# Ariadne Documentation

*Release 0.1*

**Mirumee Software**

**Dec 12, 2018**

# Contents

Ariadne is a Python library for implementing GraphQL servers.

It presents a simple, easy-to-learn and extend API inspired by Apollo Server, with a declaratory approach to type definition that uses a standard Schema Definition Language shared between GraphQL tools, production-ready WSGI middleware, simple dev server for local experiments and an awesome GraphQL Playground for exploring your APIs.

**Note:** While most of GraphQL standard is already supported, Ariadne is currently missing support for the following features: unions, interfaces and inheritance, and subscriptions.

# CHAPTER 1

# Requirements and installation

Ariadne requires Python 3.5 or 3.6 and can be installed from Pypi:

```
pip install ariadne
```

# CHAPTER 2

# Table of contents

## 2.1 Introduction

Welcome to Ariadne!

This guide will introduce you to the basic concepts behind creating GraphQL APIs, and show how Ariadne helps you to implement them with just a little Python code.

At the end of this guide you will have your own simple GraphQL API accessible through the browser, implementing a single field that returns a "Hello" message along with a client's user agent.

Make sure that you've installed Ariadne using `pip install ariadne`, and that you have your favorite code editor open and ready.

### 2.1.1 Defining schema

First, we will with describe what data can be obtained from our API.

In Ariadne this is achieved by defining Python strings with content written in Schema Definition Language (SDL), a special language for declaring GraphQL schemas.

We will start with defining the special type `Query` that GraphQL services use as entry point for all reading operations. Next we will specify a single field on it, named `hello`, and define that it will return a value of type `String`, and that it will never return `null`.

Using the SDL, our `Query` type definition will look like this:

```
type_defs = """
    type Query {
        hello: String!
    }
"""
```

The `type Query { }` block declares the type, `hello` is the field definition, `String` is the return value type, and the exclamation mark following it means that returned value will never be `null`.

### 2.1.2 Resolvers

The resolvers are functions mediating between API consumers and the application's business logic. Every type has fields, and every field has a resolver function that takes care of returning the value that the client has requested.

We want our API to greet clients with a "Hello (user agent)!" string. This means that the `hello` field has to have a resolver that somehow finds the client's user agent, and returns a greeting message from it.

We know that a resolver is a function that returns value, so let's begin with that:

```
def resolve_hello(*_):
    return "Hello..."  # What's next?
```

The above code is perfectly valid, minimal resolver meeting the requirements of our schema. It takes any arguments, does nothing with them and returns blank greeting string.

Real-world resolvers are rarely that simple: they usually read data from some source such as a database, process inputs, or resolve value in the context of a parent object. How should our basic resolver look to resolve a client's user agent?

In Ariadne every field resolver is called with at least two arguments: `obj` parent object, and the query's execution `info` that usually contains the `context` attribute that is GraphQL's way of passing additional information from the application to its query resolvers.

Default GraphQL server implementation provided by Ariadne defines `info.context` as Python `dict` containing a single key named `environ` containing basic request data. We can use this in our resolver:

```
def resolve_hello(_, info):
    request = info.context["environ"]
    user_agent = request.get("HTTP_USER_AGENT", "guest")
    return "Hello, %s!" % user_agent
```

Notice that we are discarding the first argument in our resolver. This is because `resolve_hello` is special type of resolver: it belongs to a field defined on a root type (*Query*), and such fields, by default, have no parent that could be passed to their resolvers. This type of resolver is called a *root resolver*.

Now we need to map our resolver to the `hello` field of type `Query`. To do this, we will create special dictionary where every key is named after a type in the schema. This key's value will, in turn, be another dictionary with keys named after type fields, and with resolvers as values:

```
resolvers = {
    "Query": {
        "hello": resolve_hello
    }
}
```

A dictionary mapping resolvers to schema is called a *resolvers map*.

### 2.1.3 Testing the API

Now we have everything we need to finish our API, with only piece missing being the http server that would receive the HTTP requests, execute GraphQL queries and return responses.

This is where Ariadne comes into play. One of the utilities that Ariadne provides to developers is a WSGI middleware that can also be run as simple http server for developers to experiment with GraphQL locally.

**Warning:** Please never run `GraphQLMiddleware` in production without a proper WSGI container such as uWSGI or Gunicorn.

