# Ariadne Documentation

***Release 0.3***

**Mirumee Software**

**Apr 08, 2019**

# Contents

Ariadne is a Python library for implementing GraphQL servers.

It presents a simple, easy-to-learn and extend API, 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.
- Subscriptions.
- Custom scalars and enums.
- Unions and interfaces.
- Defining schema using SDL strings.
- Loading schema from `.graphql` files.
- WSGI middleware for implementing GraphQL in existing sites.
- Opt-in automatic resolvers mapping between *camelCase* 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.

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. In Ariadne every GraphQL 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 `request` containing a request object. We can use this in our resolver:

```
def resolve_hello(_, info):
    request = info.context["request"]
    user_agent = request.headers.get("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 set our resolver on the `hello` field of type `Query`. To do this, we will use the `QueryType` class that sets resolver functions to the `Query` type in the schema. First, we will update our imports:

```
from ariadne import QueryType, gql
```

Next, we will instantiate the `QueryType` and set our function as resolver for `hello` field using it's field decorator:

```
# Create QueryType instance for Query type defined in our schema...
query = QueryType()


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

### 3.1.4 Making executable schema

Before we can run our server, we need to combine our textual representation of the API's shape with the resolvers we've defined above into what is called an "executable schema". Ariadne provides a function that does this for you:

```
from ariadne import make_executable_schema
```

You pass it your type definitions and resolvers that you want to use:

```
schema = make_executable_schema(type_defs, query)
```

In Ariadne the process of adding the Python logic to GraphQL schema is called *binding to schema*, and special types that can be passed to the `make_executable_schema` second argument are called *bindables*. `QueryType` introduced earlier is one of many *bindables* provided by Ariadne that developers will use when creating their GraphQL APIs. Next chapters will

In our first API we are passing only single instance to the `make_executable_schema`, but most of your future APIs will likely pass list of bindables instead, for example:

```
make_executable_schema(type_defs, [query, user, mutations, fallback_resolvers])
```

---

**Note:** Passing bindables to `make_executable_schema` is not required, but will result in your API handling very limited number of use cases: browsing schema types and, if you've defined root resolver, accessing root type's fields.

---

### 3.1.5 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.

Use an ASGI server like uvicorn, daphne, or hypercorn to serve your application:

```
$ pip install uvicorn
```

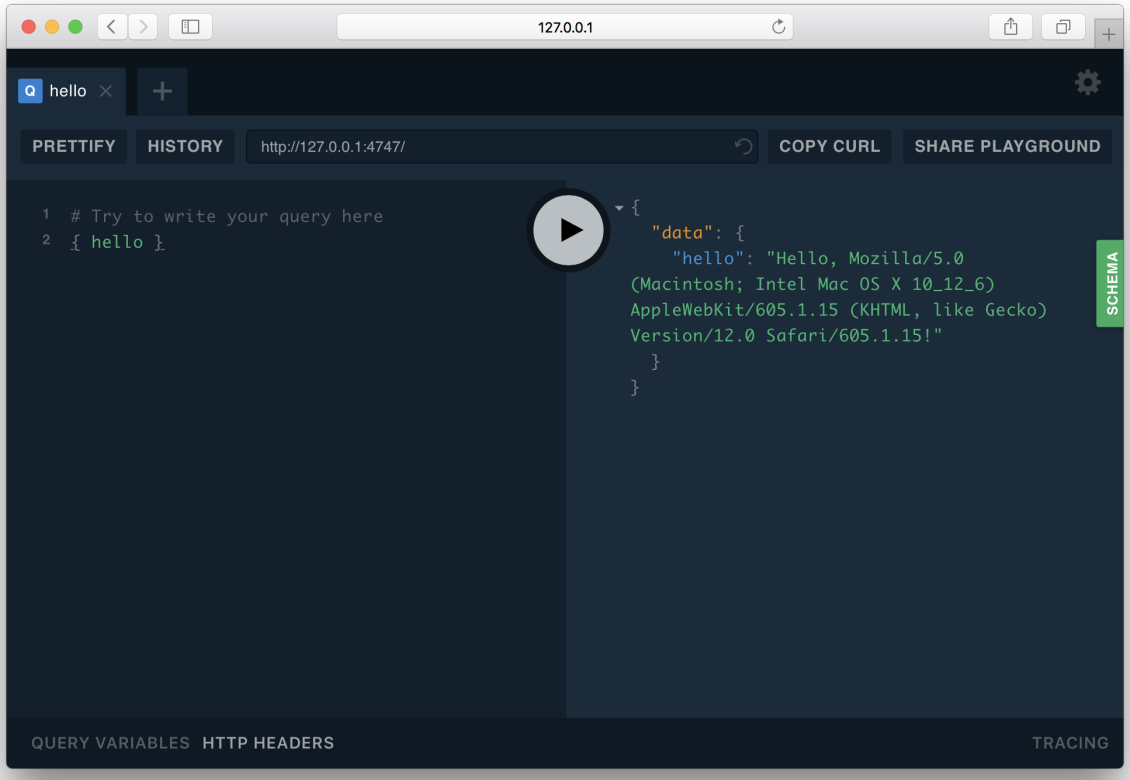
Create a `ariadne.asgi.GraphQL` instance for your schema:

```
from ariadne.asgi import GraphQL

app = GraphQL(schema, debug=True)
```

