Overview

In Airlock, a user can be either a *host* or a *guest*. These two user types share some common attributes. For example, they both have names and profile pictures. They also have a few attributes that are specific to their role: only *hosts* have a profile bio and listings, and only *guests* have bookings.

To implement this business logic into our GraphQL schema, we can use an interface.

In this lesson, we will:

- Learn how to implement interface types in a GraphQL schema
- Learn how to resolve interface types

What is an interface?

An **interface** is an abstract type that defines a common set of fields that any number of object types can then include.

Interfaces are often used to represent an important relationship among different types with some shared behavior. For example, the Airlock schema defines separate types for `Host` and `Guest`, but the shared attributes between these two types are captured in a `User` interface.

When an object type uses an interface, it's called an **implementing object type**. We also say that the type "implements" the interface. For example, the `Host` type *implements* the `User` interface.

An interface specifies a contract that its implementing types must follow. In other words, an implementing object type *must* include all the fields defined on that interface. For this reason, we recommend creating meaningful interfaces, to avoid

unnecessary schema maintenance as your graph evolves.

Learn more: What makes an interface meaningful?

An implementing object type can also define any number of additional fields that *aren't* part of the interface.

Defining an interface

In a GraphQL schema, we define an interface using the `interface` keyword, then the name of the interface. After the curly braces, we define the fields as we've done before in other schemas using the schema definition language.

Here's what Airlock's `User` interface looks like:

server/schema.graphql

```
"Represents an Airlock user's common properties"
interface User {
  id: ID!
  "The user's first and last name"
  name: String!
  "The user's profile photo URL"
  profilePicture: String!
}
```

Any type that implements our `User` interface must define these exact fields with these exact return types (including nullability). It can also define any number of other fields. (More on that in a moment...)

Implementing the interface

Once an interface has been defined, it can be implemented by other types in the schema.

To define an implementing object type, we start by writing the keyword `type`, followed by the name of the type. Then, we add the keyword `implements`, followed by the name of the interface. Next, we add all the fields defined by the interface to the type definition.

Here's how Airlock defines the `Host` and `Guest` types, which both implement the `User` interface from the previous section:

server/schema.graphql

```
type Host implements User {
  id: ID!
  "The user's first and last name"
  name: String!
  "The user's profile photo URL"
  profilePicture: String!
  "The host's profile bio description, will be shown in the listing"
  profileDescription: String!
}

type Guest implements User {
  id: ID!
  "The user's first and last name"
  name: String!
  "The user's profile photo URL"
  profilePicture: String!
  "The reservations guest has"
  bookings: [Booking]!
}
```

Note that the `Host` and `Guest` types both also have additional types that *aren't* part of the `User` interface (`Host.profileDescription` and `Guest.bookings`).

Why do we need to repeat shared fields in the implementing types?

Returning an interface

An interface can also be used in the schema as a return type. In the Airlock schema, the `Review.author` field is a perfect example of a field that returns a `User` type. A review author can be any user, whether a host or a guest.

server/schema.graphql

```
type Review {
  # ... other fields
  "User that wrote the review"
  author: User!
}
```

> **Note:** The `Query.me` field is also a good example. We'll take a closer look at that field in the next lesson.

Resolving an interface

A field that returns an interface can return any object type that implements that interface. But this raises a question: for a given operation, how do we know which implementing type the field is returning?

For example, in Airlock the `Review.author` field might return a `Host` object, or it might return a `Guest` object. How would we know whether the object that's returned is a host or a guest?

To handle this, we need to define a special resolver function called `__resolveType`.

The `__resolveType` resolver

The `__resolveType` function is responsible for determining which implementing object type is being returned. It returns a string with the name of the corresponding object type.

For example, the `User` interface's `__resolveType` function should return either "Host" or "Guest" because those are the two implementing object types defined in the schema.

Unlike our other resolver functions, `__resolveType` takes in *three* optional arguments: `obj`, `context` and `info`.

- The first argument, `obj`, is the object returned by the resolver for the field returning the interface.
- The last two arguments remain the same as we covered in Lift-off II.

```
__resolveType(obj, context, info) {
  // logic to determine which type to return goes here
}
```

The logic for determining *which* object type is being returned depends on the application! In Airlock's case, each user in the `accounts` database has a `role` attribute that is set to either "Host" or "Guest". This is perfect to use, because the two types that implement this interface are also `Host` or `Guest`.

```
User: {
  __resolveType(user) {
    return user.role; // returns "Host" or "Guest"
  },
},
```

> **Note:** Because we don't use either of the last two arguments (`context` or `info`), we omitted them from the function call. We also renamed the first argument `obj` to `user` to better clarify what the object is.

Test it in Apollo Studio

Let's use Apollo Studio to try out a query that resolves an interface.