This middleware can be imported directly from `ariadne` package, so lets add an appropriate import at the beginning of our Python script:

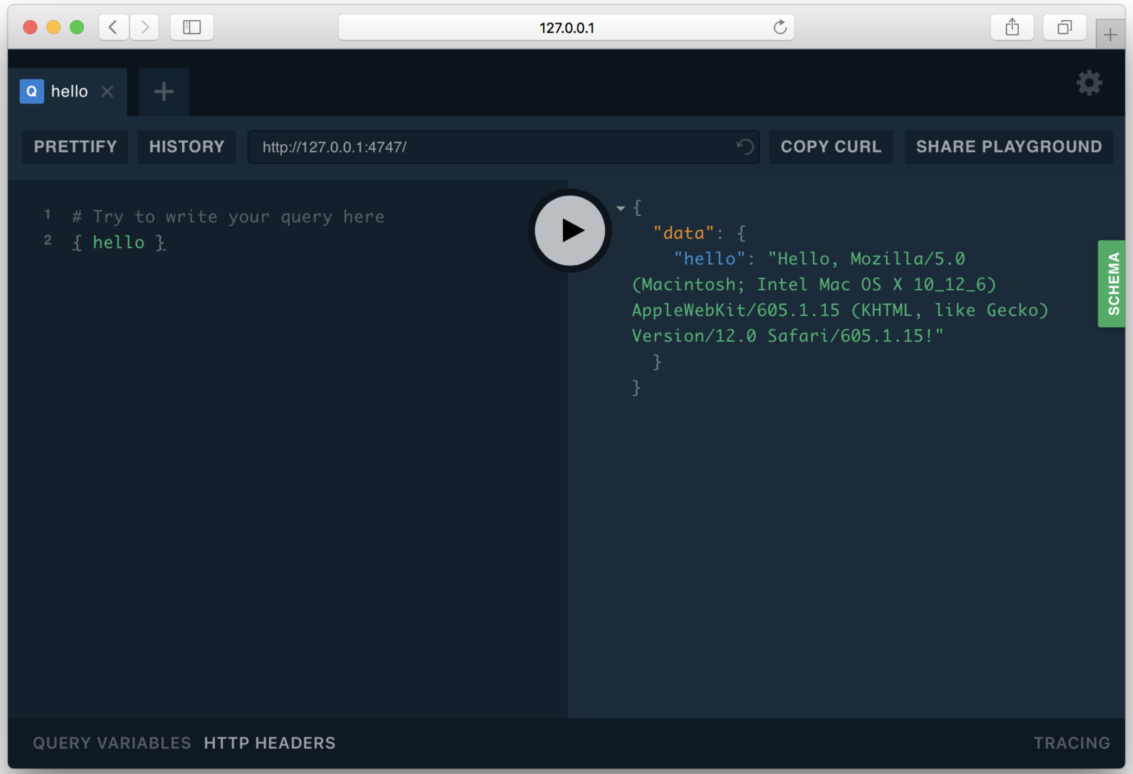
```
from ariadne import GraphQLMiddleware
```

We will now call `GraphQLMiddleware.make_simple_server` class method with `type_defs` and `resolvers` as its arguments to construct a simple dev server that we can then start:

```
print("Visit the http://127.0.0.1:8888 in the browser and say { hello }!")
my_api_server = GraphQLMiddleware.make_simple_server(type_defs, resolvers)
my_api_server.serve_forever()
```

Run your script with `python myscript.py` (remember to replace `myscript.py` with name of your file!). If all is well, you will see a message telling you to visit the http://127.0.0.1:8888 and say `{ hello }`.

This the GraphQL Playground, the open source API explorer for GraphQL APIs. You can enter `{ hello }` query on the left, press the big bright "run" button, and see the result on the right:



```
# Try to write your query here
{ hello }
```

```
{
  "data": {
    "hello": "Hello, Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0 Safari/605.1.15!"
  }
}
```

Your first GraphQL API build with Ariadne is now complete. Congratulations!

### 2.1.4 Completed code

For reference here is complete code of the API from this guide:

```python
from ariadne import GraphQLMiddleware

type_defs = """
    type Query {
        hello: String!
    }
"""


def resolve_hello(_, info):
    request = info.context["environ"]
    user_agent = request.get("HTTP_USER_AGENT", "guest")
    return "Hello, %s!" % user_agent


resolvers = {
    "Query": {
        "hello": resolve_hello
    }
}

print("Visit the http://127.0.0.1:8888 in the browser and say { hello }!")
my_api_server = GraphQLMiddleware.make_simple_server(type_defs, resolvers)
my_api_server.serve_forever()
```

## 2.2 Resolvers

### 2.2.1 Intro

In Ariadne, a resolver is any Python callable that accepts two positional arguments (`obj` and `info`):

```python
def example_resolver(obj: Any, info: ResolveInfo):
    return obj.do_something()


class FormResolver:
    def __call__(self, obj: Any, info: ResolveInfo, **data):
```

`obj` is a value returned by obj resolver. If resolver is *root resolver* (it belongs to the field defined on `Query` or `Mutation`) and GraphQL server implementation doesn't explicitly define value for this field, the value of this argument will be `None`.

`info` is the instance of `ResolveInfo` object specific for this field and query. It defines a special `context` attribute that contains any value that GraphQL server provided for resolvers on the query execution. Its type and contents are application-specific, but it is generally expected to contain application-specific data such as authentication state of the user or http request.

