The easiest way to get started with React apps that are using a GraphQL backend is with [create-react-app](https://github.com/facebookincubator/create-react-app) and [Apollo](http://dev.apollodata.com/). In this tutorial we will learn how to build an Instagram app, where users can view and post images. The complete code for this tutorial is available on [GitHub](https://github.com/graphcool-examples/react-graphql/tree/master/quickstart-with-apollo).

### **Getting your GraphQL endpoint**

For this tutorial you’ll need a GraphQL project with the following data model:

type Post @model {

id: ID! @isUnique

createdAt: DateTime!

description: String!

imageUrl: String!

}

[[1]](#footnote-0)

Using the [Graphcool CLI](https://www.npmjs.com/package/graphcool), you can generate a fully-fledged GraphQL server from your command line by providing this schema as an input argument to the graphcool initcommand:

1. In case you haven’t already, install the Graphcool CLI: npm install -g graphcool
2. Type graphcool init graphcool-server
3. cd graphcool-server and save the data model/schema above into the types.graphql file created there.
4. Create the GraphQL server: graphcool deploy  
   (you will need to login and authenticate with GitHub)

That’s it, copy the endpoint for the Simple API since we'll need it later on. Notice that you can now also manage this project in the [Graphcool console](https://console.graph.cool/) or explore it in a [GraphQL playground](https://www.graph.cool/docs/faq/tips-and-tricks-graphql-playground-ook6luephu/).

### **Getting started with create-react-app**

#### **Installation**

With create-react-app, it's super easy to start out with a new React application! It comes with a lot of nice features out-of-the-box, such as a preconfigured [Webpack](https://github.com/webpack/webpack) and [Babel](https://babeljs.io/) setup for zero build configurations. Furthermore, features like [JSX](https://jsx.github.io/) and [ES6](http://es6-features.org/) syntax as well as static typechecking with [Flow](https://flowtype.org/) are already included. The generated boilerplate code also gives you a head-start with your new app.

If you don’t have create-react-app installed on your machine yet, now is the time to do so with the following command:

npm install -g create-react-app

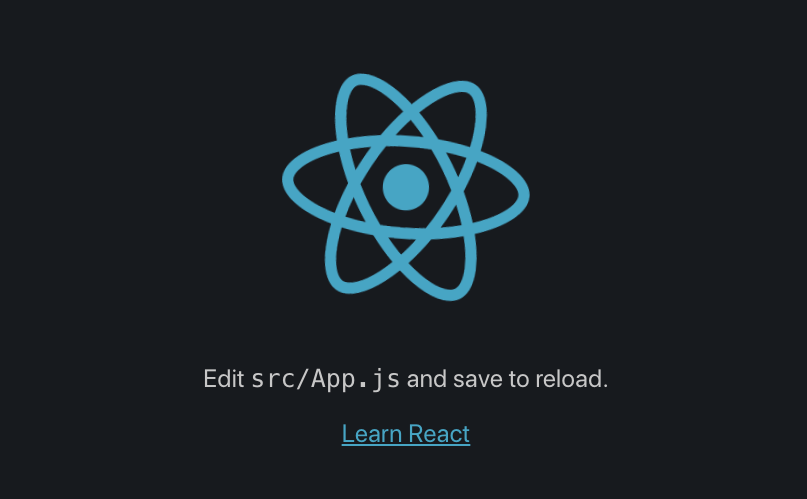
After installing, we can easily create and run a new project from the command line:

create-react-app react-apollo-instagram-example

cd react-apollo-instagram-example

npm start # open http://localhost:300

Hot-reloading and linting is included as well, the terminal window keeps us updated about errors and linter problems.



### **Integrating Apollo in a React Application**

#### **Installing the Dependencies**

[Apollo Client](http://dev.apollodata.com/react/) is one of the most popular GraphQL clients available at the moment. It implements features like caching, optimistic UI, [query batching](https://www.graph.cool/blog/improving-performance-with-apollo-query-batching-ligh7fmn38/) as well as [realtime updates using subscriptions](https://www.graph.cool/docs/tutorials/worldchat-subscriptions-example-ui0eizishe/) and generally makes interacting with a GraphQL backend a breeze.

