Ariadne Documentation

Release 0.1

Mirumee Software

Jan 07, 2019

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.

CHAPTER 1

Features

- Simple, quick to learn and easy to memorize API.
- Compatibility with GraphQL.js version 14.0.2.
- Queries, mutations and input types.
- Asynchronous resolvers and query execution.
- Custom scalars and enums.
- Defining schema using SDL strings.
- Loading schema from `.graphql` files.
- WSGI middleware for implementing GraphQL in existing sites.
- Opt-in automatic resolvers mapping between *pascalCase* and `snake_case`.
- Build-in simple synchronous dev server for quick GraphQL experimentation and GraphQL Playground.
- Support for Apollo GraphQL extension for Visual Studio Code.
- GraphQL syntax validation via `gql()` helper function. Also provides colorization if Apollo GraphQL extension is installed.

Following features should work but are not tested and documented: unions, interfaces and subscriptions.

CHAPTER 2

Requirements and installation

Ariadne requires Python 3.6 or 3.7 and can be installed from Pypi:

```
pip install ariadne
```

CHAPTER 3

Table of contents

3.1 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.

3.1.1 Defining schema

First, we will 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 by 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 the returned value will never be `null`.

3.1.2 Validating schema

Ariadne provides tiny `gql` utility function that takes single argument: GraphQL string, validates it and raises descriptive `GraphQLSyntaxError`, or returns the original unmodified string if its correct:

```
from ariadne import gql

type_defs = gql("""
    type Query {
        hello String!
    }
""")
```

If we try to run the above code now, we will get an error pointing to our incorrect syntax within our `type_defs` declaration:

```
graphql.error.syntax_error.GraphQLSyntaxError: Syntax Error: Expected :, found Name

GraphQL request (3:19)
    type Query {
        hello String!
              ^
    }
```

Using `gql` is optional; however, without it, the above error would occur during your server's initialization and point to somewhere inside Ariadne's GraphQL initialization logic, making tracking down the error tricky if your API is large and spread across many modules.

3.1.3 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.

At its simplest, resolver is a function that returns a value:

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

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

The 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"]
```

(continues on next page)

(continued from previous page)

```
    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 a 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 use the `ResolverMap` class that maps resolver functions to types in the schema. First, we will update our imports:

```
from ariadne import ResolverMap, gql
```

Next, we will create a resolver map for our only type - `Query`:

```
# Create ResolverMap for Query type defined in our schema...
query = ResolverMap("Query")


# ...and assign our resolver function to its "hello" field.
@query.field("hello")
def resolve_hello(_, info):
    request = info.context["environ"]
    user_agent = request.get("HTTP_USER_AGENT", "guest")
    return "Hello, %s!" % user_agent
```

3.1.4 Testing the API

Now we have everything we need to finish our API, with the missing only piece 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 is a `start_simple_server` that enables developers to experiment with GraphQL locally without the need for a full-fledged HTTP stack or web framework:

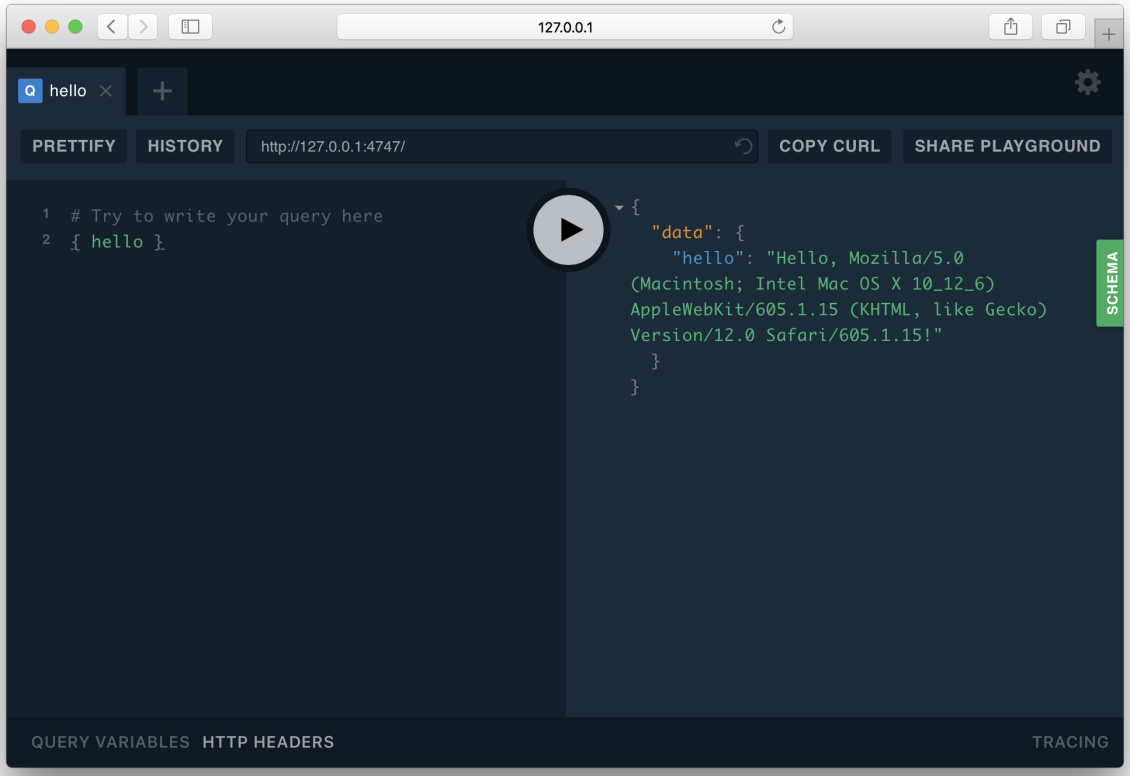
```
from ariadne import start_simple_server
```

We will now call `start_simple_server` with `type_defs` and `query` as its arguments to start a simple dev server:

```
start_simple_server(type_defs, query)
```

Run your script with `python myscript.py` (remember to replace `myscript.py` with the name of your file!). If all is well, you will see a message telling you that the simple GraphQL server is running on the http://127.0.0.1:8888. Open this link in your web browser.

You will see 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:



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

3.1.5 Completed code

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

```
from ariadne import ResolverMap, gql, start_simple_server

type_defs = gql("""
    type Query {
        hello: String!
    }
""")

# Create ResolverMap for Query type defined in our schema...
query = ResolverMap("Query")

# ...and assign our resolver function to its "hello" field.
@query.field("hello")
def resolve_hello(_, info):
    request = info.context["environ"]
    user_agent = request.get("HTTP_USER_AGENT", "guest")
    return "Hello, %s!" % user_agent


start_simple_server(type_defs, query)
```

3.2 Resolvers

3.2.1 Intro

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

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

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

`obj` is a value returned by an obj resolver. If the resolver is a *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 a `GraphQLResolveInfo` 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 `GraphQLResolveInfo`, 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. If you are interested, you can find the list of all attributes here.

3.2.2 Resolver maps

A resolver needs to be bound to a valid type's field in the schema in order to be used during the query execution.

To bind resolvers to schema, Ariadne uses a special `ResolverMap` object that is initialized with single argument - name of the type:

```
from ariadne import ResolverMap

query = ResolverMap("Query")
```

The above `ResolverMap` instance knows that it maps its resolvers to `Query` type, and enables you to assign resolver functions to these type fields. This can be done using the `field` method implemented by the resolver map:

```
from ariadne import ResolverMap

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

query = ResolverMap("Query")

@query.field("hello")
def resolve_hello(*_):
    return "Hello!"
```

