**10 Building GraphQL APIs with Python**

This chapter covers

* Creating GraphQL APIs using the Ariadne web server framework
* Validating request and response payloads
* Creating resolvers for queries and mutations
* Creating resolvers for complex object types, such as union types
* Creating resolvers for custom scalar types and object properties

In chapter 8, we designed a GraphQL API for the products service, and we produced a specification detailing the requirements for the products API. In this chapter, we implement the API according to the specification. To build the API, we’ll use the Ariadne framework, which is one of the most popular GraphQL libraries in the Python ecosystem. Ariadne allows us to leverage the benefits of documentation-driven development by automatically loading data validation models from the specification. We’ll learn to create resolvers, which are Python functions that implement the logic of a query or mutation. We’ll also learn to handle queries that return multiple types. After reading this chapter, you’ll have all the tools you need to start developing your own GraphQL APIs!

The code for this chapter is available in the GitHub repository provided with this book, under the folder ch10. Unless otherwise specified, all the file references within this chapter are relative to the ch10 folder. For example, server.py refers to the ch10/server.py file, and web/schema.py refers to the ch10/web/schema.py file. Also, to ensure all the commands used in this chapter work as expected, use the cd command to move the ch10 folder in your terminal.

**10.1 Analyzing the API requirements**

In this section, we analyze the requirements of the API specification. Before jumping into implementing an API, it’s worth spending some time analyzing the API specification and what it requires. Let’s do this analysis for the products API!

The products API specification is available under ch10/web/products.graphql in the GitHub repository for this book. The specification defines a collection of object types that represent the data we can retrieve from the API and a set of queries and mutations that expose the capabilities of the products service. We must create validation models that faithfully represent the schemas defined in the specification, as well as functions that correctly implement the functionality of the queries and mutations. We’ll work with a framework that can handle schema validation automatically from the specification, so we don’t need to worry about implementing validation models.

Our implementation will focus mainly on the queries and mutations. Most of the queries and mutations defined in the schema return either an array or a single instance of the Ingredient and Product types. Ingredient is simpler since it’s an object type, so we’ll look at queries and mutations that use this type first. Product is the union of the Beverage and Cake types, both of which implement the ProductInterface type. As we’ll see, implementing queries and mutations that return union types is slightly more complex. A query that returns a list of Product objects contains instances of both the Beverage and Cake types, so we need to implement additional functionality that makes it possible for the server to determine which type each element in the list belongs to.

With that said, let’s analyze the tech stack that we’ll use for this chapter, and then move straight into the implementation!

**10.2 Introducing the tech stack**

In this section, we discuss the tech stack that we’ll use to implement the products API. We discuss which libraries are available for implementing GraphQL APIs in Python, and we choose one of them. We also discuss the server framework that we’ll use to run the application.

Since we’re going to implement a GraphQL API, the first thing we want to look for is a good GraphQL server library. GraphQL’s website (<https://graphql.org/code/>) is an excellent resource for finding tools and frameworks for the GraphQL ecosystem. As the ecosystem is constantly evolving, I recommend you check out that website every once in a while for any new additions. The website lists four Python libraries that support GraphQL:

* *Graphene* (<https://github.com/graphql-python/graphene>) is one of the first GraphQL libraries built for Python. It’s battle tested and one of the most widely used libraries.
* *Ariadne* (<https://github.com/mirumee/ariadne>) is a library built for schema-first (or documentation-driven) development. It’s a highly popular framework, and it handles schema validation and serialization automatically.
* *Strawberry* (<https://github.com/strawberry-graphql/strawberry>) is a more recent library that makes it easy to implement GraphQL schema models by offering a clean interface inspired by Python data classes.
* *Tartiflette* (<https://github.com/tartiflette/tartiflette>) is another recent addition to the Python ecosystem that allows you to implement a GraphQL server using a schema-first approach, and it’s built on top of asyncio, which is Python’s core library for asynchronous programming.

For this chapter, we’ll use Ariadne, since it supports a schema-first or documentation-driven development approach, and it’s a mature project. The API specification is already available, so we don’t want to spend time implementing each schema model in Python. Instead, we want to use a library that can handle schema validation and serialization directly from the API specification, and Ariadne can do that.

We’ll run the Ariadne server with the help of Uvicorn, which we encountered in chapters 2 and 6 when we worked with FastAPI. To install the dependencies for this chapter, you can use the Pipfile and Pipfile.lock files available under the ch10 folder in the repository provided with this book. Copy the Pipfile and Pipfile.lock files into your ch10 folder, cd into it, and run the following command:

pipenv install

If you prefer to install the latest versions of Ariadne and Uvicorn, simply run

pipenv install ariadne uvicorn

Now that we have the dependencies installed, let’s activate the environment:

pipenv shell

With all the dependencies installed, now we are ready to start coding, so let’s do it!

**10.3 Introducing Ariadne**

In this section, we introduce the Ariadne framework, and we learn how it works by using a simple example. We’ll learn how to run a GraphQL server with Ariadne, how to load a GraphQL specification, and how to implement a simple GraphQL resolver. As we saw in chapter 9, users interact with GraphQL APIs by running queries and mutations. A GraphQL resolver is a function that knows how to execute one of those queries or mutations. In our implementation, we’ll have as many resolvers as queries and mutations there are in the API specification. As you can see from figure 10.1, resolvers are the pillars of a GraphQL server since it’s through resolvers that we can return actual data to the API users.

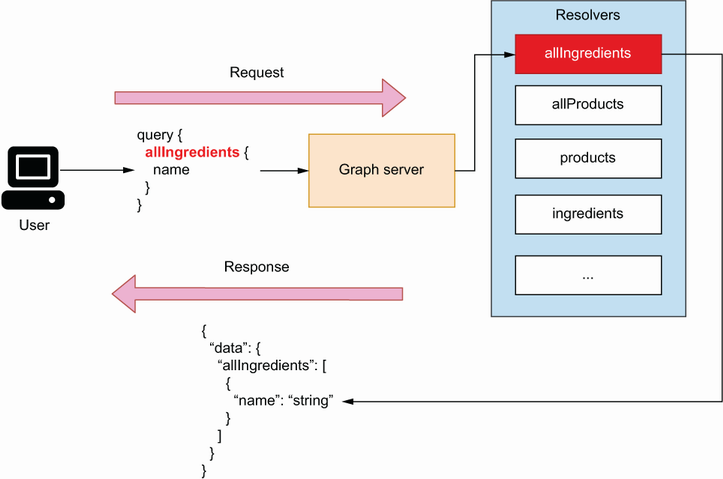


Figure 10.1 To serve data to a user, a GraphQL server uses resolvers, which are functions that know how to build the payload for a given query.

Let’s start by writing a very simple GraphQL schema. Open the server.py file and copy the following content into it:

# file: server.py

schema = '''

type Query {

hello: String

}

'''

We define a variable called schema, and we point it to a simple GraphQL schema. This schema defines only one query, named hello(), which returns a string. The return value of the hello() query is optional, which means null is also a valid return value. To expose this query through our GraphQL server, we need to implement a resolver using Ariadne.

Ariadne can run a GraphQL server from this simple schema definition. How do we do that? First, we need to load the schema using Ariadne’s make\_executable\_ schema() function. make\_executable\_schema() parses the document, validates our definitions, and builds an internal representation of the schema. As you can see in figure 10.2, Ariadne uses the output of this function to validate our data. For example, when we return the payload for a query, Ariadne validates the payload against the schema.