**Note:** `context` is just one of many attributes that can be found on `ResolveInfo`, but it is by far the most commonly used one. Other attributes enable developers to introspect the query that is currently executed and implement new utilities and abstractions, but documenting that is out of Ariadne's scope. Still, if you are interested, you can find the list of all attributes here.

### 2.2.2 Handling arguments

If GraphQL field specifies any arguments, those arguments values will be passed to the resolver as keyword arguments:

```
type_def = """
    type Query {
        holidays(year: Int): [String]!
    }
"""

def resolve_holidays(*_, year=None):
    if year:
        Calendar.get_holidays_in_year(year)
    return Calendar.get_all_holidays()
```

If field argument is marked as required (by following type with `!`, eg. `year: Int!`), you can skip the `=None` in your kwarg:

```
def resolve_holidays(*_, year):
    if year:
        Calendar.get_holidays_in_year(year)
    return Calendar.get_all_holidays()
```

### 2.2.3 Default resolver

In cases when the field has no resolver explicitly provided for it, Ariadne will fall back to the default resolver.

This resolver takes field name and, depending if obj object is `dict` or not, uses *get(field_name)* or *getattr(obj, field_name, None)* to resolve the value that should be returned:

```
type_def = """
    type User {
        username: String!
    }
"""

# We don't have to write username resolver
# for either of those "User" representations:
class UserObj:
    username = "admin"

user_dict = {"username": "admin"}
```

If resolved value is callable, it will be called and its result will be returned to response instead. If field was queried with arguments, those will be passed to the function as keyword arguments, just like how they are passed to regular resolvers:

```
type_def = """
    type User {
        likes: Int!
        initials(length: Int!): String
    }
"""

class UserObj:
    username = "admin"
```

(continues on next page)

(continued from previous page)

```
    def likes(self):
        return count_user_likes(self)

    def initials(self, length)
        return self.name[:length]

user_dict = {
    "likes": lambda obj, *_: count_user_likes(obj),
    "initials": lambda obj, *_, length: obj.username[:length])
}
```

### 2.2.4 Mapping

Ariadne provides `resolve_to` utility function, allowing easy creation of resolvers for fields that are named differently to source attributes (or keys):

```
from ariadne import resolve_to

# ...type and resolver definitions...

resolvers = {
    "User": {
        "firstName": resolve_to("first_name"),
        "role": resolve_to("title"),
    }
}
```

Resolution logic for `firstName` and `role` fields will now be identical to the one provided by default resolver described above. The only difference will be that the resolver will look at different names.

## 2.3 Mutations

So far all examples in this documentation have dealt with `Query` type and reading the data. What about creating, updating or deleting?

Enter the `Mutation` type, `Query`'s sibling that GraphQL servers use to implement functions that change application state.

**Note:** Because there is no restriction on what can be done inside resolvers, technically there's nothing stopping somebody from making `Query` fields act as mutations, taking inputs and executing state-changing logic.

In practice such queries break the contract with client libraries such as Apollo-Client that do client-side caching and state management, resulting in non-responsive controls or inaccurate information being displayed in the UI as the library displays cached data before redrawing it to display an actual response from the GraphQL.

### 2.3.1 Defining mutations

Lets define the basic schema that implements a simple authentication mechanism allowing the client to see if they are authenticated, and to log in and log out:

```
type_def = """
    type Query {
        isAuthenticated: Boolean!
    }

    type Mutation {
        login(username: String!, password: String!): Boolean!
        logout: Boolean!
    }
"""
```

In this example we have the following elements:

`Query` type with single field: boolean for checking if we are authenticated or not. It may appear superficial for the sake of this example, *but Ariadne requires* that your GraphQL API always defines `Query` type.

`Mutation` type with two mutations: `login` mutation that requires username and password strings and returns bool with status, and `logout` that takes no arguments and just returns status.

For the sake of simplicity, our mutations return bools, but really there is no such restriction. You can have a resolver that returns status code, an updated object, or an error message:

```
type_def = """
    type Mutation {
        login(username: String!, password: String!) {
            status: String!
            error: Error
            user: User
        }
    }
"""
```

### 2.3.2 Writing resolvers

Mutation resolvers are no different to resolvers used by other types. They are functions that take `parent` and `info` arguments, as well as any mutation's arguments as keyword arguments. They then return data that should be sent to client as a query result:

```
def resolve_login(_, info, username, password):
    request = info.context["request"]
    user = auth.authenticate(username, password)
    if user:
        auth.login(request, user)
        return True
    return False


def resolve_logout(_, info):
    request = info.context["request"]
    if request.user.is_authenticated:
        auth.logout(request)
        return True
    return False
```

Because `Mutation` is a GraphQL type like others, you can map resolvers to mutations using dict:

```
resolvers {
    "Mutation": {
        "login": resolve_login,
        "logout": resolve_logout,
    }
}
```

