**Part 3. Designing and building GraphQL APIs**

In part 2, you learned that REST is an API technology that allows us to alter or retrieve the state of a resource from the server. When a resource is represented by a large payload, fetching it from the server translates to a large amount of data transfer. With the emergence of API clients running in mobile devices with restricted network access and limited storage and memory capacity, exchanging large payloads often results in unreliable communication. In 2012, Facebook was acutely aware of these problems, and it developed a new technology to allow API clients to run granular data queries on the server. This technology was released in 2015 under the name of GraphQL.

GraphQL is a query language for APIs. Instead of fetching full representations of resources, GraphQL allows you to fetch one or more properties of a resource, such as the price of a product or the status of an order. With GraphQL, we can also model the relationship between different objects, which allows us to retrieve, in a single request, the properties of various resources from the server, such as a product’s ingredients and its stock availability.

Despite its benefits, many developers aren’t familiar with GraphQL or don’t know how it works, and therefore it isn’t usually the first choice of technology for building an API. In part 3, you learn everything you need to know to design and build high-quality GraphQL APIs and how to consume them. After reading part 3, you’ll know what GraphQL is, how it works, and when to use it so that you can make better decisions in your API strategy.

# 8 Designing GraphQL APIs

This chapter covers

* Understanding how GraphQL works
* Producing an API specification using the Schema Definition Language (SDL)
* Learning GraphQL’s built-in scalar types and data structures and building custom object types
* Creating meaningful connections between GraphQL types
* Designing GraphQL queries and mutations

GraphQL is one of the most popular protocols for building web APIs. It’s a suitable choice for driving integrations between microservices and for building integrations with frontend applications. GraphQL gives API consumers full control over the data they want to fetch from the server and how they want to fetch it.

In this chapter, you’ll learn to design a GraphQL API. You’ll do it by working on a practical example: you’ll design a GraphQL API for the products service of the CoffeeMesh platform. The products service owns data about CoffeeMesh’s products as well as their ingredients. Each product and ingredient contains a rich list of properties that describe their features. However, when a client requests a list of products, they are most likely interested in fetching only a few details about each product. Also, clients may be interested in being able to traverse the relationships between products, ingredients, and other objects owned by the products service. For these reasons, GraphQL is an excellent choice for building the products API.

As we build the specification for the products API, you’ll learn about GraphQL’s scalar types, designing custom object types, as well as queries and mutations. By the end of this chapter, you’ll understand how GraphQL compares with other types of APIs and when it makes the most sense to use it. We’ve got a lot to cover, so without further ado, let’s start our journey!

To follow along with the specification we develop in this chapter, you can use the GitHub repository provided with this book. The code for this chapter is available under the folder named ch08.

## 8.1 Introducing GraphQL

This section covers what GraphQL is, what its advantages are, and when it makes sense to use it. The official website of the GraphQL specification defines GraphQL as a “query language for APIs and a runtime for fulfilling those queries with your existing data.”[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/08.htm#pgfId-1309583) What does this really mean? It means that GraphQL is a specification that allows us to run queries in an API server. In the same way SQL provides a query language for databases, GraphQL provides a query language for APIs.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/08.htm#pgfId-1309586) GraphQL also provides a specification for how those queries are resolved in a server so that anyone can implement a GraphQL runtime in any programming language.[**3**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/08.htm#pgfId-1309589)

Just as we can use SQL to define schemas for our database tables, we can use GraphQL to write specifications that describe the type of data that can be queried from our servers. A GraphQL API specification is called a schema, and it’s written in a standard called Schema Definition Language (SDL). In this chapter, we will learn how to use the SDL to produce a specification for the products API.

GraphQL was first released in 2015, and since then it’s gained traction as one of the most popular choices for building web APIs. I should say there’s nothing in the GraphQL specification saying that GraphQL should be used over HTTP, but in practice, this is the most common type of protocol used in GraphQL APIs.

What’s great about GraphQL? It shines in giving users full control over which data they want to obtain from the server. For example, as we’ll see in the next section, in the products API we store many details about each product, such as its name, price, availability, and ingredients, among others. As you can see in figure 8.1, if a user wishes to get a list of just product names and prices, with GraphQL they can do that. In contrast, with other types of APIs, such as REST, you get a full list of details for each product. Therefore, whenever it’s important to give the client full control over how they fetch data from the server, GraphQL is a great choice.

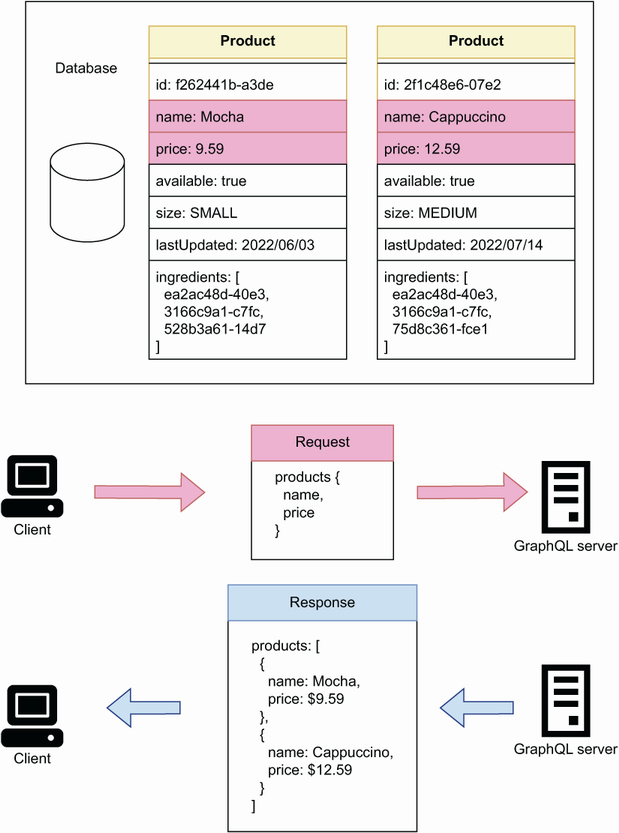


Figure 8.1 Using a GraphQL API, a client can request a list of items with specific details. In this example, a client is requesting the name and price of each product in the products API.

