# Hands-on Lab 9: Relay

**Estimated time: 60 minutes**

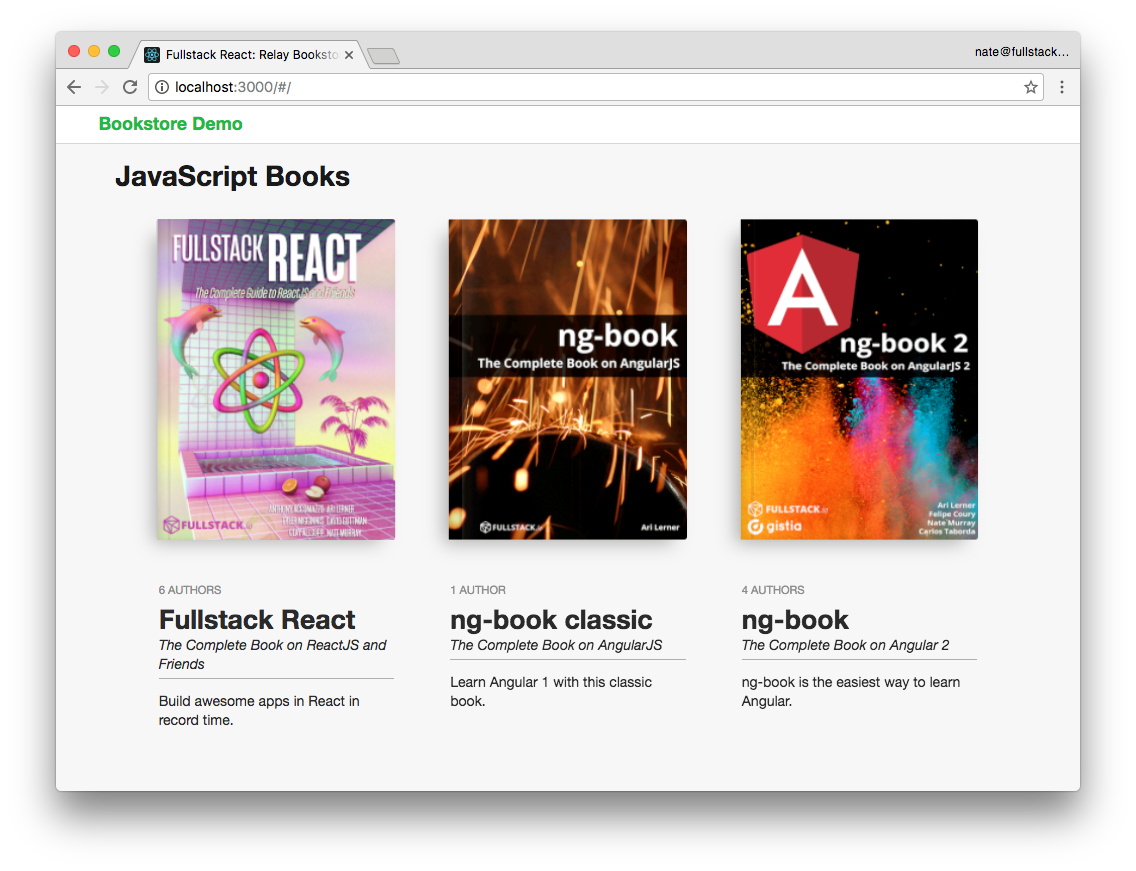
## Exercise 1:

### What We’re Building

###### The Client

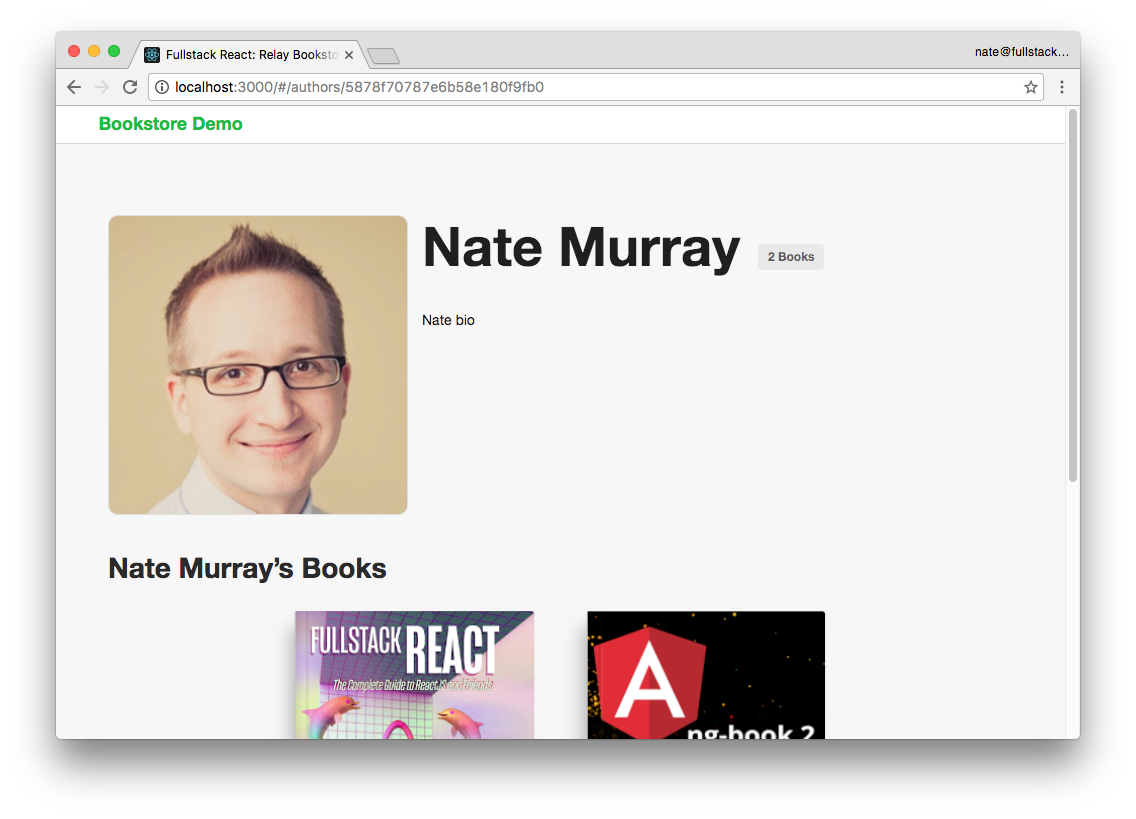
In this chapter, we’re going to build a simple bookstore. We’ll have three pages:

* A book listing page, which shows all of the books we have for sale.



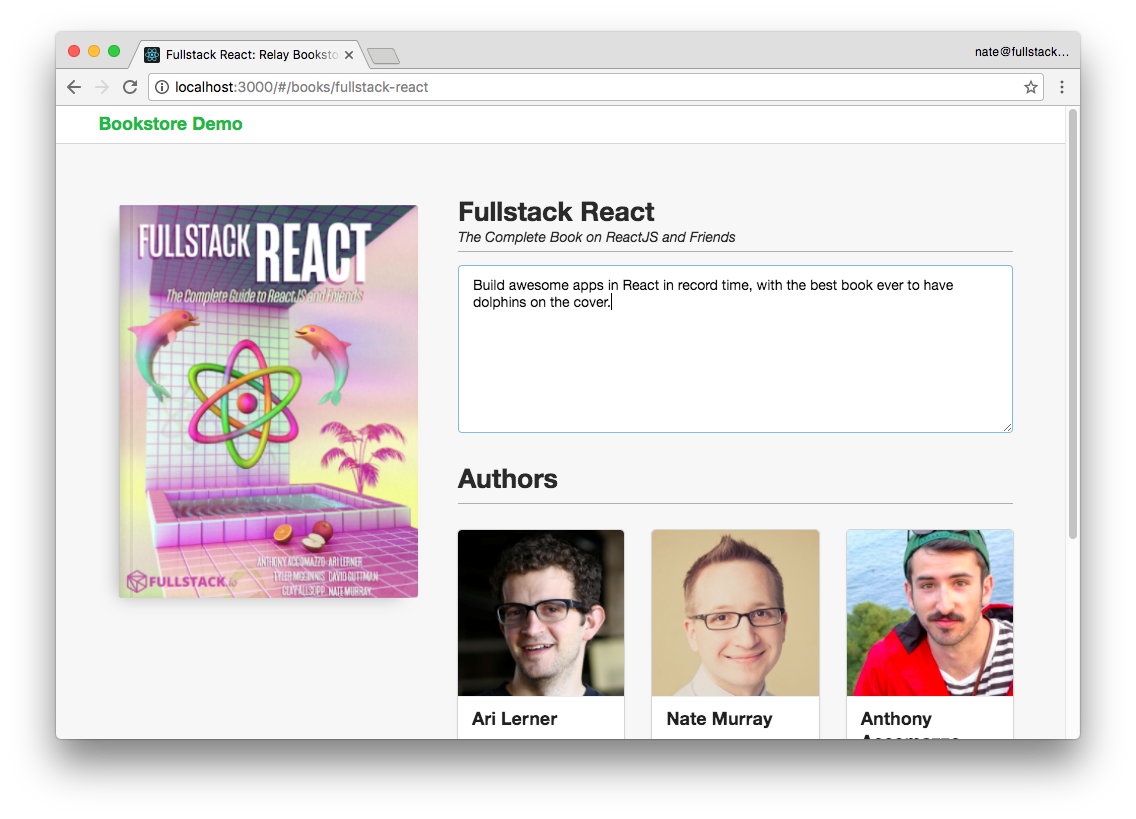
Books Store Page

* An author bio page, which shows an author’s information and the books they’ve authored.



Author Page

* A book listing page, which shows a single book and the authors for that book. We’ll also edit a book on this page.



Book Listing Page

The data model here is straightforward: a book has (and belongs to) many authors.

This data is provided to our app via an API server.

###### The Server

We’ve provided a Relay-compatible GraphQL demo server in the sample code with this app. In this chapter we aren’t going to discuss the implementation details of the server. If you’re interested in building GraphQL servers, checkout the chapter “GraphQL Server.”

###### Try The App

You can find the project for this chapter inside the relay folder of the code download:

$ cd relay/bookstore

We’ve included the completed server and completed client. Before we can run them, we need to run npm install on both the client and the server:

$ npm install

$ cd client

$ npm install $ cd ..

Next, you can run them individually in two separate tabs with:

*# tab one*

$ npm run server *# tab two*

$ npm run client

Or we’ve provided a convenience command that will run them both with:

$ npm start

When they’re running, you should be able to access the GraphQL server at http://localhost:3001 and the client app at http://localhost:3000.

In this chapter, we’re going to be spending time in our browser looking at both the client and the server. We’ve installed the GraphQL GUI tool “GraphiQL” on this demo server.

Relay is based on GraphQL, and so to get a better understanding of how it works, we’re going to be running a few queries “by hand” in this GraphiQL interface.

### Guide to the Code Structure

In this chapter, we’ve provided the completed version of the app in relay/bookstore.

In order to break up the concepts in to more digestible bites, we’ve broken up some of the components into steps. We’ve included these intermediate files in relay/bookstore/client/steps.