We will need dependencies to use it in our React application:

1. [graphql](https://github.com/graphql/graphql-js): The JavaScript reference implementation for GraphQL
2. [react-apollo](https://github.com/apollographql/react-apollo): Implements React-specific bindings for Apollo
3. [apollo-client](https://github.com/apollographql/apollo-client): Contains the general functionality of Apollo Client, and also it's related libraries:
   1. [apollo-cache-inmemory](https://www.npmjs.com/package/apollo-cache-inmemory)
   2. [apollo-link-http](https://www.npmjs.com/package/apollo-link-http)
4. [graphql-tag](https://github.com/apollographql/graphql-tag): Provides functionality for parsing the [JavaScript template literals](http://exploringjs.com/es6/ch_template-literals.html) that will contain our GraphQL queries and mutations

You can install all of these dependencies at once:

npm install graphql react-apollo apollo-client \  
 apollo-cache-inmemory apollo-link-http graphql-tag --save

Additionally, we’re using [React Router](https://github.com/ReactTraining/react-router), and it DOM bindings [react-router-dom](https://github.com/ReactTraining/react-router/blob/master/packages/react-router-dom) to deal with navigation in our app, let’s go ahead install it and that as well:

npm install react-router react-router-dom --save

Finally, for styling, we’re using [tachyons](http://tachyons.io/):

npm install tachyons --save

#### **Mocking the needed Components**

Let’s first build the components needed for our app, where we want to display, create or delete posts. Afterwards, we’ll inject the required data using Apollo and wire everything up with React Router.

These are the three components that we need:

* ListPage in src/components/ListPage.js that will list all posts from our backend

import React from 'react';

import { Link } from 'react-router-dom';

class ListPage extends React.Component {

render() {

return (

<div className="w-100 flex justify-center">

<Link to="/create" className="fixed bg-white top-0 right-0 pa4 ttu dim black no-underline">

{' '}

+ New Post

</Link>

<div className="w-100" style={{ maxWidth: 400 }}>

TODO: Display all posts...

</div>

</div>

)

}

}

export default ListPage

* CreatePage in src/components/CreatePage.js to create new posts

import React from 'react';

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

class CreatePage extends React.Component {

state = {

description: '',

imageUrl: '',

};

render () {

return (

<div className='w-100 pa4 flex justify-center'>

<div style={{ maxWidth: 400 }} className=''>

<input

className='w-100 pa3 mv2'

value={this.state.description}

placeholder='Description'

onChange={(e) => this.setState({description: e.target.value})}

/>

<input

className='w-100 pa3 mv2'

value={this.state.imageUrl}

placeholder='Image Url'

onChange={(e) => this.setState({imageUrl: e.target.value})}

/>

{this.state.imageUrl &&

<img src={this.state.imageUrl} role='presentation' className='w-100 mv3' alt={this.state.description} />

}

{this.state.description && this.state.imageUrl &&

<button className='pa3 bg-black-10 bn dim ttu pointer' onClick={this.handlePost}>Post</button>

}

</div>

</div>

)

}

handlePost = () => {

// TODO: create post before going back to ListPage

console.log(this.state);

this.props.history.replace('/')

}

}

export default withRouter(CreatePage)

* Post in src/components/Post.js to display and delete a single post

import React from 'react';

class Post extends React.Component {

render() {

return (

<div className="pa3 bg-black-05 ma3">

<div

className="w-100"

style={{

backgroundImage: `url(${this.props.post.imageUrl})`,

backgroundSize: 'cover',

paddingBottom: '100%',

}}

/>

<div className="pt3">

{this.props.post.description}&nbsp;

<span className="red f6 pointer dim" onClick={this.handleDelete}>

Delete

</span>

</div>

</div>

)