### 2.3.3 Inputs

Let’s consider the following type:

```
type_def = """
    type Discussion {
        category: Category!
        poster: User
        postedOn: Date!
        title: String!
        isAnnouncement: Boolean!
        isClosed: Boolean!
    }
"""
```

Imagine a mutation for creating `Discussion` that takes category, poster, title, announcement and closed states as inputs, and creates a new `Discussion` in the database. Looking at the previous example, we may want to define it like this:

```
type_def = """
    type Mutation {
        createDiscussion(category: ID!, title: String!, isAnnouncement: Boolean,␣
↪isClosed: Boolean) {
            status: Boolean!
            error: Error
            discussion: Discussion
        }
    }
"""
```

Our mutation takes only four arguments, but it is already too unwieldy to work with. Imagine adding another one or two arguments to it in future - its going to explode!

GraphQL provides a better way for solving this problem: `input` allows us to move arguments into a dedicated type:

```
type_def = """
    type Mutation {
        createDiscussion(input: DiscussionInput!) {
            status: Boolean!
            error: Error
            discussion: Discussion
        }
    }

    input DiscussionInput {
        category: ID!
        title: String!,
        isAnnouncement: Boolean
        isClosed: Boolean
```

(continues on next page)

(continued from previous page)

```
    }
"""
```

Now when client wants to create a new discussion, they need to provide an `input` object that matches the `DiscussionInput` definition. This input will then be validated and passed to the mutation's resolver as dict available under the `input` keyword argument:

```
def resolve_create_discussion(_, info, input):
    clean_input = {
        "category": input["category"],
        "title": input["title"],
        "is_announcement": input.get("isAnnouncement"),
        "is_closed": input.get("isClosed"),
    }

    try:
        return {
            "status": True,
            "discussion": create_new_discussion(info.context, clean_input),
        }
    except ValidationError as err:
        return {
            "status": False,
            "error: err,
        }
```

Another advantage of `input`-s is that they are reusable. If we later decide to implement another mutation for updating the Discussion, we can do it like this:

```
type_def = """
    type Mutation {
        createDiscussion(input: DiscussionInput!) {
            status: Boolean!
            error: Error
            discussion: Discussion
        }
        updateDiscussion(discussion: ID!, input: DiscussionInput!) {
            status: Boolean!
            error: Error
            discussion: Discussion
        }
    }

    input DiscussionInput {
        category: ID!
        title: String!,
        isAnnouncement: Boolean
        isClosed: Boolean
    }
"""
```

Our `updateDiscussion` mutation will now accept two arguments: `discussion` and `input`:

```
def resolve_update_discussion(_, info, discussion, input):
    clean_input = {
        "category": input["category"],
```

(continues on next page)

(continued from previous page)

```
        "title": input["title"],
        "is_announcement": input.get("isAnnouncement"),
        "is_closed": input.get("isClosed"),
    }

    try:
        return {
            "status": True,
            "discussion": update_discussion(info.context, discussion, clean_input),
        }
    except ValidationError as err:
        return {
            "status": False,
            "error: err,
        }
```

You may wonder why you would want to use `input` instead of reusing already defined type. This is because input types provide some guarantees that regular objects don't: they are serializable, and they don't implement interfaces or unions. However input fields are not limited to scalars. You can create fields that are lists, or even reference other inputs:

```
type_def = """
    input PollInput {
        question: String!,
        options: [PollOptionInput!]!
    }

    input PollOptionInput {
        label: String!
        color: String!
    }
"""
```

Lastly, take note that inputs are not specific to mutations. You can create inputs to implement complex filtering in your `Query` fields.

## 2.4 Error messaging

If you've experimented with GraphQL, you should be familiar that when things don't go according to plan, GraphQL servers include additional key `errors` to the returned response:

```
{
    "error": {
        "errors": [
            {
                "message": "Variable \"$input\" got invalid value {}.\nIn field \
↪"name\": Expected \"String!\", found null.",
                "locations": [
                    {
                        "line": 1,
                        "column": 21
                    }
                ]
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    }
}
```

Your first instinct when planning error messaging may be to use this approach to communicate custom errors (like permission or validation errors) raised by your resolvers.

**Don't do this.**

The `errors` key is, by design, supposed to relay errors to other developers working with the API. Messages present under this key are technical in nature and shouldn't be displayed to your end users.

Instead, you should define custom fields that your queries and mutations will include in result sets, to relay eventual errors and problems to clients, like this:

```
type_def = """
    type Mutation {
        login(username: String!, password: String!) {
            error: String
            user: User
        }
    }
"""
```

Depending on success or failure, your mutation resolver may return either an `error` message to be displayed to the user, or `user` that has been logged in. Your API result handling logic may then interpret the response based on the content of those two keys, only falling back to the main `errors` key to make sure there wasn't an error in query syntax, connection or application.