Our Relay app contains a server, a client, and several build tools – these all add up to quite a lot of files and directories. Here’s a brief overview of some of the directories and files you’ll find in our project. (Don’t worry if some of this is unfamiliar, we’ll explain everything you need to know in this chapter):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -- bookstore  |-- README.md  |-- client | |  | | |
| | |-- config | | // client configuration | | |
| | | |-- babelRelayPlugin.js | | // our custom babel plugin | | |
| | | |-- webpack.config.dev.js  | | `-- webpack.config.prod.js  | |-- package.json | | // webpack configuration | | |
| | |-- public  | | |-- images/  | | `-- index.html | | // images and index.html | | |
| | |-- scripts  | | |-- build.js  | | |-- start.js  | | `-- test.js | | // helper scripts | | |
| | | `-- src | |  |
| | | |-- components | | // our components are here |
| | | | |-- App.js | |  |
| | | | |-- AuthorPage.js | |  |
| | | | |-- BookItem.js | |  |
| | | | |-- BookPage.js | |  |
| | | | |-- BooksPage.js | |  |
| | | | |-- FancyBook.js | |  |
| | | | `-- TopBar.js | |  |
| | | |-- data | | // graphql metadata |
| | | | |-- schema.graphql | |  |
| | | | `-- schema.json | |  |
| | | |-- index.js | |  |
| | | |-- mutations | | // mutations make changes |
| | | | `-- UpdateBookMutation.js | |  |
| | | |-- routes.js | | // our routes |
| | | |-- steps/ | | // intermediate files |
| | | `-- styles | | // css styles here |

|-- models.js // server models

|-- package.json

|-- schema.js // graphql schema on the server

|-- server.js // our server definition

|-- start-client.js

|-- start-server.js `-- tools

`-- update-schema.js // a helper **for** generating the schema

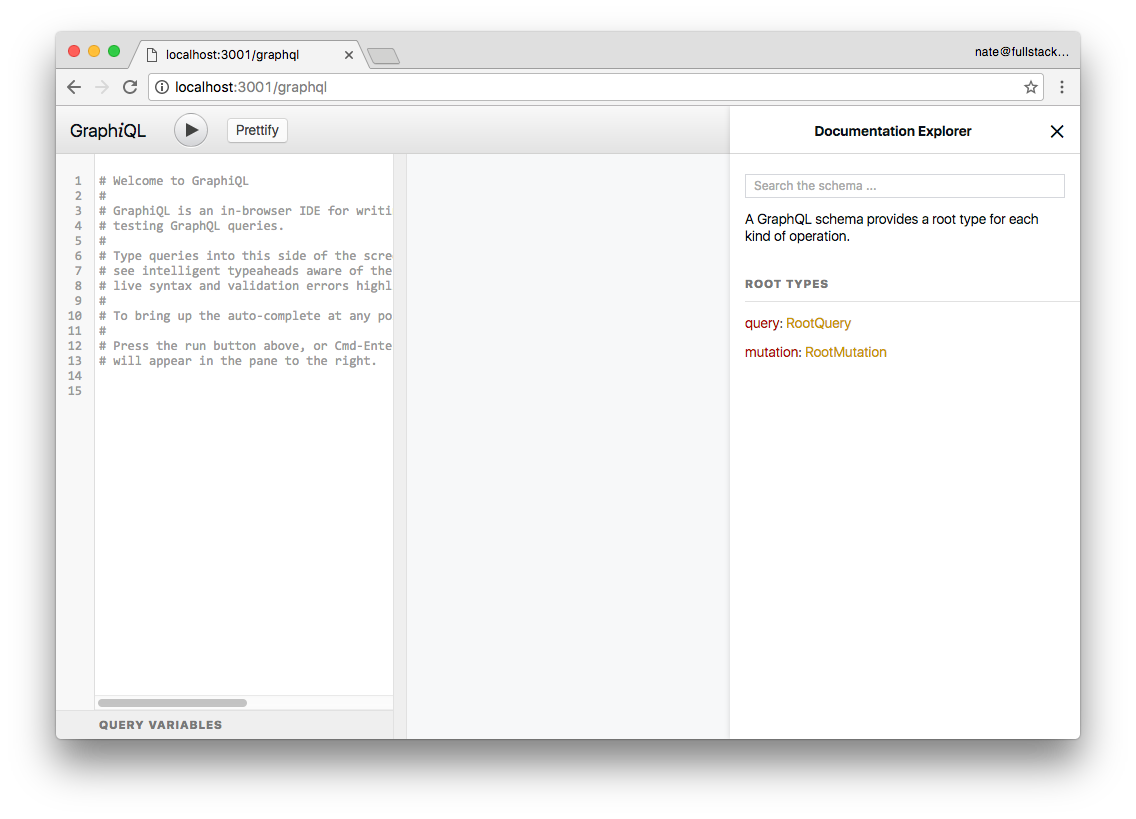
Feel free to poke around at the sample code we’ve provided, but don’t worry about understanding every file just yet. We’ll cover all of the important parts.

### Exploring Relay Conventions in GraphQL

Make sure you have the GraphQL server running, as described above, and open it in your browser at the address localhost:3001.

Remember that GraphiQL reads our schema and provides a documentation explorer to navigate the types.

Click on the “Docs” link in GraphiQL to show the “Root Types” of our schema.



GraphiQL Interface with Docs

### Fetching Objects By ID

In our server we have two models: Author and Book. The first thing we’re going to do is look-up a specific object by ID.

Imagine that we want to create a page that shows the information about a particular author. We might have a URL such as /authors/abc123, where abc123 is the ID of the author. We’d want Relay to ask our server “what is the information for the author abc123?” In that case, we’d use a GraphQL query like we’re going to define below.

However, we have a bit of a chicken and egg problem at the moment: we don’t know the IDs of any individual record.

So let’s load the entire list of authors’ names and ids and then take note of one of the IDs.

Enter the following query into GraphiQL:

query { authors {

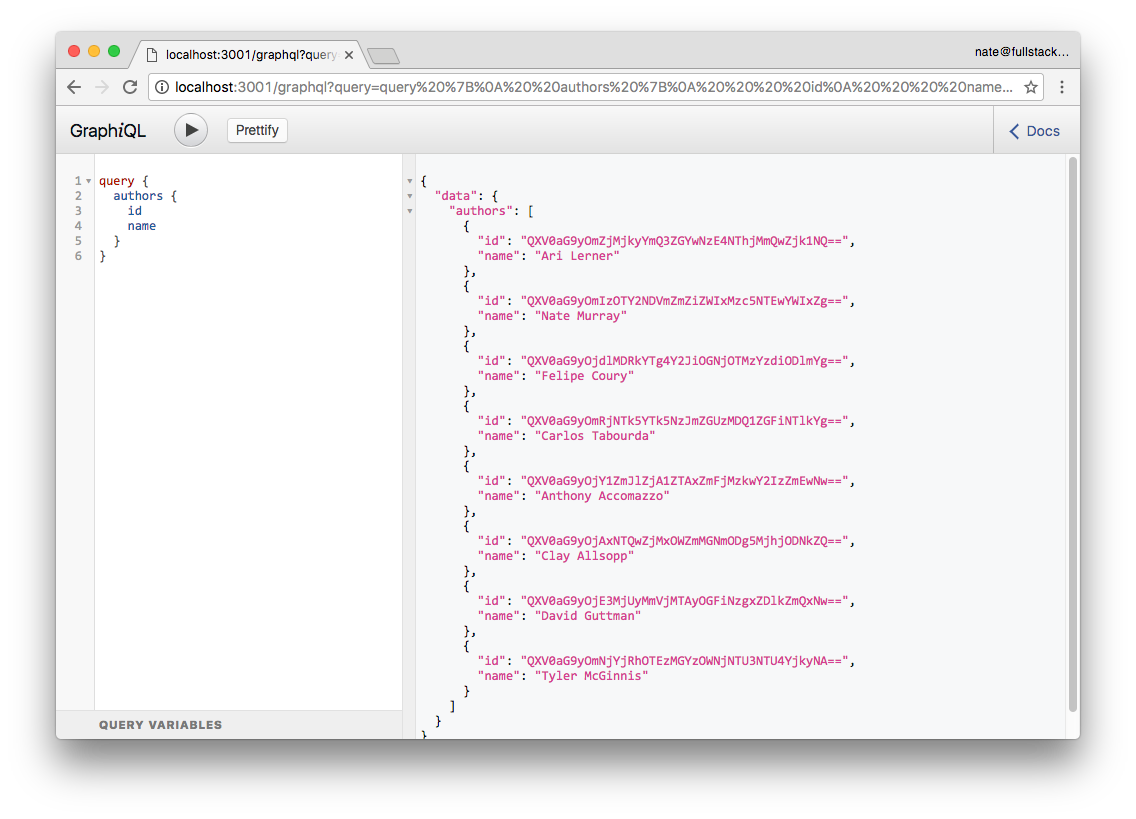
id

name

}

}

And click the play button:



GraphiQL Authors with IDs

Now that we have an author ID, we can query author to get the author with a specific id:

Query:

query { author(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {

id

name

}

}

Response:

{

"data": {

"author": {

"id": "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==",

"name": "Anthony Accomazzo"

}

}

}

While this is handy, from the author query we can only receive an object of type Author, so it doesn’t fulfill the requirement of the Relay specification. The Relay specification says that we need to have a way to query any object (Node) via the node query.

We’ve implemented the ability to lookup a Node via node on our server, so let’s try it out. First, let’s query just the id from node: Query:

query { node(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {

id

}

}

Response:

{

"data": {

"node": {

"id": "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==" }

}

}

This works! But it isn’t very useful because we don’t have any other data about the author. Let’s try fetching the name:

Query:

query { node(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {

id

name

}

}

This fails with the error:

Cannot query field "name" on type "Node". Did you mean to use an inline fragment\

on "Author" or "Book"?

What happened here? Because Node is a generic type, we can’t query the name field.

Instead we need to provide a fragment which says, if we’re querying an Author, then return Author specific fields.

So we can adjust our query like so:

Query:

query { node(id: "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==") {

id

... on Author { name

}

}

}

Response

{

"data": {

"node": {

"id": "QXV0aG9yOjY1ZmJlZjA1ZTAxZmFjMzkwY2IzZmEwNw==",

"name": "Anthony Accomazzo"

}

}

}

It works! We’re able to fetch an author using their ID. The key idea here is that we can query any object in our system by using the node query by ID (which is a Relay requirement). For instance, if we had a Book ID, we could also look up a book using the node query as well.

Relay uses the node interface particularly for re-fetching objects. When writing our apps, we can have dozens of ways of loading various objects. The idea behind the node interface is to give a consistent, easy way for Relay to ask the server “what’s the most current value for this object, given this ID?”

Now that we can query individual Nodes, we need to talk about how we traverse relationships between them. In our example app we’re going to show a list of books on the homepage, and we want to be able to load the authors who wrote that book.

In Relay we will indicate a relationship between an Author and Book by using a connection.

### Walking Connections

An Author may have contributed to several Books.

If you’re familiar with traditional relational databases, maybe you’ve seen this relationship modeled with:

* An authors table which has an id
* A books table which has an id
* and an authorships table which holds both an author\_id and a book\_id

The idea in this scenario would be that for every Author/Book pair, you’d create this new “join model” called an Authorship which represents an Author who contributed to a particular Book. This idea is sometimes referred to as “has-and-belongs-to-many”.

Analogously, Relay also defines a “join model” that should be used to denote a relationship between two models. To be precise, Relay specifies two:

1. The “connection” model which specifies a relationship between two models and keeps pagination data and
2. The “edge” model which wraps a cursor as well as the node- (model-) specific data.

This might seem a bit of overkill the first time you come to it, but recognize that this is a powerful and flexible model that provides consistent pagination across your models.

Let’s try a few queries where we traverse these connections.

Here’s a query that will get an author along with all of their book names:

query { author(id: "QXV0aG9yOmZjMjkyYmQ3ZGYwNzE4NThjMmQwZjk1NQ==") {

id

name

books {

count

edges {

node {

id

name

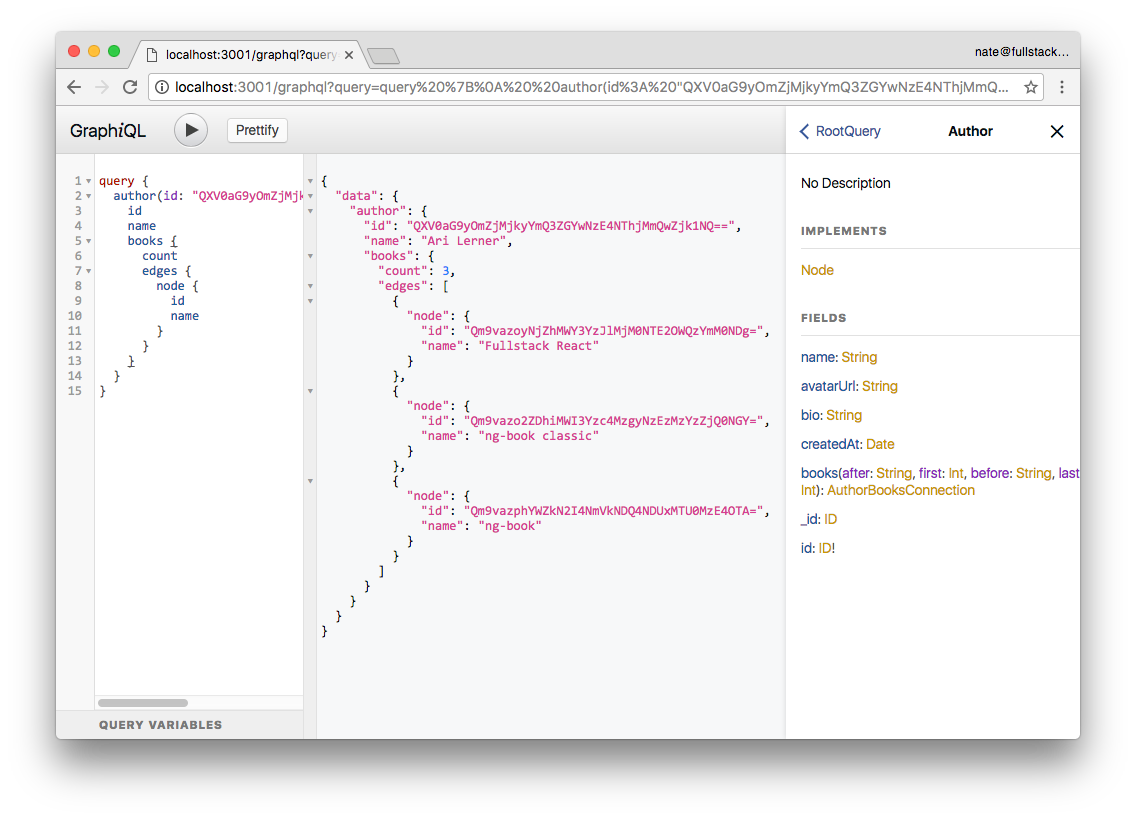
}

}

}

}

}



GraphiQL Author with Books

Click through the documentation for Author in the GraphiQL interface. Notice that on the Author type books doesn’t return an array of Book, but rather it:

1. accepts arguments, such as first or last and
2. Returns an AuthorBooksConnection



GraphiQL Author Type

Relay was designed to handle relationships that have a huge number of items. In the case where we have too many Books to load at once, we could use these arguments to limit the number of results that returned. We use the first and last arguments to denote the number of items we want to retrieve.