}

handleDelete = () => {

// TODO: delete post before reloading posts

this.props.refresh()

}

}

export default Post

#### **Setting up React Router and Apollo Client**

First, add these styles in src/index.ccs:

body {

padding: 0;

}

html {

box-sizing: border-box;

}

\*, \*:before, \*:after {

box-sizing: inherit;

}

Now we’re importing the needed packages in src/index.js

import ListPage from './components/ListPage'

import CreatePage from './components/CreatePage'

import {BrowserRouter as Router, Route} from 'react-router-dom';

import ApolloClient from 'apollo-client'

import {ApolloProvider} from 'react-apollo'

import {InMemoryCache} from 'apollo-cache-inmemory';

import {HttpLink} from 'apollo-link-http';

import 'tachyons';

Next we can create a new instance of ApolloClient below the import statements:

// replace `\_\_SIMPLE\_API\_ENDPOINT\_\_` with the endpoint from the previous step

const httpLink = new HttpLink({

uri: 'https://api.graph.cool/simple/v1/\_\_PROJECT\_ID\_\_',

});

const client = new ApolloClient({

link: httpLink,

cache: new InMemoryCache()

});

*Note: The uri that we have to pass to the* new HttpLink *constructor is the GraphQL endpoint for the Simple API that we generated in the first step using graphcool init. You can also retrieve that endpoint from the* [*Graphcool console*](https://console.graph.cool/) *by selecting your project and then clicking the ENDPOINTS-button in the bottom-left corner.*

Let’s now setup the routes for our application:

ReactDOM.render(

<ApolloProvider client={client}>

<Router>

<Route path="/" component={ListPage} />

<Route path="/create" component={CreatePage} />

</Router>

</ApolloProvider>,

document.getElementById('root'),

);

Note that the ApolloProvider is wrapping Router, which enables all child components to access the functionality from Apollo Client to send queries and mutations.

### **Using Apollo Client for Queries and Mutations**

Now we are ready to use Apollo in our components to interact with the GraphQL API!

#### **Querying all Posts in ListPage**

To display all posts in ListPage, we're adding three new imports in src/component/ListPage.js:

import Post from '../components/Post';

import { graphql } from 'react-apollo';

import gql from 'graphql-tag';

Apart from the Post component that renders a single post, we import gql and graphql. gql is used to create queries and mutations. [graphql](http://dev.apollodata.com/react/api.html#graphql) actually is a higher-order component that takes as input arguments one or more queries and/or mutations that were created with gql as well as a React component and injects the data from the query and/or the mutation function into the component as a prop.

First, let’s think about the query to display all posts:

query allPosts {

allPosts(orderBy: createdAt\_DESC) {

id

imageUrl

description

}

}

At the end of the file, outside of the Post class, we are now adding the ALL\_POSTS\_QUERY with gql which queries information about all our posts:

const ALL\_POSTS\_QUERY = gql`

query allPosts {

allPosts(orderBy: createdAt\_DESC) {

id

imageUrl

description

}

}

`;

We’re sorting the posts in descending order, so the latest posts appear on top of the list.

Now we’re replacing the current export statement with this:

export default graphql(ALL\_POSTS\_QUERY)(ListPage)

This injects a new prop called data (by default but [it can be renamed](https://www.apollographql.com/docs/react/api/react-apollo/#configname)) into ListPage. Back in the render method of ListPage, we can first check if the data has already been loaded with this.props.data.loading. If loading is set to false, the data has arrived and we can map over this.props.data.allPosts to display the posts. We're also passing the this.props.data.refetch method to every post to re-execute the query after a post has been deleted.