Another great advantage of GraphQL is the ability to create connections between different types of resources, and to expose those connections to our clients for use in their queries. For example, in the products API, products and ingredients are different but related types of resources. As you can see in figure 8.2, if a user wants to get a list of products, including their names, prices, and their ingredients, with GraphQL they can do that by leveraging the connections between these resources. Therefore, in services where we have highly interconnected resources, and where it’s useful for our clients to explore and query those connections, GraphQL makes an excellent choice.

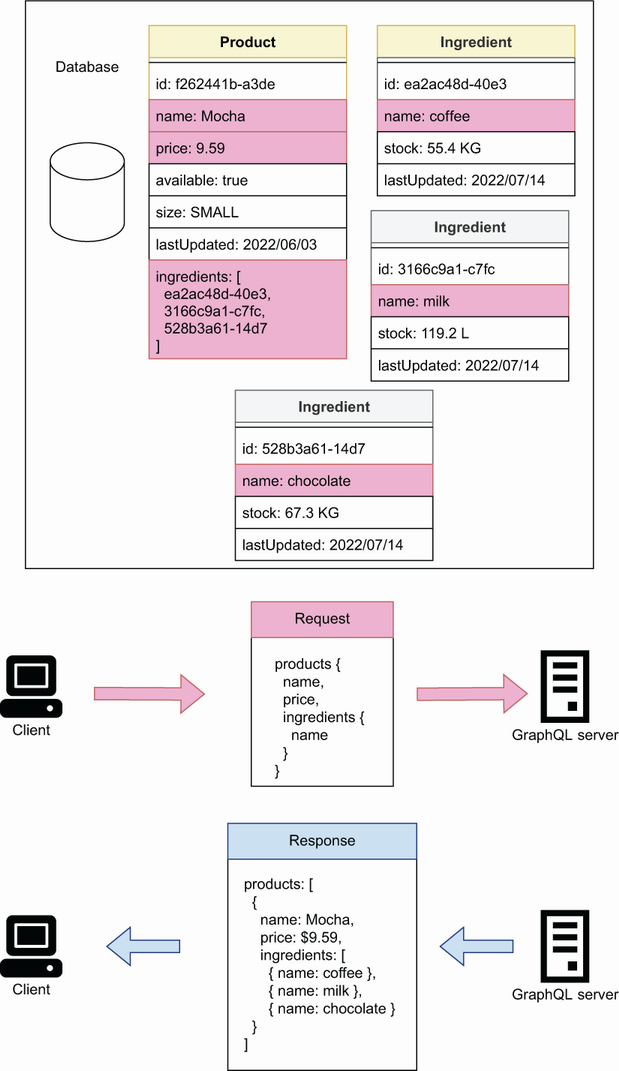


Figure 8.2 Using GraphQL, a client can request the details of a resource and other resources linked to it. In this example, the products API has two types of resources: products and ingredients, both of which are connected through product’s ingredients field. Using this connection, a client can request the name and price of each product, as well as the name of each product’s ingredient.

In the sections that follow, we’ll learn how to produce a GraphQL specification for the products service. We’ll learn how to define the types of our data, how to create meaningful connections between resources, and how to define operations for querying the data and changing the state of the server. But before we do that, we ought to understand the requirements for the products API, and that’s what we do in the next section!

## 8.2 Introducing the products API

This section discusses the requirements of the products API. Before working on an API specification, it’s important to gather information about the API requirements. As you can see in figure 8.3, the products API is the interface to the products service. To determine the requirements of the products API, we need to know what users of the products service can do with it.

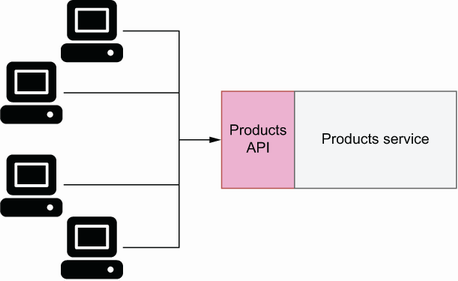


Figure 8.3 To interact with the products service, clients use the products API.

The products service owns data about the products offered by the CoffeeMesh platform. As you can see in figure 8.4, the CoffeeMesh staff must be able to use the products service to manage the available stock of each product, as well as to keep the products’ ingredients up to date. In particular, they must be able to query the stock of a product or ingredient, and to update them when new stock arrives to the warehouse. They must also be able to add new products or ingredients to the system and delete old ones. This information already gives us a complex list of requirements, so let’s break it down into specific technical requirements.

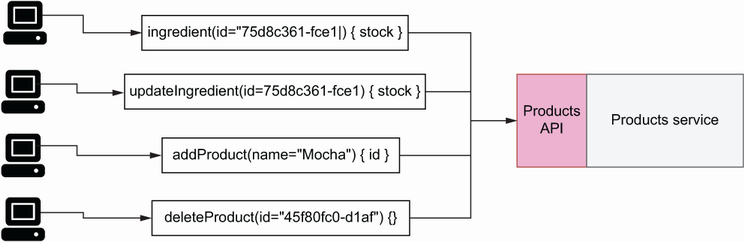


Figure 8.4 The CoffeeMesh staff uses the products service to manage products and ingredients.

Let’s start with by modeling the resources managed by the products API. We want to know which type of resources we should expose through the API and the products’ properties. From the description in the previous paragraph, we know that the products service manages two types of resources: products and ingredients. Let’s analyze products first.