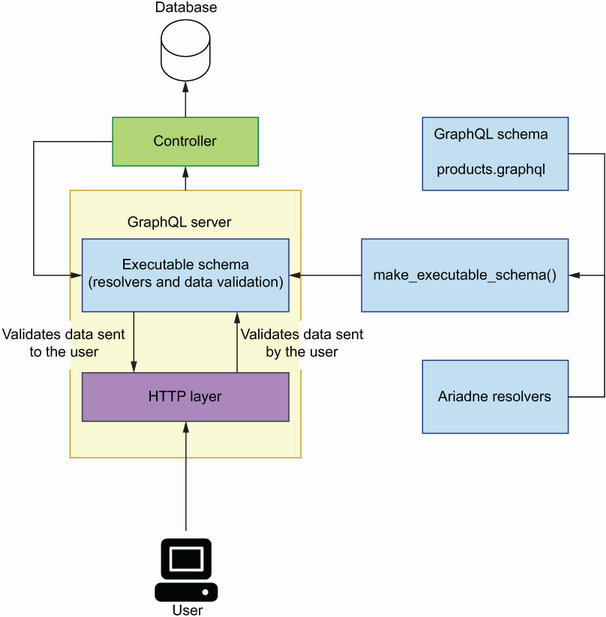


Figure 10.2 To run the GraphQL server with Ariadne, we produce an executable schema by loading the GraphQL schema for the API and a collection of resolvers for the queries and mutations. Ariadne uses the executable schema to validate data the user sent to the server, as well as data sent from the server to the user.

Once we’ve loaded the schema, we can initialize our server using Ariadne’s GraphQL class (listing 10.1). Ariadne provides two implementations of the server: a synchronous implementation, which is available under the ariande.wsgi module, and an asynchronous implementation, which is available under the ariande.asgi module. In this chapter, we’ll use the asynchronous implementation.

Listing 10.1 Initializing a GraphQL server using Ariadne

# file: server.py

from ariadne import make\_executable\_schema

from ariadne.asgi import GraphQL

schema = '''

type Query { ①

hello: String

}

'''

server = GraphQL(make\_executable\_schema(schema), debug=True) ②

① We declare a simple schema.

② We instantiate the GraphQL server.

To run the server, execute the following command from the terminal:

$ uvicorn server:server --reload

Your application will be available on http://localhost:8000. If you head over to that address, you’ll see an Apollo Playground interface to the application. As you can see in figure 10.3, Apollo Playground is similar to GraphiQL, which we learned in chapter 8. On the left-side panel, we write our queries. Write the following query:

{

hello

}

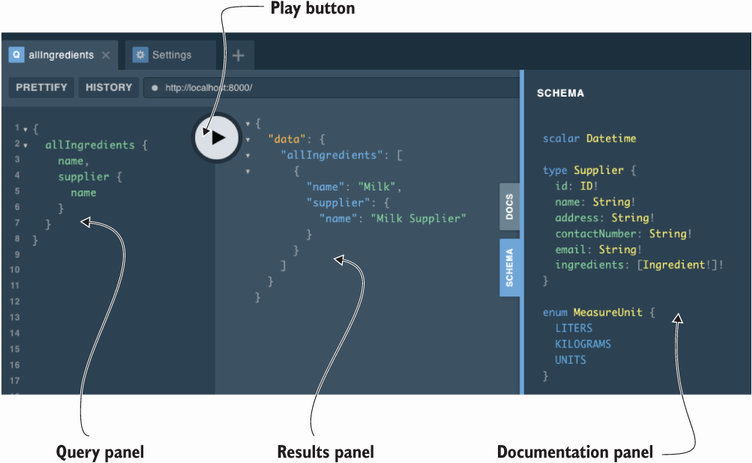


Figure 10.3 The Apollo Playground interface contains a query panel where we execute queries and mutations; a results panel where the queries and mutations are evaluated; and a documentation panel where we can inspect the API schemas.

This query executes the query function that we defined in listing 10.1. If you press the execute button, you’ll get the results of this query on the right-side panel:

{

"data": {

"hello": null

}

}

The query returns null. This shouldn’t come as a surprise, since the return value of the hello() query is a nullable string. How can we make the hello() query return a string? Enter resolvers. *Resolvers* are functions that let the server know how to produce a value for a type or an attribute. To make the hello() query return an actual string, we need to implement a resolver. Let’s create a resolver that returns a string of 10 random characters.

In Ariadne, a resolver is a Python callable (e.g., a function) that takes two positional parameters: obj and info.

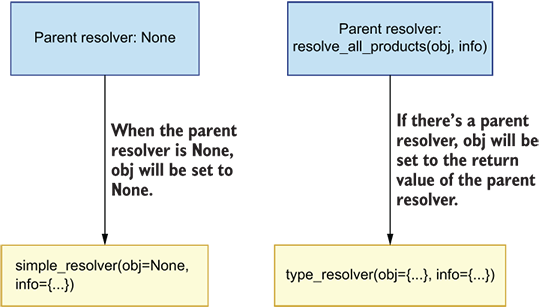
Resolver parameters in Ariadne

Ariadne’s resolvers always have two positional-only parameters, which are commonly called obj and info. The signature of a basic Ariadne resolver is

def simple\_resolver(obj: Any, info: GraphQLResolveInfo):

pass

As you can see in the figure, obj will normally be set to None, unless the resolver has a parent resolver, in which case obj will be set to the value returned by the parent resolver. We encounter the latter case when a resolver doesn’t return an explicit type. For example, the resolver for the allProducts() query, which we’ll implement in section 10.4.4, doesn’t return an explicit type. It returns an object of type Product, which is the union of the Cake and Beverage types. To determine the type of each object, Ariadne needs to call a resolver for the Product type.



When a resolver doesn't have a parent resolver, the obj parameter is set to None. When there's a parent resolver, obj will be set to the value returned by the parent resolver.

The info parameter is an instance of GraphQLResolveInfo, which contains information required to execute a query. Ariadne uses this information to process and serve each request. For the application developer, the most interesting attribute exposed by the info object is info.context, which contains details about the context in which the resolver is called, such as the HTTP context. To learn more about the obj and info objects, check out Ariadne’s documentation: <https://ariadnegraphql.org/docs/resolvers.html>.

A resolver needs to be bound to its corresponding object type. Ariadne provides bindable classes for each GraphQL type:

* ObjectType for object types.
* QueryType for query types. In GraphQL, the query type represents the collection of all queries available in a schema. As we saw in chapter 8 (section 8.8), a query is a function that reads data from a GraphQL server.
* MutationType for mutation types. As we saw in chapter 8 (section 8.9), a mutation is a function that alters the state of the GraphQL server.
* UnionType for union types.
* InterfaceType for interface types.
* EnumType for enumeration types.

Since hello() is a query, we need to bind its resolver to an instance of Ariadne’s QueryType. Listing 10.2 shows how we do that. We first create an instance of the QueryType class and assign it to a variable called query. We then use QueryType’s field() decorator method to bind our resolver, which is available on most of Ariadne’s bindable classes and allows us to bind a resolver to a specific field. By convention, we prefix our resolvers’ names with resolve\_. Ariadne’s resolvers always get two positional-only parameters by default: obj and info. We don’t need to make use of those parameters in this case, so we use a wildcard followed by an underscore (\*\_), which is a convention in Python to ignore a list of positional parameters. To make Ariadne aware of our resolvers, we need to pass our bindable objects as an array to the make\_executable\_ schema() function. The changes go under server.py.

