1 What's the GRANDstack?

This chapter covers

- Understanding the GRANDstack
- Reviewing each technology in the stack
- Looking at the full-stack application we'll build

In this chapter we look at the technologies we'll use throughout the book, specifically:

- **G**raphQL: For building our API
- **R**eact: For building our user interface and JavaScript client web application
- **A**pollo: Tools for working with GraphQL on both the server and client
- **N**eo4j **D**atabase: The database we'll use for storing and manipulating our application data

Together these technologies make up the **GRAND**stack, a full-stack framework for building applications using GraphQL (figure 1.1).

Throughout the course of this book we'll use GRANDstack to build a simple business review application, working through each technology component as we implement it in the context of our application. In the last section of this chapter we review the basic requirements of the application we'll build throughout the book.

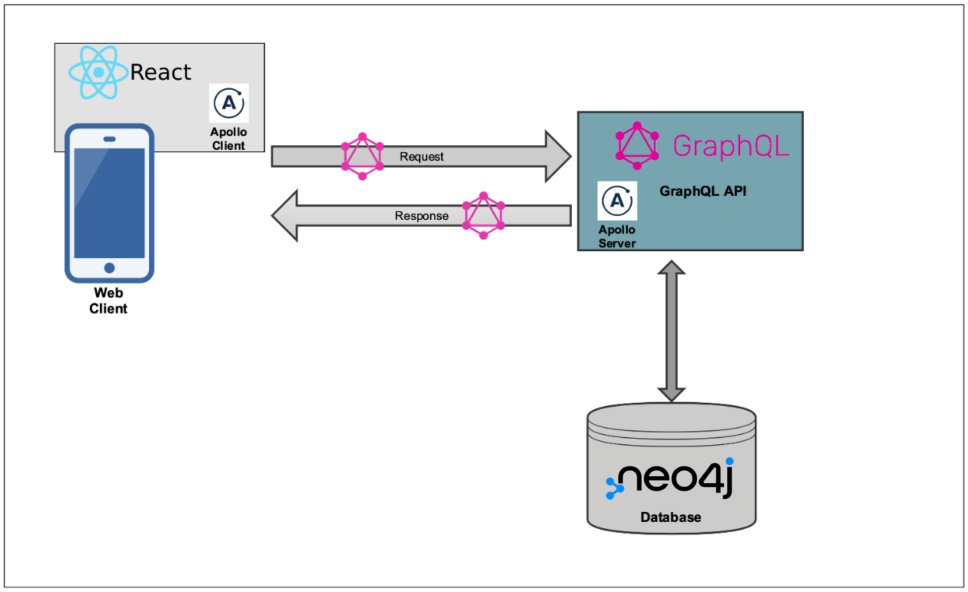


Figure 1.1 The components of GRANDstack include GraphQL, React, Apollo, and Neo4j database.

The focus of this book is learning how to build applications with GraphQL, so as we cover GraphQL it will be done in the context of building an application and using it with other technologies: how to design our schema, how to integrate with a database, how to build a web application that can query our GraphQL API, how to add authentication to our application, and so on. As a result, this book assumes basic knowledge of how web applications are typically built but doesn't necessarily require experience with each specific technology. To be successful, the reader should have a basic familiarity with JavaScript, both client-side and Node.js, and be familiar with concepts such as REST APIs and databases. You should be familiar with the `npm` command-line tool (or `yarn`) and how to use it to create Node.js projects and install dependencies. We include a brief introduction to each technology when it's introduced and suggest other resources for more in-depth coverage where needed by the reader. It's also important to note that we'll cover specific technologies to be used alongside GraphQL and that at each phase a similar technology could be substituted (for example, we could build our front-end using Vue instead of React). Ultimately, the goal of this book is to show how these technologies fit together and provide a full-stack framework for building applications with GraphQL.

1.1 GraphQL

At its heart, GraphQL is a specification for building APIs. The GraphQL specification describes an API query language and a way for fulfilling those requests. When building a GraphQL API, we describe the data available using a strict type system. These type definitions become the specification for the API, and the client is free to request the data it requires based on these type definitions, which also define the entry points for the API.