Likewise, your `Query` resolvers may return a requested object or `None` that will then cause a message such as "Requested item doesn't exist or you don't have permission to see it" to be displayed to the user in place of the requested resource.

## 2.5 Custom scalars

Custom scalars allow you to convert your Python objects to JSON-serializable form in query results, as well as convert those JSON forms back to Python objects back when they are passed as arguments or `input` values.

### 2.5.1 Example read-only scalar

Consider this API defining `Story` type with `publishedOn`:

```
type_defs = """
    type Story {
        content: String
        publishedOn: String
    }
"""
```

The `publishedOn` field's resolver returns instance of type `datetime`, but in API this field is defined as `String`. This means that our datetime will be passed throught the `str()` before being returned to client:

```
{
    "publishedOn": "2018-10-26 17:28:54.416434"
}
```

This may look acceptable, but there are better formats to serialize timestamps for later deserialization on the client, like ISO 8601. We could perform this conversion in our resolver:

```
def resolve_published_on(obj, *_):
    return obj.published_on.isoformat()
```

. . . but now you will have to remember to define custom resolver for every field that receives `datetime` as value. This really adds up the boilerplate to our API, and makes it harder to use abstractions auto-generating the resolvers for you.

Instead, you could tell GraphQL how to serialize dates by defining custom scalar type:

```
type_defs = """
    type Story {
        content: String
        publishedOn: Datetime
    }

    scalar Datetime
"""
```

If you will try to query this field now, you will get error:

```
{
    "errors": [
        {
            "message": "Expected a value of type \"Datetime\" but received: 2018-10-
↪26 17:39:55.502078",
            "path": [
                "publishedOn"
            ]
        }
    ],
}
```

This is because our custom scalar has been defined, but it's currently missing logic for serializing Python values to JSON form.

We need to add special resolver named `serialize` to our `Datetime` scalar that will implement the logic we are expecting:

```
def serialize_datetime(value):
    return value.isoformat()


resolvers = {
    "Datetime": {
        "serialize": serialize_datetime
    }
}
```

Doing so will make GraphQL server use custom logic whenever value that is not `None` is returned from resolver:

```
{
    "publishedOn": "2018-10-26T17:45:08.805278"
}
```

Now we can reuse our custom scalar across the API to serialize `datetime` instances in standardized format that our clients will understand.

### 2.5.2 Scalars as input

What will happen if now we create field or mutation that defines argument of the type `Datetime`? We can find out using basic resolver:

```
type_defs = """
    type Query {
        stories(publishedOn: Datetime): [Story!]!
    }
"""

def resolve_stories(*_, **data):
    print(data.get("publishedOn"))  # what value will "publishedOn" be?
```

`data.get("publishedOn")` will return `False`, because that is fallback value of *read-only* scalars.

To turn our *read-only* scalar into *bidirectional* scalar, we will need to add two functions to `serialize` that was implemented in previous step:

- `parse_value(value)` that will be used when scalar value is passed as part of query `variables`.
- `parse_literal(ast)` that will be used when scalar value is passed as part of query content (eg. `{ stories(publishedOn: "2018-10-26T17:45:08.805278") { ... } }`).

Those functions can be implemented as such:

```
def parse_datetime_value(value):
    try:
        # dateutil is provided by python-dateutil library
        return dateutil.parser.parse(value)
    except (ValueError, TypeError):
        return None


def parse_datetime_literal(ast):
    value = str(ast.value)
    return parse_value(value)  # reuse logic from parse_value


resolvers = {
    "Datetime": {
        "serialize": serialize_datetime,
        "parse_value": parse_datetime_value,
        "parse_literal": parse_datetime_literal,
    }
}
```

There's few things happening in above code, so let's go through them step by step:

There aren't any checks to see if arguments passed to function are `None` because if that is the case, GraphQL server skips our parsing step altogether.

When value is incorrect and either `ValueError` and `TypeError` exceptions are raised, they are silenced and function returns `None` instead. GraphQL server interprets this as sign that entered value is incorrect because it can't be transformed to internal representation, and returns appropriate error to the client. This is preferable approach to raising exceptions from parser, because in such case those exceptions interrupt query execution logic (prohibiting possible partial result), as well as result in error messages possibly **leaking implementation details** to the client.

If value is passed as part of query content, it's `ast` node is instead passed to `parse_datetime_literal` to give it chance to introspect type of node (implementations for those be found here), but we are opting in for just extracting whatever value this *ast* node had, coercing it to `str` and reusing `parse_value`.

## 2.6 Enumeration types

Ariadne supports enumeration types, which are represented as strings in Python logic:

```
from db import get_users

type_defs = """
    type Query{
        users(status: UserStatus): [User]!
    }

    enum UserStatus{
        ACTIVE
        INACTIVE
        BANNED
    }
"""


def resolve_users(*_, status):
    if status == "ACTIVE":
        return get_users(is_active=True)
    if status == "INACTIVE":
        return get_users(is_active=False)
    if status == "BANNED":
        return get_users(is_banned=True)


resolvers = {
    "Query": {
        "users": resolve_users,
    }
}
```

Above example defines resolver that returns list of users based on user status, defined using `UserStatus` enumerable from schema.

