by setting this to True.

wagtail_graphql.settings.**WAGTAIL_GRAPHQL_ALLOWED_RENDITION_FILTERS = ('fill-200x200', 'widt**

Specify a list of allowed image rendition filters that can be used in the API. Use ['*'] to disable the check.

wagtail_graphql.settings.**WAGTAIL_GRAPHQL_DEFAULT_RENDITION_FILTER = 'original'**

Specify default Wagtail's image rendition filter used by the API if not specified explicitly.

wagtail_graphql.settings.**WAGTAIL_GRAPHQL_ENABLE_DOCUMENTS = True**

Enable documents list in the GraphQL schema.

wagtail_graphql.settings.**WAGTAIL_GRAPHQL_ENABLE_IMAGES = True**

Enable images list in the GraphQL schema.

wagtail_graphql.settings.**reload_settings**(***kwargs*)

wagtail_graphql.settings.**set_settings**()

1.3.9 wagtail_graphql.utils module

wagtail_graphql.utils.**exclude_invisible_pages**(*request*, *pages*)

Exclude from the QuerySet of pages that are invisible for the current user.

Parameters

- **request** (*django.http.request.HttpRequest*) – Request used to authorize access to pages.
- **pages** – QuerySet containing pages to filter.

wagtail_graphql.utils.**exclude_restricted_collection_members**(*request*, *collection_members*)

Filter out a list of Wagtail collection members (e.g. images or documents) that have collection privacy set accordingly.

Parameters

- **request** (*django.http.request.HttpRequest*) – Request used to authorize access to pages.
- **pages** – QuerySet containing pages to filter.

wagtail_graphql.utils.**get_base_queryset_for_model_or_qs**(*model_or_qs*, *info*, ***kwargs*)

Process a query set before displaying it in the GraphQL query result.

Parameters

- **model_or_qs** – Model or a query set to be transformer.
- **info** – Graphene's info object.
- **kwargs** – Any additional keyword arguments passed from the GraphQL query.

wagtail_graphql.utils.**get_base_queryset_for_page_model_or_qs**(*page_model_or_qs*, *info*, ***kwargs*)

The same as *get_base_queryset_for_model_or_qs()* (page 27), except it adds Wagtail page-specific filters and privacy checks.

Parameters

- **model_or_qs** – Model or a query set to be transformer.
- **info** – Graphene's info object.
- **kwargs** – Any additional keyword arguments passed from the GraphQL query.

wagtail_graphql.utils.**model_to_qs**(*model_or_qs*)

Convert model to a query set if it is not already a query set.

Parameters **model_or_qs** – Model or query set to be cast as a query set.

wagtail_graphql.utils.**resolve_absolute_url**(*url*, *request*, *absolute=True*)

Transform URL to an absolute one if it already is not absolute.

Parameters

- **url** (*str*) – The URL to be resolved, relative or absolute.
- **request** (*django.http.request.HttpRequest*) – Request used to get the domain.
- **absolute** (*bool*) – Set to `True` if value should be returned as absolute.

wagtail_graphql.utils.**resolve_queryset**(*qs*, *info*, *limit=None*, *offset=None*, *search_query=None*, *id=None*, *order=None*, ***kwargs*)

Add limit, offset and search capabilities to the query. This contains argument names used by *QuerySetList* (page 21).

Parameters

- **qs** – Query set to be modified.
- **info** – Graphene's info object.
- **limit** (*int*) – Limit number of objects in the QuerySet.
- **id** – Filter by the primary key.
- **offset** (*int*) – Omit a number of objects from the beggining of the query set
- **search_query** (*str*) – Using wagtail search exclude objects that do not match the search query.
- **order** (*str*) – Use Django ordering format to order the query set.

1.3.10 Module contents

wagtail-graphql-api

class wagtail_graphql.**GraphQLEnabledModel**
Bases: object

Subclass used by all the models that are dynamically registered as a GraphQL object type.

class wagtail_graphql.**GraphQLField**(*name*, *resolve_func=None*, *graphql_type=None*)
Bases: object

Specify metadata about a model field that is to be registered at a GraphQL object type.

Parameters

- **name** – Name of the field.
- **resolve_func** (*callable*) – A custom resolve function that will be used to resolve data for this field.
- **graphql_type** – Graphene type that will be used by that field.

graphql_type

name

resolve_func