GraphQL is typically framed as an alternative to REST, which is the API paradigm you're mostly likely to be familiar with. This can be true in some cases, however GraphQL can also wrap existing REST APIs or other data sources. This is due to the benefit of GraphQL being data-layer agnostic—we can use GraphQL with any data source.

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

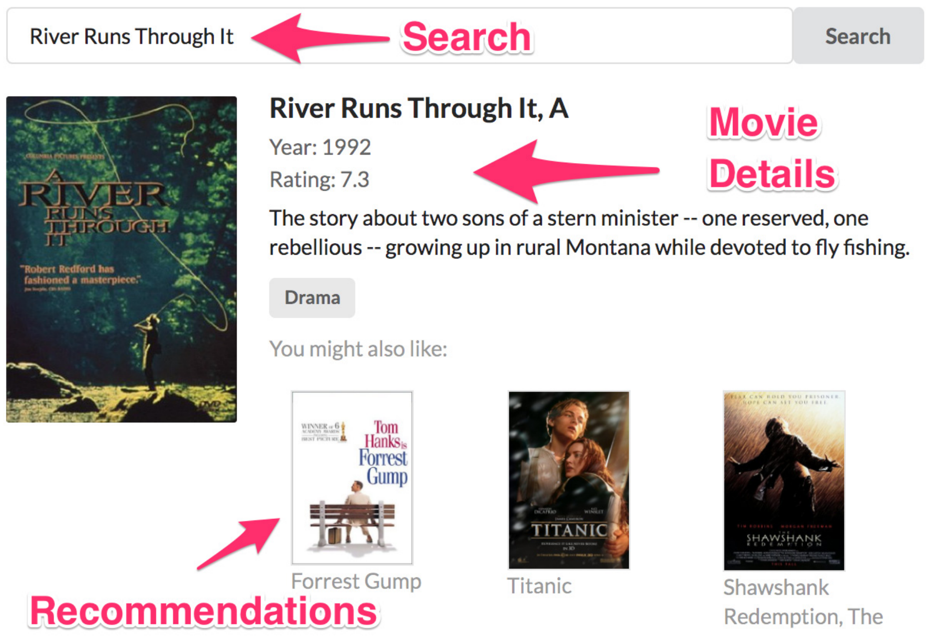
— graphql.org

Let's dive into several of the specific aspects of GraphQL.

1.1.1 GraphQL type definitions

Rather than being organized around endpoints that map to resources (as with REST), GraphQL APIs are centered around type definitions that define the data types, fields, and how they're connected in the API. These type definitions become the schema of the API, which is served from a single endpoint.

Because GraphQL services can be implemented in any language, a language-agnostic Schema Definition Language (SDL) is used to define GraphQL types. Let's look at an example, motivated by considering a simple movies application. Imagine you're hired to create a website that allows users to search a movie catalog for movie details such as title, actors, and description, as well as show recommendations for similar movies the user may find interesting (figure 1.2).



River Runs Through It, A

Year: 1992

Rating: 7.3

The story about two sons of a stern minister -- one reserved, one rebellious -- growing up in rural Montana while devoted to fly fishing.

Figure 1.2 A simple GRANDstack movies application.

Let's start by creating simple GraphQL type definitions that define the data domain of our application, as shown in the following listing.

Listing 1.1 Simple GraphQL type definitions for a movies GraphQL API

```
type Movie {
  movieId: ID!
  title: String
  actors: [Actor]
}

type Actor {
  actorId: ID!
  name: String
  movies: [Movie]
}

type Query {
  allActors: [Actor]
  allMovies: [Movie]
  movieSearch(searchString: String!): [Movie]
  moviesByTitle(title: String!): [Movie]
}
```

`Movie` is a GraphQL object type, which means a type that contains one or more fields.

`Title` is a field on the `Movie` type.

Fields can reference other types, such as a list of `Actor` objects in this case.

`actorId` is a required (or non-nullable) field on the `Actor` type, which is indicated by the `!` character.

The `Query` type is a special type in GraphQL which indicate the entry points for the API.

Fields can also have arguments, in this case the `movieSearch` field takes a required String argument, `searchString`.