Run your script with `uvicorn myscript:app` (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://127.0.0.1:8000. 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!

### 3.1.6 Completed code

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

```
from ariadne import QueryType, gql, make_executable_schema
from ariadne.asgi import GraphQL

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

# Create type instance for Query type defined in our schema...
query = QueryType()

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

schema = make_executable_schema(type_defs, query)
app = GraphQL(schema, debug=True)
```

## 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 a parent resolver. If the resolver is a *root resolver* (it belongs to the field defined on `Query`, `Mutation` or `Subscription`) 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 Binding resolvers

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 `ObjectType` class that is initialized with single argument - name of the type defined in the schema:

```
from ariadne import ObjectType

query = ObjectType("Query")
```

The above `ObjectType` 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` decorator implemented by the resolver map:

```
from ariadne import ObjectType

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

query = ObjectType("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 = ObjectType("User")
client = ObjectType("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, `set_field` method can be used to set function as field's resolver:

```
from .resolvers import resolve_email_with_permission_check

user = ObjectType("User")
user.set_field("email", 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 = ObjectType("Query")
```

(continues on next page)

(continued from previous page)

```
@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 `ObjectType.set_alias` to quickly make a field an alias for a differently-named attribute on a resolved object:

```
type_def = """
    type User {
        fullName: String
    }
"""

user = ObjectType("User")
user.set_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, make_executable_schema
from .typedefs import type_defs
from .resolvers import resolvers

schema = make_executable_schema(type_defs, resolvers + [fallback_resolvers])
```

The above example creates executable schema using types and resolvers imported from other modules, but it also adds `fallback_resolvers` to the list of bindables that should be used in creation of the schema.

Resolvers set by `fallback_resolvers` don't 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, make_executable_schema
from .typedefs import type_defs
from .resolvers import resolvers

schema = make_executable_schema(type_defs, resolvers + [snake_case_fallback_
↪resolvers])
```

### 3.2.6 Default resolver

Both `ObjectType.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
    }
"""

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 Query shortcut

Ariadne defines the `QueryType` shortcut that you can use in place of `ObjectType("Query")`:

```
from ariadne import QueryType

type_def = """
    type Query {
        systemStatus: Boolean!
    }
"""

query = QueryType()

@query.field("systemStatus")
def resolve_system_status(*_):
    ...
```

## 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!
    }ś
"""
```

In this example we have the following elements:

The `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.

The `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
```

(continues on next page)

(continued from previous page)

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

You can map resolvers to mutations using the `MutationType`:

```
from ariadne import MutationType
from . import auth_mutations

mutation = MutationType()
mutation.set_field("login", auth_mutations.resolve_login)
mutation.set_field("logout", auth_mutations.resolve_logout)
```

**Note:** `MutationType()` is just a shortcut for `ObjectType("Mutation")`.

`field()` decorator is also available for mapping resolvers to mutations:

```
mutation = MutationType()

@mutation.field("logout")
def resolve_logout(_, info):
    ...
```

### 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"]
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
        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"),
```

(continues on next page)