`@query.field` decorator is non-wrapping - it simply registers a given function as a resolver for specified field and then returns it as it is. This makes it easy to test or reuse resolver functions between different types or even APIs:

```
user = ResolverMap("User")
client = ResolverMap("Client")

@user.field("email")
@client.field("email")
def resolve_email_with_permission_check(obj, info):
    if info.context.user.is_administrator:
        return obj.email
    return None
```

Alternatively, `query.field` can also be called as regular method:

```
from .resolvers import resolve_email_with_permission_check

user = ResolverMap("User")
user.field("email", resolver=resolve_email_with_permission_check)
```

3.2.3 Handling arguments

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

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

user = ResolverMap("Query")

@query.field("holidays")
def resolve_holidays(*_, year=None):
    if year:
        Calendar.get_holidays_in_year(year)
    return Calendar.get_all_holidays()
```

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

```
@query.field("holidays")
def resolve_holidays(*_, year):
    if year:
        Calendar.get_holidays_in_year(year)
    return Calendar.get_all_holidays()
```

3.2.4 Aliases

You can use `ResolverMap.alias` to quickly make a field an alias for a differently-named attribute on a resolved object:

```
type_def = """
    type User {
```

(continues on next page)

(continued from previous page)

```
        fullName: String
    }
"""

user = ResolverMap("User")
user.alias("fullName", "username")
```

3.2.5 Fallback resolvers

Schema can potentially define numerous types and fields, and defining a resolver or alias for every single one of them can become a large burden.

Ariadne provides two special "fallback resolvers" that scan schema during initialization, and bind default resolvers to fields that don't have any resolver set:

```
from ariadne import fallback_resolvers, start_simple_server
from .typedefs import type_defs
from .resolvers import resolvers

start_simple_server(type_defs, resolvers + [fallback_resolvers])
```

The above example starts a simple GraphQL API using types and resolvers imported from other modules, but it also adds `fallback_resolvers` to the list of resolvers that should be used in creation of schema.

`fallback_resolvers` perform any case conversion and simply seek the attribute named in the same way as the field they are bound to using "default resolver" strategy described in the next chapter.

If your schema uses JavaScript convention for naming its fields (as do all schema definitions in this guide) you may want to instead use the `snake_case_fallback_resolvers` that converts field name to Python's `snake_case` before looking it up on the object:

```
from ariadne import snake_case_fallback_resolvers, start_simple_server
from .typedefs import type_defs
from .resolvers import resolvers

start_simple_server(type_defs, resolvers + [snake_case_fallback_resolvers])
```

3.2.6 Default resolver

Both `ResolverMap.alias` and fallback resolvers use an Ariadne-provided default resolver to implement its functionality.

This resolver takes a target attribute name and (depending if `obj` is `dict` or not) uses either `obj.get(attr_name)` or `getattr(obj, attr_name, None)` to resolve the value that should be returned.

In the below example, both representations of `User` type are supported by the default resolver:

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

(continues on next page)

(continued from previous page)

```
class UserObj:
    username = "admin"

    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])
}
```

3.2.7 Understanding schema binding

When Ariadne initializes GraphQL server, it iterates over a list of objects passed to a `resolvers` argument and calls `bind_to_schema` method of each item with a single argument: instance of `GraphQLSchema` object representing parsed schema used by the server.

`ResolverMap` and the fallback resolvers introduced above don't access the schema until their `bind_to_schema` method is called. It is safe to create, call methods and perform other state mutations on those objects until they are passed to Ariadne.

You can easily implement a custom utility class that can be used in Ariadne:

```
from graphql.type import GraphQLSchema

class MyResolverMap:
    def bind_to_schema(self, schema: GraphQLSchema) -> None:
        pass  # insert custom logic here
```

In later parts of the documentation, other special types will be introduced that internally use `bind_to_schema` to implement their logic.

3.3 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.

3.3.1 Defining mutations

Let's 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.

3.3.2 Writing resolvers

Mutation resolvers are no different than 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 the 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 the `ResolverMap`:

```
from ariadne import ResolverMap
from . import auth_mutations

mutation = ResolverMap("Mutation")
mutation.field("login", resolver=auth_mutations.resolve_login)
mutation.field("logout", resolver=auth_mutations.resolve_logout)
```

3.3.3 Mutation payloads

`login` and `logout` mutations introduced earlier in this guide work, but give very limited feedback to the client: they return either `false` or `true`. The application could use additional information like an error message that could be displayed in the interface after mutation fails, or an updated user state after a mutation completes.

In GraphQL this is achieved by making mutations return special *payload* types containing additional information about the result, such as errors or current object state:

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

    type LoginPayload {
        status: Boolean!
        error: Error
        user: User
    }
"""
```

The above mutation will return a special type containing information about the mutation's status, as well as either an `Error` message or a logged in `User`. In Python this payload can be represented as a simple `dict`:

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

Let's take one more look at the payload's fields:

- `status` makes it easier for the frontend logic to check if mutation succeeded or not.
- `error` contains error message returned by mutation or `null`. Errors can be simple strings, or more complex types that contain additional information for use by the client.

`user` field is especially noteworthy. Modern GraphQL client libraries like Apollo Client implement automatic caching and state management, using GraphQL types to track and automatically update stored objects data whenever a new one is returned from the API.

Consider a mutation that changes a user's username and its payload:

```
type Mutation {
    updateUsername(id: ID!, username: String!): userMutationPayload
}

type UsernameMutationPayload {
    status: Boolean!
    error: Error
    user: User
}
```

Our client code may first perform an *optimistic update* before the API executes a mutation and returns a response to client. This optimistic update will cause an immediate update of the application interface, making it appear fast and responsive to the user. When the mutation eventually completes a moment later and returns updated `user` one of two things will happen:

If the mutation succeeded, the user doesn't see another UI update because the new data returned by mutation was the same as the one set by optimistic update. If mutation asked for additional user fields that are dependant on username but weren't set optimistically (like link or user name changes history), those will be updated too.

If mutation failed, changes performed by an optimistic update are overwritten by valid user state that contains pre-changed username. The client then uses the `error` field to display an error message in the interface.

For the above reasons it is considered a good design for mutations to return updated object whenever possible.

Note: There is no requirement for every mutation to have its own `Payload` type. `login` and `logout` mutations can both define `LoginPayload` as return type. It is up to the developer to decide how generic or specific mutation payloads should be.

3.3.4 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
        ): DiscussionPayload
    }

    type DiscussionPayload {
        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!): DiscussionPayload
    }

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

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` types 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!): DiscussionPayload
        updateDiscussion(discussion: ID!, input: DiscussionInput!): DiscussionPayload
    }

    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 = {
```

(continues on next page)

(continued from previous page)

```
        "category": input["category"],
        "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.

3.4 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.

3.5 Custom scalars

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

3.5.1 Example read-only scalar

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

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

The `publishedOn` field resolver returns an instance of type `datetime`, but in the API this field is defined as `String`. This means that our datetime will be passed through 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. This conversion could be performed in a dedicated resolver:

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

However, the developer now has to remember to define a custom resolver for every field that returns `datetime`. This really adds a boilerplate to the API, and makes it harder to use abstractions auto-generating the resolvers for you.

Instead, GraphQL API can be told how to serialize dates by defining the custom scalar type:

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

    scalar Datetime
"""
```

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

```
{
    "error": "Unexpected token A in JSON at position 0"
}
```

This is because a custom scalar has been defined, but it's currently missing logic for serializing Python values to JSON form and `Datetime` instances are not JSON serializable by default.

We need to add a special serializing resolver to our `Datetime` scalar that will implement the logic we are expecting. Ariadne provides `Scalar` class that enables just that:

```
from ariadne import Scalar

datetime_scalar = Scalar("Datetime")