The CoffeeMesh platform offers two types of products: cakes and beverages. As you can see in figure 8.5, both cakes and beverages have a common set of properties, including the product’s name, price, size, list of ingredients, and its availability. Cakes have two additional properties:

* hasFilling—Indicates whether the cake has a filling
* hasNutsToppingOption—Indicates whether the customer can add a topping of nuts to the cake

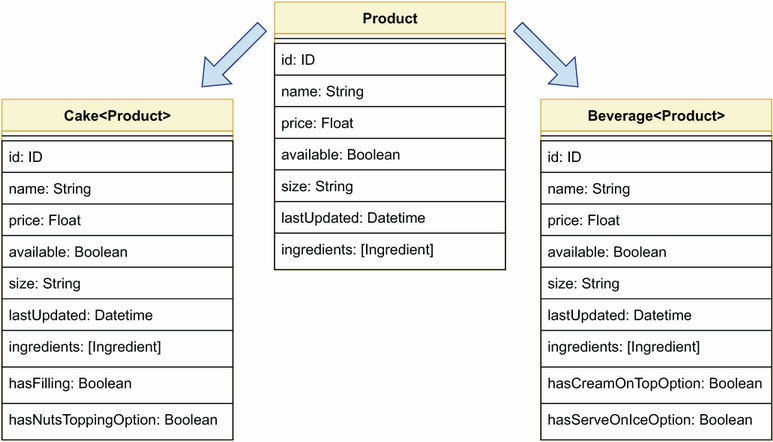


Figure 8.5 CoffeeMesh exposes two types of products: Cake and Beverage, both of which share a common list of properties.

Beverages have the following two additional properties:

* hasCreamOnTopOption—Indicates whether the customer can top the beverage with cream
* hasServeOnIceOption—Indicates whether the customer can choose to get the beverage served on ice

What about ingredients? As you can see in figure 8.6, we can represent all ingredients through one entity with the following attributes:

* name—The ingredient’s name.
* stock—The ingredient’s available stock. Since different ingredients are measured with different units, such as kilograms or liters, we express the available stock in terms of amounts of per unit of measure.
* description—A collection of notes that CoffeeMesh employees can use to describe and qualify the product.
* supplier—Information about the company that supplies the ingredient to CoffeeMesh, including their name, address, contact number, and email.

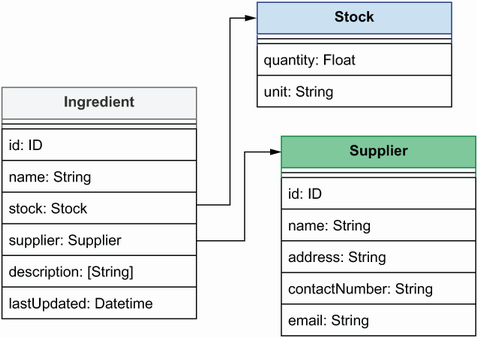


Figure 8.6 List of properties that describe an ingredient. The ingredient’s supplier is described by a resource called Supplier, while the ingredient’s stock is described through a Stock object.

Now that we’ve modeled the main resources managed by the products service, let’s turn our attention to the operations we must expose through the API. We’ll distinguish read operations from write/delete operations. This distinction will make sense when we look more closely at these operations in sections 8.8 and 8.9.

Based on the previous discussion, we’ll expose the following read operations:

* allProducts()—Returns the full list of products available in the CoffeeMesh catalogue
* allIngredients()—Returns the full list of ingredients used by CoffeeMesh to make their products
* products()—Allows users to filter the full list of products by certain criteria such as availability, maximum price, and others
* product()—Allows users to obtain information about a single product
* ingredient()—Allows users to obtain information about a single ingredient

In terms of write/delete operations, from the previous discussion it’s clear that we should expose the following capabilities:

* addIngredient()—To add new ingredients
* updateStock()—To update an ingredient’s stock
* addProduct()—To add new products
* updateProduct()—To update existing products
* deleteProduct()—To delete products from the catalogue

Now that we understand the requirements of the products API, it’s time to move on to creating the API specification! In the following sections, we’ll learn to create a GraphQL specification for the products API, and along the way we’ll learn how GraphQL works. Our first stop is GraphQL’s type system, which we’ll use to model the resources managed by the APIs.

## 8.3 Introducing GraphQL’s type system

In this section, we introduce GraphQL’s type system. In GraphQL, types are definitions that allow us to describe the properties of our data. They’re the building blocks of a GraphQL API, and we use them to model the resources owned by the API. In this section, you’ll learn to use GraphQL’s type system to describe the resources we defined in section 8.2.

### 8.3.1 Creating property definitions with scalars

This section explains how we define the type of a property using GraphQL’s type system. We distinguish between scalar types and object types. As we’ll see in section 8.3.2, object types are collection of properties that represent entities. *Scalar types* are types such as Booleans or integers. The syntax for defining a property’s type is very similar to how we use type hints in Python: we include the name of the property followed by a colon, and the property’s type to the right of the colon. For example, in section 8.2 we discussed that cakes have two distinct properties: hasFilling and hasNutsToppingOption, both of which are Booleans. Using GraphQL’s type system, we describe these properties like this:

hasFilling: Boolean

hasNutsToppingOption: Boolean

GraphQL supports the following types of scalars:

* *Strings* (String)—For text-based object properties.
* *Integers* (Int)—For numerical object properties.
* *Floats* (Float)—For numerical object properties with decimal precision.
* *Booleans* (Boolean)—For binary properties of an object.
* *Unique identifiers* (ID)—For describing an object ID. Technically, IDs are strings, but GraphQL checks and ensures that the ID of each object is unique.

In addition to defining the type of a property, we can also indicate whether the property is non-nullable. Nullable properties are properties that can be set to null when we don’t know their value. We mark a property as non-nullable by placing an exclamation point at the end of the property definition:

name: String!

This line defines a property name of type String, and it marks it as non-nullable by using an exclamation point. This means that, whenever we serve this property from the API, it will always be a string.

Now that we’ve learned about properties and scalars, let’s see how we use this knowledge to model resources!

### 8.3.2 Modeling resources with object types

This section explains how we use GraphQL’s type system to model resources. Resources are the entities managed by the API, such as the ingredients, cakes, and beverages that we discussed in section 8.2. In GraphQL, each of these resources is modeled as an object type. *Object types* are collections of properties, and as the name indicates, we use them to define objects. To define an object type, we use the type keyword followed by the object name, and the list of object properties wrapped between curly braces. A property is defined by declaring the property name followed by a colon, and its type on the right side of the colon. In GraphQL, ID is a type with a unique value. An exclamation point at the end of a property indicates that the property is non-nullable. The following illustrates how we describe the cake resource as an object type. The listing contains the basic properties of the cake type, such as the ID, the name, and its price.

Listing 8.1 Definition of the Cake object type

type Cake { ①

id: ID! ②

name: String!

price: Float

available: Boolean!

hasFilling: Boolean!

hasNutsToppingOption: Boolean!

