Version: v20.1.0

GraphQL and Relay

This section is an overview to situate Relay in relation to GraphQL, React, and the other parts of the stack. Don't worry about understanding every detail, just try to get the gist and then proceed to the next section to start working with code. Much more specifics will be explained as we go through working examples throughout the tutorial.

GraphQL is a language for querying and modifying data on servers. The unique thing about GraphQL is that rather than having a fixed set of API endpoints, your server provides a palette of options that the client can use to request any combination of data that it may need. This allows front-end developers to move more quickly because there is no need to write and deploy new endpoints as data requirements change. It also means that when a new version of the client is released, it can request just the data it needs, without extra fields leftover for compatibility with older versions.

GraphQL provides a unified interface for querying data across any kind of back-end. Whether your data is in a relational SQL database, a graph-oriented database, or an armada of microservices, a GraphQL server can collect the data from multiple back-ends and send it to the client in a single response, which is more efficient than issuing separate queries to each service from the client.

In a traditional HTTP API, there are URLs that each respond with a fixed set of information:

```
Request:
GET /person?id=24601

Response:
{"id": "24601", "name": "Jean Valjean", "age": 64, "occupation":
"Mayor"}
```

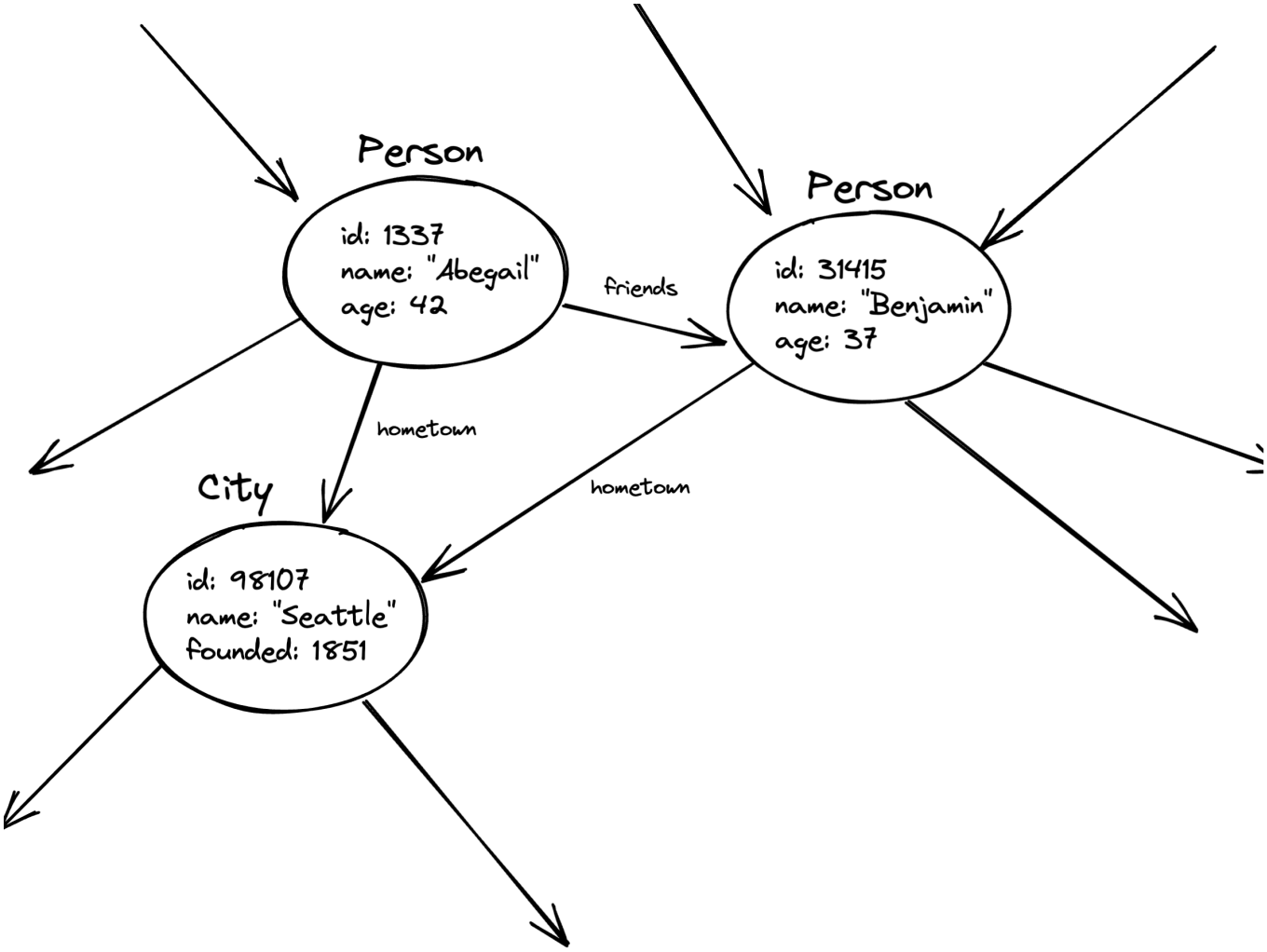
In GraphQL, the client asks for the specific information that it wants, and the server responds with just the information that was requested:

```
Request:
query {
  person(id: "24601") {
    name
    occupation
  }
}

Response:
{
  "person": {
    "name": "Jean Valjean",
    "occupation": "Mayor"
  }
}
```