Listing 10.2 Implementing a GraphQL resolver with Ariadne

# file: server.py

import random

import string

from ariadne import QueryType, make\_executable\_schema

from ariadne.asgi import GraphQL

query = QueryType() ①

@query.field('hello') ②

def resolve\_hello(\*\_): ③

return ''.join(

random.choice(string.ascii\_letters) for \_ in range(10) ④

)

schema = '''

type Query { ⑤

hello: String

}

'''

server = GraphQL(make\_executable\_schema(schema, [query]), debug=True) ⑥

① Instance of QueryType

② We bind a resolver for the hello() query using QueryType’s field() decorator.

③ We skip positional-only parameters.

④ We return a list of randomly generated ASCII characters.

⑤ We declare our GraphQL schema.

⑥ Instance of the GraphQL server

Since we’re running the server with the hot reloading flag (--reload), the server automatically reloads once you save the changes to the file. Go back to the Apollo Playground interface in http://127.0.0.1:8000 and run the hello() query again. This time, you should get a random string of 10 characters as a result.

This completes our introduction to Ariadne. You’ve learned how to load a GraphQL schema with Ariadne, how to run the GraphQL server, and how to implement a resolver for a query function. In the rest of the chapter, we’ll apply this knowledge as we build the GraphQL API for the products service.

**10.4 Implementing the products API**

In this section, we’ll use everything we learned in the previous section to build the GraphQL API for the products service. Specifically, you’ll learn to build resolvers for the queries and mutations of the products API, to handle query parameters, and to structure your project. Along the way, we’ll learn additional features of the Ariadne framework and various strategies for testing and implementing GraphQL resolvers. By the end of this section, you’ll be able to build GraphQL APIs for your own microservices. Let the journey begin!

**10.4.1 Laying out the project structure**

In this section, we structure our project for the products API implementation. So far, we’ve included all our code under the server.py file. To implement a whole API, we need to split our code into different files and add structure to the project; otherwise, the codebase would become difficult to read and to maintain. To keep the implementation simple, we’ll use an in-memory representation of our data.

If you followed along with the code in the previous section, delete the code we wrote earlier under server.py, which represents the entry point to our application and therefore will contain an instance of the GraphQL server. We’ll encapsulate the web server implementation within a folder called web/. Create this folder, and within it, create the following files:

* data.py will contain the in-memory representation of our data.
* mutations.py will contain resolvers for the mutations in the products API.
* queries.py will contain resolvers for queries.
* schema.py will contain all the code necessary to load an executable schema.
* types.py will contain resolvers for object types, custom scalar types, and object properties.

The products.graphql specification file also goes under the web folder, since it’s handled by the code under the web/schema.py file. You can copy the API specification from the ch10/web/products.graphql file in the GitHub repository for this book. The directory structure for the products API looks like this:

.

├── Pipfile

├── Pipfile.lock

├── server.py

└── web

├── data.py

├── mutations.py

├── products.graphql

├── queries.py

├── schema.py

└── types.py

The GitHub repository for this book contains an additional module called exceptions.py, which you can check for examples of how to handle exceptions in your GraphQL APIs. Now that we have structured our project, it’s time to start coding!

**10.4.2 Creating an entry point for the GraphQL server**

Now that we have structured our project, it’s time to work on the implementation. In this section, we’ll create the entry point for the GraphQL server. We need to create an instance of Ariadne’s GraphQL class and load an executable schema from the products specification.

As we mentioned in section 10.4.1, the entry point for the products API server lives under server.py. Include the following content in this file:

# file: server.py

from ariadne.asgi import GraphQL

from web.schema import schema

server = GraphQL(schema, debug=True)

Next, let’s create the executable schema under web/schema.py:

# file: web/schema.py

from pathlib import Path

from ariadne import make\_executable\_schema

schema = make\_executable\_schema(

(Path(\_\_file\_\_).parent / 'products.graphql').read\_text()

)

The API specification for the products API is available under the web/products.graphql file. We read the schema file contents and pass them on to Ariadne’s make\_executable\_ schema() function. We then pass the resulting schema object to Ariadne’s GraphQL class to instantiate the server. If you haven’t started the server, you can do it now by executing the following command:

$ uvicorn server:server --reload

Like before, the API is available on http://localhost:8000. If you visit this address again, you’ll see the familiar Apollo Playground UI. At this point, we could try running any of the queries defined in the products API specification; however, most of them will fail since we haven’t implemented any resolvers. For example, if you run the following query

{

allIngredients {

name

}

}

you’ll get the following error message: “Cannot return null for non-nullable field Query.allProducts.” The server doesn’t know how to produce a value for the Ingredient type since we don’t have a resolver for it, so let’s build it!

**10.4.3 Implementing query resolvers**

In this section, we learn to implement query resolvers. As you can see from figure 10.4, a query resolver is a Python function that knows how to return a valid payload for a given query. We’ll build a resolver for the allIngredients() query, which is one of the simplest queries in the products API specification (listing 10.3).

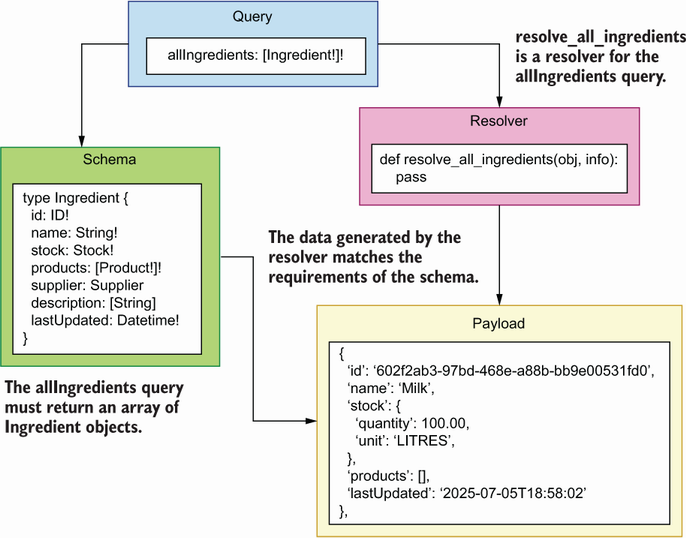


Figure 10.4 GraphQL uses resolvers to serve the query requests sent by the user to the server. A resolver is a Python function that knows how to return a valid payload for a given query.

To implement a resolver for the allIngredients() query, we simply need to create a function that returns a data structure with the shape of the Ingredient type, which has four non-nullable properties: id, name, stock, and products. The stock property is, in turn, an instance of the Stock object type, which, as per the specification, must contain the quantity and unit properties. Finally, the products property must be an array of Product objects. The contents of the array are non-nullable, but an empty array is a valid return value.

Listing 10.3 Specification for the Ingredient type

# file: web/products.graphql

type Stock { ①

quantity: Float! ②

unit: MeasureUnit!

}

type Ingredient {

id: ID!

name: String!

stock: Stock!

products: [Product!]! ③

supplier: Supplier ④

description: [String!]

lastUpdated: Datetime!

}

① We declare the Stock type.

② quantity is a non-nullable float.

③ products is a non-nullable list of products.

④ supplier is a nullable through type that points to the Supplier type.

Let’s add a list of ingredients to the in-memory list representation of our data under the web/data.py file:

# file: web/data.py

from datetime import datetime

ingredients = [

{

'id': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',

'name': 'Milk',

'stock': {

'quantity': 100.00,

'unit': 'LITRES',

},

'supplier': '92f2daae-a4f8-4aae-8d74-51dd74e5de6d',

'products': [],

'lastUpdated': datetime.utcnow(),

},

]

Now that we have some data, we can use it in the allIngredients()’ resolver. Listing 10.4 shows what allIngredients()’ resolver looks like. As we did in section 10.3, we first create an instance of the QueryType class, and we bind the resolver with this class. Since this is a resolver for a query type, the implementation goes under the web/queries.py file.

