":`Business`"	["Business"]	"state"	["String"]	true
":`Business`"	["Business"]	"address"	["String"]	true
":`Business`"	["Business"]	"location"	["Point"]	true
":`Business`"	["Business"]	"businessId"	["String"]	true

We'll use this table in a few moments when we construct GraphQL type definitions that describes this graph.

NODE.JS APP

Now that we have our Neo4j database loaded with our sample dataset let's set up a new node.js project for our GraphQL API:

```
npm init -y
```

And install our dependencies:

- *neo4j*-graphql.js: A package that makes it easier to use GraphQL and Neo4j together. neo4j-graphql-js translates GraphQL queries to a single Cypher query, eliminating the need to write queries in GraphQL resolvers and for batching queries. It also exposes the Cypher query language through GraphQL via the `@cypher` schema directive.
- apollo-server: Apollo Server is an open-source GraphQL server that works with any GraphQL schema built with graphql.js, including neo4j-graphql.js. It also has options for working with many different Node.js webserver frameworks, or the default Express.js.
- neo4j-driver: The Neo4j drivers allow for connecting to a Neo4j instance, either local or remote, and executing Cypher queries over the Bolt protocol. Neo4j drivers are available in many different languages and here we use the Neo4j JavaScript driver.

```
npm install neo4j-graphql-js apollo-server neo4j-driver
```

Now create a new file `index.js` and let's add initial code in the following listing.

Listing 4.4 index.js: Initial GraphQL API code

```
const { ApolloServer } = require("apollo-server");
const neo4j = require("neo4j-driver");
const { makeAugmentedSchema } = require("neo4j-graphql-js");

const typeDefs = /* GraphQL */ ``;

const schema = makeAugmentedSchema({
  typeDefs
});
```

Here we pull in our dependencies.

This line serves as a placeholder for our GraphQL type definitions to be filled in later.

makeAugmentedSchema generates resolvers for our type definitions.

Here we start the GraphQL server.

Our generated GraphQL schema is passed to Apollo Server.

```
const server = new ApolloServer({
  schema
});

server.listen().then(({ url }) => {
  console.log(`GraphQL server ready at ${url}`);
});
```

This is the basic structure for our GraphQL API application code. We can run it on the command line using the `node` command:

```
node index.js
```

However, we'll quickly see an error message complaining that we haven't provided GraphQL type definitions.

```
➜ node index.js
/Users/lyonwj/api/node_modules/neo4j-graphql-js/dist/index.js:293
  if (!typeDefs) throw new Error('Must provide typeDefs');
  ...
```

We must provide either GraphQL type definitions or a GraphQL schema object to `makeAugmentedSchema` that defines the GraphQL API, so the next step is to fill in our GraphQL type definitions.

4.3.2 Generated GraphQL schema from type definitions

Following the GraphQL-First approach described previously, our GraphQL type definitions drive the API specification. In this case, we know what data we want to expose (our sample dataset loaded in Neo4j), so we can refer to the table of node properties above and apply a simple rule as we create our GraphQL type definitions: node labels become types, taking on the node properties as fields. We also need to define relationship fields in our GraphQL type definitions. Let's first look at the complete type definitions and then explore how we define relationship fields in the following listing.

Listing 4.5 index.js: GraphQL type definitions

```
const typeDefs = /* GraphQL */ `
  type Business {
    businessId: ID!
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
  }

  type User {
    userID: ID!
    name: String!
    reviews: [Review] @relation(name: "WROTE", direction: "OUT")
  }
```

```
  type Review {
    reviewId: ID!
    stars: Float!
    date: Date!
    text: String
    user: User @relation(name:"WROTE", direction: "IN")
    business: Business @relation(name: "REVIEWS", direction: "OUT")
  }

  type Category {
    name: String!
    businesses: [Business] @relation(name: "IN_CATEGORY", direction: "IN")
  }
`;
```

@RELATION GRAPHQL SCHEMA DIRECTIVE

In the property graph model used by Neo4j, every relationship has a direction and type. To represent this in GraphQL, we use GraphQL schema directives, specifically the `@relation` schema directive. A directive is like an annotation in our GraphQL type definitions. It's an identifier preceded by the `@` character, which may then optionally contain a list of named arguments. Schema directives are GraphQL's built-in extension mechanism, indicating some custom logic on the server.