Notice that only the specific fields that the client requested were included in the response.

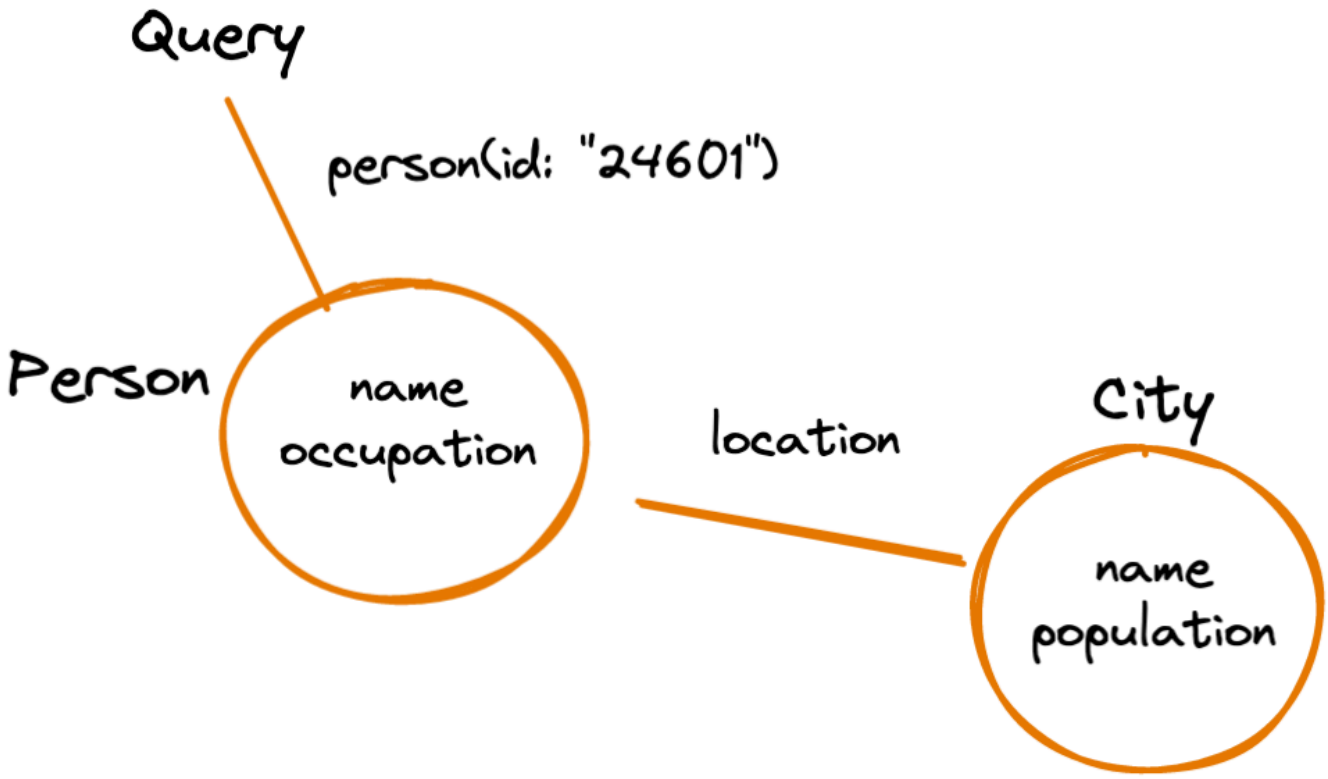
As the name suggests, GraphQL organizes data into a *graph*. The graph consists of *nodes* (like objects or records) and *edges* (pointers from one node to another):