Listing 10.4 A resolver for the allIngredients() query

# file: web/queries.py

from ariadne import QueryType

from web.data import ingredients

query = QueryType()

@query.field('allIngredients') ①

def resolve\_all\_ingredients(\*\_):

return ingredients ②

① We bind allIngredients()’ resolver using the decorator.

② We return a hardcoded response.

To enable the query resolver, we have to pass the query object to the make\_executable\_ schema() function under web/schema.py:

# file: web/schema.py

from pathlib import Path

from ariadne import make\_executable\_schema

**from web.queries import query**

schema = make\_executable\_schema(

(Path(\_\_file\_\_).parent / 'products.graphql').read\_text(), **[query]**

)

If we go back to the Apollo Playground UI and we run the query

{

allIngredients {

name

}

}

we get a valid payload. The query selects only the ingredient’s name, which in itself is not very interesting, and it doesn’t really tell us whether our current resolver works as expected for other fields. Let’s write a more complex query to test our resolver more thoroughly. The following query selects the id, name, and description of an ingredient, as well as the name of each product it’s related to:

{

allIngredients {

id,

name,

products {

...on ProductInterface {

name

}

},

description

}

}

The response payload to this query is also valid:

{

"data": {

"allIngredients": [

{

"id": " "602f2ab3-97bd-468e-a88b-bb9e00531fd0",

"name": "Milk",

"products": [],

"description": null

}

]

}

}

The products list is empty because we haven’t associated any products with the ingredient, and description is null because this is a nullable field. Now that we know how to implement resolvers for simple queries, in the next section, we’ll learn to implement resolvers that handle more complex situations.

**10.4.4 Implementing type resolvers**

In this section, we’ll learn to implement resolvers for queries that return multiple types. The allIngredients() query is fairly simple since it only returns one type of object: the Ingredient type. Let’s now consider the allProducts() query. As you can see from figure 10.5, allProducts() is more complex since it returns the Product type, which is a union of the Beverage and Cake types, both of which implement the ProductInterface type.

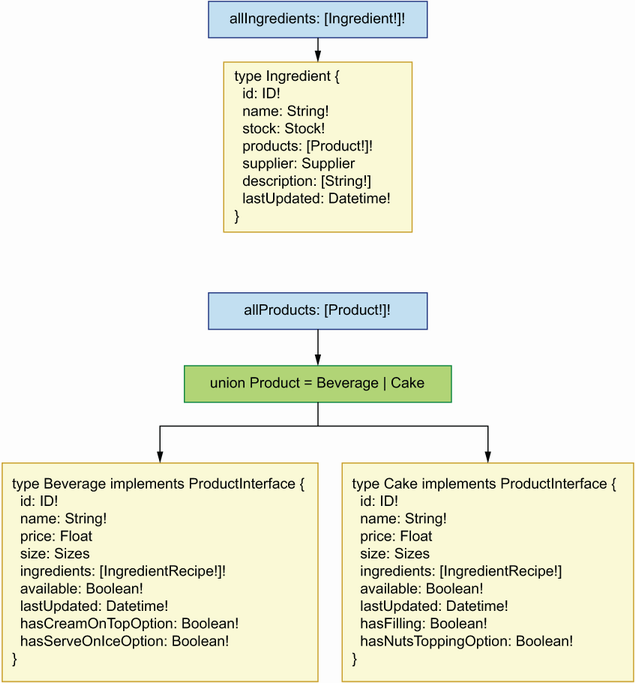


Figure 10.5 The allIngredients() query returns an array of Ingredient objects, while the allProducts() query returns an array of Product objects, where Product is the union of two types: Beverage and Cake.

Let’s begin by adding a list of products to our in-memory list of data under the web/data.py file. We’ll add two products: one Beverage and one Cake. What fields should we include in the products? As you can see in figure 10.6, since Beverage and Cake implement the ProductInterface type, we know they both require an id, a name, a list of ingredients, and a field called available, which signals if the product is available. On top of these common fields inherited from ProductInterface, Beverage requires two additional fields: hasCreamOnTopOption and hasServeOnIceOption, both of which are Booleans. In turn, Cake requires the properties hasFilling and hasNutsToppingOption, which are also Booleans.

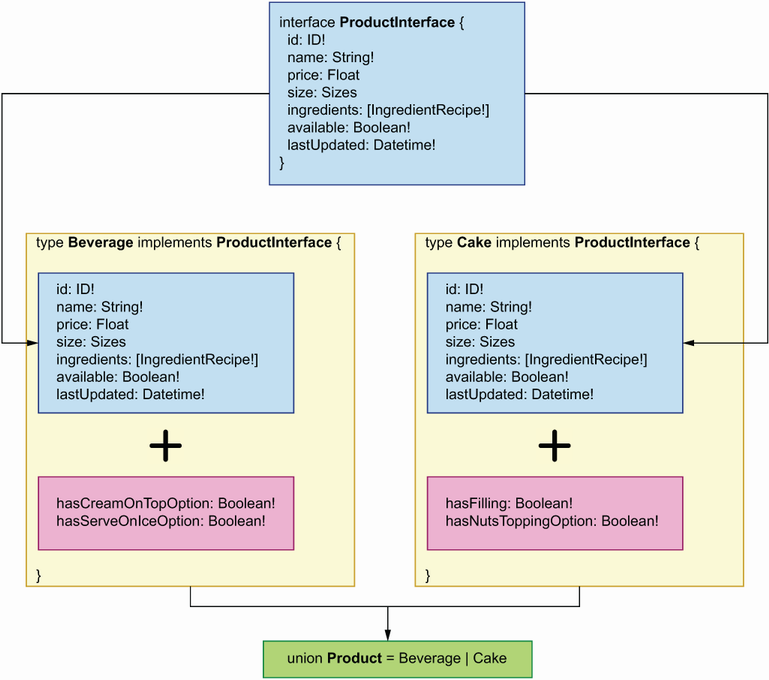


Figure 10.6 Product is the union of the Beverage and the Cake types, both of which implement the ProductInterface type. Since Beverage and Cake implement the same interface, both types share the properties inherited from the interface. In addition to those properties, each type has its own specific properties, such as hasFilling in the case of the Cake type.

Listing 10.5 Resolver for the allProducts() query

# file: web/data.py

...

**products = [**

**{**

**'id': '6961ca64-78f3-41d4-bc3b-a63550754bd8',**

**'name': 'Walnut Bomb',**

**'price': 37.00,**

**'size': 'MEDIUM',**

**'available': False,**

**'ingredients': [**

**{**

**'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',** ①

**'quantity': 100.00,**

**'unit': 'LITRES',**

**}**

**],**

**'hasFilling': False,**

**'hasNutsToppingOption': True,**

**'lastUpdated': datetime.utcnow(),**

**},**

**{**

**'id': 'e4e33d0b-1355-4735-9505-749e3fdf8a16',**

**'name': 'Cappuccino Star',**

**'price': 12.50,**

**'size': 'SMALL',**

**'available': True,**

**'ingredients': [**

**{**

**'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',**

**'quantity': 100.00,**

**'unit': 'LITRES',**

**}**

**],**

**'hasCreamOnTopOption': True,**

**'hasServeOnIceOption': True,**

**'lastUpdated': datetime.utcnow(),**

**},**

**]**

① This ID references the ID of the milk ingredient we added earlier to web/data.py.

Now that we have a list of products, let’s use it in the allProducts()’ resolver.