}

① We define an object type.

② We define a non-nullble ID property.

**TYPES AND OBJECT TYPES** For convenience, throughout the book, we use the concepts of type and object type interchangeably unless otherwise stated.

Some of the property definitions in listing 8.1 end with an exclamation point. In GraphQL, an exclamation point means that a property is non-nullable, which means that every cake object returned by our API will contain an ID, a name, its availability, as well as the hasFilling and hasNutsToppingOption properties. It also guarantees that none of these properties will be set to null. For API client developers, this information is very valuable because they know they can count on these properties to always be present and build their applications with that assumption. The following code shows the definitions for the Beverage and Ingredient types. It also shows the definition for the Supplier type, which contains information about the business that supplies a certain ingredient, and in section 8.5.1 we’ll see how we connect it with the Ingredient type.

Listing 8.2 Definitions of the Beverage and Ingredient object types

type Beverage {

id: ID!

name: String!

price: Float

available: Boolean!

hasCreamOnTopOption: Boolean!

hasServeOnIceOption: Boolean!

}

type Ingredient {

id: ID!

name: String!

}

type Supplier {

id: ID!

name: String!

address: String!

contactNumber: String!

email: String!

}

Now that we know how to define object types, let’s complete our exploration of GraphQL’s type system by learning how to create our own custom types!

### 8.3.3 Creating custom scalars

This section explains how we create custom scalar definitions. In section 8.3.1, we introduced GraphQL’s built-in scalars: String, Int, Float, Boolean, and ID. In many cases, this list of scalar types is sufficient to model our API resources. In some cases, however, GraphQL’s built-in scalar types might prove limited. In such cases, we can define our own custom scalar types. For example, we may want to be able to represent a date type, a URL type, or an email address type.

Since the products API is used to manage products and ingredients and make changes to them, it is useful to add a lastUpdated property that tells us the last time a record changed. lastUpdated should be a Datetime scalar. GraphQL doesn’t have a built-in scalar of that type, so we have to create our own. To declare a custom date-time scalar, we use the following statement:

scalar Datetime

We also need to define how this scalar type is validated and serialized. We define the rules for validation and serialization of a custom scalar in the server implementation, which will be the topic of chapter 10.

Listing 8.3 Using a custom Datetime scalar type

scalar Datetime ①

type Cake {

id: ID!

name: String!

price: Float

available: Boolean!

hasFilling: Boolean!

hasNutsToppingOption: Boolean!

lastUpdated: Datetime! ②

}

① We declare a custom Datetime scalar.

② We declare a non-nullable property with type Datetime.

This concludes our exploration of GraphQL scalars and object types. You’re now in a position to define basic object types in GraphQL and create your own custom scalars. In the following sections, we’ll learn to create connections between different object types, and we’ll learn how to use lists, interfaces, enumerations, and more!

## 8.4 Representing collections of items with lists

This section introduces GraphQL lists. Lists are arrays of types, and they’re defined by surrounding a type with square brackets. Lists are useful when we need to define properties that represent collections of items. As discussed in section 8.2, the Ingredient type contains a property called description, which contains collections of notes about the ingredient, as shown in the following code.

Listing 8.4 Representing a list of strings

type Ingredient {

id: ID!

name: String!

**description: [String!]**  ①

}

① We define a list of non-nullable items.

Look closely at the use of exclamation points in the description property: we’re defining it as a nullable property with non-nullable items. What does this mean? When we return an ingredient from the API, it may or may not contain a description field, and if that field is present, it will contain a list of strings.

When it comes to lists, you must pay careful attention to the use of exclamation points. In list properties, we can use two exclamation points: one for the list itself and another for the item within the list. To make both the list and its contents non-nullable, we use exclamation points for both. The use of exclamation points for list types is one of the most common sources of confusion among GraphQL users. Table 8.1 summarizes the possible return values for each combination of exclamation points in a list property definition.

**USE EXCLAMATION POINTS AND LISTS CAREFULLY!** In GraphQL, an exclamation point indicates that a property is non-nullable, which means that the property needs to be present in an object and its value cannot be null. When it comes to lists, we can use two exclamation points: one for the list itself and another for the item within the list. Different combinations of the exclamation points will yield different representations of the property. Table 8.1 shows which representations are valid for each combination.

Table 8.1 Valid return values for list properties

|  | [Word] | [Word!] | [Word]! | [Word!]! |
| --- | --- | --- | --- | --- |
| null | Valid | Valid | Invalid | Invalid |
| [] | Valid | Valid | Valid | Valid |
| ["word"] | Valid | Valid | Valid | Valid |
| [null] | Valid | Invalid | Valid | Invalid |
| ["word", null] | Valid | Invalid | Valid | Invalid |

Now that we’ve learned about GraphQL’s type system and list properties, we’re ready to explore one of the most powerful and exciting features of GraphQL: connections between types.

## 8.5 Think graphs: Building meaningful connections between object types

This section explains how we create connections between objects in GraphQL. One of the great benefits of GraphQL is being able to connect objects. By connecting objects, we make it clear how our entities are related. As we’ll see in the next chapter, this makes our GraphQL API more easily consumed.

### 8.5.1 Connecting types through edge properties

This section explains how we connect types by using *edge properties*: properties that point to another type. Types can be connected by creating a property that points to another type. As you can see in figure 8.7, a property that connects with another object is called an *edge*. The following code shows how we connect the Ingredient type with the Supplier type by adding a property called supplier to Ingredient that points to Supplier.

Listing 8.5 Edge for one-to-one connection

type Ingredient {

id: ID!

name: String!

**supplier: Supplier!**  ①

description: [String!]

}

① We use an edge property to connect the Ingredient and the Supplier types.

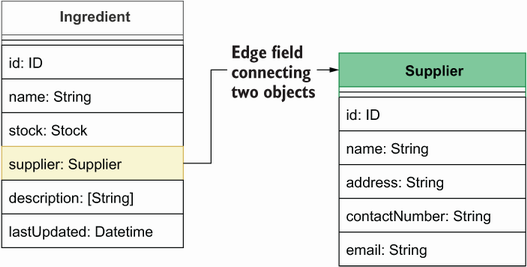


Figure 8.7 To connect the Ingredient type with the Supplier type, we add a property to Ingredient called supplier, which points to the Supplier type. Since the Ingredient’s supplier property is creating a connection between two types, we call it an edge.