Our GraphQL type definitions declare the types used in the API, their fields, and how they're connected. When defining an object type (such as `Movie`) all fields available on the object and the type of each field is also specified (we can also add fields later, using the `extend` keyword). In this case we define `title` to be a scalar `String` type—a type that resolves to a single value, as opposed to an object type. Here `actors` is a field on the `Movie` type that resolves to an array of `Actor` objects, indicating that the actors and movies are connected (the foundation of the "graph" in GraphQL).

Fields can be either optional or required. The `actorId` field on the `Actor` object type is required (or non-nullable). Every `Actor` type must have a value for `actorId`. Fields that don't include a `!` are nullable, meaning values for those fields are optional.

The fields of the `Query` type become the entry points for queries into the GraphQL service. GraphQL schemas may also contain a `Mutation` type, which defines the entry points for write operations into the API. A third special entry point related type is the `Subscription` type, which defines events to which a client can subscribe.

> **NOTE** We're skipping over many important GraphQL concepts here, such as mutation operations, interface and union types, and so on, but don't worry, we're getting started and will get to these soon enough!

At this point you may wonder where the "graph" is in "GraphQL". It turns out we've defined a graph using our GraphQL type definitions earlier in this chapter. A graph is a data structure composed of nodes (the entities or objects in our data model) and relationships that connect nodes, which is exactly the structure we've defined in our type definitions using SDL. The GraphQL type definitions above have defined a simple graph with this structure (figure 1.3).

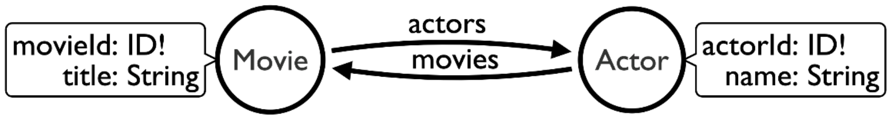


Figure 1.3 GraphQL type definitions, expressed as a graph.

Graphs are all about describing connected data and here we've defined how our movies and actors are connected in a graph.

When a GraphQL service receives a query, it's validated and executed against the GraphQL schema defined by these type definitions. Let's look at an example query that could be executed against a GraphQL service defined using the previous type definitions.

1.1.2 Querying with GraphQL

GraphQL queries define a traversal through the graph defined by our type definitions as well as requesting a subset of fields to be returned by the query; this is known as the selection set. In this query we start from the `allMovies` entry point and traverse the graph to find actors connected to each movie. Then for each of these actors, we traverse to all the other movies they're connected to. Our GraphQL query is shown in the following listing.

Listing 1.2 Simple GraphQL query

```
query FetchSomeMovies {
  allMovies {
    title
    actors {
      name
      movies {
        title
      }
    }
  }
}
```

Optional naming of operation, `query` is the default operation and can therefore be omitted. Naming the query, in this case `FetchSomeMovies` is also optional and can be omitted.

Here we specify the entry point, which is a field on either the `Query` or `Mutation` type, in this case our entry point for the query is the `allMovies` query field.

The selection set defines the fields to be returned by the query.

In the case of object fields, a nested selection set is used to specify the fields to be returned.

A further nested selection set is needed for the fields on `movies` to be returned.

Note that our query is nested and describes how to traverse the graph of related objects (in this case movies and actors), as in figure 1.4 and listing 1.3.

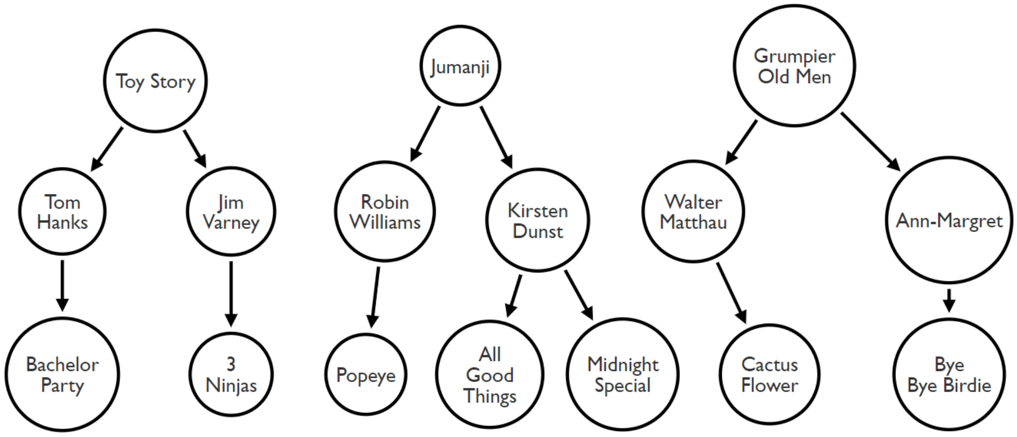


Figure 1.4 A GraphQL query traversal through the data graph.