@datetime_scalar.serializer
def serialize_datetime(value):
    return value.isoformat()
```

Include the `datetime_scalar` in the list of `resolvers` passed to your GraphQL server. Custom serialization logic will now be used when a resolver for the `Datetime` field returns a value other than `None`:

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

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

3.5.2 Scalars as input

What will happen if now we create a field or mutation that defines an argument of the type `Datetime`? We can find out using a 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 print whatever value was passed to the argument, coerced to the respective Python type. For some scalars this may do the trick, but for this one it's expected that input gets converted back to the `datetime` instance.

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

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

Those functions can be implemented as such:

```
@datetime_scalar.value_parser
def parse_datetime_value(value):
    # dateutil is provided by python-dateutil library
    if value:
        return dateutil.parser.parse(value)

@datetime_scalar.literal_parser
def parse_datetime_literal(ast):
    value = str(ast.value)
    return parse_datetime_value(value)  # reuse logic from parse_value
```

There are a few things happening in the above code, so let's go through it step by step:

If the value is passed as part of query's `variables`, it's passed to `parse_datetime_value`.

If the value is not empty, `dateutil.parser.parse` is used to parse it to the valid Python `datetime` object instance that is then returned.

If value is incorrect and either a `ValueError` or `TypeError` exception is raised by the `dateutil.parser.parse` GraphQL server interprets this as a sign that the entered value is incorrect because it can't be transformed to internal representation and returns an automatically generated error message to the client that consists of two parts:

- Part supplied by GraphQL, for example: `Expected type Datetime!, found "invalid string"`
- Exception message: `time data 'invalid string' does not match format '%Y-%m-%d'`

Complete error message returned by the API will look like this:

```
Expected type Datetime!, found "invalid string"; time data 'invalid string' does not␣
↪match format '%Y-%m-%d'
```

Note: You can raise either `ValueError` or `TypeError` in your parsers.

Warning: Because the error message returned by the GraphQL includes the original exception message from your Python code, it may contain details specific to your system or implementation that you may not want to make known to the API consumers. You may decide to catch the original exception with `except (ValueError, TypeError)` and then raise your own `ValueError` with a custom message or no message at all to prevent this from happening.

If a value is specified as part of query content, its `ast` node is instead passed to `parse_datetime_literal` to give Scalar a chance to introspect type of the node (implementations for those be found here).

Logic implemented in the `parse_datetime_literal` may be completely different from that in the `parse_datetime_value`, however, in this example `ast` node is simply unpacked, coerced to `str` and then passed to `parse_datetime_value`, reusing the parsing logic from that other function.

3.6 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,
    }
}
```

The above example defines a resolver that returns a 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 a 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
                    }
                ]
            }
        ]
    }
}
```

3.7 Modularization

Ariadne allows you to spread your GraphQL API implementation over multiple files, with different strategies being available for schema and resolvers.

Internally Ariadne uses special function named `make_executable_schema` for GraphQL server creation. This function is called by all other code that creates GraphQL servers, with values of `type_defs` and `resolvers`. Following guides apply for all Ariadne functions and classes that take those arguments.

3.7.1 Defining schema in `.graphql` files

Recommended way to define schema is by using the `.graphql` files. This approach offers certain advantages:

- First class support from developer tools like Apollo GraphQL plugin for VS Code.
- Easier cooperation and sharing of schema design between frontend and backend developers.
- Dropping whatever python boilerplate code was used for SDL strings.

To load schema from file or directory, you can use the `load_schema_from_path` utility provided by the Ariadne:

```
from ariadne import load_schema_from_path, start_simple_server

# Load schema from file...
schema = load_schema_from_path("/path/to/schema.graphql")

# ...or construct schema from all *.graphql files in directory
schema = load_schema_from_path("/path/to/schema/")
```

(continues on next page)

(continued from previous page)