GraphQL lets you follow those edges from one node to another and ask for information about each node that you visit. For example, here we go from a person to their city and get information about the city:

Request Response

```
query {
  person(id: "24601") {
    name
    occupation
    location {
      name
      population
    }
  }
}
```



This means we can retrieve information about a whole panoply of objects all in one query — in others words, you can efficiently get all the data for a screen in a single request instead of sending many requests one after the other. But you achieve this without writing and maintaining a separate endpoint for each screen in your UI.

Instead, your GraphQL server provides a *schema*, which describes what kinds of nodes there are, how they're connected, and what information each node contains. Then, you pick and choose from this schema to select the information you want.

The example app in this tutorial is a newsfeed app, so its schema consists of types such as

- `Story`, which represents a newsfeed story — it has fields such as its title, an image, and an *edge* to the person or organization who posted it
- `Person`, with information such as their name, email, and a list of friends (which are edges to other Persons).
- `Viewer`, which represents the person viewing the app and has information like their list of newsfeed stories
- `Image`, which has a URL for the image itself as well as an `alt` text description.

The GraphQL language includes a type system and language for specifying the schema. Here's a snippet from the schema definition for our example app — don't worry about every detail, it's just to give you a general idea:

```
// A newsfeed story. It has fields, some of which are scalars
(e.g. strings
// and numbers) and some that are edges that point to other
nodes in the graph,
// such as the 'thumbnail' and 'poster' fields:
type Story {
  id: ID!
  category: Category
  title: String
  summary: String
  thumbnail: Image
  poster: Actor
}

// An Actor is an entity that can do something on the site. This
is an
// interface that multiple different types can implement, in
this case
// Person and Organization:
interface Actor {
  id: ID!
  name: String
  profilePicture: Image
}

// This is a specific type that implements that interface:
type Person implements Actor {
  id: ID!
  name: String
  email: String
  profilePicture: Image
  location: Location
}

// The schema also lets you define enums, such as the category
// of a newsfeed story:
enum Category {
  EDUCATION
  NEWS
  COOKING
}
```

Besides queries, GraphQL also lets you send *mutations* that ask the server to update its data. If queries are analogous to HTTP GET requests, then mutations are the equivalent of POST requests. Like POSTs, they let the server respond with updated data. GraphQL also has *subscriptions* which allow for an open connection for realtime updates.

(GraphQL is usually implemented over HTTP, so queries and mutations are not only *analogous* to GET and POST, but may be sent as such as well.)