This is an example of *one-to-one connection*: a property in an object that points to exactly one object. The property in this case is called an edge because it connects the Ingredient type with the Supplier type. It’s also an example of a *directed connection*: as you can see in figure 8.7, we can reach the Supplier type from the Ingredient type, but not the other way around, so the connection only works in one direction.

To make the connection between Supplier and the Ingredient bidirectional,**[4](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/08.htm" \l "pgfId-1309896" \t "_blank)** we need to add a property to the Supplier type that points to the Ingredient type. Since a supplier can provide more than one ingredient, the ingredients property points to a list of Ingredient types. This is an example of a *one-to-many connection*. Figure 8.8 shows what the new relationship between the Ingredient and the Supplier types looks like.

Listing 8.6 Bidirectional relationship between Supplier and Ingredient

type Supplier {

id: ID!

name: String!

address: String!

contactNumber: String!

email: String!

**ingredients: [Ingredient!]!**  ①

}

① We create a bidirectional relationship between the Ingredient and the Supplier types.

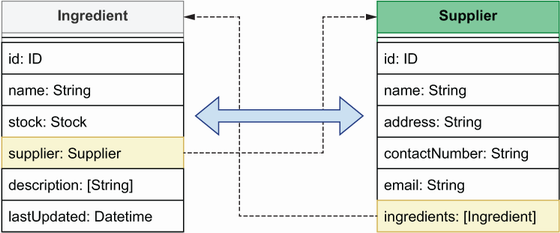


Figure 8.8 To create a bidirectional relationship between two types, we add properties to each of them that point to each other. In this example, the Ingredient’s supplier property points to the Supplier type, while the Supplier’s ingredients property points to a list of ingredients.

Now that we know how to create simple connections through edge properties, let’s see how we create more complex connections using dedicated types.

### 8.5.2 Creating connections with through types

This section discusses *through types*: types that tell us how other object types are connected. They add additional information about the connection itself. We’ll use through types to connect our products, cakes, and beverages, with their ingredients. We could connect them by adding a simple list of ingredients to Cake and Beverage, as shown in figure 8.9, but this wouldn’t tell us how much of each ingredient goes into a product’s recipe.

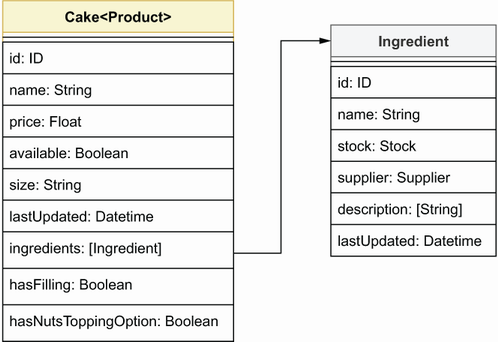


Figure 8.9 We can express Cake’s ingredients field as a list of Ingredient types, but that wouldn’t tell us how much of each ingredient goes into a cake recipe.

To connect cakes and beverages with their ingredients, we’ll use a through type called IngredientRecipe. As you can see in figure 8.10, IngredientRecipe has three properties: the ingredient itself, its amount, and the unit in which the amount is measured. This gives us more meaningful information about how our products relate to their ingredients.

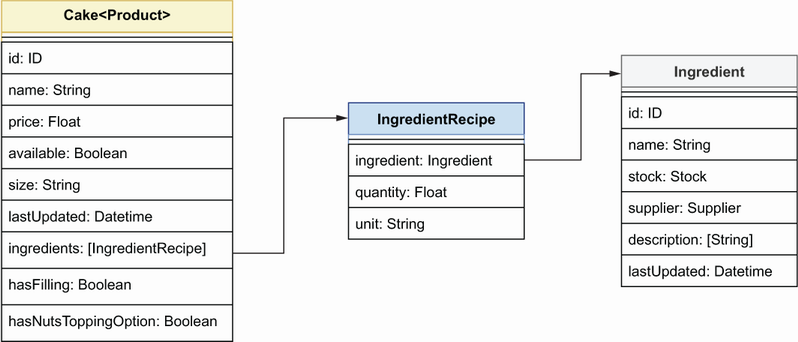


Figure 8.10 To express how an Ingredient is connected with a Cake, we use the IngredientRecipe through type, which allows us to detail how much of each ingredient goes into a cake recipe.

Listing 8.7 Through types that represent a relationship between two types

**type IngredientRecipe {**  ①

**ingredient: Ingredient!**

**quantity: Float!**

**unit: String!**

**}**

type Cake {

id: ID!

name: String!

price: Float

available: Boolean!

hasFilling: Boolean!

hasNutsToppingOption: Boolean!

lastUpdated: Datetime!

**ingredients: [IngredientRecipe!]!**  ②

}

type Beverage {

id: ID!

name: String!

price: Float

available: Boolean!

hasCreamOnTopOption: Boolean!

hasServeOnIceOption: Boolean!

lastUpdated: Datetime!

**ingredients: [IngredientRecipe!]!**

}

① We declare the IngredientRecipe through type.

② We declare ingredients as a list of IngredientRecipe through types.

By creating connections between different object types, we give our API consumers the ability to explore our data by just following the connecting edges in the types. And by creating bidirectional relationships, we give users the ability to traverse our data graph back and forth. This is one of the most powerful features of GraphQL, and it’s always worth spending the time to design meaningful connections across our data.

More often than not, we need to create properties that represent multiple types. For example, we could have a property that represents either cakes or beverages. This is the topic of the next section.

## 8.6 Combining different types through unions and interfaces

This section discusses how we cope with situations where we have multiple types of the same entity. You’ll often have to deal with properties that point to a collection of multiple types. What does this mean in practice, and how does it work? Let’s look at an example from the products API!

