side because we're spending less compute resources at the data layer, less overall data sent over the wire, and reduced latency by being able to render our application view with a single network request to the API service.

GraphQL specification

GraphQL is a specification for client-server communication that describes the features, functionality, and capability of the GraphQL API query language. Having this specification gives a clear guide of how to implement your GraphQL API and clearly defines what is and what isn't GraphQL.

REST doesn't have a specification, instead there are many different implementations, from REST-ish to Hypermedia As The Engine Of Application State (HATEOAS). Having a specification as part of GraphQL simplifies debates over endpoints, status codes, and documentation. All this comes built in with GraphQL which leads to productivity wins for developers an API designer. The specification provides a clear path for API implementors.

With GraphQL it's graphs all the way down

REST models itself as a hierarchy of resources, yet most interactions with APIs are done in terms of relationships. For example, given our movie query, for this movie, show me all of the actors connected to it, and for each actor show me all the other movies they've acted in—we're querying for relationships.

GraphQL can also help unify data from disparate systems. Because GraphQL is data-layer agnostic we can build GraphQL APIs that integrate data from multiple services or microservices together and provide a clear way to integrate data from these different system into a single GraphQL schema.

GraphQL can also be used to compartmentalize data fetching in the application in a component-based data interaction pattern. Because each GraphQL query can describe exactly the graph traversal and fields to be returned, encapsulating these queries with application components can help simplify application development and testing. We'll see how to apply this once we start building our React application.

Introspection

Introspection is a powerful feature of GraphQL that allows us to ask a GraphQL API for the types and queries it supports. Introspection becomes a way of self-documenting the API. Tools that make use of introspection can provide human readable API documentation, visualization tooling, and leverage code generation to create API clients.

1.1.4 Disadvantages of GraphQL

GraphQL isn't a silver bullet, and we shouldn't think of GraphQL as the solution to all of our API-related problems. One of the most notable challenges of adopting GraphQL is that several well-understood practices from REST don't apply when using GraphQL. For example, HTTP status codes are commonly used to convey success, failure, and other cases of a REST request: 200 OK means our request was successful and 404 Not Authorized means we forgot an auth token or don't have the correct permissions for the resource requested. However, with GraphQL, each request returns 200 OK regardless if the request was a complete success or not. This makes error handling a bit different in the GraphQL world. Instead of a single status code describing the result of our request, GraphQL errors are typically returned at the field level of the selection set. This means that we may have successfully retrieved a part of our GraphQL query, while other fields returned errors.

Web caching is another well understood area from REST that's handled a bit differently with GraphQL. With REST, caching the response for `/movie/123` is possible because we can return the same exact result for each GET request. This isn't possible with GraphQL because each request could contain a different selection set, meaning we can't simply return a cached result. This is mitigated by most GraphQL clients implementing client caches at the application level and in practice most of the time our GraphQL requests are in an authenticated environment where caching isn't applicable.

Another challenge is that of exposing arbitrary complexity to the client and related performance considerations. If the client is free to compose queries as they wish, how can we ensure these queries don't become so complex as to impact performance significantly or overwhelm the computing resources of our backend infrastructure? Fortunately, GraphQL tooling allows us to restrict the depth of the queries used and further restrict the queries that can be run to a whitelisted selection of queries (known as persisted queries). A related challenge is implementing rate limiting. With REST we could restrict the number of requests that can be made in a certain time period; however, with GraphQL this becomes more complicated since the client could be requesting multiple objects in a single query. This results in bespoke query costing implementations to address rate limiting.

Finally, the so called "n+1 query problem" is a common problem in GraphQL data fetching implementations that can result in multiple round trips to the database and negatively impact performance. Consider the case where we request information about a movie and all actors of the movie. In the database we might store a list of actor IDs associated with each movie, which is returned with our request for the movie details. In a naive GraphQL implementation we then need to retrieve the actor details, and we must make a separate request to the database for each actor object, resulting in a total of n (the number of actors) + 1 (the movie) queries to the database. To address the n+1 query problem, tools such as DataLoader allow us to batch requests to the database, increasing performance, and database integrations such as neo4j-graphql.js and PostGraphile allow us to generate a single database query from an arbitrary GraphQL request.