```
# Start server that can't execute any queries, but allows you to browse your schema
start_simple_server(schema)
```

`load_schema_from_path` validates syntax of every loaded file, and will raise an `ariadne.exceptions.GraphQLFileSyntaxError` if file syntax is found to be invalid.

3.7.2 Defining schema in multiple modules

Because Ariadne expects `type_defs` to be either string or list of strings, it's easy to split types across many string variables in many modules:

```
query = """
    type Query {
        users: [User]!
    }
"""

user = """
    type User {
        id: ID!
        username: String!
        joinedOn: Datetime!
        birthDay: Date!
    }
"""

scalars = """
    scalar Datetime
    scalar Date
"""

start_simple_server([query, user, scalars])
```

The order in which types are defined or passed to `type_defs` doesn't matter, even if those types depend on each other.

3.7.3 Defining resolver maps in multiple modules

Just like `type_defs` can be a string or list of strings, `resolvers` can be a single resolver map instance, or a list of resolver maps:

```
from ariadne import ResolverMap, Scalar

schema = ... # valid schema definition

query = ResolverMap("Query")

user = ResolverMap("User")

datetime_scalar = Scalar("Datetime)
date_scalar = Scalar("Date)

start_simple_server(schema, [query, user, datetime_scalar, date_scalar])
```

The order in which objects are passed to the `resolvers` argument matters. `ResolverMap` and `Scalar` objects replace previously bound resolvers with new ones, when more than one is defined for the same GraphQL type.

Fallback resolvers are safe to put anywhere in the list, because those explicitly avoid replacing already set resolvers.

3.7.4 Reusing resolver functions

`ResolverMap` and `Scalar` objects don't wrap or otherwise change resolver functions in any way, making it easy to reuse functions for many resolver maps, scalars and even fields:

```
from ariadne import ResolverMap

# Create resolver maps for two types
staff = ResolverMap("Staff")
client = ResolverMap("Client")

# Reuse same resolver function for 3 fields
@staff.field("email")
@client.field("email")
@client.field("contactEmail")
def resolve_email(obj, *_):
    return obj.email

# Define new user type and reuse email resolver
reseller = ResolverMap("Reseller")
reseller.field("email", resolver=resolve_email)
```

Note that if you are mixing other decorators with Ariadne's `@type.field` syntax, the order of decorators will matter.

3.8 WSGI Middleware

Ariadne provides `GraphQLMiddleware` that realizes the 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 set up a full-fledged web framework.

3.8.1 Using as Middleware

To add GraphQL API to your project using `GraphQLMiddleware`, instantiate it with your existing WSGI application as a 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
```

(continues on next page)

(continued from previous page)

```
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 a fourth argument:

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

3.8.2 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.

The 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)
```

3.8.3 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** – *ResolverMap* or list of resolver map objects.
- **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
```

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

Note: `ariadne.start_simple_server` is actually a simple shortcut that internally creates HTTP server with `GraphQLMiddleware.make_simple_server`, starts it with *serve_forever*, displays instruction message and handles `KeyboardInterrupt` gracefully.

3.9 Custom server example

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

3.9.1 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 ResolverMap, make_executable_schema

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

(continues on next page)

(continued from previous page)

```
    }
"""

query = ResolverMap("Query")

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

schema = make_executable_schema(type_defs, query)
```

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, variable_values={})
```

3.9.2 Basic GraphQL server with Django

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

```
import json

from ariadne import ResolverMap, 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!
    }
"""

query = ResolverMap("Query")


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


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


# Create GraphQL view
class GraphQLView(View):
```

(continues on next page)

(continued from previous page)

```
    # 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()

        # 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_value=request,  # expose request as info.context
            variable_values=data.get("variables"),
            operation_name=data.get("operationName"),
        )

        # Build valid GraphQL API response
        response = {"data": result.data}
        if result.errors:
            response["errors"] = [format_error(e) for e in result.errors]

        # Send response to client
        return JsonResponse(response)
```

3.10 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`:

3.10.1 Complete logo

3.10.2 Labyrinth only

3.10.3 Vertical logo

Python Module Index

a

Index