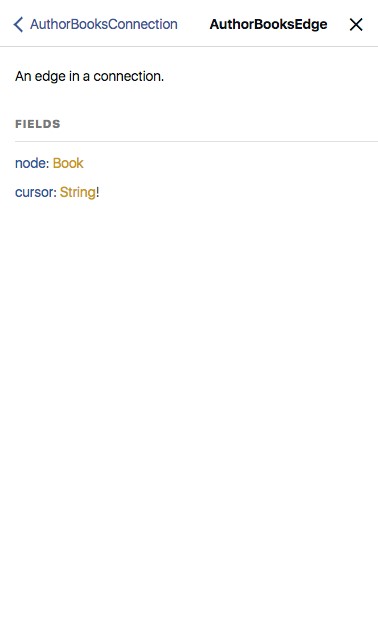
The AuthorBooksConnection has three fields:

* pageInfo
* edges and
* count

count gives us the total number of edges we have (the number of Books this person has authored, in this case).

pageInfo gives us information such as if there is a previous or next page and what the start and end cursor for this page are. We could use these cursors later on to ask for the next (or previous) page.

edges contains the list of AuthorBooksEdge. Look at the AuthorBooksEdge type in the GraphiQL interface.



GraphiQL **AuthorBooksEdge** Type The AuthorBooksEdge has two fields:

* node and
* cursor

The cursor is a string we can use for pagination, from this record. Here we can see that the node is of type Book. Within that node is the hard data that we’re looking for.

While we are using these connections for many-to-many relationships, you’d navigate one-to-many relationships the same way. By using this standard for traversing relationships between models it’s easy to know how to get access to related models, regardless of the type of relationship. Everything is nodes and edges in a graph.

Additionally, by making pagination part of the standard, we’re making accommodations for realworld data sets where we’re not going to load every piece of data in a relationship (or table) at the same time.

Having standardized pagination as a part of Relay make it that much easier to standardize pagination in our React apps.

### Changing Data with Mutations

Mutations are how we change data in Relay/GraphQL. To change data with mutations we need to:

1. Locate the mutation we want to call
2. Specify the input arguments and
3. Specify what data we want to be returned after the mutation is finished.

Say, for instance, that we want to change an author’s bio. We could perform the following:

Query:

mutation {

updateAuthor(input: {

id: "QXV0aG9yOmZjMjkyYmQ3ZGYwNzE4NThjMmQwZjk1NQ==", bio: "all around great guy"

}) {

changedAuthor {

id

name

bio

}

}

}

Now we can see in the response that the bio has been changed to "all around great guy".

Response:

{

"data": {

"updateAuthor": {

"changedAuthor": {

"id": "QXV0aG9yOmZjMjkyYmQ3ZGYwNzE4NThjMmQwZjk1NQ==",

"name": "Ari Lerner",

"bio": "all around great guy"

}

}

}

}

In this case, the mutation is updateAuthor. As you may recall from the GraphQL chapter, we change data in GraphQL by creating a mutation query. Our server defines a mutation query for common data operations.

For instance, if you’re familiar with the REST paradigm of Create-Read-Update-Delete (CRUD), we might define similar mutations for our models here: createAuthor, updateAuthor, deleteAuthor.

While mutations are used generally in GraphQL, Relay, unsurprisingly, specifies some constraints around how it expects mutations to be defined.

Mutations on the client-side can be a bit daunting. We’ll talk more about them later in the chapter.

### Relay GraphQL Queries Summary

Above, we’ve just covered the three types of queries we’ll use when writing Relay apps:

1. Fetching individual records from our server
2. Traversing relationships between records, using connections and edges
3. Mutating data with mutation queries.

Now that we’ve explored our data and reviewed the conventions on our server, let’s take a look at how Relay works inside our app.

## Adding Relay to Our App

### Quick Look at the Goal

Now let’s switching gears into writing our React app.

There are quite a few steps to installing Relay into our app. Before we walk through them, let’s take a look at the goal: a basic Relay container component that loads data from our server and renders it.

Once we have one component that can load data from Relay, building our app gets much easier from there.

Below we’ll walk through how to build out a fully working Relay app, but it can also be helpful to take a look at a single-file, “hello world” example. Below is the code for a minimal, but working Relay app. Many of the specifics will be unfamilar, and the rest of the chapter is dedicated to explaining in detail how to use each idea.

For now, just skim this code to get an idea of the different parts involved in working with Relay.

**relay/bookstore/client/src/steps/index.minimal.js**

/\* eslint-disable react/prefer-stateless-function \*/

**import React from** 'react';

**import ReactDOM from** 'react-dom';

**import Relay from** 'react-relay';

**import** '../semantic-dist/semantic.css';

**import** './styles/index.css';

// Customize this based on your server's URL

const graphQLUrl = 'http://localhost:3001/graphql';

// Configure Relay **with** a "NetworkLayer"

Relay.injectNetworkLayer(new Relay.DefaultNetworkLayer(graphQLUrl));

// Create the top-level query that we'll execute

**class AppQueries** extends Relay.Route {

static routeName = 'AppQueries';

static queries = {

viewer: () => Relay.QL`

query {

viewer

}

`

};

}

// A basic component that renders the list of authors

**class App** extends React.Component {

render() {

**return** (

<div>

<h1>Authors list</h1>

<ul>

{this.props.viewer.authors.edges.map(edge => (

<li key={edge.node.id}>{edge.node.name}</li>

))}

</ul>

</div>

);

}

}

// A Relay Container that specifies the fragment to be used **in** our query above

const AppContainer = Relay.createContainer(App, {

fragments: {

viewer: () => Relay.QL`

fragment on Viewer {

authors(first: 100) {

edges {

node {

id

name

}

}

}

}

`

}

});

ReactDOM.render(

<Relay.Renderer

environment={Relay.Store}

Container={AppContainer}

queryConfig={new AppQueries()}

/>,

document.getElementById('root')

);

In the example above, our AppContainer fetches a collection of authors and renders them in a list. Notice that this code is declarative. That is, there’s no part of the code where we’re saying “make a post request to the server and interpret it as JSON” etc. Instead, our component declares the data it needs, and Relay fetches it from the server and provides it to our component.

### A Preview of the Author Page

Let’s look at another example, taken from our Bookstore app. Here’s a version of the AuthorPage component:

**relay/bookstore/client/src/steps/AuthorPage.minimal.js**

**class** AuthorPage **extends** React.Component {

render() {

**const** {author} = **this**.props;

**return** (

<div>

<img src={author.avatarUrl} />

<h1>{author.name}</h1>

<p>

{*/\* e.g. '2 Books' or '1 Book' \*/*}

{author.books.count}

{author.books.count > 1 ? ' Books' : ' Book'}

</p>

<p>{author.bio}</p>

</div>

);

}

}

**export default** Relay.createFragmentContainer(AuthorPage, {

fragments: {

author: () => Relay.QL`

fragment on Author {

name

avatarUrl

bio

books {

count

}

}`

}

});

There are two parts: 1. the Relay.createContainer statement (with a Relay.QL query) and 2. the render() function.

At a high level, what’s happening here is that our Relay.QL query specifies the data we want to load for this author. Then the author is passed in to the component via props and we render the data.

We haven’t yet talked about all of the setup that goes in to getting our app to this point. We will. For now, just notice that once we have everything setup, it becomes extremely easy to load data into our components (and it’s easy to modify our queries if we change our mind).

Notice that when we use Relay on our component, we have two things: 1. the createContainer and 2. a fragment.

### Validating Our Relay Queries at Compile Time

One of the great things about GraphQL is that we have a type-schema for our API. We can use this to our advantage when writing client side code.

When we write query fragments in our components, it looks like this:

**code/relay/bookstore/client/src/steps/AuthorPage.minimal.js**

fragments: {

author: () => Relay.QL`

fragment on Author {

name

avatarUrl

bio

books {

count

}

}`

}

But notice that we’re putting the query in a long JavaScript string. It’s easy to write, but in a naive implementation of handling these queries, typos could be the source of many bugs because there’s no way to validate that the contents are well formed. In the case where something was mis-typed, we wouldn’t even know there was a typo until one of these queries was run.

However, the good news is that there is a custom Babel plugin which will verify these queries against our schema at compile time.

For this we’ll use the [babel-relay-plugin](https://www.npmjs.com/package/babel-relay-plugin)[[1]](#footnote-1). This plugin:

* 1. reads these Relay.QL backtick strings
  2. parses the GraphQL query
  3. validates them against our schema
  4. and converts our Relay.QL queries to a expanded function calls[[2]](#footnote-2)

But in order to verify our schema, we need to have the schema available to our client-side build tools.

Remember that our schema is defined by our server. It’s our GraphQL server that defines the data models (Books and Authors, in this case) and the corresponding GraphQL schema.

So how do we make our server’s schema available to our client build-tools? We export the schema from the server as a JSON file and copy it over.

###### Building the schema.json

To provide the schema to our client build tools we’ll write a script in the server that will export the schema to a JSON file.

We’ll then configure babel-relay-plugin to use this JSON file when we compile our client code. It’s a little bit of work up-front, but the benefit is that we’ll get compile-time validation of or Relay queries in our client app. This can save our team tons of time trying to track down bugs because of invalid queries.

Here’s the script we’re using to generate our schema.json:

**relay/bookstore/tools/update-schema.js**

**import** fs from 'fs';

**import** path from 'path';

**import** { graphql } from 'graphql';

**import** { introspectionQuery, printSchema } from 'graphql/utilities';

**import** schema from '../schema';

*// Save JSON of full schema introspection for the Babel Relay Plugin to use*

**const** generateJSONSchema = async () => {

**var** result = await (graphql(schema, introspectionQuery));

**if** (result.errors) {

console.error(

'ERROR introspecting schema: ',

JSON.stringify(result.errors, **null**, 2)

);

} **else** {

fs.writeFileSync(

path.join(\_\_dirname, '../client/src/data/schema.json'),

JSON.stringify(result, **null**, 2)

);

}

*// Save user readable type system shorthand of schema*

fs.writeFileSync(

path.join(\_\_dirname, '../client/src/data/schema.graphql'),

printSchema(schema)

);

};

generateJSONSchema().then(() => {

console.log("Saved to client/src/data/schema.{json,graphql}");

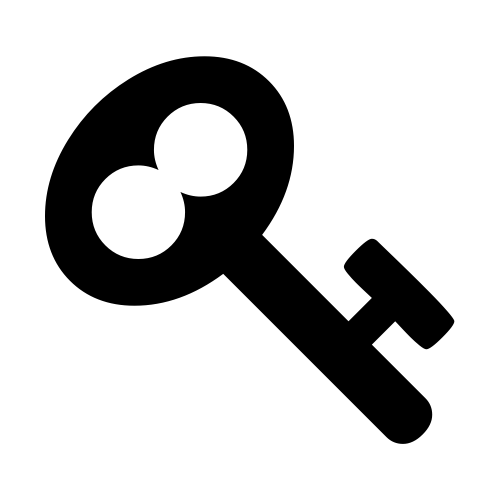
})

Take a look at the dependencies we’re loading here. Besides fs and path, we have:

* 1. graphql, introspectionQuery, printSchema from graphql and
  2. schema from ../schema.

One thing to notice is that we’re not loading anything from the relay library. This is all GraphQL. The relay library isn’t involved. Our GraphQL schema conforms to the Relay standard, but we don’t depend on any Relay-specific functionality. This process of exporting your schema can be done with any GraphQL server.

What’s happening here is that we’re telling the graphql library to take our schema and then introspectionQuery and print out our schema into two files:

The introspectionQuery is a query which asks GraphQL information about what queries it supports. You can read more about GraphQL Introspection her[e](http://graphql.org/learn/introspection/)[[3]](#footnote-3)

1. a machine-readable schema.json file (for our client) and
2. a human-readable schema.graphql file

Here’s a sample from the schema.graphql file:

**relay/bookstore/client/src/data/schema.graphql**

type Author implements Node {

name: String

avatarUrl: String

bio: String

createdAt: Date

books(after: String, first: Int, before: String, last: Int): AuthorBooksConnection

\_id: ID

# The ID of an object id: ID!

}

# A connection to a list of items.

type AuthorBooksConnection {

# Information to aid in pagination.

pageInfo: PageInfo!