Listing 10.6 Adding the allProducts() resolver

# file: web/queries.py

from ariadne import QueryType

from web.data import ingredients, **products**

query = QueryType()

...

**@query.field('allProducts')**  ①

**def resolve\_all\_products(\*\_):**

**return products**  ②

① We bind allProducts()’ resolver using the field() decorator.

② We return a hardcoded response.

Let’s run a simple query to test the resolver:

{

allProducts {

...on ProductInterface {

name

}

}

}

If you run this query, you’ll get an error saying that the server can’t determine what types each of the elements in our list are. In these situations, we need a type resolver. As you can see in figure 10.7, a *type resolver* is a Python function that determines what type an object is, and it returns the name of the type.

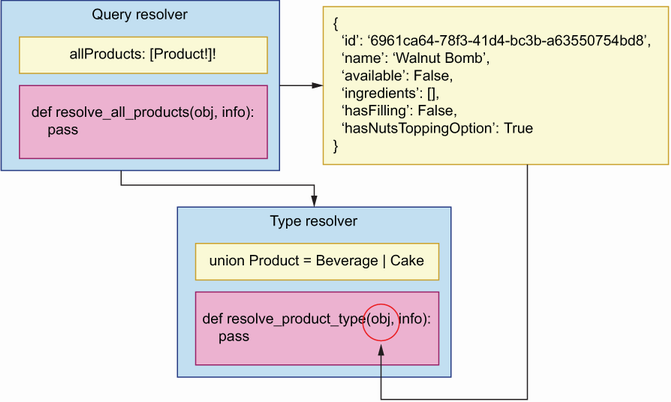


Figure 10.7 A type resolver is a function that determines the type of an object. This example shows how the resolve\_product\_type() resolver determines the type of an object returned by the resolve\_all\_products() resolver.

We need type resolvers in queries and mutations that return more than one object type. In the products API, this affects all queries and mutations that return the Product type, such as allProducts(), addProduct(), and product().

**RETURNING MULTIPLE TYPES** Whenever a query or mutation returns multiple types, you’ll need to implement a type resolver. This applies to queries and mutations that return union types and object types that implement interfaces.

Listing 10.7 shows how we implement a type resolver for the Product type in Ariadne. The type resolver function takes two positional parameters, the first of which is an object. We need to determine the type of this object. As you can see in figure 10.8, since we know that Cake and Beverage have different required fields, we can use this information to determine their types: if the object has a hasFilling property, we know it’s a Cake; otherwise, it’s a Beverage.

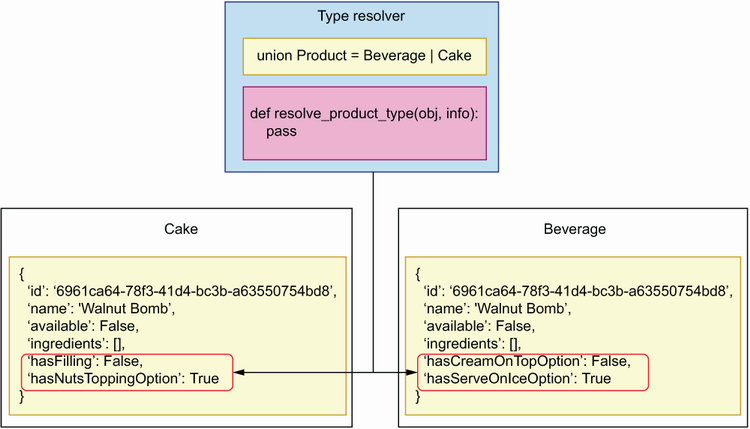


Figure 10.8 A type resolver inspects the properties of a payload to determine its type. In this example, resolve\_product\_type() looks for distinguishing properties that differentiate a Cake from a Beverage type.

The type resolver must be bound to the Product type. Since Product is a union type, we create a bindable object of it using the UnionType class. Ariadne guarantees that the first argument in a resolver is an object, and we inspect this object to resolve its type. We don’t need any other parameters, so we ignore them with Python’s \*\_ syntax, which is standard for ignoring positional parameters. To resolve the type of the object, we check if it has a hasFilling attribute. If it does, we know it’s a Cake object; otherwise, it’s a Beverage. Finally, we pass the product bindable to the make\_executable\_schema() function. Since this is a type resolver, this code goes into the web/types.py.

Listing 10.7 Implementing a type resolver for the Product union type

# file: web/types.py

from ariadne import UnionType

product\_type = UnionType('Product') ①

@product\_type.type\_resolver ②

def resolve\_product\_type(obj, \*\_): ③

if 'hasFilling' in obj:

return 'Cake'

return 'Beverage'

① We create a bindable object for the Product type using the UnionType class.

② We bind Product’s resolver using the resolver() decorator.

③ We capture the resolver’s first positional argument as obj.

To enable the type resolver, we need to add the product object to the make\_executable\_ schema() function under web/schema.py:

# file: web/schema.py

from pathlib import Path

from ariadne import make\_executable\_schema

from web.queries import query

**from web.types import product\_type**

schema = make\_executable\_schema(

(Path(\_\_file\_\_).parent / 'products.graphql').read\_text(),

[query, **product\_type**]

)

Let’s run the allProducts() query again:

{

allProducts {

...on ProductInterface {

name

}

}

}

You’ll now get a successful response. You have just learned to implement type resolvers and to handle queries that return multiple types! In the next section, we continue exploring queries by learning how to handle query parameters.

**10.4.5 Handling query parameters**

In this section, we learn to handle query parameters in the resolvers. Most of the queries in the products API accept filtering parameters, and all the mutations require at least one parameter. Let’s see how we access parameters by studying one example from the products API: the products() query, which accepts an input filter object whose type is ProductsFilter. How do we access this filter object in a resolver?

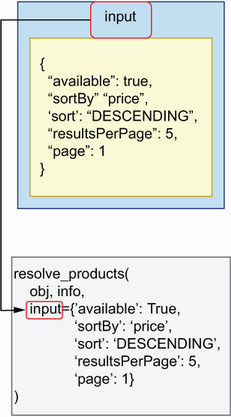


Figure 10.9 Query parameters are passed to our resolvers as keyword arguments. This example illustrates how the resolve\_products() resolver is called, with the input parameter passed as a keyword argument. The parameter input is an object of type ProductsFilter, and therefore it comes in the form of a dictionary.

As you can see in figure 10.9, when a query or mutation takes parameters, Ariadne passes those parameters to our resolvers as keyword arguments. Listing 10.8 shows how we access the input parameter for the products() query resolver. Since the input parameter is optional and therefore nullable, we set it by default to None. The input parameter is an instance of the ProductsFilter input type, so when it’s present in the query, it comes in the form of a dictionary. From the API specification, we know that ProductsFilter guarantees the presence of the following fields:

* available—Boolean field that filters products by whether they’re available
* sortBy—An enumeration type that allows us to sort products by price or name
* sort—Enumeration type that allows us to sort the results in ascending or descending order
* resultsPerPage—Indicates how many results should be shown per page
* page—Indicates which page of the results we should return

In addition to these parameters, ProductsFilter may also include two optional parameters: maxPrice, which filters results by maximum price, and minPrice, which filters results by minimum price. Since maxPrice and minPrice are not required fields, we check for their presence using the Python dictionary’s get() method, which returns None if they’re not found. Let’s implement the filtering and sorting functionality first, and deal with pagination afterwards. The following code goes under web/ queries.py.

Listing 10.8 Accessing input parameters in a resolver

# file: web/queries.py

...

Query = QueryType()

...