GraphQL limitations

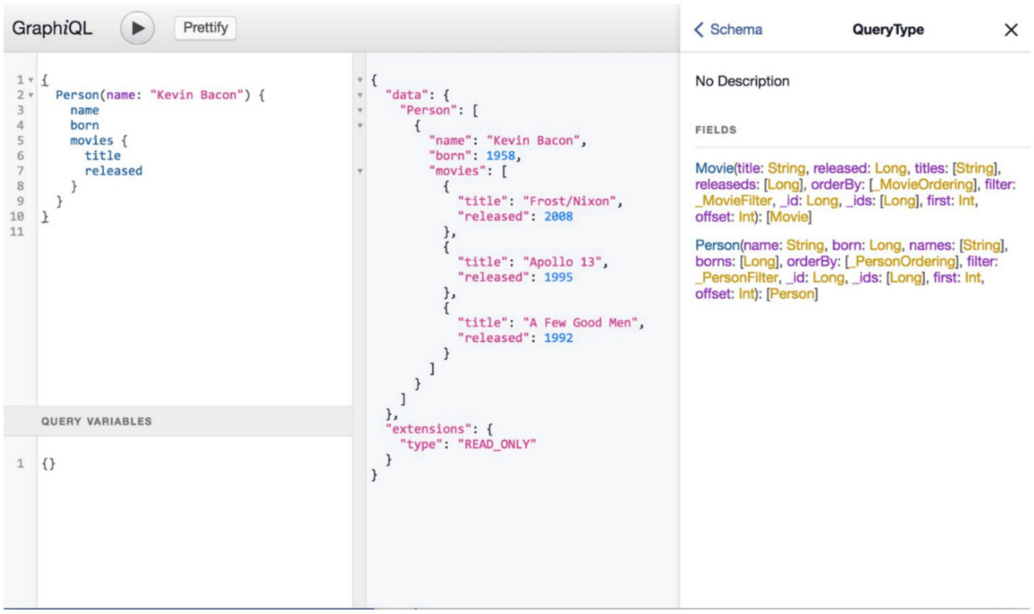
It's important to understand that GraphQL is an API query language and not a database query language. GraphQL lacks semantics for many complex operations required of database query languages, such as aggregations, projects, and variable length path traversals.

1.1.5 GraphQL tooling

In this section we review GraphQL specific tooling that will help us build, test, and query our GraphQL API.

GraphiQL

GraphiQL is an in-browser tool for exploring and querying GraphQL APIs. With GraphQL we can execute GraphQL queries against a GraphQL API and view the results. Thanks to GraphQL's introspection feature we can view the types, fields, and queries supported by the GraphQL API we've connected to. In addition, because of the GraphQL type system we have immediate query validation as we construct our query (figure 1.5).



```
{
  Person(name: "Kevin Bacon") {
    name
    born
    movies {
      title
      released
    }
  }
}
```

```
{}
```

```
{
  "data": {
    "Person": [
      {
        "name": "Kevin Bacon",
        "born": 1958,
        "movies": [
          {
            "title": "Frost/Nixon",
            "released": 2008
          },
          {
            "title": "Apollo 13",
            "released": 1995
          },
          {
            "title": "A Few Good Men",
            "released": 1992
          }
        ]
      }
    ]
  },
  "extensions": {
    "type": "READ_ONLY"
  }
}
```