When defining relationship fields using the `@relation` directive, the `name` argument indicates the relationship type stored in Neo4j and the `direction` argument indicates the relationship direction.

In addition to schema directives, directives can also be used in GraphQL queries to indicate specific behavior. We'll see examples of query directives when we explore GraphQL clients.

4.3.3 Generated data fetching

The autogenerated resolvers in our GraphQL schema need to access our Neo4j database using the Neo4j JavaScript driver and they expect a driver instance to be injected into the context object that's passed to each resolver. By convention, the driver is injected under the key `driver` in the context object, as shown in the following listing.

Listing 4.6 index.js: Creating a Neo4j driver instance

```
const driver = neo4j.driver(
  "bolt://localhost:7687",
  neo4j.auth.basic("neo4j", "letmein")
);

const server = new ApolloServer({
  schema,
  context: { driver }
});
```

Creating a Neo4j driver instance using credentials for our Neo4j database.

We inject this driver instance into the context object.

4.3.4 Configuring the generated API

We mentioned that the schema augmentation process adds queries and mutations for each type defined in the type definitions. We can configure the generated API, disabling queries or mutations altogether, or specify that certain types be excluded.

DISABLING AUTO-GENERATED QUERIES AND MUTATIONS

Because we're initially only focusing on the query API, let's disable all generated mutations in the following listing.

Listing 4.7 index.js: Disabling mutations

```
const schema = makeAugmentedSchema({
  typeDefs,
  resolvers,
  config: {
      mutation: false
  }
});
```

EXCLUDING TYPES

We can also pass an array of types to be excluded from the generated queries or mutations, as shown in the following listing.

Listing 4.8 index.js: Disabling specific types

```
const schema = makeAugmentedSchema({
  typeDefs,
  resolvers,
  config: {
      mutation: false,
      query: {
          exclude: ["MySecretType"]
      }
  }
});
```

Now, let's run our API application:

```
node index.js
```

As output, we should see the address where our API application is listening, in this case on port 4000 on `localhost`.

```
➜  node index.js
GraphQL server ready at http://localhost:4000/
```

Navigate to http://localhost:4000 in your web browser and you should see the familiar GraphQL Playground interface. Open the "Docs" tab in GraphQL to see the generated API (figure 4.5). Spend a few minutes looking through the Query field descriptions, and you'll notice arguments have been added to types for things such as ordering, pagination, and filtering.

4.4 Basic queries

Now that we have our GraphQL server powered by Apollo Server and neo4j-graphql.js up and running, let's start querying our API using GraphQL Playground. Looking at the `Docs` tab in GraphQL Playground, we can see the API entrypoints (in GraphQL

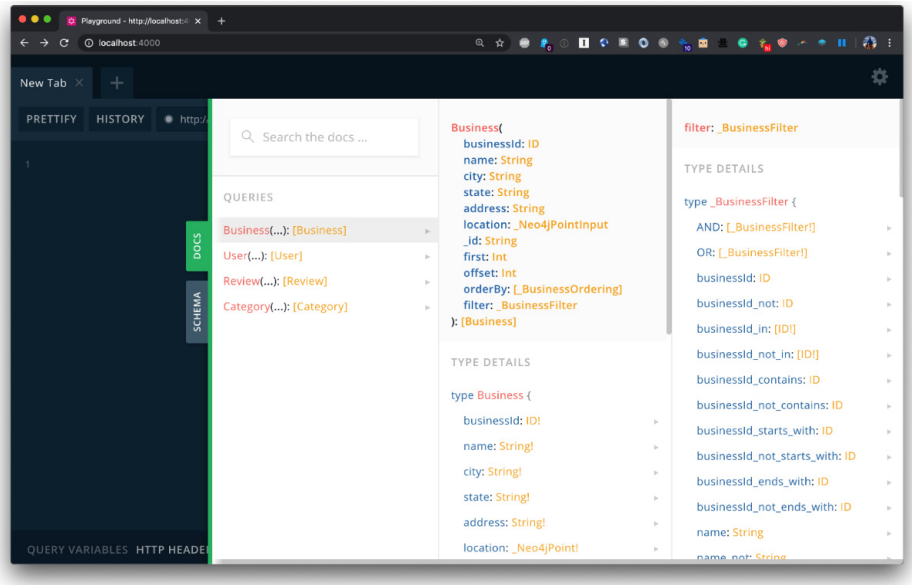

Figure 4.5 **GraphQL Playground showing our generated API.**

parlance, each Query type field is an entry point to the API) available to us: `Business`, `User`, `Review`, `Category`, one for each type defined in our type definitions.