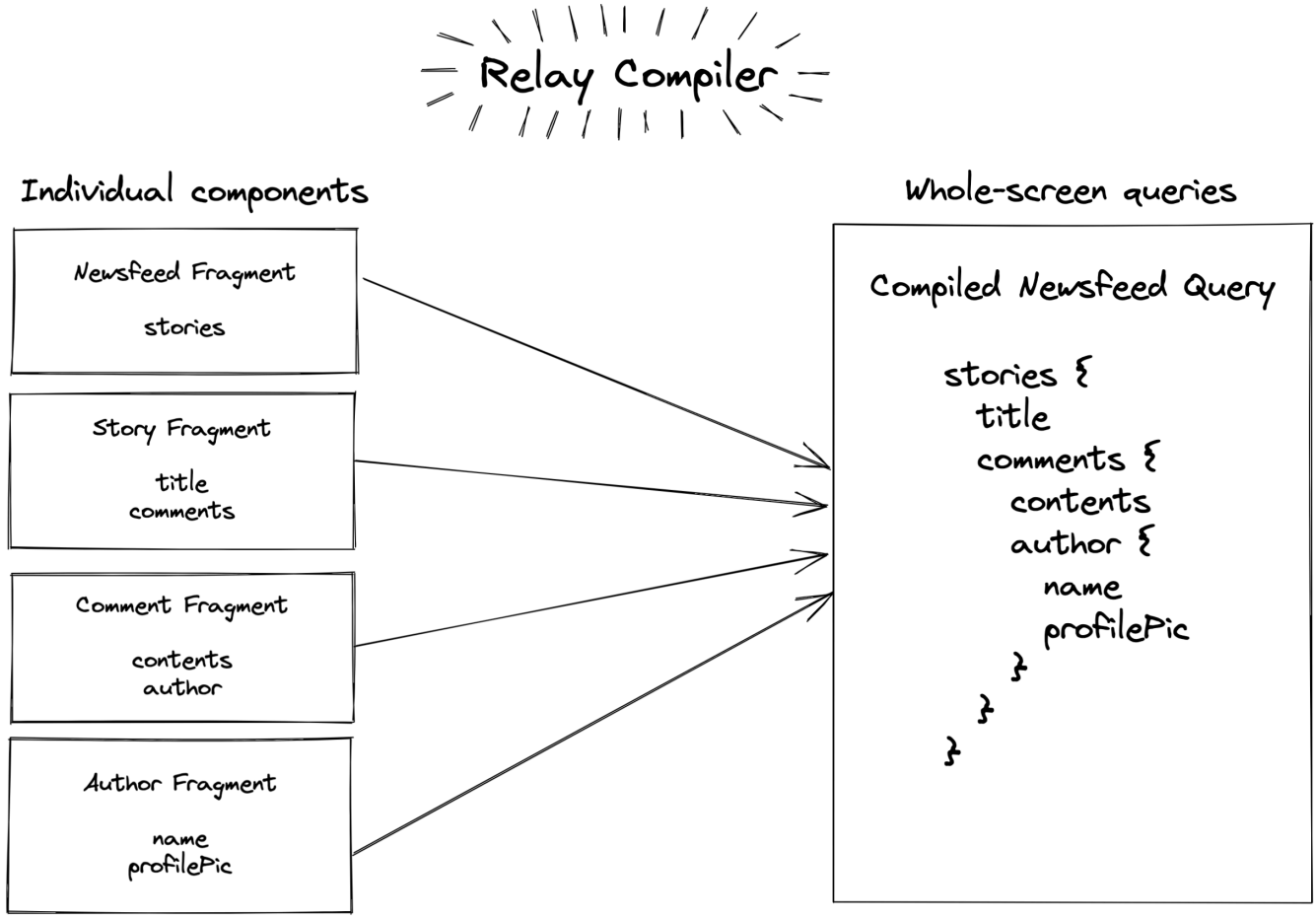
Now that we've talked about GraphQL, let's talk about Relay. It has a few different parts and pieces that we'll briefly go over before diving into the code.

Relay is a data management library for the client that's oriented around GraphQL, but uses it in a very specific way that gets the most benefit from it.