**@query.field('products')**  ①

**def resolve\_products(\*\_, input=None):**  ②

**filtered = [product for product in products]**  ③

**if input is None:**  ④

**return filtered**

**filtered = [**  ⑤

**product for product in filtered**

**if product['available'] is input['available']**

**]**

**if input.get('minPrice') is not None:**  ⑥

**filtered = [**

**product for product in filtered**

**if product['price'] >= input['minPrice']**

**]**

**if input.get('maxPrice') is not None:**

**filtered = [**

**product for product in filtered**

**if product['price'] <= input['maxPrice']**

**]**

**filtered.sort(**  ⑦

**key=lambda product: product.get(input['sortBy'], 0],**

**reverse=input['sort'] == 'DESCENDING'**

**)**

**return filtered**  ⑧

① We bind products()’ resolver using the field() decorator.

② We ignore the default positional arguments and instead capture the input parameter.

③ We copy the list of products.

④ If input is None, we return the whole dataset.

⑤ We filter products by availability.

⑥ We filter products by minPrice.

⑦ We sort the filtered dataset.

⑧ We return the filtered dataset.

Let’s run a query to test this resolver:

{

products(input: {available: true}) {

...on ProductInterface {

name

}

}

}

You should get a valid response from the server. Now that we have filtered the results, we need to paginate them. Listing 10.9 adds a generic pagination function called get\_page() to web/queries.py. Just a word of warning: in normal circumstances, you’ll be storing your data in a database and delegating filtering and pagination to the database. The examples here are to illustrate how you use the query parameters in the resolver. We paginate the results using the islice() function from the itertools module.

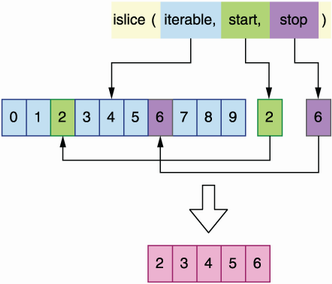


Figure 10.10 The islice() function from the itertools module allows you to get a slice of an iterable object by selecting the start and stop indices of the subset that you want to slice.

As you can see in figure 10.10, islice() allows us to extract a slice of an iterable object. islice() requires us to provide the start and stop indices of the portion that we want to slice. For example, a list of 10 items comprising the numbers 0 to 9, providing a start index of 2 and a stop index of 6, would give us a slice with the following items: [2, 3, 4, 5]. The API paginates results starting at 1, while islice() uses zero-based indexing, so get\_page() subtracts one unit from the page parameter to account for that difference.

Listing 10.9 Paginating results

# file: web/queries.py

**From itertools import islice**  ①

from ariadne import QueryType

from web.data import ingredients, products

...

**def get\_page(items, items\_per\_page, page):**

**page = page - 1**

**start = items\_per\_page \* page if page > 0 else page**  ②

**stop = start + items\_per\_page**  ③

**return list(islice(items, start, stop))**  ④

@query.field('products')

def resolve\_products(\*\_, input=None):

**...**

**return get\_page(filtered, input['resultsPerPage'], input['page'])** ⑤

① We import islice().

② We resolve the start index.

③ We calculate the stop index.

④ We return a slice of the list.

⑤ We paginate the results.

Our hardcoded dataset only contains two products, so let’s test the pagination with resutlsPerPage set to 1, which will split the list into two pages:

{

products(input: {resultsPerPage: 1, page: 1}) {

...on ProductInterface {

name

}

}

}

You should get exactly one result. Once we implement the addProduct() mutation in the next section, we’ll be able to add more products through the API and make more use of the pagination parameters.

You just learned how to handle query parameters! We’re now in a good position to learn how to implement mutations. Mutation resolvers are similar to query resolvers, but they always have parameters. But that’s enough of a spoiler; move on to the next section to learn more about mutations.

**10.4.6 Implementing mutation resolvers**

In this section, we learn to implement mutation resolvers. Implementing a mutation resolver follows the same guidelines we saw for queries. The only difference is the class we use to bind the mutation resolvers. While queries are bound to an instance of the QueryType class, mutations are bound to an instance of the MutationType class.

Let’s have a look at implementing the resolver for the addProduct() mutation. From the specification, we know that the addProduct() mutation has three required parameters: name, type, and input. The shape of the input parameter is given by the AddProductInput object type. AddProductInput defines additional properties that can be set when creating a new product, all of which are optional and therefore nullable. Finally, the addProduct() mutation must return a product type.

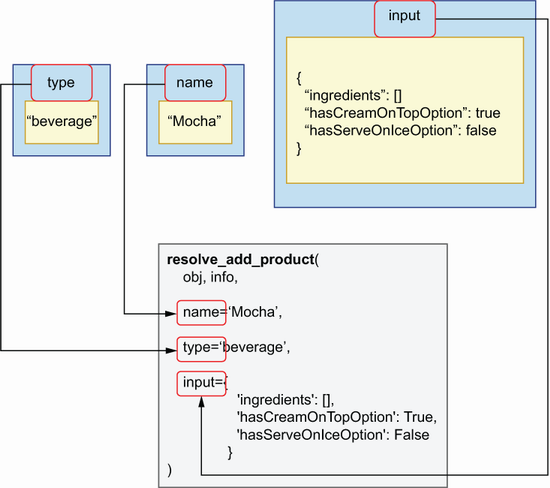


Figure 10.11 Mutation parameters are passed to our resolvers as keyword arguments. This example illustrates how the resolve\_add\_product() resolver is called, with the name, type, and input parameters passed as keyword arguments.

Listing 10.10 shows how we implement the resolver for the addProduct() mutation (see figure 10.11 for an illustration). We first import the MutationType bindable class and instantiate it. We then declare our resolver and bind it to MutationType using its field() decorator. We don’t need to use Ariadne’s default positional parameters obj and info, so we skip them using a wildcard followed by an underscore (\*\_). We don’t set default values for addProduct()’s parameters, since the specification states they’re all required. addProduct() must return a valid Product object, so we build the object with its expected attributes in the body of the resolver. Since Product is the union of the Cake and Beverage types, and each type requires different sets of properties, we check the type parameter to determine which fields we should add to our object. The following code goes into the web/mutations.py file.

Listing 10.10 Resolver for the addProduct() mutation

# file: web/mutations.py

import uuid

from datetime import datetime

from ariadne import MutationType

from web.data import products

mutation = MutationType() ①

@mutation.field('addProduct') ②

def resolve\_add\_product(\*\_, name, type, input): ③

product = { ④

'id': uuid.uuid4(), ⑤

'name': name,

'available': input.get('available', False), ⑥

'ingredients': input.get('ingredients', []),

'lastUpdated': datetime.utcnow(),

}

if type == 'cake': ⑦

product.update({

'hasFilling': input['hasFilling'],

'hasNutsToppingOption': input['hasNutsToppingOption'],

})

else:

product.update({

'hasCreamOnTopOption': input['hasCreamOnTopOption'],

'hasServeOnIceOption': input['hasServeOnIceOption'],

})

products.append(product) ⑧

return product

① Bindable object for mutations

② We bind addProduct()’s resolver using the field() decorator.

③ We capture addProduct()’s parameters.

④ We declare the new product as a dictionary.

⑤ We set server-side properties such as the ID.

⑥ We parse optional parameters and set their default values.

⑦ We check whether the product is a Beverage or a Cake.

⑧ We return the newly created product.