Putting it all together, this is the render method that we end up with:

render () {

if (this.props.data.loading) {

return (<div>Loading</div>)

}

return (

<div className='w-100 flex justify-center'>

<Link to='/create' className='fixed bg-white top-0 right-0 pa4 ttu dim black no-underline'>

+ New Post

</Link>

<div className='w-100' style={{ maxWidth: 400 }}>

{this.props.data.allPosts.map((post) =>

<Post key={post.id} post={post} refresh={() => this.props.data.refetch()} />

)}

</div>

</div>

)

}

#### **Creating Posts in CreatePage**

Adding mutations to React components is similar to adding queries, but instead of injected data, functions are injected for each mutation. Again, we need to import the Apollo related packages at the top of src/components/CreatePage.js:

import { graphql } from 'react-apollo';

import gql from 'graphql-tag';

The mutation to create a new post looks as follows:

mutation addPost($description: String!, $imageUrl: String!) {

createPost(description: $description, imageUrl: $imageUrl) {

description

imageUrl

}

}

Now, at the end of the file, outside of the CreatePage class, but before the export default withRouter(CreatePage) statement, we can add a new mutation with gql:

const CREATE\_POST\_MUTATION = gql`

mutation addPost($description: String!, $imageUrl: String!) {

createPost(description: $description, imageUrl: $imageUrl) {

id

description

imageUrl

}

}

`;

Similar to the ALL\_POSTS\_QUERY before, we now have to replace the export statement with the following code snippet:

const CreatePageWithMutation = *graphql*(CREATE\_POST\_MUTATION, {

props: ({mutate}) => ({

addPost: ({description, imageUrl}) =>

mutate({

variables: {description, imageUrl},

}),

})

})(CreatePage);

export default *withRouter*(CreatePageWithMutation);

This injects a new function to the props of CreatePage accessible with this.props.addPost. This function takes a description and the imageUrl as arguments so that we can provide the necessary info for each post that's being created. Using that, we can implement the handlePost method of the CreatePage class to create a post:

handlePost = () => {

const { description, imageUrl } = this.state

this.props.addPost({ description, imageUrl }).then(() => {

this.props.history.replace('/');

})

}

If you try creating a "+ New Post" you may notice a problem. The new post is not visible, at least not immediately. However, if you refresh the page, you will find that it is there. We need to add a componentWillReceiveProps

to the ListPage so that we do a refetch:

componentWillReceiveProps(nextProps) {

if (this.props.location.key !== nextProps.location.key) {

this.props.data.*refetch*();

}

}

Try it again! The User Experience should be much better.

#### **Deleting Posts in Post**

To delete a post, we again need to import the Apollo Client related packages at the top of src/components/Post.js:

import { graphql } from 'react-apollo'

import gql from 'graphql-tag'

Let’s take a look at the mutation that we can use to delete a post from a playground:

mutation deletePost($id: ID!) {

deletePost(id: $id) {

id

}

}

First let's update the handleDelete:

*handleDelete* = async () => {

await this.props.deletePostMutation({variables: {id: this.props.post.id}});

this.props.refresh();

}

Instead of using a Promise, this time we are using ES7's async/await, but the effect is exactly the same.

Now we can define that deletePost mutation in our JavaScript code and then inject it into our component by wrapping it with the graphql function. As before, we'll do that right before the export default Post statement:

const DELETE\_POST\_MUTATION = gql`

mutation deletePost($id: ID!) {

deletePost(id: $id) {

id

}

}

`;

const PostWithMutation = *graphql*(DELETE\_POST\_MUTATION,

{ name: 'deletePostMutation'})(Post);

export default PostWithMutation;

Notice that this time, instead of using the default prop name 'data', we are explicitly naming it 'deletePostMutation'.

That should give us a fully functioning application with create, read & delete functionality.

### **Conclusion**

That’s it! Using create-react-app and Apollo Client, it's easy to write React applications that work with a GraphQL backend. If you want to dive deeper in the example code, you can check it out on [GitHub](https://github.com/graphcool-examples/react-graphql/tree/master/quickstart-with-apollo).

With thanks and credit to [Nikolas Burk](https://www.prisma.io/blog/how-to-use-create-react-app-with-graphql-apollo-62e574617cff)

1. [↑](#footnote-ref-0)