No Description

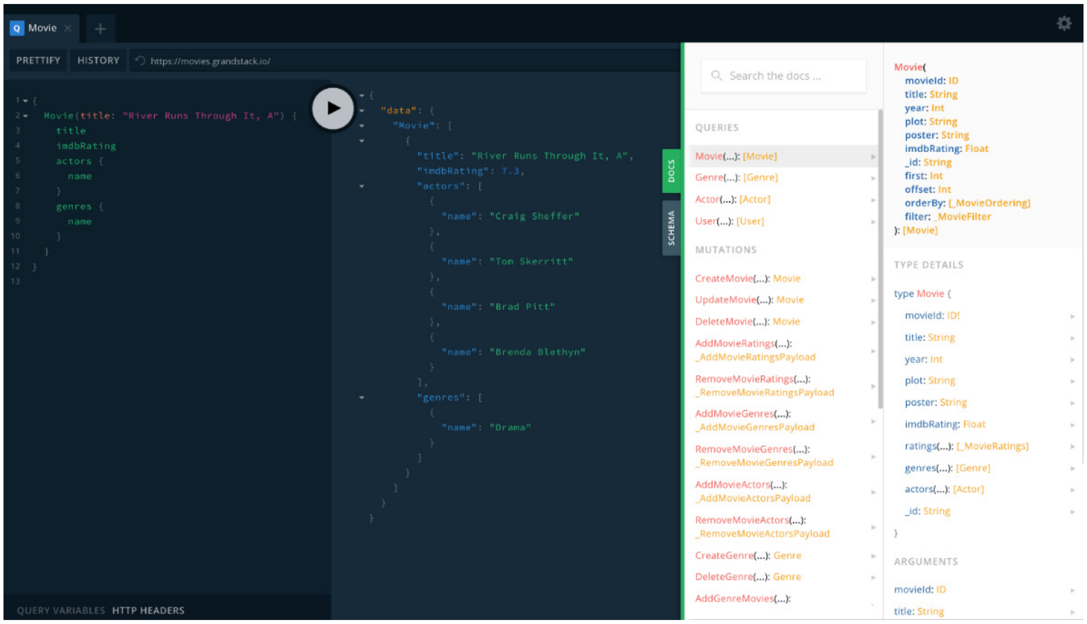
Movie(title: String, released: Long, titles: [String], releaseds: [Long], orderBy: [_MovieOrdering], filter: _MovieFilter, _id: Long, _ids: [Long], first: Int, offset: Int): [Movie]

Person(name: String, born: Long, names: [String], borns: [Long], orderBy: [_PersonOrdering], filter: _PersonFilter, _id: Long, _ids: [Long], first: Int, offset: Int): [Person]

Figure 1.5 GraphiQL screenshot.

GraphQL Playground

Similar to GraphiQL, GraphQL Playground (figure 1.6) is an in-browser tool for executing GraphQL queries, viewing the results, and exploring the GraphQL API's schema, powered by GraphQL's introspection features. GraphQL Playground has a few additional features such as viewing GraphQL type definitions, searching through the GraphQL schema, and easily adding request headers (such as those required for authentication). We include GraphQL Playground in addition to GraphiQL because certain tools (such as Apollo Server) expose GraphQL Playground by default, while others expose GraphiQL.



```
{
  Movie(title: "River Runs Through It, A") {
    title
    imdbRating
    actors {
      name
    }
    genres {
      name
    }
  }
}
```

```
{
  "data": {
    "Movie": [
      {
        "title": "River Runs Through It, A",
        "imdbRating": 7.3,
        "actors": [
          {
            "name": "Craig Sheffer"
          },
          {
            "name": "Tom Skerritt"
          },
          {
            "name": "Brad Pitt"
          },
          {
            "name": "Brenda Blethyn"
          }
        ],
        "genres": [
          {
            "name": "Drama"
          }
        ]
      }
    ]
  }
}
```

QUERIES

Movie(...): [Movie]
Genre(...): [Genre]
Actor(...): [Actor]
User(...): [User]

MUTATIONS

CreateMovie(...): Movie
UpdateMovie(...): Movie
DeleteMovie(...): Movie
AddMovieRatings(...): _AddMovieRatingsPayload
RemoveMovieRatings(...): _RemoveMovieRatingsPayload
AddMovieGenres(...): _AddMovieGenresPayload
RemoveMovieGenres(...): _RemoveMovieGenresPayload
AddMovieActors(...): _AddMovieActorsPayload
RemoveMovieActors(...): _RemoveMovieActorsPayload
CreateGenre(...): Genre
DeleteGenre(...): Genre
AddGenreMovies(...):

```
Movie(
  movieId: ID
  title: String
  year: Int
  plot: String
  poster: String
  imdbRating: Float
  _id: String
  first: Int
  offset: Int
  orderBy: [_MovieOrdering]
  filter: _MovieFilter
): [Movie]
```

TYPE DETAILS

```
type Movie {
  movieId: ID!
  title: String
  year: Int
  plot: String
  poster: String
  imdbRating: Float
  ratings(...): [_MovieRatings]
  genres(...): [Genre]
  actors(...): [Actor]
  _id: String
}
```

ARGUMENTS

movieId: ID
title: String

Figure 1.6 GraphQL Playground screenshot.

1.2 React

React is a declarative, component-based library for building user interfaces using JavaScript. React uses a virtual DOM (a copy of the actual Document Object Model) to efficiently calculate DOM updates required to render views as state and data changes. This means users design views that map to application data and React handles rendering the DOM updates efficiently. Components encapsulate data handling and user interface rendering logic without exposing their internal structure so they can be easily composed together to build complex applications.