1. In a web browser, open `http://localhost:4000` in Apollo Studio Sandbox.

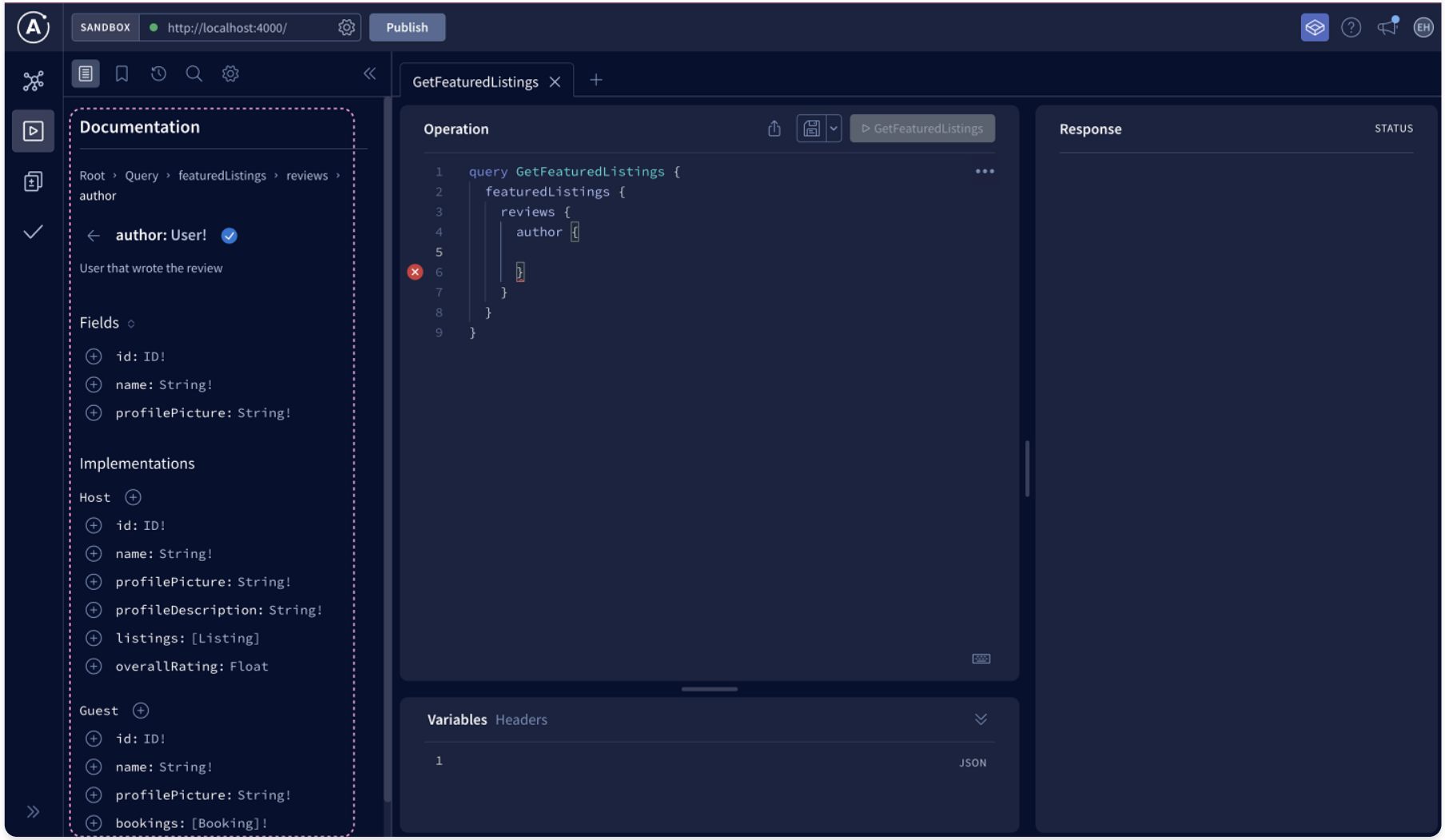
2. In the *Operations* tab, let's start building up our query. We'll start by adding the `featuredListings` field and its `reviews` subfield. From here, we can add the `author` field, which we know resolves to the `User` interface.

```
query GetFeaturedListings {
  featuredListings {
    reviews {
      author {
        # TODO!
      }
    }
  }
}
```

Copy

When we add the `author` field to our query, notice how the *Documentation* panel updates to show us the possible *Implementations* for this interface. Here, we can see the shared fields encapsulated in the `User` interface, alongside those that are specific either to `Host` or `Guest`.

https://studio.apollographql.com/sandbox/explorer



For now, let's just look at the fields that both implementing types *share* in the `User` interface: `id`, `name` and `profilePicture`.

3. Add the `id`, `name`, and `profilePicture` fields to the query in Apollo Studio. Here's what the final query should look like:

```
query GetFeaturedListings {
  featuredListings {
    reviews {
```

Copy

```
      author {
        id
        name
        profilePicture
      }
    }

  }
}
```

4. When we run the query, we see our data returning. Fantastic! The response should look something like the object below.

See JSON response ›

See it in the Airlock codebase

Check out the interfaces defined in the Airlock codebase. You can find them in the `server/schema.graphql` file.

Practice

When should you use an interface in a GraphQL schema?

- [ ] To prevent implementing object types from defining fields that aren't in the interface
- [ ] To represent meaningful shared behavior and relationships between object types
- [ ] To return one of multiple object types from a particular schema field
- [ ] To define a common set of fields for object types

Submit

Use the following schema to complete the code challenge below:

```
type Query {
  availableBooks: [Book]
  borrowedBooks(userId: ID!): [Book]
}

interface Book {
  isbn: ID!
  title: String!
  genre: String!
}

type PictureBook implements Book {
  isbn: ID!
  title: String!
  genre: String!
  numberOfPictures: Int
  isInColor: Boolean
}

type YoungAdultNovel implements Book {
  isbn: ID!
  title: String!
  genre: String!
  wordCount: Int
  numberOfChapters: Int
}
```

Code Challenge!

Write the `__resolveType` resolver for the `Book` interface. To determine the type of `Book`, you can use the book's `hasPictures` property. `PictureBook` types have this property set to `true`, while `YoungAdultNovel` types do not.

```
const resolvers = {
  Book: {

  }
};
```

Run

Key takeaways

- An interface defines a common set of fields that any number of object types must include.
- We recommend creating meaningful interfaces to avoid unnecessary schema maintenance as the graph evolves.

Up next

So far, we've learned how to implement an interface and resolve the fields that are shared between all implementing types.

In the next lesson, we'll see how to query fields that belong exclusively to one implementing type and not another. To do that, we'll need to learn about **query fragments**.

← Previous

2 tasks remaining ↑