Implementing logic validating if `status` value is allowed is not required - this is done on GraphQL level. This query will produce error:

```
{
    users(status: TEST)
}
```

GraphQL failed to find `TEST` in `UserStatus`, and returned error without calling `resolve_users`:

```
{
    "error": {
        "errors": [
            {
                "message": "Argument \"status\" has invalid value TEST.\nExpected␣
↪type \"UserStatus\", found TEST.",
                "locations": [
                    {
                        "line": 2,
                        "column": 14
                    }
                ]
            }
        ]
    }
}
```

## 2.7 Modularization

Ariadne allows you to spread your GraphQL API implementation over multiple Python modules.

Types can be defined as list of strings instead of one large string and resolvers can be defined as list of dicts of dicts for same effect. Here is example of API that moves scalars and `User` to dedicated modules:

```
# graphqlapi.py
from ariadne import GraphQLMiddleware
from . import scalars, users

# Defining Query and Mutation types in root module is optional
# but makes it easier to see what features are implemented by the API
# without having to run and introspect it with GraphQL Playground.
root_type_defs = """
    type Query {
        users: [Users!]
    }
"""

graphql_server = GraphQLMiddleware.make_simple_server(
    [root_type_defs, scalars.type_defs, users.type_defs],
    [scalars.resolvers, users.resolvers]
)

# scalars.py
type_defs = """
    scalar Date
    scalar Datetime
"""

resolvers = {
    "Date": {},
    "Datetime": {},
}

# users.py
type_defs = """
```

(continues on next page)

(continued from previous page)

```
    type User {
        username: String!
        joinedOn: Date!
        lastVisitOn: Datetime
    }
"""


def resolve_users(*_):
    return get_some_users()


resolvers = {
    "User": {},  # User resolvers will be merged with other resolvers
    "Query": {
        "users": resolve_users, # Add resolvers for root type too
    },
}
```

## 2.8 WSGI Middleware

Ariadne provides `GraphQLMiddleware` that realizes following goals:

- is production-ready WSGI middleware that can be added to existing setups to start building GraphQL API quickly.
- it's designed to encourage easy customization through extension.
- provides reference implementation for Ariadne GraphQL server.
- implements *make_simple_server* utility for running local development servers without having to setup full-fledged web framework.

### 2.8.1 Using as Middleware

To add GraphQL API to your project using `GraphQLMiddleware` instantiate it with your existing WSGI application as first argument, type defs as second and resolvers as third:

```
# in wsgi.py
import os

from django.core.wsgi import get_wsgi_application
from ariadne import GraphQLMiddleware
from mygraphql import type_defs, resolvers

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mydjangoproject.settings")

django_application = get_wsgi_application()
application = GraphQLMiddleware(django_application, type_defs, resolvers)
```

Now direct your WSGI container to *wsgi.application*. GraphQL API is available on `/graphql/` by default, but this can be customized by passing path as fourth argument:

```
# GraphQL will now be available on "/graphql-v2/" path
application = GraphQLMiddleware(
    django_application, type_defs, resolvers, "/graphql-v2/"
)
```

### 2.8.2 Customizing context or root

`GraphQLMiddleware` defines two methods that you can redefine in inheriting classes:

`GraphQLMiddleware.`**`get_query_root`**(*environ*, *request_data*)

**Parameters**

- **`environ`** – *dict* representing HTTP request received by WSGI server.
- **`request_data`** – json that was sent as request body and deserialized to *dict*.

**Returns** value that should be passed to root resolvers as first argument.

`GraphQLMiddleware.`**`get_query_context`**(*environ*, *request_data*)

**Parameters**

- **`environ`** – *dict* representing HTTP request received by WSGI server.
- **`request_data`** – json that was sent as request body and deserialized to *dict*.

**Returns** value that should be passed to resolvers as `context` attribute on `info` argument.

Following example shows custom GraphQL middleware that defines its own root and context:

```
from ariadne import GraphQLMiddleware:
from . import DataLoader, MyContext


class MyGraphQLMiddleware(GraphQLMiddleware):
    def get_query_root(self, environ, request_data):
        return DataLoader(environ)

    def get_query_context(self, environ, request_data):
        return MyContext(environ, request_data)
```

### 2.8.3 Using simple server

`GraphQLMiddleware` and inheriting types define class method `make_simple_server` with following signature:

`GraphQLMiddleware.`**`make_simple_server`**(*type_defs*, *resolvers*, *host="127.0.0.1"*, *port=8888*)

**Parameters**

- **`type_defs`** – *str* or list of *str* with SDL for type definitions.
- **`resolvers`** – *dict* or list of *dict* with resolvers.
- **`host`** – *str* of host on which simple server should list.
- **`port`** – *int* of port on which simple server should run.