(continued from previous page)

```
    }

    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 = {
        "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:

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

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.

### 3.4.1 Debugging errors

By default individual `errors` elements contain very limited amount of information about errors occurring inside the resolvers, forcing developer to search application's logs for details about possible error's causes.

Developer experience can be improved by including the `debug=True` in the list of arguments passed to Ariadne's `GraphQL` object:

```
app = GraphQL(schema, debug=True)
```

This will result in each error having additional `exception` key containing both complete traceback, and current context for which the error has occurred:

```
{
    "errors": [
        {
            "message": "'dict' object has no attribute 'build_name'",
            "locations": [
                [
                    3,
                    5
                ]
            ],
            "path": [
                "people",
                0,
                "fullName"
            ],
            "extensions": {
                "exception": {
                    "stacktrace": [
                        "Traceback (most recent call last):",
                        "  File \"/Users/lib/python3.6/site-packages/graphql/
↪execution/execute.py\", line 619, in resolve_field_value_or_error",
                        "    result = resolve_fn(source, info, **args)",
                        "  File \"myapp.py\", line 40, in resolve_person_fullname",
                        "    return get_person_fullname(person)",
                        "  File \"myapp.py\", line 47, in get_person_fullname",
                        "    return person.build_name()",
                        "AttributeError: 'dict' object has no attribute 'build_name'"
                    ],
                    "context": {
                        "person": "{'firstName': 'John', 'lastName': 'Doe', 'age': 21}
↪"
                    }
                }
            }
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

### 3.4.2 Replacing default error formatter

Default error formatter used by Ariadne performs following tasks:

- Formats error by using it's `formatted` property.
- Unwraps `GraphQL` error by accessing its `original_error` property.
- If unwrapped error is available and `debug` argument is set to `True`, update already formatted error to also include `extensions` entry with `exception` dictionary containing `traceback` and `context`.

If you wish to change or customize this behavior, you can set custom function in `error_formatter` of `GraphQL` object:

```
from ariadne import format_error

def my_format_error(error: GraphQLError, debug: bool = False) -> dict:
    if debug:
        # If debug is enabled, reuse Ariadne's formatting logic (not required)
        return format_error(error, debug)

    # Create formatted error data
    formatted = error.formatted
    # Replace original error message with custom one
    formatted["message"] = "INTERNAL SERVER ERROR"
    return formatted

app = GraphQL(schema, error_formatter=my_format_error)
```

## 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 `ScalarType` class that enables just that:

```
from ariadne import ScalarType

datetime_scalar = ScalarType("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 `ScalarType` 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.

### 3.5.3 Configuration reference

In addition to decorators documented above, `ScalarType` provides two more ways for configuring it's logic.

You can pass your functions as values to `serializer`, `value_parser` and `literal_parser` keyword arguments on instantiation:

```
from ariadne import ScalarType
from thirdpartylib import json_serialize_money, json_deserialize_money

money = ScalarType("Money", serializer=json_serialize_money, value_parser=json_
↪deserialize_money)
```

Alternatively you can use `set_serializer`, `set_value_parser` and `set_literal_parser` setters:

```
from ariadne import ScalarType
from thirdpartylib import json_serialize_money, json_deserialize_money

money = ScalarType("Money")
money.set_serializer(json_serialize_money)
money.set_value_parser(json_deserialize_money)
money.set_literal_parser(json_deserialize_money)
```

## 3.6 Enumeration types

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

```
from ariadne import QueryType
from db import get_users

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

(continues on next page)

(continued from previous page)

```
    enum UserStatus{
        ACTIVE
        INACTIVE
        BANNED
    }
"""

query = QueryType()

@query.field("users")
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)
```

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
                    }
                ]
            }
        ]
    }
}
```

### 3.6.1 Mapping to internal values

By default enum values are represented as Python strings, but Ariadne also supports mapping GraphQL enums to custom values.

Imagine posts on social site that can have weights like “standard”, “pinned” and “promoted”:

```
type Post {
    weight: PostWeight
}

enum PostWeight {
    STANDARD
    PINNED
    PROMOTED
}
```

In the database, the application may store those weights as integers from 0 to 2. Normally, you would have to implement a custom resolver transforming GraphQL representation to the integer but, like with scalars, you would have to remember to use this boiler plate on every use.

Ariadne provides an `EnumType` utility class thats allows you to delegate this task to GraphQL server:

```
import enum

from ariadne import EnumType

class PostWeight(enum.IntEnum):
    STANDARD = 1
    PINNED = 2
    PROMOTED = 3

post_weight = EnumType("PostWeight", PostWeight)
```

Include the `post_weight` instance in list of types passed to your GraphQL server, and it will automatically translate Enums between their GraphQL and Python values.

Instead of `Enum` you may use `dict`:

```
from ariadne import EnumType

post_weight = EnumType(
    "PostWeight",
    {
        "STANDARD": 1,
        "PINNED": 2,
        "PROMOTED": 3,
    },
)
```

Both `Enum` and `IntEnum` are supported by the `EnumType`.

## 3.7 Union types

When designing your API, you may run into a situation where you want your field to resolve to one of a few possible types. It may be an `error` field that can resolve to one of many error types, or an activity feed made up of different types.

The most obvious solution may be creating a custom "intermediary" type that would define dedicated fields to different types:

```
type MutationPayload {
    status: Boolean!
```

(continues on next page)

(continued from previous page)

```
    validationError: ValidationError
    permissionError: AccessError
    user: User
}

type FeedItem {
    post: Post
    image: Image
    user: User
}
```

GraphQL provides a dedicated solution to this problem in the form of dedicated `Union` type.

### 3.7.1 Union example

Consider an earlier error example. The union representing one of a possible three error types can be defined in schema like this:

```
union Error = NotFoundError | AccessError | ValidationError
```

This `Error` type can be used just like any other type:

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

Your union will also need a special resolver named *type resolver*. This resolver will we called with an object returned from a field resolver and current context, and should return a string containing the name of an GraphQL type, or `None` if received type is incorrect:

```
def resolve_error_type(obj, *_):
    if isinstance(obj, ValidationError):
        return "ValidationError"
    if isinstance(obj, AccessError):
        return "AccessError"
    return None
```

---

**Note:** Returning `None` from this resolver will result in `null` being returned for this field in your query's result. If field is not nullable, this will cause the GraphQL query to error.

---

Ariadne relies on dedicated `UnionType` class for bindinding this function to Union in your schema:

```
from ariadne import UnionType

error = UnionType("Error")

@error.type_resolver
def resolve_error_type(obj, *_):
    ...
```

If this function is already defined elsewhere (e.g. 3rd party package), you can instantiate the `UnionType` with it as second argument:

```
from ariadne import UnionType
from .graphql import resolve_error_type

error = UnionType("Error", resolve_error_type)
```

Lastly, your `UnionType` instance should be passed to `make_executable_schema` together will other types:

```
schema = make_executable_schema(type_defs, [query, error])
```

### 3.7.2 `__typename` field

Every type in GraphQL has a special `__typename` field that is resolved to a string containing the type's name.

Including this field in your query may simplify implementation of result handling logic in your client:

```
query getFeed {
    feed {
        __typename
        ... on Post {
            text
        }
        ... on Image {
            url
        }
        ... on User {
            username
        }
    }
}
```

Assuming that the feed is a list, the query could produce the following response:

```
{
    "data": {
        "feed": [
            {
                "__typename": "User",
                "username": "Bob"
            },
            {
                "__typename": "User",
                "username": "Aerith"
            },
            {
                "__typename": "Image",
                "url": "http://placekitten.com/200/300"
            },
            {
                "__typename": "Post",
                "text": "Hello world!"
            },
            {
                "__typename": "Image",
                "url": "http://placekitten.com/200/300"
            }
        ]
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Client code could check the `__typename` value of every item in the feed to decide how it should be displayed in the interface.

## 3.8 Interface types

Interface is an abstract GraphQL type that defines certain set of fields and requires other types *implementing* it to also define same fields in order for schema to be correct.

### 3.8.1 Interface example

Consider an application implementing a search function. Search can return items of different type, like `Client`, `Order` or `Product`. For each result it displays a short summary text that is a link leading to a page containing the item's details.

An `Interface` can be defined in schema that forces those types to define the `summary` and `url` fields:

```
interface SearchResult {
    summary: String!
    url: String!
}
```

Type definitions can then be updated to `implement` this interface:

```
type Client implements SearchResult {
    first_name: String!
    last_name: String!
    summary: String!
    url: String!
}

type Order implements SearchResult {
    ref: String!
    client: Client!
    summary: String!
    url: String!
}

type Product implements SearchResult {
    name: String!
    sku: String!
    summary: String!
    url: String!
}
```

GraphQL standard requires that every type implementing the `Interface` also explicitly defines fields from the interface. This is why the `summary` and `url` fields repeat on all types in the example.