# A list of edges.

edges: [AuthorBooksEdge]

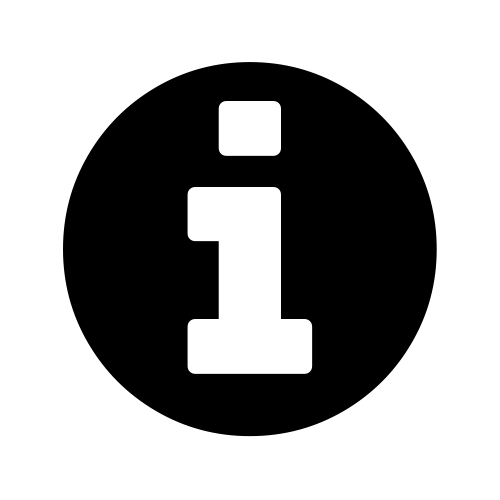
count: Float

}

If we’re in the root directory for this project (relay) then we can generate this schema by running:

1 npm run generateSchema

If we ever change our schema (like adding a field to a model or adding a new model) then we need to run this script to regenerate our schema. If we don’t regenerate the schema and try to use the new data in our client app then babel-relay-plugin will throw compiler errors because it will using the old schema. So make sure if you change your models, you regenerate your schema.

Watch out for schema caching

Some webpack configurations (such as the default that is generated when you eject from create-react-app) cache compiled scripts. If you’re using such a feature (as we are in this app) then your schema is also cached.

This means that when you update your schema, you have to clear your react-scripts cache. On my machine, this folder is kept in node\_modules/.cache/react-scripts. So whenever we regenerate the schema we run the following command as well to clear the cache: rm -rf client/node\_modules/.cache/react-scripts/

Failure to clear this cache when regenerating your schema may result in your client loading the old, cached schema which can be confusing.

**Installing babel-relay-plugin**

To use babel-relay-plugin we have to:

1. npm install babel-relay-plugin
2. Tell babel-relay-plugin about our schema.json
3. Configure babel to use our plugin

To do #2, see the file client/config/babelRelayPlugin.js like so:

**relay/bookstore/client/config/babelRelayPlugin.js**

**var** getBabelRelayPlugin = require('babel-relay-plugin');

**var** schema = require('../src/data/schema.json');

module.exports = getBabelRelayPlugin(schema.data, {

debug: **true**,

suppressWarnings: **false**,

enforceSchema: **true**

});

What we’re doing here is just configuring the babel-relay-plugin by adding in our own schema and setting a few options.

Now we need to add this script as a plugin to our babel config.

To do this, we’ve added the following to the client’s package.json:

"babel": {

"presets": [

"react-app"

],

"plugins": [

"./config/babelRelayPlugin"

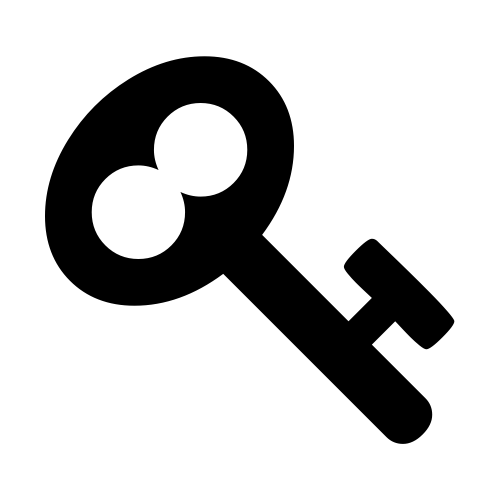
]

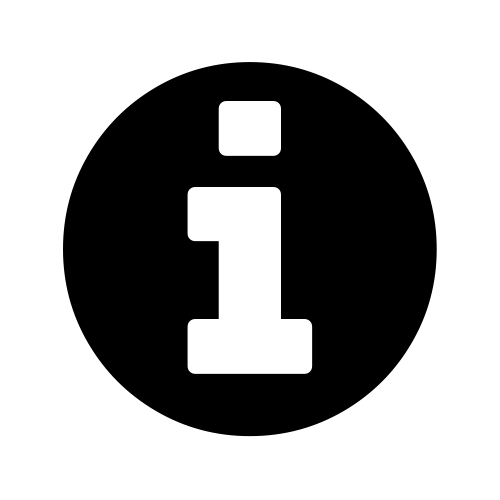
},

Above, we’ve added "./config/babelRelayPlugin" to the plugins section of our babel configuration in the package.json.

### Setting Up Routing

Now that we have our build tools in place, we can start integrating Relay with React. Because we’re building a multi-page app, we’re going to need a router. For this app we’re going to use react-router with react-router-relay.

If you’d like to see an example of a Relay app that uses Relay directly (without using react-router then checkout the [relay-starter-kit](https://github.com/relayjs/relay-starter-kit)[[4]](#footnote-4) .

react-router-relay uses react-router v2.8 (not router v4, which we covered in the chapter “Routing.”

Unfortunately, as you can see her[e](https://github.com/relay-tools/react-router-relay/issues/193)[[5]](#footnote-5) , there are no plans to directly support Relay in ReactRouter v4.

That said, it is possible to integrate Relay with any routing framework including React Router v4. Doing so is beyond the scope of this chapter.

In this chapter, we’ll provide all of the route configuration needed to run the app, but we’re not going to be discussing the React Router AP I . If you need to look it up, you can find the docs for Router her[e](https://github.com/ReactTraining/react-router)[[6]](#footnote-6) .

react-router-relay provides a convenient way to execute Relay queries whenever the route changes. react-router-relay also will read parameters from the URL and pass them as parameters to our Relay queries. We’ll take a close look at this feature in a it. To install react-router-relay into our app, we’ll do the following:

1. Configure Relay
2. Configure our Router
3. Use the react-router-relay middleware to connect Relay to our Router.

Let’s look at the code we use to do this:

**relay/bookstore/client/src/index.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** createHashHistory from 'history/lib/createHashHistory';

**import** Relay, {DefaultNetworkLayer} from 'react-relay/classic';

**import** applyRouterMiddleware from 'react-router/lib/applyRouterMiddleware';

**import** Router from 'react-router/lib/Router';

**import** useRouterHistory from 'react-router/lib/useRouterHistory';

**import** useRelay from 'react-router-relay';

**import** routes from './routes';

**import** './semantic-dist/semantic.css';

**import** './styles/index.css';

*// Customize this based on your server's URL*

**const** graphQLUrl = 'http://localhost:3001/graphql';

*// Configure Relay with a "NetworkLayer"*

Relay.injectNetworkLayer(**new** DefaultNetworkLayer(graphQLUrl));

**const** history = useRouterHistory(createHashHistory)();

ReactDOM.render(

<Router

history={history}

routes={routes}

render={applyRouterMiddleware(useRelay)}

environment={Relay.Store}

/>,

document.getElementById('root')

);

The first section imports our dependencies.

Next we configure the URL to our server in the variable graphQLUrl. If your server is at a different URL, configure it here. For instance, we’ll often use an environment-specific variable here.

Next we configure Relay with a “Network Layer”. In this case, we’re using DefaultNetworkLayer which will make HTTP requests, but you could use this to, say, mock out a Relay server for testing or use a different protocol entirely.

We’re going to use hash-based routing for this app, so we configure history to use createHashHistory.

We tie our Router root component to Relay by:

* 1. Using applyRouterMiddleware(useRelay) – which comes from react-router-relay and
  2. Setting our Relay.Store onto the Routerenvironment.

Setting up our Router this way makes Relay available to our app, but to actually perform Relay queries we have one more step: we need to configure Relay queries on our routes.

### Adding Relay to Our Routes

Let’s take a look at our routes.js:

**relay/bookstore/client/src/steps/routes.author.js**

**import** Relay from 'react-relay';

**import** React from 'react';

**import** IndexRoute from 'react-router/lib/IndexRoute';

**import** Route from 'react-router/lib/Route';

**import** App from './components/App';

**import** AuthorPage from './components/AuthorPage';

**const** AuthorQueries = {

author: () => Relay.QL`

query {

author(id: $authorId)

}`,

};

**export default** (

<Route

path='/'

component={App}

>

<Route

path='/authors/:authorId'

component={AuthorPage}

queries={AuthorQueries}

/>

</Route>

);

For these initial routes we have a parent route that uses the App component and a child route that uses the AuthorPage component. Eventually we will have a child component for each page in our app, but for now let’s look at the following in order:

* 1. Our parent App Component
  2. The AuthorQueries and how the relate to the AuthorPage then 3. Dig in to the AuthorPage component.

### App Component

Our top-level App component establishes the wrapper for the rest of the app:

**relay/bookstore/client/src/components/App.js**

**import** React, {Children, Component} from 'react';

**import** {withRouter} from 'react-router';

**import** TopBar from './TopBar';

**import** '../styles/App.css';

**class** App **extends** React.Component {

render() {

**return** (

<div className="ui grid">

<TopBar />

<div className="ui grid container">

{Children.map(**this**.props.children, c => React.cloneElement(c))}

</div>

</div>

);

}

}

**export default** withRouter(App);

Here we render the TopBar and the markup that will wrap any child components (this.props.children).

Before we exportApp, we wrap it using withRouter from react-router. withRouter [is a helper function](https://github.com/ReactTraining/react-router/blob/master/docs/API.md#withroutercomponent-options)166 that provides props.router on our component.

166<https://github.com/ReactTraining/react-router/blob/master/docs/API.md#withroutercomponent-options>

### AuthorQueries Component

Now that we understand the App component, hop back into routes.js and look at the AuthorQueries:

**relay/bookstore/client/src/steps/routes.author.js**

**const** AuthorQueries = {

author: () => Relay.QL`

query {

author(id: $authorId)

}`,

};

Remember that in Relay in order to fetch the data that we need for our components we have to execute queries. When using react-router-relay we specify that when a particular route is visited, the queries defined on that route will be executed.

Notice that the AuthorQueries has one query: author. This query also has a variable $authorId. Where does the $authorId variable come from? It comes from the route path parameter:

**relay/bookstore/client/src/steps/routes.author.js**

<Route

path='/authors/:authorId'

component={AuthorPage}

queries={AuthorQueries}

/>

Here in this route we’re saying that we’ll match the route /authors/ and whatever follows will be interpreted as the authorId. This authorId is passed as a variable to the Relay query.

Notice something else odd about the author query: it doesn’t specify any “leaf nodes” of data to fetch. The query stops at author(id: $authorId). This is because the query is leaving the decision of what particular data to fetch to the component.

It is in our component (well, our Relay Container, to be precise), that we will specify the fields that are needed to render that component.

With that in mind, let’s turn our attention to the AuthorPage component and look at the Relay query there.

### AuthorPage Component

Here’s the code that specifies what fields we need to render the minimal AuthorPage:

**relay/bookstore/client/src/steps/AuthorPage.minimal.js**

**export default** Relay.createFragmentContainer(AuthorPage, {

fragments: {

author: () => Relay.QL`

fragment on Author {

name

avatarUrl

bio

books {

count

}

}`

}

});

On the “inside” of the query it’s easy to see that we’re asking for the name, avatarUrl, and bio of the author. We’re even able to dip into the books relationship and get the count of the number of books this person has authored.

However, it may not be clear how this ties into the query above. There are two constraints we need to follow.

The first is that the key names of these fragments must match the keys names of the queries. In this case, because this component is being rendered with a query key name of author (in AuthorQueries) the fragment key name must also be author (in AuthorPagefragments).



Relay Fragment Naming

The second constraint is that this fragment will be a **fragment on** the type of the inner field of **query**. That is, we wouldn’t put fragment on Book here, because the containing query “ends” on an Author. We can see this by looking at the GraphQL schema in GraphiQL.



Relay Fragment on Type

When a route matching /authors/:authorId is visited, the AuthorPage author fragment will be pulled into the AuthorQueries and results will be fetched from the server and passed down into our component.

### Try It Out

We’ve only setup the AuthorPage so far, but it’s enough for us to try out.

To do this, ensure you have both the GraphQL server started, as well as the client, as described above.

Then visit the GraphiQL server at http://localhost:3001/graphql and try the following query:

query { authors {

id

name

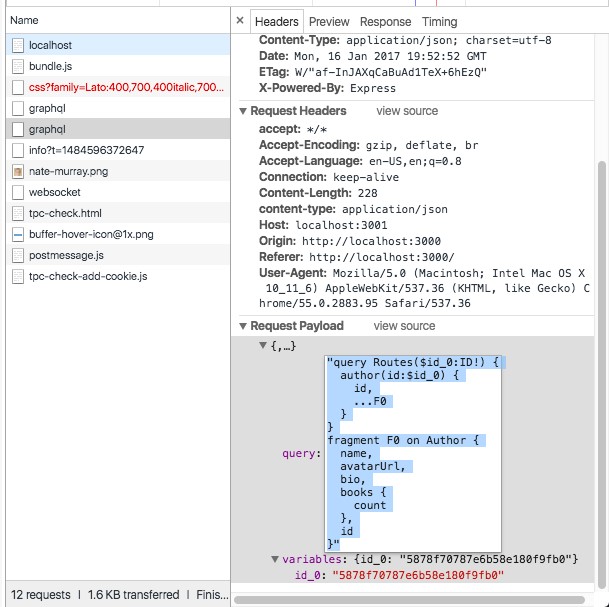
}

}

Copy an author ID and then visit the client app with that ID. Something like:

<http://localhost:3000/#/authors/QXV0aG9yOmIzOTY2NDVmZmZiZWIxMzc5NTEwYWIxZg>==

In our browser, if we look at our network pane we can see the GraphQL query that was called:



Relay GraphQL Call in Network Pane

In this case, the query sent over the wire looks something like:

query Routes($id\_0:ID!) {

author(id:$id\_0) {

id,

...F0

}

}

fragment F0 on Author {

name,

avatarUrl,

bio,

books {

count

},

id

}

The first part of the query (query Routes...) comes from AuthorQueries and the fragment F0 comes from our author fragment on AuthorPage. Again, note that it wouldn’t make any sense for this fragment F0 to be on any type other than *Author* because the child of query > author is of type Author. Trying to put a Book- or any other-typed fragment here would be invalid.