Let's start by querying for all businesses and return only the name field, as shown in the following listing.

Listing 4.9 GraphQL query

```
{
  Business {
    name
  }
}
```

If we run this query in GraphQL Playground, we should see the following results listing businesses and their names only.

```
{
  "data": {
    "Business": [
      {
        "name": "Missoula Public Library"
      },
      {
        "name": "Ninja Mike's"
      },
      {
        "name": "KettleHouse Brewing Co."
      },
```

```
      {
        "name": "Imagine Nation Brewing"
      },
      {
        "name": "Market on Front"
      },
      {
        "name": "Hanabi"
      },
      {
        "name": "Zootown Brew"
      },
      {
        "name": "Ducky's Car Wash"
      },
      {
        "name": "Neo4j"
      }
    ]
  }
}
```

Neat! This data has been fetched from our Neo4j instance for us, and we didn't even need to write any resolvers!

If we check the console output in the terminal, we can see the generated Cypher query logged to the terminal in the following listing.

Listing 4.10 Generated Cypher query

```
MATCH (`business`:`Business`) RETURN `business` { .name } AS `business`
```

We can add additional fields to the GraphQL query and those fields will be added to the generated Cypher query, returning only the data needed.

For example, the following GraphQL query adds the address of the business as well as the name field, as shown in the following listing.

Listing 4.11 GraphQL query

```
{
  Business {
    name
    address
  }
}
```

The Cypher translation of the GraphQL query also now includes the address field, as shown in the following listing.

Listing 4.12 Generated Cypher query

```
MATCH (`business`:`Business`) RETURN `business` { .name , .address } AS
     `business`
```

And finally, when we examine the results of the GraphQL query, we now see an address listed for each business.


```
{
  "data": {
    "Business": [
      {
        "name": "Missoula Public Library",
        "address": "301 E Main St"
      },
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St"
      },
      {
        "name": "KettleHouse Brewing Co.",
        "address": "313 N 1st St W"
      },
      {
        "name": "Imagine Nation Brewing",
        "address": "1151 W Broadway St"
      },
      {
        "name": "Market on Front",
        "address": "201 E Front St"
      },
      {
        "name": "Hanabi",
        "address": "723 California Dr"
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St"
      },
      {
        "name": "Ducky's Car Wash",
        "address": "716 N San Mateo Dr"
      },
      {
        "name": "Neo4j",
        "address": "111 E 5th Ave"
      }
    ]
  }
}
```


Next, let's take advantage of several of the features of the generated GraphQL API.

4.5 Ordering and pagination

Each input type field includes `first`, `offset` and `orderBy` arguments to enable ordering and pagination. In the next listing, we search for the first three businesses, ordered by the value of the `name` field.

Listing 4.13 Initial GraphQL API code

```
{
  Business(first: 3, orderBy: name_asc) {
    name
  }
}
```

Ordering enums are generated for each type, offering ascending and descending options for each field. For example, in the next listing we see the ordering enum generated for the `Business` type.

Listing 4.14 Initial GraphQL API code

```
enum _BusinessOrdering {
  businessId_asc
  businessId_desc
  name_asc
  name_desc
  city_asc
  city_desc
  state_asc
  state_desc
  address_asc
  address_desc
  _id_asc
  _id_desc
}
```

Running our query returns businesses now ordered by name, as shown in the following listing.

Listing 4.15 Initial GraphQL API code

```
{
  "data": {
    "Business": [
      {
        "name": "Ducky's Car Wash"
      },
      {
        "name": "Hanabi"
      },
      {
        "name": "Imagine Nation Brewing"
      }
    ]
  }
}
```

If we switch to the terminal, we can see the Cypher query generated from our GraphQL query, which now includes `ORDER BY` and `LIMIT` clauses that map to our `first` and `orderBy` GraphQL arguments. The ordering and limiting is executed in