Like with the union, the `SearchResult` interface will also need a special resolver named *type resolver*. This resolver will we called with an object returned from a field resolver and current context, and should return a string containing the name of a GraphQL type, or `None` if the received type is incorrect:

```
def resolve_search_result_type(obj, *_):
    if isinstance(obj, Client):
        return "Client"
    if isinstance(obj, Order):
        return "Order"
    if isinstance(obj, Product):
        return "Product"
    return None
```

**Note:** Returning `None` from this resolver will result in `null` being returned for this field in your query's result. If a field is not nullable, this will cause the GraphQL query to error.

Ariadne relies on a dedicated `InterfaceType` class for binding this function to the `Interface` in your schema:

```
from ariadne import InterfaceType

search_result = InterfaceType("SearchResult")

@search_result.type_resolver
def resolve_search_result_type(obj, *_):
    ...
```

If this function is already defined elsewhere (e.g. 3rd party package), you can instantiate the `InterfaceType` with it as a second argument:

```
from ariadne import InterfaceType
from .graphql import resolve_search_result_type

search_result = InterfaceType("SearchResult", resolve_search_result_type)
```

Lastly, your `InterfaceType` instance should be passed to `make_executable_schema` together with other types:

```
schema = make_executable_schema(type_defs, [query, search_result])
```

### 3.8.2 Field resolvers

Ariadne's `InterfaceType` instances can optionally be used to set resolvers on implementing types fields.

`SearchResult` interface from previous section implements two fields: `summary` and `url`. If resolver implementation for those fields is same for multiple types implementing the interface, `InterfaceType` instance can be used to set those resolvers for those fields:

```
@search_result.field("summary")
def resolve_summary(obj, *_):
    return str(obj)


@search_result.field("url")
def resolve_url(obj, *_):
    return obj.get_absolute_url()
```

`InterfaceType` extends the *ObjectType*, so `set_field` and ``set_alias` are also available:

```
search_result.set_field("summary", resolve_summary)
search_result.alias("url", "absolute_url")
```

**Note:** `InterfaceType` assigns the resolver to a field only if that field has no resolver already set. This is different from `ObjectType` that sets resolvers fields if field already has other resolver set.

## 3.9 Subscriptions

Let's introduce a third type of operation. While queries offer a way to query a server once, subscriptions offer a way for the server to notify the client each time new data is available and that no other data will be available for the given request.

This is where the `Subscription` type comes useful. It's similar to `Query` but as each subscription remains an open channel you can send anywhere from zero to millions of responses over its lifetime.

**Warning:** Because of their nature, subscriptions are only possible to implement in asynchronous servers that implement the WebSockets protocol.

*WSGI*-based servers (including Django) are synchronous in nature and *unable* to handle WebSockets which makes them incapable of implementing subscriptions.

If you wish to use subscriptions with Django, consider wrapping your Django application in a Django Channels container and using Ariadne as an *ASGI* server.

### 3.9.1 Defining subscriptions

In schema definition subscriptions look similar to queries:

```
type_def = """
    type Query {}

    type Subscription {
        counter: Int!
    }
"""
```

This example contains:

The `Query` type with no fields. Ariadne requires you to always have a `Query` type.

The `Subscription` type with a single field: `counter` that returns a number.

When defining subscriptions you can use all of the features of the schema such as arguments, input and output types.

### 3.9.2 Writing subscriptions

Subscriptions are more complex than queries as they require us to provide two functions for each field:

A `generator` is a function that yields data we're going to send to the client. It has to implement the `AsyncGenerator` protocol.

A `resolver` that tells the server how to send data to the client. This is similar to the ref:*resolvers we wrote earlier <resolvers>*.

**Note:** Make sure you understand how asynchronous generators work before attempting to use subscriptions.

The signatures are as follows:

```
async def counter_generator(
    obj: Any, info: GraphQLResolveInfo
) -> AsyncGenerator[int, None]:
    for i in range(5):
        await asyncio.sleep(1)
        yield i

def counter_resolver(
    count: int, info: GraphQLResolveInfo
) -> int:
    return count + 1
```

Note that the resolver consumes the same type (in this case `int`) that the generator yields.