**Returns** instance of `wsgiref.simple_server.WSGIServer` with middleware running as WSGI app handling *all* incoming requests.

The make_simple_server respects inheritance chain, so you can use it in custom classes inheriting from GraphQLMiddleware:

```
from ariadne import GraphQLMiddleware:
from . import type_defs, resolvers


class MyGraphQLMiddleware(GraphQLMiddleware):
    def get_query_context(self, environ, request_data):
        return MyContext(environ, request_data)

simple_server = MyGraphQLMiddleware(type_defs, resolvers)
simple_server.serve_forever()  # info.context will now be instance of MyContext
```

## 2.9 Custom server example

In addition to simple GraphQL server implementation in form of GraphQLMiddleware, Ariadne provides building blocks for assembling custom GraphQL servers.

### 2.9.1 Creating executable schema

The key piece of the GraphQL server is an *executable schema* - a schema with resolver functions attached to fields.

Ariadne provides a make_executable_schema utility function that takes type definitions as a first argument and a resolvers map as the second, and returns an executable instance of GraphQLSchema:

```
from ariadne import make_executable_schema

type_defs = """
    type Query {
        hello: String!
    }
"""


def resolve_hello(_, info):
    request = info.context["environ"]
    user_agent = request.get("HTTP_USER_AGENT", "guest")
    return "Hello, %s!" % user_agent


resolvers = {
    "Query": {
        "hello": resolve_hello
    }
}

schema = make_executable_schema(type_defs, resolvers)
```

This schema can then be passed to the graphql query executor together with the query and variables:

```
from graphql import graphql

result = graphql(schema, query, variables={})
```

### 2.9.2 Basic GraphQL server with Django

The following example presents a basic GraphQL server using a Django framework:

```
import json

from ariadne import make_executable_schema
from ariadne.constants import PLAYGROUND_HTML
from django.http import (
    HttpResponse, HttpResponseBadRequest, JsonResponse
)
from django.views import View
from graphql import format_error, graphql

type_defs = """
    type Query {
        hello: String!
    }
"""


def resolve_hello(_, info):
    request = info.context["environ"]
    user_agent = request.get("HTTP_USER_AGENT", "guest")
    return "Hello, %s!" % user_agent


resolvers = {
    "Query": {
        "hello": resolve_hello
    }
}

# Create executable schema instance
schema = make_executable_schema(type_defs, resolvers)


# Create GraphQL view
class GraphQLView(View):
    # On GET request serve GraphQL Playground
    # You don't need to provide Playground if you don't want to
    # bet keep on mind this will nor prohibit clients from
    # exploring your API using desktop GraphQL Playground app.
    def get(self, request, *args, **kwargs):
        return HttpResponse(PLAYGROUND_HTML)

    # GraphQL queries are always sent as POSTd
    def post(self, request, *args, **kwargs):
        # Reject requests that aren't JSON
        if request.content_type != "application/json":
            return HttpResponseBadRequest()

        # Naively read data from JSON request
        try:
            data = json.loads(request.data)
        except ValueError:
            return HttpResponseBadRequest()
```

(continues on next page)

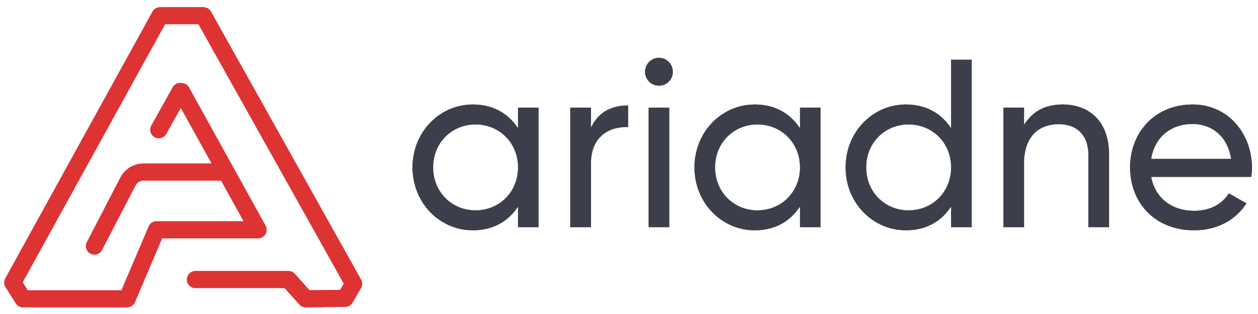
(continued from previous page)

```
        # Check if instance data is not empty and dict
        if not data or not isinstance(data, dict):
            return HttpResponseBadRequest()

        # Check if variables are dict:
        variables = data.get("variables")
        if variables and not isinstance(variables, dict):
            return HttpResponseBadRequest()

        # Execute the query
        result = graphql(
            schema,
            data.get("query"),
            context=request,  # expose request as info.context
            variables=data.get("variables"),
            operation_name=data.get("operationName"),
        )

        # Build valid GraphQL API response
        status = 200
        response = {}
        if result.errors:
            response["errors"] = map(format_error, result.errors)
        if result.invalid:
            status = 400
        else:
            response["data"] = result.data

        # Send response to client
        return JsonResponse(response, status=status)
```