1.2.1 React components

Let's examine a simple React component in the following listing.

Listing 1.5 A simple GraphQL component

Components can be either class components or functional components. Here we implement a class component, inheriting from the `React.Component` class.

```
class MovieTitleComponent extends React.Component {
  constructor() {
    super()
    this.state = {title: 'River Runs Through It, A'}
  }

  render() {
    return (
```

The constructor for our class component is called when our component is first initialized.

Class components can store data in state. We can set the initial state value in the constructor.

The `render` method defines what user interface elements the component will render.

```
      <div>{this.state.title}</div>
    )
  }
}
```

Here we access the `title` value from our component state and render that inside a `div` tag.

Component libraries

Because components encapsulate data handling and user interface rendering logic and are easily composable, it's practical to distribute libraries of components that can be used as dependencies of your project for quickly leveraging complex styling and user interface design. We'll use the Material-UI component library that will allow us to import many popular common user interface components such as a grid layout, data table, navigation, and inputs.

1.2.2 JSX

React is typically used with a JavaScript language extension called JSX. JSX looks similar to XML and is a powerful way of building user interfaces in React and composing React components. It is possible to use React without JSX, but most users prefer the readability and maintainability that JSX offers. We'll introduce JSX in chapter 5.

We'll cover React concepts such as unidirectional data flow, props and state, and data fetching in chapter 5.

1.2.3 React tooling

Here we review useful tooling that will help us build, develop, and troubleshoot React applications.

Create React App

Create React App is a command line tool that can be used to quickly create the scaffolding for a React application, taking care of configuring build settings, installing dependencies, and templating a simple React application to get started.

React Chrome Devtools

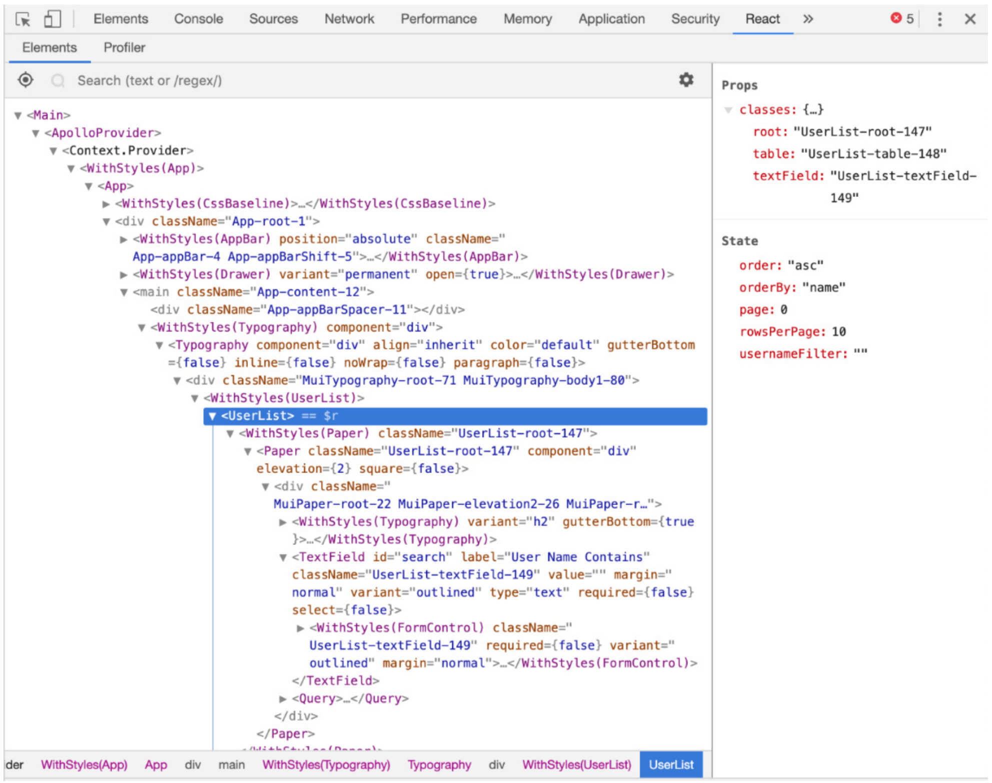
React Chrome Devtools is a browser integration that lets us inspect a React application, seeing under the hood the component hierarchy and the props and state of each component while our application is running (figure 1.7).