Each time our source yields a response, its getting sent to our resolver. The above implementation counts from zero to four, each time waiting for one second before yielding a value.

The resolver increases each number by one before passing them to the client so the client sees the counter progress from one to five.

After the last value is yielded the generator returns, the server tells the client that no more data will be available, and the subscription is complete.

We can map these functions to subscription fields using the `SubscriptionType` class that extends `ObjectType` with support for `source`:

```
from ariadne import SubscriptionType
from . import counter_subscriptions

sub_map = SubscriptionType()
sub_map.set_field("counter", counter_subscriptions.counter_resolver)
sub_map.set_source("counter", counter_subscriptions.counter_generator)
```

You can also use the `source` decorator:

```
@sub_map.source
async def counter_generator(
    obj: Any, info: GraphQLResolveInfo
) -> AsyncGenerator[int, None]:
    ...
```

## 3.10 Documenting a GraphQL schema

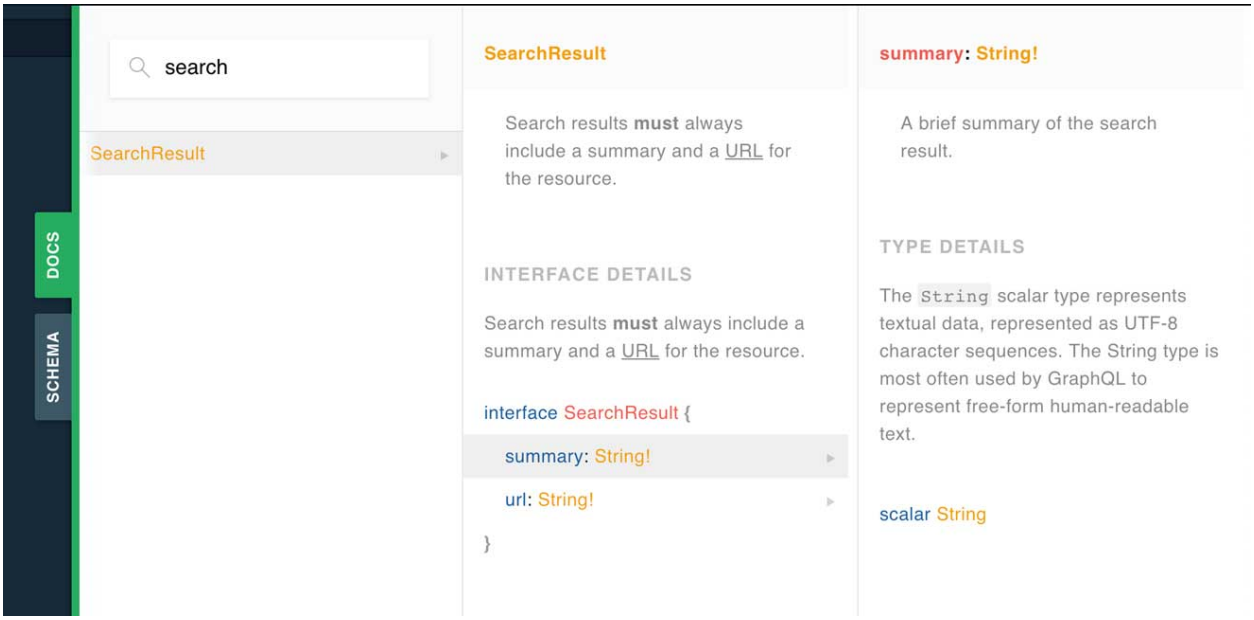
The GraphQL specification includes two features that make documentation and schema exploration easy and powerful. Those features are descriptions and introspection queries.

There is now a rich ecosystem of tools built on top of those features. Some of which include IDE plugins, code generators and interactive API explorers.

### 3.10.1 GraphQL Playground

Ariadne ships with GraphQL Playground, a popular interative API explorer.

GraphQL Playground allows developers and clients to explore the relationships between types across your schema in addition to reading detail about individual types.




### 3.10.2 Descriptions

GraphQL schema definition language supports a special description syntax. This allows you to provide additional context and information alongside your type definitions, which will be accessible both to developers and API consumers.