### AuthorPage with Styles

The minimal AuthorPage we’ve rendered so far doesn’t look so great. Let’s quickly add a bit of markup so that the page looks better.

Like many of the examples in this book, we’re using [Semantic UI](http://semantic-ui.com/)[[7]](#footnote-7) for the CSS framework. When you see CSS class names like sixteen wide column or ui grid centered these are coming from Semantic UI.

Keeping the same Relay query, we’re going to change the AuthorPage markup to the following:

**relay/bookstore/client/src/steps/AuthorPage.styled.js**

**class** AuthorPage **extends** React.Component {

render() {

**const** { author } = **this**.props;

**return** (

<div className='authorPage bookPage sixteen wide column'>

<div className='spacer row' />

<div className='ui divided items'>

<div className='item'>

<div className='ui'>

<img src={author.avatarUrl}

alt={author.name}

className='ui medium rounded bordered image'

/>

</div>

<div className='content'>

<div className='header authorName'>

<h1>{ author.name }</h1>

<div className='extra'>

<div className='ui label'>

{ author.books.count }

{ author.books.count > 1 ? ' Books' : ' Book' }

</div>

</div>

</div>

<div className='description'>

<p> { author.bio } </p>

</div>

</div>

</div>

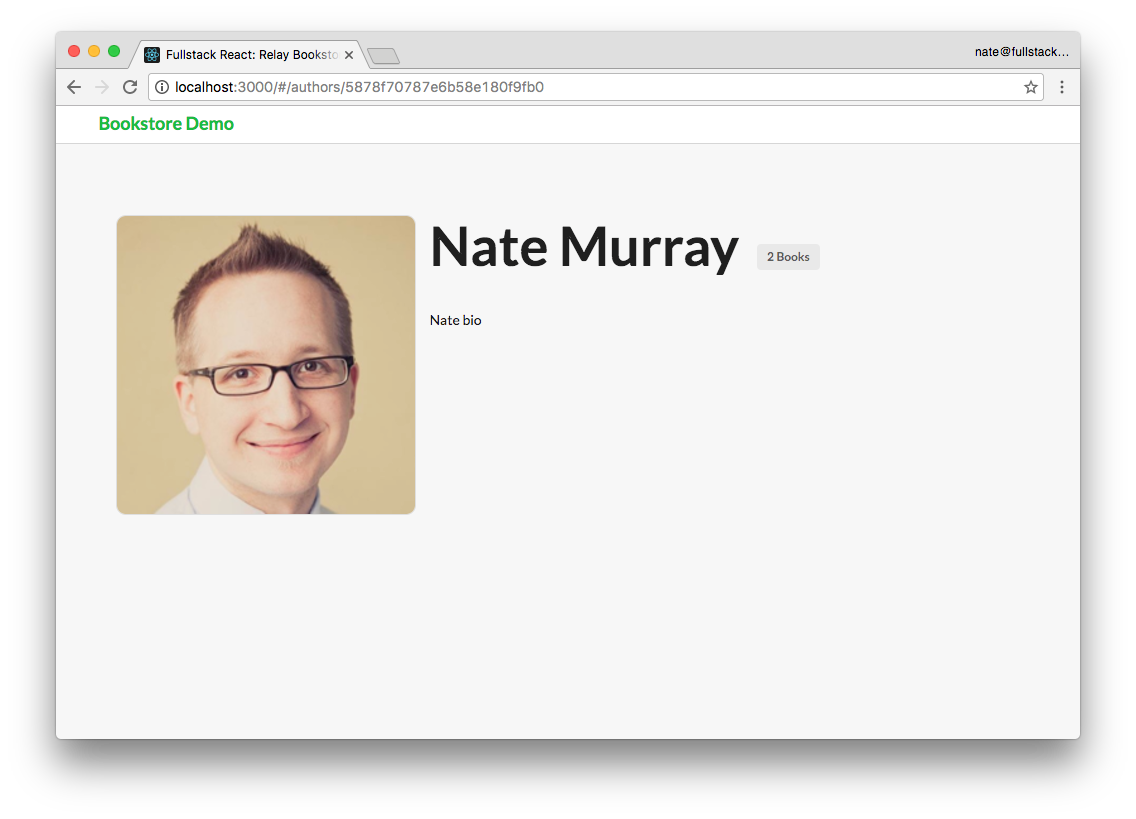
</div>

</div>

);

}

}

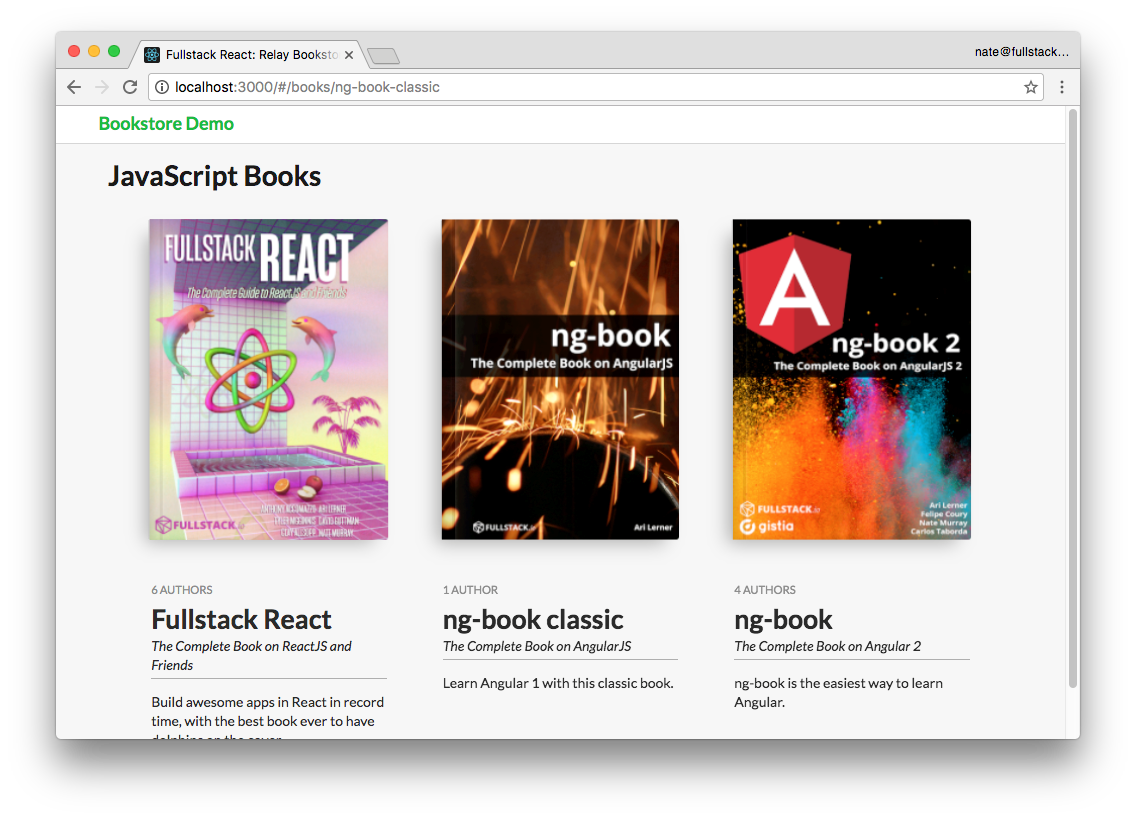


Author Page with Styling

We’ll add the list of this author’s books to this page later, but for now let’s build the “index” page of the site, which will show the list of all of the books available.

## BooksPage

Here’s what the list of books page will look like when we’re finished:



Books Page

The first thing we need to do is create the route for the BooksPage and the queries for that route.

### BooksPage Route

The BooksPage is going to be the default page for our app so we will use the IndexRoute helper to define this route:

**relay/bookstore/client/src/routes.js**

<IndexRoute

component={BooksPage}

queries={ViewerQueries}

/>

Let’s take a look at the ViewerQueries:

**relay/bookstore/client/src/routes.js**

**const** ViewerQueries = {

viewer: () => Relay.QL`query { viewer }`,

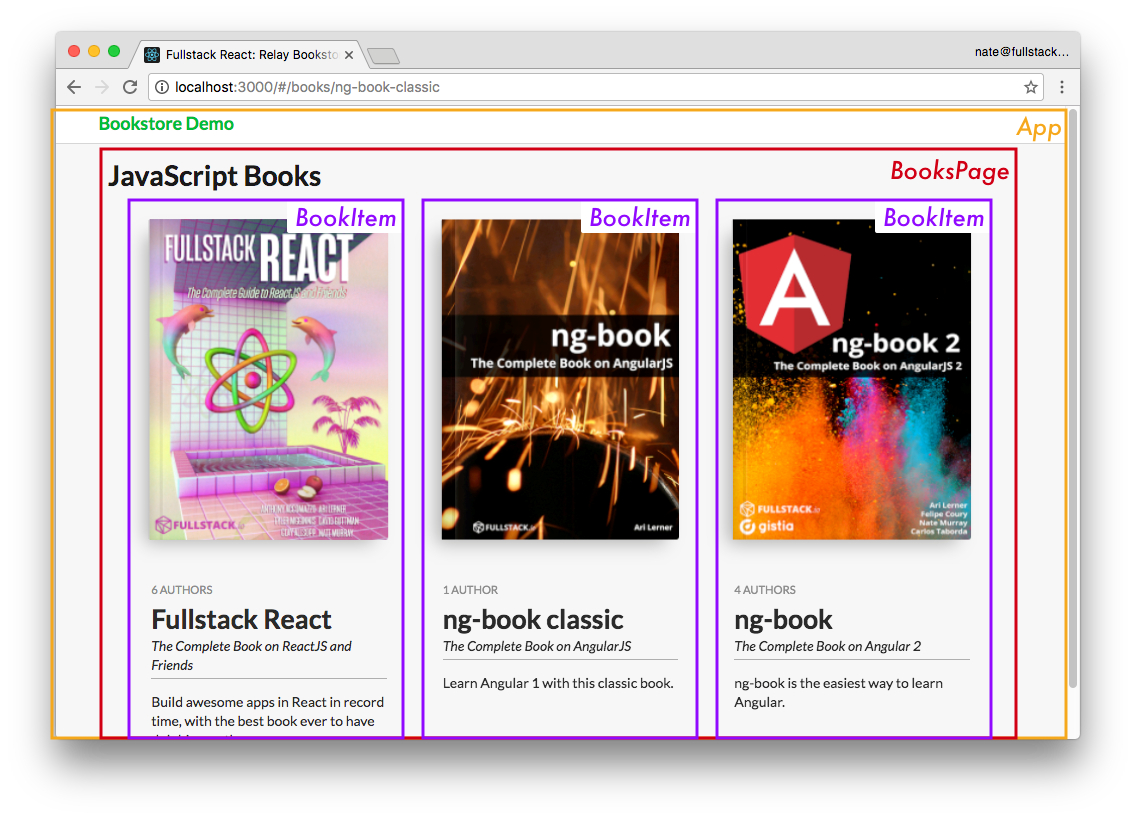
};

On this page, we’re going to be looking up the list of books via the viewer node.

The viewer node is not strictly part of Relay but it’s a pattern that you see in many GraphQL apps. The idea is that the “viewer” is the current user of the app. So, often in a real application you’d see that the viewer field accepts a user-identity field as an argument, such as an authentication token.

Imagine creating an app with a social feed. We could use the viewer field to get the feed for a particular user as opposed to the “firehose” feed for all users. In this case, we’re going to use the Viewer to load a list of Books.

### BooksPage Component



Books Page Components Here we have two Relay containers:

1. the BooksPage which holds all the books
2. a BookItem which renders a particular book

Because our BooksPage route specified queries on viewer we’re going to specify a fragment on the key viewer (on the Viewer-type). Let’s take a look at the query:

**relay/bookstore/client/src/components/BooksPage.js**

**export default** Relay.createContainer(BooksPage, {

initialVariables: {

count: 100

},

fragments: {

viewer: () => Relay.QL`

fragment on Viewer {

books(first: $count) {

count

edges {

node {

slug

**${**BookItem.getFragment('book')**}**

}

}

}

}

`

}

});

There’s a couple of new things here:

* 1. initialVariables
  2. Composing a fragment, using getFragment()

Fragment Variables

We can set variables for our queries by using the initialVariables field. Here you can see that we’re telling Relay we want to set $count to 100. In the query you can see that we’re asking for the first 100 books.