In the products API, Cake and Beverage are two types of products. In section 8.4.2, we saw how we connect Cake and Beverage with the Ingredient type. But how do we connect Ingredient to Cake and Beverage? We could simply add a property called products to the Ingredient type, which points to a list of Cakes and Beverages, like this:

products: [Cake, Beverage]

This works, but it doesn’t allow us to represent Cakes and Beverages as a single product entity. Why would we want to do that? Because of the following reasons:

* Cake and Beverage are the same thing: a product, and as such, it makes sense to treat them as the same entity.
* As we’ll see in sections 8.8 and 8.9, we’ll have to refer to our products in other parts of the code, and it will be very helpful to be able to use one single type for that.
* If we add new types of products to the system in the future, we don’t want to have to change all parts of the specification that refer to products. Instead, we want to have a single type that represents them all and update only that type.

GraphQL offers two ways to bring various types together under a single type: unions and interfaces. Let’s look at each in detail.

Interfaces are useful when we have types that share properties in common. This is the case for the Cake and the Beverage types, which share most of their properties. GraphQL interfaces are similar to class interfaces in programming languages, such as Python: they define a collection of properties that must be implemented by other types. Listing 8.8 shows how we use an interface to represent the collection of properties shared by Cake and Beverage. As you can see, we declare interface types using the interface keyword. The Cake and Beverage types implement ProductInterface, and therefore they must define all the properties defined in the ProductInterface type. By looking at the ProductInterface type, any user of our API can quickly get an idea of which properties are accessible on both the Beverage and Cake types.

Listing 8.8 Representing common properties through interfaces

**interface ProductInterface {**  ①

**id: ID!**

**name: String!**

**price: Float**

**ingredients: [IngredientRecipe!]**

**available: Boolean!**

lastUpdated: Datetime!

**}**

**type Cake implements ProductInterface** { ②

id: ID!

name: String!

price: Float

available: Boolean!

hasFilling: Boolean! ③

hasNutsToppingOption: Boolean!

lastUpdated: Datetime!

ingredients: [IngredientRecipe!]!

}

**type Beverage implements ProductInterface** { ④

id: ID!

name: String!

price: Float

available: Boolean!

hasCreamOnTopOption: Boolean!

hasServeOnIceOption: Boolean!

lastUpdated: Datetime!

ingredients: [IngredientRecipe!]!

}

① We declare the ProductInterface interface type.

② The Cake type implements the ProductInterface interface.

③ We define properties specific to Cake.

④ Beverage implements the ProductInterface interface.

By creating interfaces, we make it easier for our API consumers to understand the common properties shared by our product types. As we’ll see in the next chapter, interfaces also make the API easier to consume.

While interfaces help us define the common properties of various types, unions help us bring various types under the same type. This is very helpful when we want to treat various types as a single entity. In the products API, we want to be able to treat the Cake and Beverage types as a single Product type, and unions allow us to do that. A union type is the combination of different types using the pipe (|) operator.

Listing 8.9 A union of different types

type Cake implements ProductInterface {

id: ID!

name: String!

price: Float

available: Boolean!

hasFilling: Boolean!

hasNutsToppingOption: Boolean!

lastUpdated: Datetime!

ingredients: [IngredientRecipe!]!

}

type Beverage implements ProductInterface {

id: ID!

name: String!

price: Float

available: Boolean!

hasCreamOnTopOption: Boolean!

hasServeOnIceOption: Boolean!

lastUpdated: Datetime!

ingredients: [IngredientRecipe!]!

}

**union Product = Beverage | Cake** ①

① We create a union of the Beverage and the Cake types.

Using unions and interfaces makes our API easier to maintain and to consume. If we ever add a new type of product to the API, we can make sure it offers a similar interface to Cake and Beverage by making it implement the ProductInterface type. And by adding the new product to the Product union, we make sure it’s available on all operations that use the Product union type.

Now that we know how to combine multiple object types, it’s time to learn how we constrain the values of object type properties through enumerations.

## 8.7 Constraining property values with enumerations

This section covers GraphQL’s enumeration type. Technically, an *enumeration* is a specific type of scalar that can only take on a predefined number of values. Enumerations are useful in properties that can accept a value only from a constrained list of choices. In GraphQL, we declare enumerations using the enum keyword followed by the enumeration’s name, and we list its allowed values within curly braces.

In the products API, we need enumerations for expressing the amounts of the ingredients. For example, in section 8.5.2, we defined a through type called IngredientRecipe, which indicates the amount of each ingredient that goes into a product. IngredientRecipe expresses amounts in terms of quantity per unit of measure. We can measure ingredients in different ways. For example, we can measure milk in pints, liters, ounces, gallons, and so on. For the sake of consistency, we want to ensure that everyone uses the same units to describe the amounts of our ingredients, so we’ll create an enumeration type called MeasureUnit that can be used to constrain the values for the unit property.

Listing 8.10 Using the MeasureUnit enumeration type

**enum MeasureUnit {**  ①

**LITERS**  ②

**KILOGRAMS**

**UNITS**

**}**

type IngredientRecipe {

ingredient: Ingredient!

quantity: Float!

**unit: MeasureUnit!**  ③

}

① We declare an enumeration.

② We list the allowed values within this enumeration.

③ unit is a non-nullable property of type MeasureUnit.

We also want to use the MeasureUnit enumeration to describe the available stock of an ingredient. To do so, we define a Stock type, and we use it to define the stock property of the Ingredient type.

Listing 8.11 Using the Stock enumeration type

**type Stock {**  ①

**quantity: Float!**

**unit: MeasureUnit!**  ②

**}**

type Ingredient {

id: ID!

name: String!

**stock: Stock**  ③

products: [Product!]!

supplier: Supplier!

description: [String!]

}

① We declare the Stock type to help us express information about the available stock of an ingredient.

② Stock’s unit property is an enumeration.

③ We connect the Ingredient type with the Stock type through Ingredient’s stock property.

Enumerations are useful to ensure that certain values remain consistent through the interface. This helps avoid errors that happen when you let users choose and write those values by themselves.

This concludes our journey through GraphQL’s type system. Types are the building blocks of an API specification, but without a mechanism to query or interact with them, our API is very limited. To perform actions on the server, we need to learn about GraphQL queries and mutations. Those will be the topic of the rest of the chapter!