To enable the resolver implemented in listing 10.10, we need to add the mutation object to the make\_executable\_schema() function in web/schema.py:

# file: web/schema.py

from pathlib import Path

from ariadne import make\_executable\_schema

**from web.mutations import mutation**

from web.queries import query

from web.types import product\_type

schema = make\_executable\_schema(

(Path(\_\_file\_\_).parent / 'products.graphql').read\_text(),

[query, **mutation**, product\_type]

)

Let’s put the new mutation to work by running a simple test. Go to the Apollo Playground running on http://127.0.0.1:8000, and run the following mutation:

mutation {

addProduct(name: "Mocha", type: beverage, input:{ingredients: []}) {

...on ProductInterface {

name,

id

}

}

}

You’ll get a valid response, and a new product will be added to our list. To verify things are working correctly, run the following query and check that the response contains the new item just created:

{

allProducts {

...on ProductInterface {

name

}

}

}

Remember that we are running the service with an in-memory list representation of our data, so if you stop or reload the server, the list will be reset and you’ll lose any newly created data.

You just learned how to build mutations! This is a powerful feature: with mutations, you can create and update data in a GraphQL server. We’ve now covered nearly all the major aspects of the implementation of a GraphQL server. In the next section, we’ll take this further by learning how to implement resolvers for custom scalar types.

**10.4.7 Building resolvers for custom scalar types**

In this section, we learn how to implement resolvers for custom scalar types. As we saw in chapter 8, GraphQL provides a decent amount of scalar types, such as Boolean, integer, and string. And in many cases, GraphQL’s default scalar types are sufficient to develop an API. Sometimes, however, we need to define our own custom scalars. The products API contains a custom scalar called Datetime. The lastUpdated field in both the Ingredient and Product types have a Datetime scalar type. Since Datetime is a custom scalar, Ariadne doesn’t know how to handle it, so we need to implement a resolver for it. How do we do that?

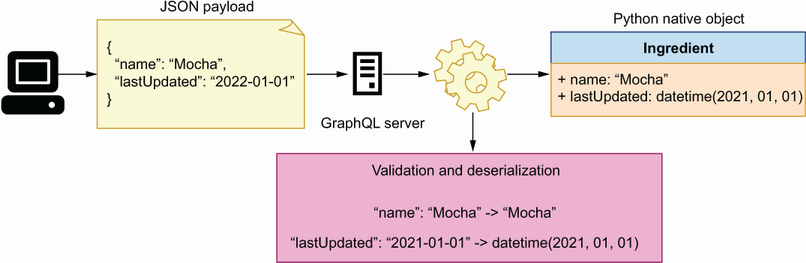


Figure 10.12 When a GraphQL server receives data from the user, it validates and deserializes the data into native Python objects. In this example, the server deserializes the name "Mocha" into a Python string, and the date "2021-01-01" into a Python datetime.

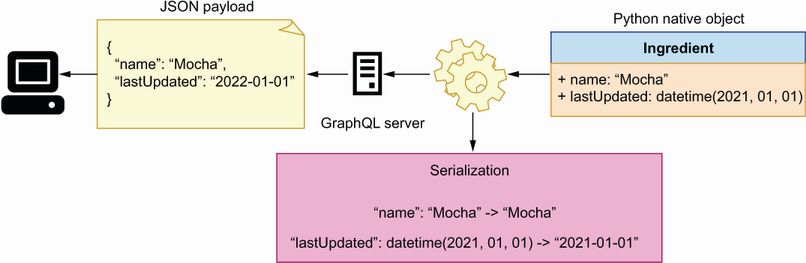


Figure 10.13 When the GraphQL server sends data to the user, it transforms native Python objects into serializable data. In this example, the server serializes the both the name and the date as strings.

As you can see in figures 10.12 and 10.13, when we encounter a custom scalar type in a GraphQL API, we need to make sure we can perform the following three actions on the custom scalar:

* *Serialization*—When a user requests data from the server, Ariadne has to be able to serialize the data. Ariadne knows how to serialize GraphQL’s built-in scalars, but for custom scalars, we need to implement a custom serializer. In the case of the Datetime scalar in the products API, we have to implement a method to serialize a datetime object.
* *Deserialization*—When a user sends data to our server, Ariadne deserializes the data and makes it available to us as a Python native data structure, such as a dictionary. If the data includes a custom scalar, we need to implement a method that lets Ariadne know how to parse and load the scalar into a native Python data structure. For the Datetime scalar, we want to be able to load it as a datetime object.
* *Validation*—GraphQL enforces validation of each scalar and type, and Ariadne knows how to validate GraphQL’s built-in scalars. For custom scalars, we have to implement our own validation methods. In the case of the Datetime scalar, we want to make sure it has a valid ISO format.

Ariadne provides a simple API to handle these actions through its ScalarType class. The first thing we need to do is create an instance of this class:

from ariadne import ScalarType

datetime\_scalar = ScalarType('Datetime')

ScalarType exposes decorator methods that allow us to implement serialization, deserialization, and validation. For serialization, we use ScalarType’s serializer() decorator. We want to serialize datetime objects into ISO standard date format, and Python’s datetime library provides a convenient method for ISO formatting, the isoformat() method:

@datetime\_scalar.serializer

def serialize\_datetime(value):

return value.isoformat()

For validation and deserialization, ScalarType provides the value\_parser() decorator. When a user sends data to the server containing a Datetime scalar, we expect the date to be in ISO format and therefore parsable by Python’s datetime.fromisoformat() method:

from datetime import datetime

@datetime\_scalar.value\_parser

def parse\_datetime\_value(value):

return datetime.fromisoformat(value)

If the date comes in the wrong format, fromisoformat() will raise a ValueError, which will be caught by Ariadne and shown to the user with the following message: “Invalid isoformat string.” The following code goes under web/types.py since it implements a type resolver.

Listing 10.11 Serializing and parsing a custom scalar

# file: web/types.py

import uuid

**from datetime import datetime**

from ariadne import UnionType, **ScalarType**

...

**datetime\_scalar = ScalarType('Datetime')** ①

**@datetime\_scalar.serializer**  ②

**def serialize\_datetime\_scalar(date):**  ③

**return date.isoformat()**  ④

**@datetime\_scalar.value\_parser**  ⑤

**def parse\_datetime\_scalar(date):**  ⑥

**return datetime.fromisoformat(date)** ⑦

① We create a bindable object for the Datetime scalar using the ScalarType class.

② We bind Datetime’s serializer using the serializer() decorator.

③ We capture the serializer’s argument as date.

④ We serialize the date object.

⑤ We bind Datetime’s parser using the value\_parser() decorator.

⑥ We capture the parser’s argument.

⑦ We parse a date.

To enable the Datetime resolvers, we add datetime\_scalar to the array of bindable objects for the make\_executable\_schema() function under web/schema.py:

from pathlib import Path

from ariadne import make\_executable\_schema

from web.mutations import mutation

from web.queries import query

from web.types import product\_type, **datetime\_scalar**

schema = make\_executable\_schema(

(Path(\_\_file\_\_).parent / 'products.graphql').read\_text(),

[query, mutation, product\_type, **datetime\_scalar**]

)

Let’s put the new resolvers to the test! Go back to the Apollo Playground running on http://127.0.0.1:8000 and execute the following query:

# Query document

{

allProducts {

...on ProductInterface {

name,

lastUpdated

}

}

}

# result:

{

"data": {

"allProducts": [

{

"name": "Walnut Bomb",

"lastUpdated": "2022-06-19T18:27:53.171870"

},

{

"name": "Cappuccino Star",

"lastUpdated": "2022-06-19T18:27:53.171871"

}

]

}

}

You should get a list of all products with their names, and with an ISO-formatted date in the lastUpdated field. You now have the power to implement your own custom scalar types in GraphQL. Use it wisely! Before we close the chapter, there’s one more topic we need to explore: implementing resolvers for the fields of an object type.