GraphQL descriptions are declared using a format that feels very similar to Python's *docstrings*:

```
query = '''
    """
    Search results must always include a summary and a URL for the resource.
    """
    interface SearchResult {
        "A brief summary of the search result."
        summary: String!

        "The URL for the resource the search result is describing."
        url: String!
    }
'''
```

Note that GraphQL descriptions also support Markdown (as specified in CommonMark):

```
query = '''
    """
    Search results **must** always include a summary and a
    [URL](https://en.wikipedia.org/wiki/URL) for the resource.
    """
    interface SearchResult {
```

(continues on next page)

(continued from previous page)

```
        # ...
    }
'''
```

### 3.10.3 Introspection Queries

The GraphQL specification also defines a programmatic way to learn about a server's schema and documentation. This is called introspection.

The Query type in a GraphQL schema also includes special introspection fields (prefixed with a double underscore) which allow a user or application to ask for information about the schema itself:

```
query IntrospectionQuery {
    __schema {
        types {
            kind
            name
            description
        }
    }
}
```

The result of the above query might look like this:

```
{
    "__schema": {
        "types": [
            {
                "kind": "OBJECT",
                "name": "Query",
                "description": "A simple GraphQL schema which is well described.",
            }
        ]
    }
}
```

**Note:** Tools like GraphQL Playground use introspection queries internally to provide the live, dynamic experiences they do.

## 3.11 Modularization

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

### 3.11.1 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
from ariadne.asgi import GraphQL

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

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

# Build an executable schema
schema = make_executable_schema(type_defs)

# Create an ASGI app for the schema
app = GraphQL(schema)
```

The above app won't be able to execute any queries but it will allow you to browse your 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.

### 3.11.2 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
"""

schema = make_executable_schema([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.

### 3.11.3 Defining types in multiple modules

Just like `type_defs` can be a string or list of strings, `bindables` can be a single instance, or a list of instances:

```
schema = ... # valid schema definition

from .types import query, user
from .scalars import scalars

resolvers = [query, user]
resolvers += scalars  # [date_scalar, datetime_scalar]

schema = make_executable_schema(schema, resolvers)
```

The order in which objects are passed to the `bindables` argument matters. Most bindables replace previously set resolvers with new ones, when more than one is defined for the same GraphQL type, with `InterfaceType` and fallback resolvers being exceptions to this rule.

## 3.12 Bindables

In Ariadne bindables are special types implementing the logic required for *binding* Python callables and values to the GraphQL schema.

### 3.12.1 Schema validation

Standard bindables provided by the library include validation logic that raises `ValueError` when bindable's GraphQL type is not defined by the schema, is incorrect or missing a field.

### 3.12.2 Creating custom bindable

While Ariadne already provides bindables for all GraphQL types, you can also create your own bindables. Potential use cases for custom bindables include be adding abstraction or boiler plate for mutations or some of types used in the schema.

Custom bindable should extend the `SchemaBindable` base type and define `bind_to_schema` method that will receive single argument, instance of `GraphQLSchema` from *graphql-core-next <https://github.com/graphql-python/graphql-core-next>* when on executable schema creation:

```
from graphql.type import GraphQLSchema
from ariadne import SchemaBindable

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

## 3.13 Local development

### 3.13.1 Starting a local server

You will need an ASGI server such as uvicorn, daphne, or hypercorn:

```
$ pip install uvicorn
```

Pass an instance of `ariadne.asgi.GraphQL` to the server to start your API server:

```
from ariadne import make_executable_schema
from ariadne.asgi import GraphQL
from . import type_defs, resolvers

schema = make_executable_schema(type_defs, resolvers)
app = GraphQL(schema)
```

Run the server pointing it to your file:

```
$ uvicorn example:app
```

## 3.14 ASGI app

Ariadne provides a `GraphQL` class that implements a production-ready ASGI application.

### 3.14.1 Using with an ASGI server

First create an application instance pointing it to the schema to serve:

```
# in myasgi.py
import os

from ariadne import make_executable_schema
from ariadne.asgi import GraphQL
from mygraphql import type_defs, resolvers

schema = make_executable_schema(type_defs, resolvers)
application = GraphQL(schema)
```

Then point an ASGI server such as uvicorn at the above instance.