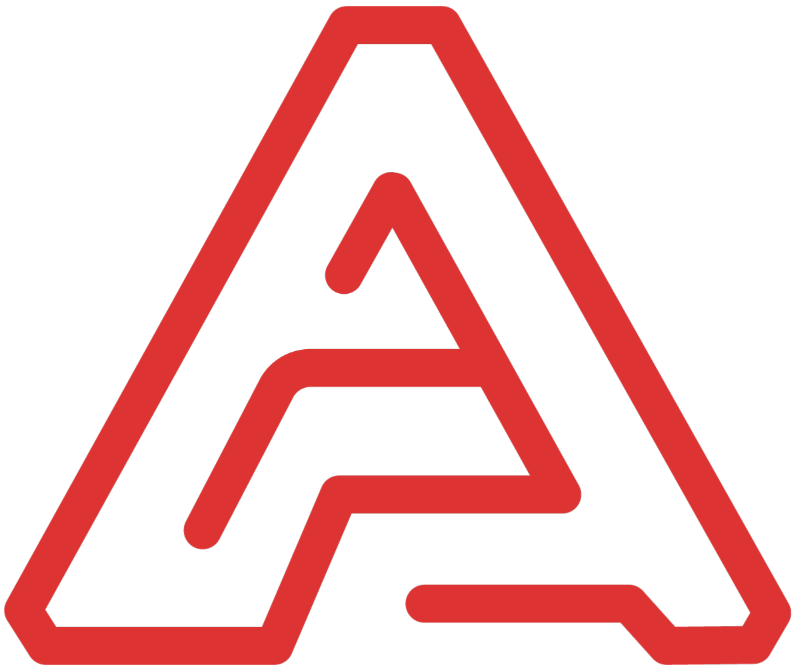
## 2.10 Ariadne logo

Ariadne logo is an “A” shaped labyrinth. If your project uses Ariadne and you want to share the love, feel free to place the logo somewhere on your site and link back to `https://github.com/mirumee/ariadne`:

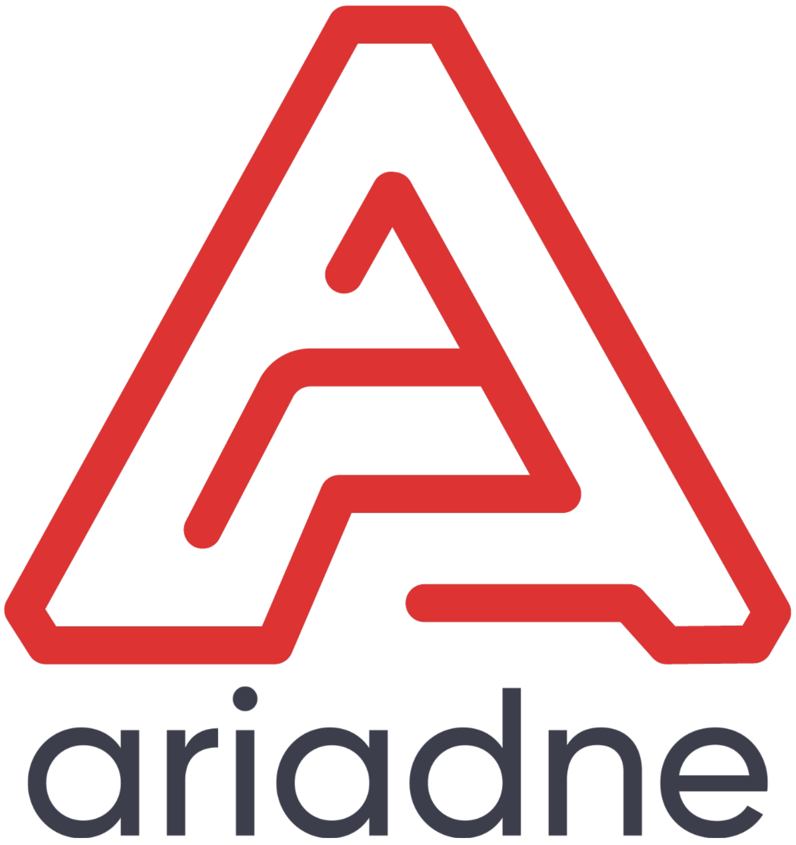
### 2.10.1 Complete logo



### 2.10.2 Labyrinth only



### 2.10.3 Vertical logo



# Python Module Index

# Index