## 8.8 Defining queries to serve data from the API

This section introduces GraphQL *queries*: operations that allow us to fetch or read data from the server. Serving data is one of the most important functions of any web API, and GraphQL offers great flexibility to create a powerful query interface. Queries correspond to the group of read operations that we discussed in section 8.2. As a reminder, these are the query operations that the products API needs to support:

* allProducts()
* allIngredients()
* products()
* product()
* ingredient()

We’ll work on the allProducts() query first since it’s the simplest, and then move on to the products() query. As we work on products(), we’ll see how we add arguments to our query definitions, we’ll learn about pagination, and, finally, we’ll learn how to refactor our query parameters into their own type to improve readability and maintenance.

The specification of a GraphQL query looks similar to the signature definition of a Python function: we define the query name, optionally define a list of parameters for the query between parentheses, and specify the return type after a colon. The following code shows the simplest query in the products API: the allProducts() query, which returns a list of all products. allProducts() doesn’t take any parameters and simply returns a list of all products that exist in the server.

Listing 8.12 Simple GraphQL query to return a list of products

type Query { ①

allProducts: [Products!]! ②

}

① All queries are defined under the Query object type.

② We define the allProducts() query. After the colon, we indicate what the return type of the query is.

allProducts() returns a list of all products that exist in the CoffeeMesh database. Such a query is useful if we want to run an exhaustive analysis of all products, but in real life our API consumers want to be able to filter the results. They can do that by using the products() query, which, according to the requirements we gathered in section 8.2, returns a filtered list of products.

Query arguments are defined within parentheses, similar to how we define the parameters of a Python function. Listing 8.13 shows how we define the products() query. It includes arguments that allows our API consumers to filter products by availability, or by maximum and minimum price. All the arguments are optional. API consumers are free to use any or all of the query arguments, or none. If they don’t specify any of the arguments when using the products() query, they’ll get a list of all the products.

Listing 8.13 Simple GraphQL query to return a list of products

type Query {

products(available: Boolean, maxPrice: Float, minPrice: Float):

[Product!] ①

}

① Query parameters are defined within parentheses.

In addition to filtering the list of products, API consumers will likely want to be able to sort the list and paginate the results. Pagination is the ability to deliver the result of a query in different sets of a specified size, and it’s commonly used in APIs to ensure that API clients receive a sensible amount of data in each request. As illustrated in figure 8.11, if the result of a query yields 10 or more records, we can divide the query result into groups of five items each and serve one set at a time. Each set is called a *page*.

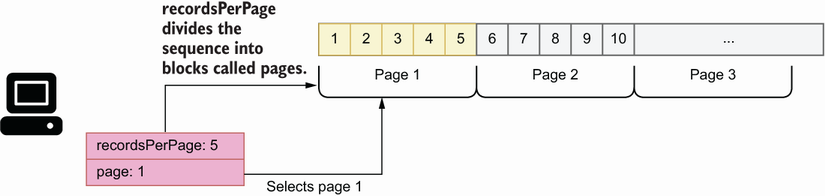


Figure 8.11 A more common approach to pagination is to let users decide how many results per page they want to see and let them select the specific page they want to get.

We enable pagination by adding a resultsPerPage argument to the query, as well as a page argument. To sort the result set, we expose a sort argument. The following snippet shows in bold the changes to the products() query after we add these arguments:

type Query {

products(available: Boolean, maxPrice: Float, minPrice: Float, **sort: String,**

**resultsPerPage: Int, page: Int):** [Product!]!

}

Offering numerous query arguments gives a lot of flexibility to our API consumers, but it can be cumbersome to set values for all of them. We can make our API easier to use by setting default values for some of the arguments. We’ll set a default sorting order, as well as a default value for the resultsPerPage argument and a default value for the page argument. The following code shows how we assign default values to some of the arguments in the products() query and includes a SortingOrder enumeration that constrains the values of the sort argument to either ASCENDING or DESCENDING.

Listing 8.14 Setting default values for query arguments

**enum SortingOrder {**  ①

**ASCENDING**

**DESCENDING**

**}**

type Query {

products(

maxPrice: Float

minPrice: Float

available: **Boolean = true**  ②

**sort: SortingOrder = DESCENDING** ③

**resultsPerPage: Int = 10**

**page: Int = 1**

): [Product!]!

}

① We declare the SortingOrder enumeration.

② We assign default values for some of the parameters.

③ We constrain sort’s values by setting its type to the SortingOrder enumeration.

The signature of the products() query is becoming a bit cluttered. If we keep adding arguments to it, it will become difficult to read and maintain. To improve readability, we can refactor the arguments out of the query specification into their own type. In GraphQL, we can define lists of parameters by using input types, which have the same look and feel as any other GraphQL object type, but they’re meant for use as input for queries and mutations.

Listing 8.15 Refactoring query arguments into input types

**input ProductsFilter {**  ①

**maxPrice: Float**  ②

**minPrice: Float**

**available: Boolean = true,**  ③

**sort: SortingOrder = DESCENDING**

**resultsPerPage: Int = 10**

**page: Int = 1**

**}**

type Query {

products(**input: ProductsFilter**): [Product!]! ④

}

① We declare the ProductsFilter input type.

② We define ProductsFilter’s parameters.

③ We assign default values to some parameters.

④ We set the input parameter’s type to ProductsFilter.

The remaining API queries, namely, allIngredients(), product(), and ingredient(), are shown in listing 8.16 in bold. allIngredients() returns a full list of ingredients and therefore takes no arguments, as in the case of the allProducts() query. Finally, product() and ingredient() return a single product or ingredient by ID, and therefore have a required id argument of type ID. If a product or ingredient is found for the provided ID, the queries will return the details of the requested item; otherwise, they’ll return null.

Listing 8.16 Specification for all the queries in the products API

type Query {

allProducts: [Product!]!

**allIngredients: [Ingredient!]!**

products(input: ProductsFilter!): [Product!]!

**product(id: ID!): Product**  ①

**ingredient(id: ID!): Ingredient**

}

① product() returns a nullable result of type Product.

Now that we know how to define queries, it’s time to learn about mutations, which are the topic of the next section.