**10.4.8 Implementing field resolvers**

In this section, we learn to implement resolvers for the fields of an object type. We’ve implemented nearly all the resolvers that we need to serve all sorts of queries on the products API, but there’s still one type of query that our server can’t resolve: queries involving fields that map to other GraphQL types. For example, the Products type has a field called ingredients, which maps to an array of IngredientRecipe objects. According to the specification, the shape of the IngredientRecipe type looks like this:

# file: web/products.graphql

type IngredientRecipe {

ingredient: Ingredient!

quantity: Float!

unit: String!

}

Each IngredientRecipe object has an ingredient field, which maps to an Ingredient object type. This means that, when we query the ingredients field of a product, we should be able to pull information about each ingredient, such as its name, description, or supplier information. In other words, we should be able to run the following query against the server:

{

allProducts {

...on ProductInterface {

name,

ingredients {

quantity,

unit,

ingredient{

name

}

}

}

}

}

If you run this query in Apollo Playground at this juncture, you’ll get an error with the following message: “Cannot return null for non-nullable field Ingredient.name.”

Why is this happening? If you look at the list of products in listing 10.5, you’ll notice that the ingredients field maps to an array of objects with three fields: ingredient, quantity, and unit. For example, the Walnut Bomb has the following ingredients:

# file: web/data.py

ingredients = [

{

'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',

'quantity': 100.00,

'unit': 'LITRES',

}

]

The ingredient field maps to an ingredient ID, not a full ingredient object. This is our internal representation of the product’s ingredients. It’s how we store product data in our database (in-memory list in this implementation). And it’s a useful representation since it allows us to identify each ingredient by ID. However, the API specification tells us that the ingredients field should map to an array of IngredientRecipe objects and that each ingredient should represent an Ingredient object, not just an ID.

How do we solve this problem? We can use different approaches. For example, we could make sure that each ingredient payload is correctly built in the resolvers for each query that returns a Product type. For example, listing 10.12 shows how we can modify the allProducts() resolver to accomplish this. The snippet modifies every product’s ingredients property to make sure it contains a full ingredient payload. Since every product is represented by a dictionary, we make a deep copy of each product to make sure the changes we apply in this function don’t affect our in-memory list of products.

Listing 10.12 Updating products to contain full ingredient payloads, not just IDs

# file: web/queries.py

...

@query.field('allProducts')

def resolve\_all\_products(\*\_):

**products\_with\_ingredients = [deepcopy(product) for product in products]**①

**for product in products\_with\_ingredients:**

**for ingredient\_recipe in product['ingredients']:**

**for ingredient in ingredients:**

**if ingredient['id'] == ingredient\_recipe['ingredient']:**

**ingredient\_recipe['ingredient'] = ingredient**  ②

**return products\_with\_ingredients**  ③

① We make a deep copy of each object in the products list.

② We update the ingredient property with a full representation of the ingredient.

③ We return the list of products with ingredients.

The approach in listing 10.12 is perfectly fine, but as you can see, it makes the code grow in complexity. If we had to do this for a few more properties, the function would quickly become difficult to understand and to maintain.

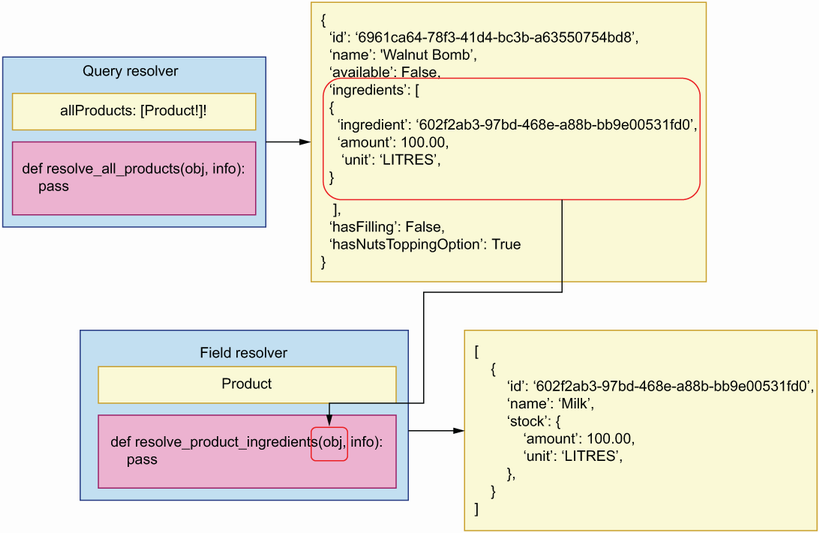


Figure 10.14 GraphQL allows us to create resolvers for specific fields of an object. In this example, the resolve\_product\_ingredients() resolver takes care of returning a valid payload for the ingredients property of a product.

As you can see in figure 10.14, GraphQL offers an alternative way of resolving object properties. Instead of modifying the product payload within the allProducts() resolver, we can create a specific resolver for the product’s ingredients property and make any necessary changes within that resolver. Listing 10.13 shows what the resolver for the product’s ingredients property looks like and goes under web/types.py since it implements a resolver for object properties.

Listing 10.13 Implementing a field resolver

# file: web/types.py

...

**@product\_interface.field('ingredients')**

**def resolve\_product\_ingredients(product, \_):**

**recipe = [**  ①

**copy.copy(ingredient)**

**for ingredient in product.get("ingredients", [])**

**]**

**for ingredient\_recipe in recipe:**

**for ingredient in ingredients:**

**if ingredient['id'] == ingredient\_recipe['ingredient']:**

**ingredient\_recipe['ingredient'] = ingredient**

**return recipe**

① We create a deep copy of each ingredient.

Object property resolvers help us keep our code more modular because every resolver does only one thing. They also help us avoid repetition. By having a single resolver that takes care of updating the ingredients property in product payloads, we avoid having to perform this operation in every resolver that returns a product type. On the downside, property resolvers may be more difficult to trace and debug. If something is wrong with the ingredients payload, you won’t find the bug within the allProducts() resolver. You have to know that there’s a resolver for products’ ingredients and look into that resolver. Application logs will help to point you in the right direction when debugging this kind of issues, but bear in mind that this design will not be entirely obvious to other developers who are not familiar with GraphQL. As with everything else in software design, make sure that code reusability doesn’t impair the readability and ease of maintenance of your code.

**Summary**

* The Python ecosystem offers various frameworks for implementing GraphQL APIs. See GraphQL’s official website for the latest news on available frameworks: <https://graphql.org/code/>.
* You can use the Ariadne framework to implement GraphQL APIs following a schema-first approach, which means we first design the API, and then we implement the server against the specification. This approach is beneficial since it allows the server and client development teams to work in parallel.
* Ariadne can validate request and response payloads automatically using the specification, which means we don’t have to spend time implementing custom validation models.
* For each query and mutation in the API specification, we need to implement a resolver. A resolver is a function that knows how to process the request for a given query or mutation. Resolvers are the code that allow us to expose the capabilities of a GraphQL API and therefore represent the backbone of the implementation.
* To register a resolver, we use one of Ariadne’s bindable classes, such as QueryType or MutationType. These classes expose decorators that allow us to bind a resolver function.
* GraphQL specifications can contain complex types, such as union types, which combine two or more object types. If our API specification contains a union type, we must implement a resolver that knows how to determine the type of an object; otherwise, the GraphQL server doesn’t know how to resolve it.
* With GraphQL, we can define custom scalars. If the specification contains a custom scalar, we must implement resolvers that know how to serialize, parse, and validate the custom scalar type; otherwise, the GraphQL server doesn’t know how to handle them.