1.3 Apollo

Apollo is a collection of tooling to make it easier to use GraphQL, both on the server and the client. We'll use Apollo Server, a Node.js library for building our GraphQL API and Apollo Client, a client-side JavaScript library for querying our GraphQL API from our application.

1.3.1 Apollo Server

Apollo Server allows us to easily spin up a Node.js server serving a GraphQL endpoint by defining our type definitions and resolver functions. Apollo Server can be used with many different web frameworks; however, the default and most popular is



```
▼ <Main>
  ▼ <ApolloProvider>
    ▼ <Context.Provider>
      ▼ <WithStyles(App)>
        ▼ <App>
          ▶ <WithStyles(CssBaseline)>…</WithStyles(CssBaseline)>
          ▼ <div className="App-root-1">
            ▶ <WithStyles(AppBar) position="absolute" className="
              App-appBar-4 App-appBarShift-5">…</WithStyles(AppBar)>
            ▶ <WithStyles(Drawer) variant="permanent" open={true}>…</WithStyles(Drawer)>
            ▼ <main className="App-content-12">
                <div className="App-appBarSpacer-11"></div>
              ▼ <WithStyles(Typography) component="div">
                ▼ <Typography component="div" align="inherit" color="default" gutterBottom
                  ={false} inline={false} noWrap={false} paragraph={false}>
                  ▼ <div className="MuiTypography-root-71 MuiTypography-body1-80">
                    ▼ <WithStyles(UserList)>
                      ▼ <UserList> == $r
                        ▼ <WithStyles(Paper) className="UserList-root-147">
                          ▼ <Paper className="UserList-root-147" component="div"
                            elevation={2} square={false}>
                            ▼ <div className="
                              MuiPaper-root-22 MuiPaper-elevation2-26 MuiPaper-r…">
                              ▶ <WithStyles(Typography) variant="h2" gutterBottom={true
                                }>…</WithStyles(Typography)>
                              ▼ <TextField id="search" label="User Name Contains"
                                className="UserList-textField-149" value="" margin="
                                normal" variant="outlined" type="text" required={false}
                                select={false}>
                                ▶ <WithStyles(FormControl) className="
                                  UserList-textField-149" required={false} variant="
                                  outlined" margin="normal">…</WithStyles(FormControl)>
                                </TextField>
                              ▶ <Query>…</Query>
                              </div>
                            </Paper>
```

```
Props
▼ classes: {…}
    root: "UserList-root-147"
    table: "UserList-table-148"
    textField: "UserList-textField-
              149"

State
  order: "asc"
  orderBy: "name"
  page: 0
  rowsPerPage: 10
  usernameFilter: ""
```

Figure 1.7 React Chrome Devtool.

Express.js. Apollo Server can also be used with serverless functions such as Amazon Lambda and Google Cloud Functions.

Apollo Server can be installed with npm: `npm install apollo-server`.

1.3.2 Apollo Client

Apollo Client is a JavaScript library for querying GraphQL APIs and has integrations with many frontend frameworks, including React, Vue.js as well as native mobile versions for iOS and Android. We'll use the React Apollo Client integration to implement data fetching via GraphQL in our React components. Apollo Client handles client data caching and can also be used to manage local state data.

The React Apollo Client library can be installed with npm: `npm install react-apollo`.

1.4 Neo4j database

Neo4j is an open-source native graph database. Unlike other databases that use tables or documents for the data model, the data model used with Neo4j is a graph, specifically known as the Property Graph data model, and allows us to model, store, and query our data as a graph. Graph databases such as Neo4j are optimized for working with graph data and executing complex graph traversals, such as those defined by GraphQL queries.

One of the benefits of using a graph database with GraphQL is that we maintain the same data model throughout our application stack, working with graphs on both the frontend, API, and backend. Another benefit has to do with the performance optimizations graph databases make versus other database systems, such as relational databases. Many GraphQL queries end up nested several levels deep—the equivalent of a `JOIN` operation in a relational database. Graph databases are optimized for performing these graph traversal operations efficiently, so a graph database is a natural fit for the backend of a GraphQL API.

NOTE It's important to note that we aren't querying the database directly with GraphQL. While there are database integrations for GraphQL, it's important to realize the GraphQL API is a layer that sits between our application and the database.