By using this viewer we can’t say “I want all the books”. The reason for this is that this server is designed for large apps and you generally don’t want to load every single item in your database. Typically we would use pagination.

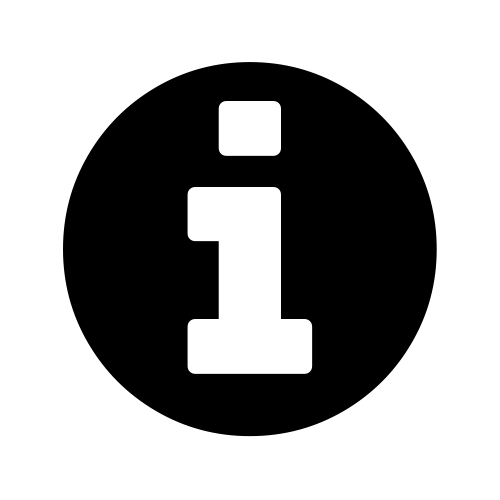
In this case though, we’re just going to set the count to 100 because that’s more than enough for this simple app. But say that we wanted to change the variable count. How would we do it?

You use this.props.relay.setVariables like this:

this.props.relay.setVariables({count: 2});

If we called the above function you would see that we load only 2 books instead of the whole set.

More generally, we can use Relay variables when we want our component to be able to change parameters about the query that’s being executed.

If you forget to set the proper variables (e.g. no first field) when trying to get a list of items through a connection, you might get an error like the following: {lang=shell,linenumbers=off } Uncaught Error: Relay transform error You supplied the `edges` field on a connection named `authors`, but you did not supply an argument necessary to do so. Use either the `find`, `first`, or `last` argument. in file code/relay/bookstore/client/src/components/BookPage.js. Try updating your GraphQL schema if an argument/field/type was recently added.

Fragment Composition

This is important: if you use a child Relay component you need to embed the child’s fragment in the parent’s query by using getFragment().

Remember that fragments are parts of queries and they aren’t realized until they’re part of an executed query. If you want a child component like BookItem to be able to render each book, you have to include the **BookItem** fragment in the **BooksPage** query.

This takes a little getting used to, and we haven’t even looked at the render function of BooksPage yet, so let’s do that and come back to this idea of fragment composition when we have a bit more context.

### BooksPagerender()

Here’s the rendering for BooksPage:

**relay/bookstore/client/src/components/BooksPage.js**

**class** BooksPage **extends** React.Component {

render() {

**const** books = **this**.props.viewer.books.edges.map(**this**.renderBook);

**return** (

<div className="sixteen wide column">

<h1>JavaScript Books</h1>

<div className="ui grid centered">{books}</div>

</div>

);

}

renderBook(bookEdge) {

**return** (

<Link

to={`/books/**${**bookEdge.node.slug**}**`}

key={bookEdge.node.slug}

className="five wide column book"

>

<BookItem book={bookEdge.node} />

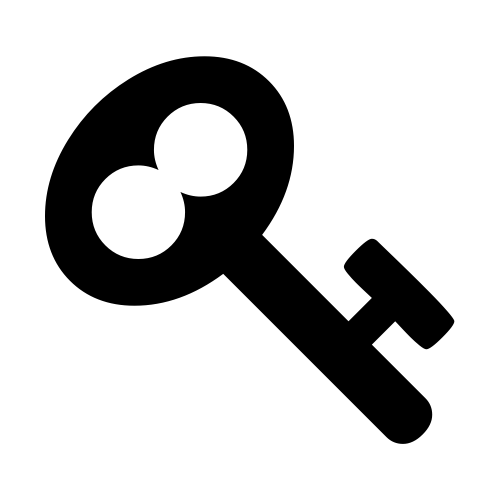
</Link>

);

}

}

Our render function has a basic wrapper. We’re taking this.props.viewer.books.edges and calling renderBook on each edge.

 Notice that we’re not calling renderBook on the book itself (node), but on the edge.

Within renderBook, we render a:

* 1. Link to /books/${bookEdge.node.slug} with a
  2. BookItem component, populating it with a book, found in bookEdge.node

Let’s dig deeper into BookItem and then we’ll pop back up to BooksPage to take a closer look at how Relay manages parent/child fragment values.

### BookItem

Here’s our Relay Query for BookItem:

**relay/bookstore/client/src/components/BookItem.js**

**export default** Relay.createContainer(BookItem, {

fragments: {

book: () => Relay.QL`

fragment on Book {

name

slug

tagline

coverUrl

pages

description

authors {

count

}

}

`

}

});

We’re defining a fragment with the key of book on type Book. This query is asking for basic book information like the name, the slug, the number of pages, and we also dip into authors to get the count.

Here’s the component definition:

**relay/bookstore/client/src/components/BookItem.js**

**class** BookItem **extends** React.Component {

render() {

**return** (

<div className="bookItem">

<FancyBook book={**this**.props.book} />

<div className="bookMeta">

<div className="authors">

{**this**.props.book.authors.count}

{**this**.props.book.authors.count > 1 ? ' Authors' : ' Author'}

</div>

<h2>{**this**.props.book.name}</h2>

<div className="tagline">{**this**.props.book.tagline}</div>

<div className="description">{**this**.props.book.description}</div>

</div>

</div>

);

}

}

Inside the divbookMeta we show the basic information such as the number of authors, the book name, tagline and description.

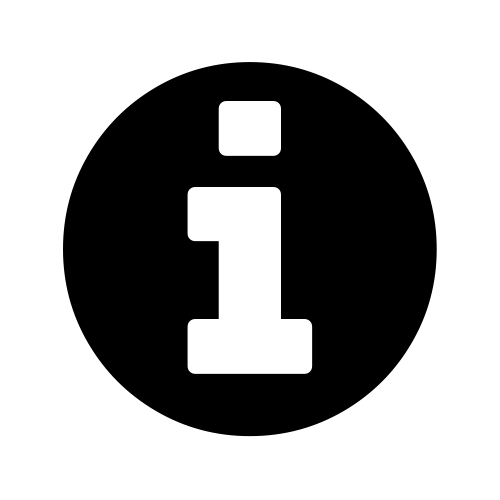
We also pass this.props.book to a pure component FancyBook, which simply provides the markup for the fancy 3D CSS effect on the book.

### BookItem Fragment

What’s interesting about the BookItem fragment book is that it isn’t tied to a particular query. Remember that the query we’re using in BooksPage starts on a Viewer.

What we’re doing here is saying, to use this component, it’s concern is with something on a Book type. As long as we compose this Book fragment into the parent’s fragment at a location where a Book fragment is valid then we’ll be able to use this BootItem component.

The rule of thumb here is that if you are composing Relay components, make sure you compose their fragments.

 If you get a warning that reads like the following:

warning.js:36 Warning: RelayContainer: component BookItem was rendered with variables that differ from the variables used to fetch fragment book.

This means that you forgot to include the child fragment in parent’s query.

### Fragment Value Masking

There’s another important feature of Relay that we haven’t talked about yet: data masking.

The idea is components can only see data they asked for explicitly. If the component didn’t ask for a specific field, Relay will actively hide that field from the component even if that data was loaded.

For instance, look at the queries on BookItem and BooksPage side-by-side . Notice that:

* BookItem loads the slug field for the book and
* BooksPage uses the slug field for the Link URL

For review, here’s BookItem’s Relay.QL query:

**relay/bookstore/client/src/components/BookItem.js**

**export default** Relay.createContainer(BookItem, {

fragments: {

book: () => Relay.QL`

fragment on Book {

name

slug

tagline

coverUrl

pages

description

authors {

count

}

}

`

}

});

And BooksPage’s Relay.QL query:

**relay/bookstore/client/src/components/BooksPage.js**

**export default** Relay.createContainer(BooksPage, {

initialVariables: {

count: 100

},

fragments: {

viewer: () => Relay.QL`

fragment on Viewer {

books(first: $count) {

count

edges {

node {

slug

**${**BookItem.getFragment('book')**}**

}

}

}

}

`

}

});

You might think that because we’re calling ${BookItem.getFragment('book')} in the BooksPage query that means that slug would be available to the BooksPage component, but this is not the case.

Fragment composition does not make that child-fragment’s data available to our parent query. We must explicitly list the slug field in the BooksPage query if we want to be able to use that field in our render function of BooksPage.

Data masking is one of those features that can feel like an inconvenience at first, but it turns out to be super helpful as your app (and team) grows because it prevents bugs that result from changes outside of your component.

For instance, say that BooksPage was depending on the slug field from BookItem to be loaded. But, somewhere down the line, someone is cleaning up BookItem and they remove the slug field. What would happen? BooksPage would now break.

The idea is that it is bad to have this coupling between components. Each component should define everything it needs to render properly. Data masking is a way to ensure that every component explicitly defines the set of data it needs to operate.

Masking Works Both Ways

This masking also goes the other way: since we do define that BooksPage (the parent) needs the slug field, even though we pass the book (via <BookItem book={bookEdge.node}/>) the child cannot access **slug** if it does not ask for it explicitly.

Any Relay manged props will have masking applied to them according to the queries. So be mindful of data masking when writing your applications – if you find that you unexpectedly have missing data (that you know you loaded somewhere else), check your queries to make sure you’ve explicitly request those specific fields in the component where they’re needed.

### Improving the AuthorPage

Now that we can render a list of books, a nice touch to the author’s page would be to show a list of the books that person has authored.

Given what we’ve just covered, we know that we can add this list of books by doing the following:

1. Request the author’s books in our Relay query
2. Include the BookItem fragment in this query
3. Take the list of the author’s books and render a BookItem for each one

Let’s do this now.

Requesting an Author’s Books

Right now the AuthorPage only includes the count of the number of books an author appears on. Let’s change the AuthorPage query to include more data on author’s books in order to show a complete book thumbnail and link to them:

**relay/bookstore/client/src/components/AuthorPage.js**

**export default** Relay.createContainer(AuthorPage, {

fragments: {

author: () => Relay.QL`

fragment on Author {

\_id

name

avatarUrl

bio

books(first: 100) {

count

edges {

node {

slug

**${**BookItem.getFragment('book')**}**

}

}

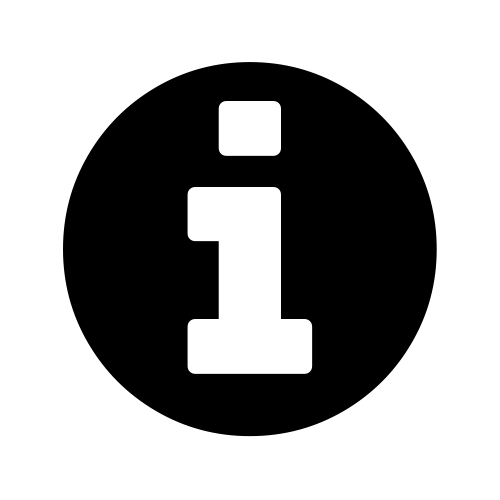
}

}`

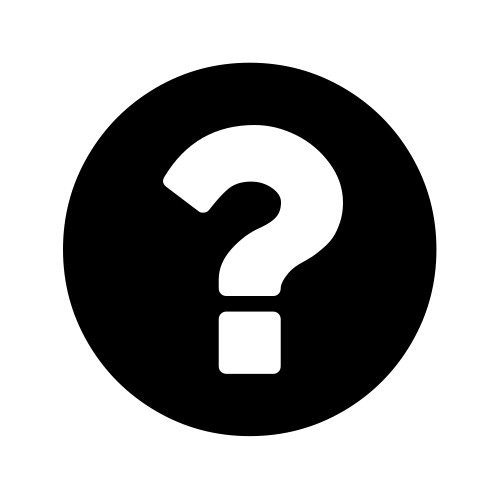
}

});

This change shows one of the things that’s great about Relay and GraphQL. Consider what this type of change might look like if we were working with a traditional REST-based API. We’d have to write code that looks up the author’s id and makes a request to the server asking for that author’s books. We’d have to add code to gracefully handle errors. Here, with Relay, all we have to do is add a books() section to the query and populate it with the fields we want.

Notice that we just hard-code that we want the first 100 books. In a real app, we’d probably set a variable like we did on the BooksPage.

The other thing we have to remember to do is include the BookItem fragment book. Again, note that the “parent” query here is author. You might recall that this is on an Author type up in the router. But, you don’t really need to remember that. All you need to know is that BookItem’s book fragment is on type Book and so you can place it in your query wherever a Book type is allowed.

How do I know what fragments my child components need?

If you’re building your first Relay app it might not be immediately clear how you even know what fragments to request. Essentially, it should be part of the documentation of the component. In the same way that in order to use a “regular” component, you’re probably going to know what props it needs, in the same way, the author of the component you’re using needs to document what fragments you’ll use to render that component.