Example using uvicorn:

```
$ uvicorn myasgi:application
```

### 3.14.2 Customizing context or root

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

`GraphQL.`**`root_value_for_document`**(*query*, *variables*)

**Parameters**

- **`query`** – *DocumentNode* representing the query sent by the client.
- **`variables`** – an optional *dict* representing the query variables.

**Returns** value that should be passed to root resolvers as the parent (first argument).

`GraphQL.`**`context_for_request`**(*request*)

**Parameters** **`request`** – either a *Request* sent by the client or a message sent over a *WebSocket*.

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

The following example shows custom a GraphQL server that defines its own root and context:

```
from ariadne.asgi import GraphQL:
from . import DataLoader, MyContext


class MyGraphQL(GraphQL):
    def root_value_for_document(self, query, variables):
        return DataLoader()

    def context_for_request(self, request):
        return MyContext(request)
```

## 3.15 WSGI app

Ariadne provides a `GraphQL` class that implements a production-ready WSGI application.

Ariadne also provides `GraphQLMiddleware` that allows you to route between a `GraphQL` instance and another WSGI app based on the request path.

### 3.15.1 Using with a WSGI server

First create an application instance pointing it to the schema to serve:

```
# in mywsgi.py
import os

from ariadne import make_executable_schema
from ariadne.wsgi import GraphQL
from mygraphql import type_defs, resolvers

schema = make_executable_schema(type_defs, resolvers)
application = GraphQL(schema)
```

Then point a WSGI server such as uWSGI or Gunicorn at the above instance.

Example using Gunicorn:

```
$ gunicorn mywsgi::application
```

Example using uWSGI:

```
$ uwsgi --http :8000 --wsgi-file mywsgi
```

### 3.15.2 Customizing context or root

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

`GraphQL.`**`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.

GraphQL.**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 a GraphQL server that defines its own root and context:

```
from ariadne.wsgi import GraphQL:
from . import DataLoader, MyContext


class MyGraphQL(GraphQL):
    def get_query_root(self, environ, request_data):
        return DataLoader(environ)

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

### 3.15.3 Using the middleware

To add GraphQL API to your project using `GraphQLMiddleware`, instantiate it with your existing WSGI application as a first argument and your schema as the second:

```
# in wsgi.py
import os

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

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

schema = make_executable_schema(type_defs, resolvers)
django_application = get_wsgi_application()
graphql_application = GraphQL(schema)
application = GraphQLMiddleware(django_application, graphql_application)
```

Now direct your WSGI server to *wsgi.application*. The GraphQL API is available on `/graphql/` by default but this can be customized by passing a different path as the third argument:

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

## 3.16 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.

### 3.16.1 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 bindables as the second, and returns an executable instance of `GraphQLSchema`:

```
from ariadne import QueryType, make_executable_schema

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

query = QueryType()

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

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={})
```

### 3.16.2 Basic GraphQL server with Django

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

```
import json

from ariadne import QueryType, graphql_sync, make_executable_schema
from ariadne.constants import PLAYGROUND_HTML
from django.conf import settings
from django.http import (
    HttpResponseBadRequest, JsonResponse
)
from django.views.decorators.csrf import csrf_exempt
from graphql import graphql_sync

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

query = QueryType()


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

(continues on next page)

(continued from previous page)

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


# Create the view
@csrf_exempt
def graphql_view(request):
    # On GET request serve GraphQL Playground
    # You don't need to provide Playground if you don't want to
    # but keep on mind this will not prohibit clients from
    # exploring your API using desktop GraphQL Playground app.
    if request.method == "GET":
        return HttpResponse(PLAYGROUND_HTML)

    # GraphQL queries are always sent as POST
    if request.method != "POST":
        return HttpResponseBadRequest()

    if request.content_type != "application/json":
        return HttpResponseBadRequest()

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

    # Execute the query
    success, result = graphql_sync(
        schema,
        data,
        context_value=request,  # expose request as info.context
        debug=settings.DEBUG,
    )

    status_code = 200 if success else 400
    # Send response to client
    return JsonResponse(result, status=status_code)
```

## 3.17 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`:

### 3.17.1 Complete logo


### 3.17.2 Labyrinth only

### 3.17.3 Vertical logo

# Python Module Index

## a

# Index