## 8.9 Altering the state of the server with mutations

This section introduces GraphQL *mutations*: operations that allow us to trigger actions that change the state of the server. While the purpose of a query is to let us fetch data from the server, mutations allow us to create new resources, to delete them, or to alter their state. Mutations have a return value, which can be a scalar, such as a Boolean, or an object. This allows our API consumers to verify that the operation completed successfully and to fetch any values generated by the server, such as IDs.

In section 8.2, we discussed that the products API needs to support the following operations for adding, deleting, and updating resources in the server:

* addIngredient()
* updateStock()
* addProduct()
* updateProduct()
* deleteProduct()

In this section, we’ll document the addProduct(), updateProduct(), and deleteProduct() mutations. The specification for the other mutations is similar to these, and you can check them out in the GitHub repository provided with this book.

A GraphQL mutation looks similar to the signature of a function in Python: we define the name of the mutation, describe its parameters between parentheses, and provide its return type after a colon. Listing 8.17 shows the specification for the addProduct() mutation. addProduct() accepts a long list of arguments, and it returns a Product type. All the arguments are optional except name and type. We use type to indicate what kind of product we’re creating, a cake or a beverage. We also include a ProductType enumeration to constrain the values of the type argument to either cake or beverage. Since this mutation is used to create cakes and beverages, we allow users to specify properties of each type, namely hasFilling and hasNutsToppingOption for cakes, as well as hasCreamOnTopOption and hasServeOnIceOption for beverages, but we set them by default to false to make the mutation easier to use.

Listing 8.17 Defining a GraphQL mutation

enum ProductType { ①

cake

beverage

}

input IngredientRecipeInput {

ingredient: ID!

quantity: Float!

unit: MeasureUnit!

}

enum Sizes {

SMALL

MEDIUM

BIG

}

type Mutation { ②

addProduct(

name: String!

type: ProductType!

price: String

size: Sizes

ingredients: [IngredientRecipeInput!]!

hasFilling: Boolean = false

hasNutsToppingOption: Boolean = false

hasCreamOnTopOption: Boolean = false

hasServeOnIceOption: Boolean = false

): Product! ③

}

① We declare a ProductType enumeration.

② We declare mutations under the Mutation object type.

③ We specify the return type of addProduct().

You’d agree that the signature definition of the addProduct() mutation looks a bit cluttered. We can improve readability and maintainability by refactoring the list of parameters into their own type. Listing 8.18 shows how we refactor the addProduct() mutation by moving the list of parameters into an input type. AddProductInput contains all the optional parameters that can be set when we create a new product. We set aside the name parameter, which is the only required parameter when we create a new product. As we’ll see shortly, this allows us to reuse the AddProductInput input type in other mutations that don’t require the name parameter.

Listing 8.18 Refactoring parameters with input types

**input AddProductInput {** ①

**price: String** ②

**size: Sizes**

**ingredients: [IngredientRecipeInput!]!**

**hasFilling: Boolean = false** ③

**hasNutsToppingOption: Boolean = false**

**hasCreamOnTopOption: Boolean = false**

**hasServeOnIceOption: Boolean = false**

**}**

type Mutation {

addProduct(

name: String!

type: ProductType!

**input: AddProductInput!**

): Product! ④

}

① We declare the AddProductInput input type.

② We list AddProductInput’s parameters.

③ We assign default values to some parameters.

④ addProduct()’s input parameter has the AddProduct input type.

Input types not only help us make our specification more readable and maintainable, but they also allow us to create reusable types. We can reuse the AddProductInput input type in the signature of the updateProduct() mutation. When we update the configuration for a product, we may want to change only some of its parameters, such as the name, the price, or its ingredients. The following snippet shows how we reuse the AddProductInput parameters in updateProduct(). In addition to AddProductInput, we also include a mandatory product id parameter, which is necessary to identify the product we want to update. We also include the name parameter, which in this case is optional:

type Mutation {

updateProduct(id: ID!, input: AddProductInput!): Product!

}

Let’s now look at the deleteProduct() mutation, which removes a product from the catalogue. To do that, the user must provide the ID for the product they want to delete. If the operation is successful, the mutation returns true; otherwise, it returns false. The next snippet shows the specification for the deleteProduct() mutation:

deleteProduct(id: ID!): Boolean!

This concludes our journey through GraphQL’s SDL! You’re now equipped with everything you need to define your own API schemas. In chapter 9, we’ll learn how to launch a mock server using the products API specification and how to consume and interact with the GraphQL API.

## Summary

* GraphQL is a popular protocol for building web APIs. It shines in scenarios where it’s important to give API clients full control over the data they want to fetch and in situations where we have highly interconnected data.
* A GraphQL API specification is called a schema, and it’s written using the Schema Definition Language (SDL).
* We use GraphQL’s scalar types to define the properties of an object type: Booleans, strings, floats, integers, and IDs. In addition, we can also create our own custom scalar types.
* GraphQL’s object types are collections of properties, and they typically represent the resource or entities managed by the API server.
* We can connect objects by using edge properties, namely, properties that point to another object, and by using through types. Through types are object types that add additional information about how two objects are connected.
* To constrain the values of a property, we use enumeration types.
* GraphQL queries are operations that allow API clients to fetch data from the server.
* GraphQL mutations are operations that allow API clients to trigger actions that change the state of the server.
* When queries and mutations have long lists of parameters, we can refactor them into input types to increase readability and maintainability. Input types can also be reused in more than one query or mutation.

**1** This definition appears in the home page of the GraphQL specification: <https://graphql.org/>.

**2** I owe the comparison between GraphQL and SQL to Eve Porcello and Alex Banks, *Learning GraphQL, Declarative Data Fetching for Modern Web Apps* (O’Reilly, 2018), pp. 31–32.

**3** The GraphQL website maintains a list of runtimes available for building GraphQL servers in different languages: <https://graphql.org/code/>.

**4** In the literature about GraphQL, you’ll often find a digression about how GraphQL is inspired by graph theory, and how we can use some of the concepts from graph theory to illustrate the relationships between types. Following that tradition, the bidirectional relationship we refer to here is an example of an undirected graph, since the Supplier type can be reached from the Ingredient type, and vice versa. For a good discussion of graph theory in the context of GraphQL, see Eve Porcello and Alex Banks, *Learning GraphQL, Declarative Data Fetching for Modern Web Apps* (O’Reilly, 2018), pp. 15–30.