**AuthorPagerenderBook**

Now that we have the books we can render them like so:

**relay/bookstore/client/src/components/AuthorPage.js**

**class** AuthorPage **extends** React.Component {

renderBook(bookEdge) {

**return** (

<Link

to={`/books/**${**bookEdge.node.slug**}**`}

key={bookEdge.node.slug}

className="five wide column book"

>

<BookItem book={bookEdge.node} />

</Link>

);

}

and in the render() function:

render() {

**const** author = **this**.props.author;

**const** books = **this**.props.author.books.edges.map(**this**.renderBook);

**return** (

{*/\* ... truncated \*/*}

<div className='sixteen wide column'>

<h1>{ author.name }&rsquo;s Books</h1>

<div className='ui grid centered'>

{ books }

</div>

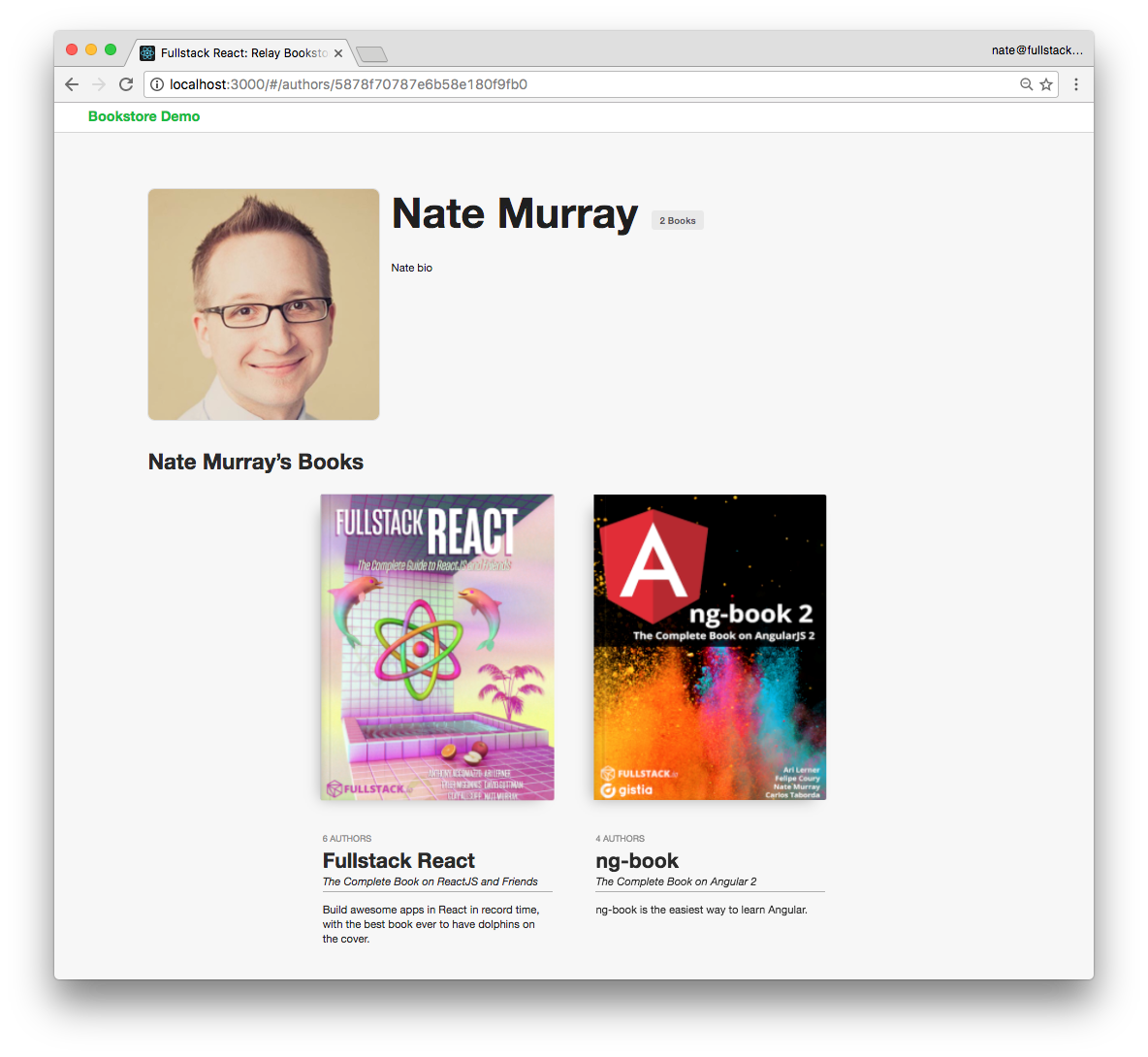
</div>

{*/\* ... truncated \*/*}

);

}

Here’s a screenshot of what our author page looks like now that we have the books added in:



Author with Books Page

## Changing Data With Mutations

Now we’ve seen how to:

1. read a single record of data from Relay
2. read a list of data from Relay
3. load data from child components

There’s one critical issue we haven’t covered: changing data. In Relay, changing data is done through mutations.

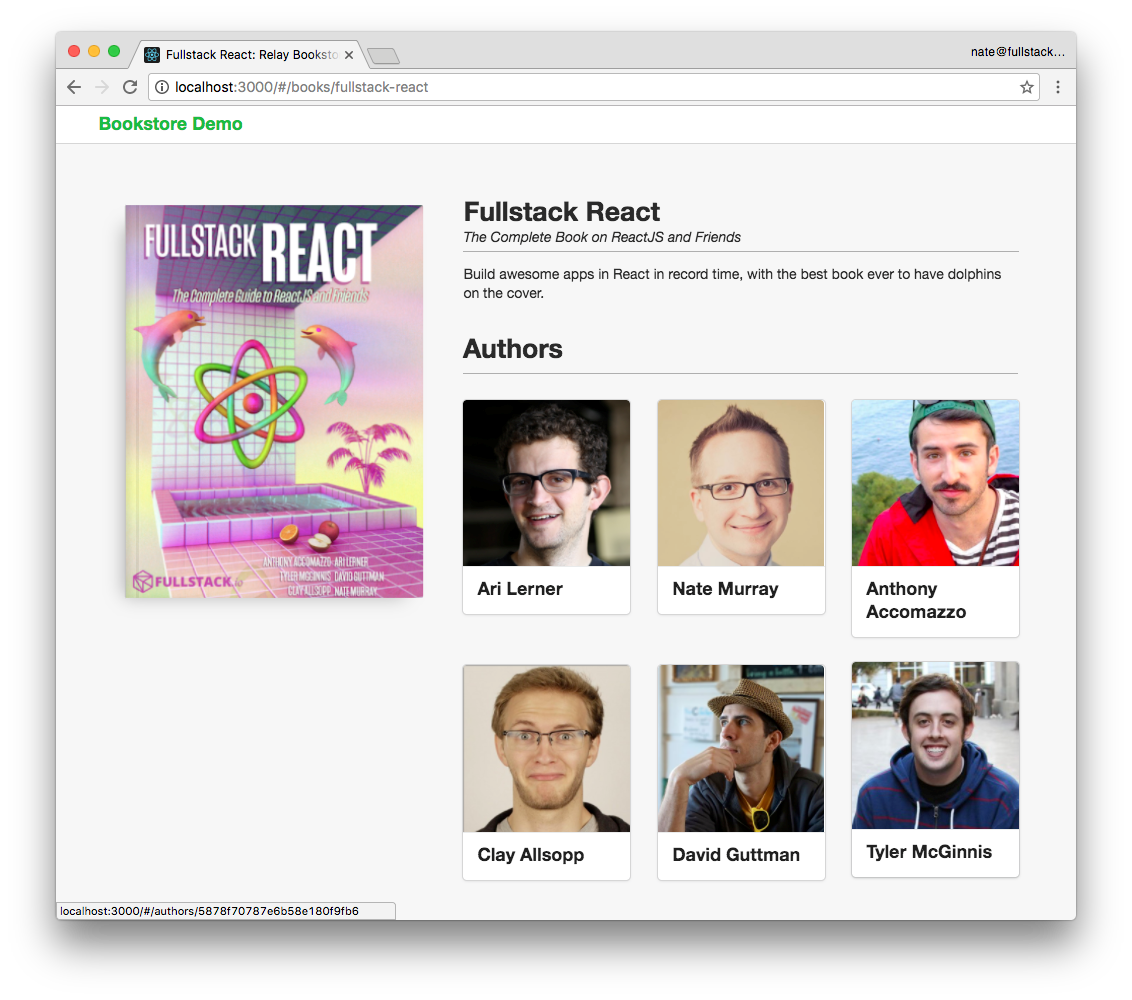
We’re going to add the ability to edit the metadata of a book.

First, let’s create a new page for an individual book. We’ll read the initial data from Relay, as have been doing. Then we’ll talk about how to change the data with mutations.

## Building a Book’s Page

The read-only version of the individual book page doesn’t have a lot of new ideas in it, so let’s talk through it briefly.

First, here’s a screenshot:



Individual Book Page, Read Only

You can see that we show the basic book image, each author, and the book cover.

Here’s the Relay container:

**relay/bookstore/client/src/steps/BookPage.reads.js**

**export default** Relay.createContainer(BookPage, {

fragments: {

book: () => Relay.QL`

fragment on Book {

id

name

tagline

coverUrl

description

pages

authors(first: 100) {

edges {

node {

\_id

name

avatarUrl

bio

}

}

}

}`,

},

});

In this fragment we’re loading the Book metadata as well as the first 100 authors.

Let’s look at the render() function:

**relay/bookstore/client/src/steps/BookPage.reads.js**

render() {

**const** { book } = **this**.props;

**const** authors = book.authors.edges.map(**this**.renderAuthor);

**return** (

<div className='bookPage sixteen wide column'>

<div className='spacer row' />

<div className='ui grid row'>

<div className='six wide column'>

<FancyBook book={book} />

</div>

<div className='ten wide column'>

<div className='content ui form'>

<h2>{book.name}</h2>

<div className='tagline hr'>

{book.tagline}

</div>

<div className='description'>

<p>

{book.description}

</p>

</div>

</div>

<div className='ten wide column authorsSection'>

<h2 className='hr'>Authors</h2>

<div className='ui three column grid link cards'>

{authors}

</div>

</div>

</div>

</div>

</div>

);

}

Most of the markup here is divs with Semantic UI classes. Basically we’re just show the book information with a reasonable layout.

For each author edge we call renderAuthor:

**relay/bookstore/client/src/steps/BookPage.reads.js**

renderAuthor(authorEdge) {

**return** (

<Link

key={authorEdge.node.\_id}

to={`/authors/**${**authorEdge.node.\_id**}**`}

className='column'

>

<div className='ui fluid card'>

<div className='image'>

<img src={authorEdge.node.avatarUrl}

alt={authorEdge.node.name}

/>

</div>

<div className='content'>

<div className='header'>{authorEdge.node.name}</div>

</div>

</div>

</Link>

);

}

### Book Page Editing

Now let’s add in some basic editing for this book’s data. What we’d like to be able to do is click on any field like the title or description and edit it in-place. To make this simple, let’s use the existing inline-edit component [React Edit Inline Kit1](https://github.com/kaivi/riek)68.

React Edit Inline Kit (or RIEK for short) has a simple API. We specify:

1. The original value
2. The property name of this value 3. A callback to run when this value changes

So that we can isolate what we’re doing, let’s integrate RIEK into our app without changing the data in Relay. The first step is just to get the inline form editing working and then we’ll deal with what to do with the changes once we have them.