Listing 1.3 Query results

```
"data": {
  "allMovies": [
    {
      "title": "Toy Story",
      "actors": [
        {
          "name": "Tom Hanks",
          "movies": [
            {
              "title": "Bachelor Party"
            }
          ]
        },
        {
          "name": " Jim Varney",
          "movies": [
            {
              "title": "3 Ninjas: High Noon On Mega Mountain"
            }
          ]
        }
      ]
    },
    {
      "title": "Jumanji",
      "actors": [
        {
          "name": "Robin Williams",
          "movies": [
            {
              "title": "Popeye"
            }
          ]
        },
        {
          "name": "Kirsten Dunst",
            "movies": [
              {
                "title": "Midnight Special"
              },
              {
                "title": "All Good Things"
              }
            ]
        }
      ]
    },
    {
      "title": "Grumpier Old Men",
      "actors": [
        {
          "name": "Walter Matthau",
          "movies": [
            {
              "title": "Cactus Flower"
            }
          ]
        },
        {
          "name": " Ann-Margret",
          "movies": [
            {
              "title": "Bye Bye Birdie"
            }
          ]
        }
      ]
    }
  ]
}
```

As you can see from the results, the response matches the form of the query—exactly the fields requested in the query are returned.

But where does the data come from? The data fetching logic for GraphQL APIs is defined in functions called resolvers, which contain the logic for resolving the data for an arbitrary GraphQL request from the data layer. GraphQL is data-layer agnostic so the resolvers could query one or more databases or even fetch data from another API, even a REST API. We cover resolvers in depth in the next chapter.

1.1.3 Advantages of GraphQL

Now that we've seen our first GraphQL query you may wonder, "OK that's nice, but I can fetch data about movies with REST, too. What's so great about GraphQL?". Let's review some of the benefits of GraphQL

Overfetching and underfetching

Overfetching refers to a pattern commonly associated with REST where unnecessary and unused data is sent over the network in response to an API request. Because REST is modeling resources, when we make a `GET` request for say `/movie/tt0105265` we get back the representation of that movie, nothing more nothing less, as shown in the following listing.

Listing 1.4 REST API response for GET /movie/tt0105265

```
{
  "title": "A River Runs Through It",
  "year": 1992,
  "rated": "PG",
  "runtime": "123 min",
  "plot": "The story about two sons of a stern minister -- one reserved, one
     rebellious -- growing up in rural Montana while devoted to fly
     fishing.",
  "movieId": "tt0105265",
  "actors": ["nm0001729", "nm0000093", "nm0000643", "nm0000950"],
  "language": "English",
  "country": "USA",
  "production": "Sony Pictures Home Entertainment",
  "directors": ["nm0000602"],
  "writers": ["nm0533805", "nm0295030"],
  "genre": "Drama",
  "averageReviews": 7.3
}
```

But what if the view of our application only needs to render the title and year of the movie? Then we've unnecessarily sent too much data over the wire. Further, several of those movie fields may be expensive to compute. For example, if we need to calculate `averageReviews` by aggregating across all movie reviews for each request, but we're not even showing that in the application view, that's wasted compute time and unnecessarily impacts the performance of our API (in the real world we may cache this, but this is an example).

Similarly, underfetching is a pattern associated with REST where not enough data is returned by the request.

Let's say our application view needs to render the name of each actor in a movie. First, we make a GET request for `/movie/tt0105265`. As seen previously, we have an array of IDs for the actors connected to this movie. Now to get the data required for our application we need to iterate over this array of actor IDs to get the name of each actor to render in our view:

```
/actor/nm0001729
/actor/nm0000093
/actor/nm0000643
/actor/nm0000950
```

With GraphQL we can accomplish this in a single request, solving both the overfetching and underfetching problems. This results in improved performance on the server