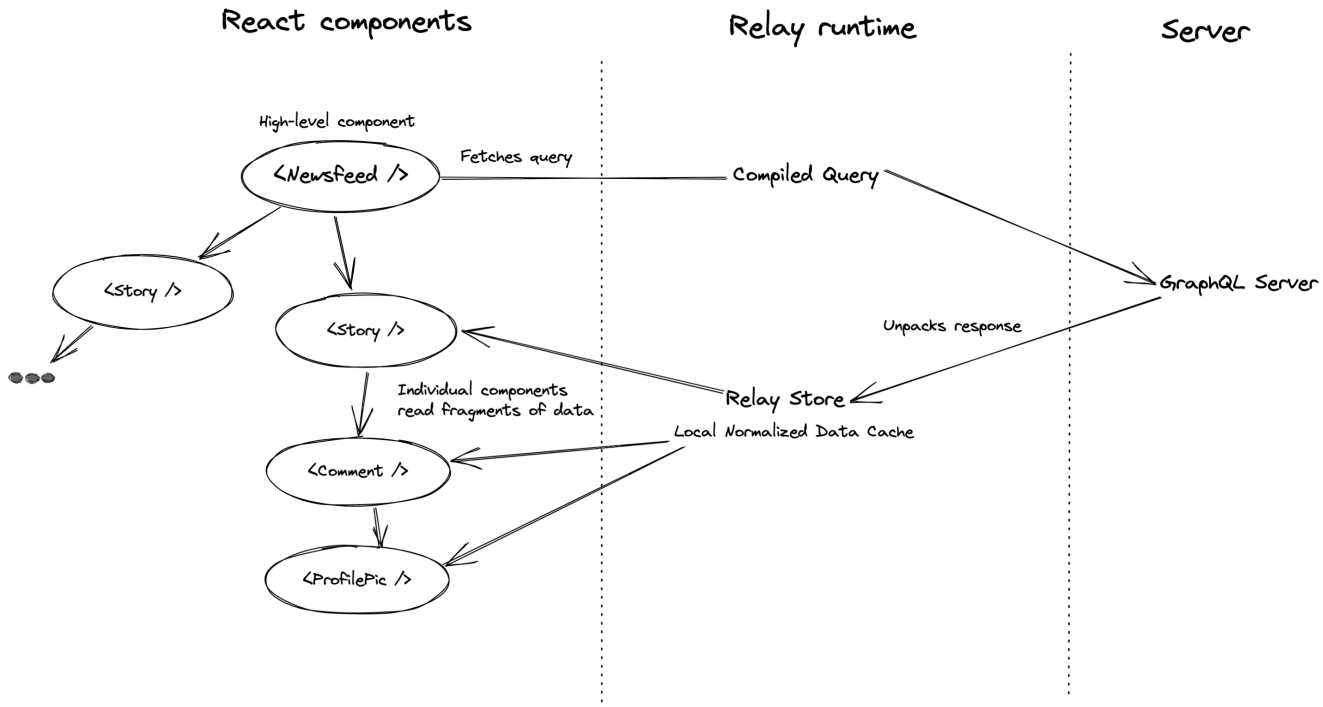
For the best performance, you want your app to issue a single request at the beginning of each screen or page instead of having individual components issue their own requests. But the problem with that is that it couples components and screens together, creating a big maintenance problem: If you need some additional data in a specific component, you have to find every screen where that component is used and add the new field to that screen's query. On the other hand, if you remove the need for a particular field, you have to remove that field from every query again — but this time, are you sure the field isn't still in use by some *other* component? It becomes very difficult to maintain these big screen-wide queries.

Relay's unique strength is to avoid this tradeoff by letting each component declare its own data requirements locally, but then stitching those requirements together into larger queries. That way you get both performance and maintainability.

Relay does this with a *compiler* that scans your JavaScript code for fragments of GraphQL, and then stitches those fragments together into complete queries.



Besides the compiler, Relay has runtime code that manages the fetching and processing of GraphQL. It maintains a local cache of all the data that has been retrieved (called the *Store*), and vends out to each component the data that belongs to it:



The advantage of having a centralized Store is that it lets you keep your data consistent when it's updated. For instance, if your UI has a way for somebody to edit their name, then you can make that update in a single place and every component that displays that person's name will see the new information, even if they're on different screens and therefore used different queries to initially retrieve the data. This is because Relay *normalizes* the data as it comes in, meaning that it merges all the data it sees for a single graph node into one place, so it doesn't have multiple copies of the same node.

Indeed, Relay doesn't just query data, it provides functions to manage the entire lifecycle of querying and updating, including support for optimistic updates and rollbacks. You can paginate, refresh data — all of the basic operations you'll need to create a UI. Whenever data in the Store is updated, Relay efficiently re-renders just those components that are displaying that particular data.

Summary

GraphQL is a language for modeling data as a graph and querying that data from a server (as well as updating the data). Relay is a React-based client library for GraphQL that lets you build up queries from individual fragments that are co-located with each React component. Once the data has been queried, Relay maintains consistency and re-renders components as the data is updated.


Edit this page