1.4.1 Property graph data model

Like many graph databases Neo4j uses a property graph model. The components of the property graph model are (figure 1.8):

- Nodes: The entities, or objects, in our data model
- Relationships: Connections between nodes
- Labels: A grouping semantic for nodes
- Properties: Key-value pair attributes, stored on nodes and relationships

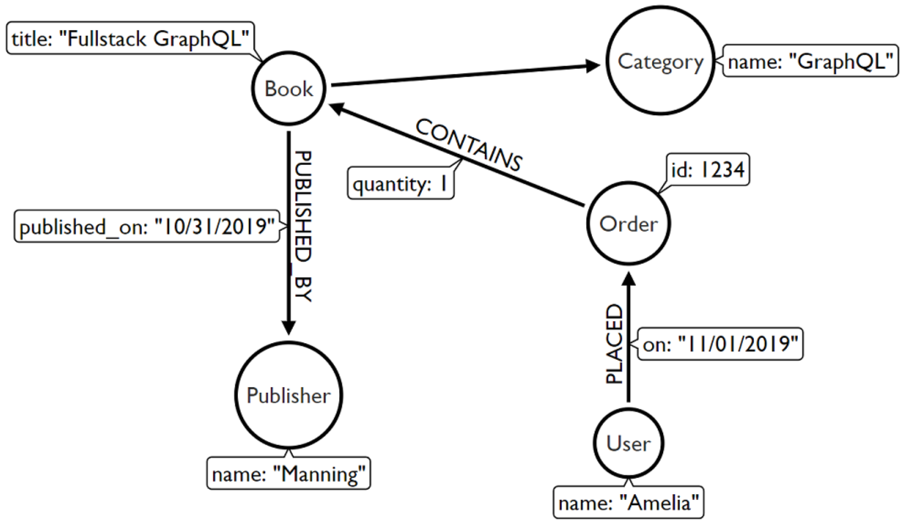


Figure 1.8 Property graph example modeling an order for a book placed by a user and how the data is connected as a graph.

The property graph data model allows us to express complex, connected data in a flexible way. This data model also has the additional benefit of closely mapping to the way we often think about data when dealing with a domain.

1.4.2 Cypher query language

Cypher is a declarative graph query language, used by Neo4j and other graph databases and graph compute engines. You can think of Cypher as similar to SQL, but instead designed for graph data. A major feature of Cypher is that of pattern matching. With graph pattern matching in Cypher, we can define the graph pattern using ASCII-art like notation. Cypher is an open standard through the openCypher project. Let's look at a simple Cypher example in the following listing, querying for movies and actors connected to these movies.

Listing 1.6 Simple Cypher query

`MATCH` is used to search for a graph pattern described using an ASCII-art like notation. In the pattern, nodes are defined within parentheses, for example `(m:Movie)`. The `:Movie` indicates we should match nodes with the label `Movie` and the `m` before the colon becomes a variable that is bound to any nodes that match the pattern. We can refer to `m` later throughout the query. Relationships are defined by square brackets, for example • `[r:ACTED_IN]` – and follow a similar convention where `:ACTED_IN` declares the `ACTED_IN` relationship type and `r` becomes a variable we can refer to later in the query to represent any relationships matching that pattern.

```
MATCH (m:Movie)<-[r:ACTED_IN]-(a:Actor)
RETURN m,r,a #B
```

In the `RETURN` clause we specify the data to be returned by the query. Here we specify the variables `m`, `r`, and `a`, variables that were defined in the `MATCH` clause above.

Cypher is an open source query language through the openCypher project and several other graph databases and graph systems implement Cypher.

1.4.3 Neo4j tooling

We'll use Neo4j Desktop for managing our Neo4j instances locally, and Neo4j Browser, a developer tool for querying and interacting with our Neo4j database. For querying Neo4j from our GraphQL API we will use the JavaScript Neo4j client driver as well as neo4j-graphql.js, a GraphQL integration for Neo4j.

NEO4J DESKTOP

Neo4j Desktop is Neo4j's command center. From Neo4j Desktop (figure 1.9) we can manage Neo4j database instances, including editing configuration, installing plugins and graph apps (such as visualization tools), and access admin level features such as dump/load database. Neo4j Desktop is the default download experience for Neo4j.

NEO4J BROWSER

Neo4j Browser (figure 1.10) is a query workbench for Neo4j. With Neo4j Browser we can query the database with Cypher and visualize the results, either as a graph visualization or tabular results.