Here’s our new render() function on BookPage with RIEK:

**relay/bookstore/client/src/steps/BookPage.iek.js**

render() {

**const** { book } = **this**.props;

**const** authors = book.authors.edges.map(**this**.renderAuthor);

**return** (

<div className='bookPage sixteen wide column'>

<div className='spacer row' />

<div className='ui grid row'>

<div className='six wide column'>

<FancyBook book={book} />

</div>

<div className='ten wide column'>

<div className='content ui form'>

<h2>

<RIEInput

value={book.name}

propName={'name'}

change={**this**.handleBookChange}

/>

</h2>

<div className='tagline hr'>

<RIEInput

value={book.tagline}

propName={'tagline'}

change={**this**.handleBookChange}

/>

</div>

<div className='description'>

<p>

<RIETextArea

value={book.description}

propName={'description'}

change={**this**.handleBookChange}

/>

</p>

</div>

</div>

<div className='ten wide column authorsSection'>

<h2 className='hr'>Authors</h2>

<div className='ui three column grid link cards'>

{authors}

</div>

</div>

</div>

</div>

</div>

);

}

We’ve added three new components using RIEInput and RIETextArea. All three will call handleBookChange when the data changes. Here’s the current implementation:

**relay/bookstore/client/src/steps/BookPage.iek.js**

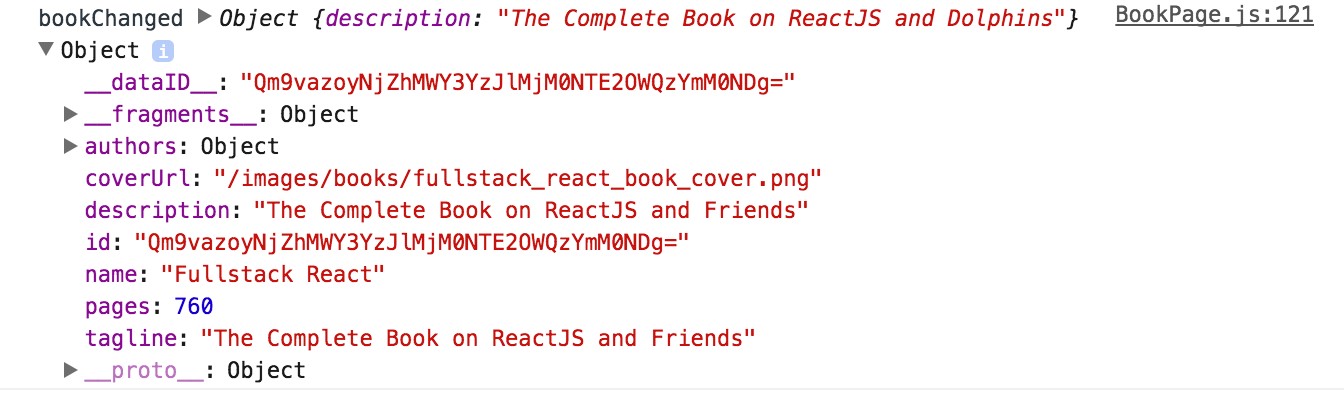
handleBookChange(newState) {

console.log('bookChanged', newState, **this**.props.book);

}

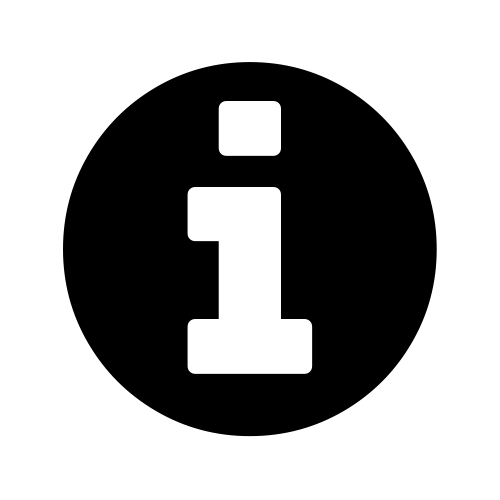
All we’re going to do at this point is log out the changes to **console**.

Here’s a screenshot of what happens when I change the tagline to "The Complete Book on ReactJS and Dolpins":



**console.log** the tagline

You can see here that newState has the new values we need to update the book. Our task at hand is to tell Relay about this change via a mutation.

Just to be clear, RIEK is a completely arbitrary choice for a forms library, selected here for convenience. It has nothing to do with Relay. The steps that we’re about to take to create mutations in Relay could be done with any form library. If you’d like to build your own forms, that’s perfectly fine. Checkout the chapter “Forms.”

## Mutations

A mutation in Relay is an object that describes a change. Mutations are one of the most difficult aspects to get used to in Relay. That said, the complexity of mutations comes trade-offs that are inherent to building client-server applications. Let’s take a look at the steps necessary to create mutations in Relay: To mutate data in Relay:

1. We define a mutation object
2. We create an instance of that object, passing configuration variables and then
3. We send it to Relay using Relay.Store.commitUpdate

There are 5 major types of mutations in Relay:

* FIELDS\_CHANGE
* NODE\_DELETE
* RANGE\_ADD
* RANGE\_DELETE
* REQUIRED\_CHILDREN

We are going to update the fields for an existing object, and so we will write a FIELDS\_CHANGE mutation.

### Defining a Mutation Object

Mutations in Relay are objects. To define a new mutation we subclass Relay.Mutation. Then when we want to execute a mutation, we create an instance of this class, configure it with the appropriate properties, and give it to Relay (which handles communicating with the server and updating the Relay store).

When we create a Relay.Mutation subclass, we have to define 6 things that describe the behavior of the mutation:

1. What GraphQL method are we using for this mutation?
2. What variables will be used as the input to this mutation?
3. What fields does this mutation depend on in order to run properly?
4. What fields could change as a result of this mutation?
5. If everything goes smoothly, what’s the expected outcome of this mutation?
6. How should Relay handle the actual response that comes back from the server?

We specify each of these things by defining a function for each one.

This might be more care than we usually take when we’re using a fire-and-forget API, but remember, by specifically describing each of these steps in our mutation we get the benefits of Relay managing the bookkeeping for our data.

This is much easier if we walk through a concrete example, so lets make an “update” with FIELDS\_CHANGE.

Open up client/src/mutations/UpdateBookMutation.js. Starting at the top let’s look at .getMutation:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

**export default class** UpdateBookMutation **extends** Relay.Mutation {

*// ...*

getMutation() {

**return** Relay.QL`mutation { updateBook }`;

}

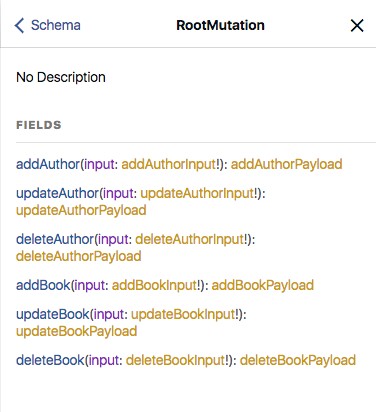
*// ...*

}

The first thing we need to specify when we create a Relay.Mutation subclass is the node of the GraphQL mutation. Here we’ve specified that we’re using the updateBook mutation.

You can find this mutation in the schema by using GraphiQL and picking the mutation field at the root.

We can see here that updateBook takes a single argument input, which is of type updateBookInput, and it returns an item of type updateBookPayload.



For updateBook, the values that we use for updateBookInput will be used as the new values for the book we specify.

That is, we’ll send an id, which will be used to look up the specific Book we’re changing, and we’ll send new values such as the name, tagline, or description.

When it comes time to send this mutation, the values passed in to updateBookInput will come from the function getVariables:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js updateBook Mutation**

getVariables() {

**return** {

id: **this**.props.id,

name: **this**.props.name,

tagline: **this**.props.tagline,

description: **this**.props.description

};

}

This function describes how to create the arguments for updateBookInput – in this case we’re going to look at this.props for values for the id, name, tagline, and description.

One of the things we have to be careful of is to ensure that this mutation has all of the fields it needs to operate properly. For instance, this mutation especially needs a Bookid because that’s how we’re going to reference the object we’re changing.

To do this, we specify fragments, much like we do for Relay container components:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

**static** fragments = {

book: () => Relay.QL`

fragment on Book {

id

name

tagline

description

}

`

};

Notice that in getVariables we’re also sending along the name, tagline, and description – without checking if they have any value.

Relay will mask **prop** values from mutations just like components. So there’s a subtle bug that could be introduced here. If we forget to specify the proper fragment, the mutation can accidentally set values to null.

What’s the solution? In the same way that parent components need to include their child components’ fragments, any component that uses a mutation also needs to include that mutation’s fragments. We’ll show how to do this when we use the UpdateBookMutation in our app below.

After we send this mutation it will be evaluated on the server. In this case, what changes on the server is pretty simple: we’re updating the field values for one object.

That said, for many mutations the effects are probably more nuanced – we can’t always know the full set of side effects that will occur from a mutation operation.

Furthermore, we don’t actually have confirmation that this operation succeeded at all.

To deal with this, we’re going to ask the server to send back to us all the fields that we think might have changed.

To do this, we’ll specify what’s called a “fat query”. It’s “fat” because we’re trying to capture everything that might have changed:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

getFatQuery() {

**return** Relay.QL`

fragment on updateBookPayload {

changedBook

}

`;

}

The fat query is a GraphQL query. In this case, we’re just asking for the changedBook. So once the mutation is run on the server we’re asking the server to send us back the new, updated values for the book that we (hopefully) changed.

Relay will take that changedBook (which is an object of type Book), look at its ID, and update the Relay Store accordingly.

However, before the server returns our actual response, we have the option to make a performance optimization and specify an “optimistic” response. The optimistic response answers the question: assuming this mutation executed successfully, what would be the response?

Here’s our implementation of getOptimisticResponse:

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

getOptimisticResponse() {

**const** {book, id, name, tagline, description} = **this**.props;

**const** newBook = Object.assign({}, book, {id, name, tagline, description});

**const** optimisticResponse = {

changedBook: newBook

};

console.log('optimisticResponse', optimisticResponse);

**return** optimisticResponse;

}

In this function we’re merging together the old book with the argument values. Our optimistic response returns a newBook that looks like the response we’re about to get from getFatQuery.

This mutation is straightforward: we have the original book and we have the updated fields. In this case, we know what the result of the mutation is going to be without even asking the server.

So what we can do is fake to the user that their operation was successful. This can give our user the feeling of a super-responsive app, because they’re able to see the effects of their changes immediately without waiting for a network call.

If the mutation succeeds, the user is none the wiser.

If the mutation fails, then we’ve given the user false confirmation. But we’ll still have the opportunity to handle that case and inform the user, perhaps telling them that they need to try again.

If the consistency trade-offs are acceptable to your application, optimistic responses can greatly improve the feel of the response time of your app.

When the real response is returned from the server, Relay needs to be told how to handle that data.

Because there are several different ways to mutate data, Relay mutations must implement a getConfigs method which describes the way Relay is going to handle the actual data that changed.

**relay/bookstore/client/src/mutations/UpdateBookMutation.js**

getConfigs() {

**return** [

{

type: 'FIELDS\_CHANGE',

fieldIDs: {

changedBook: **this**.props.book.id

}

}

];

}

In this case, because we’re using a FIELDS\_CHANGE mutation, we’re specifying that we’ll look at changedBook to find an ID that matches the Book we’re talking about.

### Inline Editing

Now that we have our mutation defined, we’re now prepared to implement click-to-edit on our

BookPage.

Remember that our mutation is going to be making a query to check with the server for the current value of the Books data after the mutation finishes. Because of this, we need to compose our mutation’s fragment into our **BookPage** query.

Just like a child Relay component needs its fragments composed, any mutations a component uses also needs to have its fragments composed.

Here is our BookPage query now that we’re adding in the mutation’s fragments:

**relay/bookstore/client/src/components/BookPage.js**

fragments: {

book: () => Relay.QL`

fragment on Book {

id

name

tagline

coverUrl

description

pages

authors(first: 100) {

edges {

node {

\_id

name

avatarUrl

bio

}

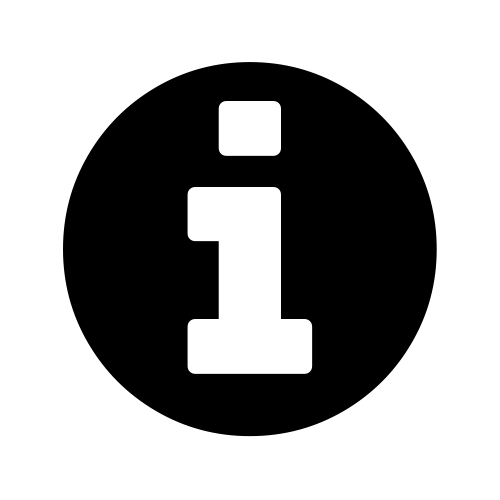
}

}

**${**UpdateBookMutation.getFragment('book')**}**

}`

}

In the future, if you forget to add your mutation fragment to your query, you might get this:

* 1. Warning: RelayMutation: Expected prop `book` supplied to `UpdateBookMutation` to\
  2. be data fetched by Relay. This is likely an error unless you are purposely pass\ 3 ing **in** mock data that conforms to the shape **of this** mutation's fragment.

The solution: add your mutation fragment to the calling component’s query.

Now we have everything in place to actually execute our mutation! Let’s update handleBookChange to send out the mutation:

**relay/bookstore/client/src/components/BookPage.js**

handleBookChange(newState) {

console.log('bookChanged', newState, **this**.props.book);

**const** book = Object.assign({}, **this**.props.book, newState);

Relay.Store.commitUpdate(

**new** UpdateBookMutation({

id: book.id,

name: book.name,

tagline: book.tagline,

description: book.description,

book: **this**.props.book

})

);

}

Here we create a new object that is the merger of this.props.book and newState by using Object.assign. Next we execute our mutation by calling Relay.Store.commitUpdate and passing a new UpdateBookMutation to it. Behind the scenes, Relay will:

* + immediately update our view (and the Relay store) using the optimistic response
  + call out to the GraphQL server and try to execute our mutation
  + receive the reply from the GraphQL server and update the Relay store

Hopefully, at this point our optimistic response matches the actual response. But if not, the actual value will be represented on our page.

1. <https://www.npmjs.com/package/babel-relay-plugin> [↑](#footnote-ref-1)
2. This is similar to how JSX works [↑](#footnote-ref-2)
3. <http://graphql.org/learn/introspection/> [↑](#footnote-ref-3)
4. <https://github.com/relayjs/relay-starter-kit> [↑](#footnote-ref-4)
5. <https://github.com/relay-tools/react-router-relay/issues/193> [↑](#footnote-ref-5)
6. <https://github.com/ReactTraining/react-router> [↑](#footnote-ref-6)
7. <http://semantic-ui.com/> [↑](#footnote-ref-7)