CHAPTER 25

Consuming GraphQL

In this chapter—the last in this book—I show you the different ways that a GraphQL service can be consumed by a React application. I show you how to work directly with HTTP requests, how to integrate GraphQL with a data store, and how to use a dedicated GraphQL client.

Preparing for This Chapter

In this chapter, I continue using the `productapp` project from Chapter 24 and the GraphQL service it contains. To prepare for this chapter, the changes described in the following sections are required.

Adding Packages to the Project

Later in the chapter, I create components that receive GraphQL data directly, which requires additional packages. Open a new command prompt, navigate to the `productapp` folder, and run the commands shown in Listing 25-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-react-16`.

Listing 25-1. Adding Packages to the Example Project

```
npm install apollo-boost@0.1.22
npm install react-apollo@2.3.2
```

Table 25-1 describes the purpose of the new packages.

Table 25-1. *The Packages Added to the Project*

Name	Description
`apollo-boost`	This package contains the Apollo GraphQL client with a configuration that is suitable for most projects.
`react-apollo`	This package contains the React integration for the Apollo client.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_25

Changing the Data for the GraphQL Server

In Chapter 24, I used the same data as I had previously for the web service to highlight the different ways that REST and GraphQL approach the same problems. For this chapter, I want to make it obvious when the example application stops obtaining data using REST and starts using GraphQL. I created a file called `graphqlData.js` in the `productapp` folder with the contents shown in Listing 25-2.

Listing 25-2. The Contents of the graphqlData.js File in the productapp Folder

```
module.exports = function () {
    var data = {
        products: [
            { id: 1, name: "Trail Shoes", category: "Running", price: 120 },
            { id: 2, name: "Heated Gloves", category: "Running", price: 20.95 },
            { id: 3, name: "Padded Shorts", category: "Cycling", price: 19.50 },
            { id: 4, name: "Puncture Kit", category: "Cycling", price: 34.95 },
            { id: 5, name: "Mirror Goggles", category: "Swimming", price: 79500 },

        ],
        suppliers: [
            { id: 1, name: "Just Running", city: "Houston", products: [1, 2] },
            { id: 2, name: "Miles and Smiles", city: "Paris", products: [3, 4] },
            { id: 3, name: "Deep Dive", city: "New York", products: [5] },
        ]
    }
    return data
}
```

Updating the Schema and Resolvers

To prepare for this chapter, I need to extend the GraphQL schema to define mutations for deleting data, as shown in Listing 25-3. I have also removed the input type so that the `storeProduct` and `storeSupplier` mutations are consistent.

Listing 25-3. Defining and Updating Mutations in the schema.graphql File in the src/graphql Folder

```
type product {
    id: ID!,
    name: String!,
    category: String!
    price: Float!
}

type supplier {
    id: ID!,
    name: String!,
    city: String!,
    products(nameFilter: String = ""): [product]
}
```

```
type Query {
    products: [product],
    product(id: ID!): product,
    suppliers: [supplier]
    supplier(id: ID!): supplier
}

type Mutation {
    storeProduct(id: ID, name: String!, category: String!, price: Float!): product
    storeSupplier(id: ID, name: String!, city: String!, products: [Int]): supplier
    deleteProduct(id: ID!): ID
    deleteSupplier(id: ID!): ID
}
```

In Listing 25-4, I have defined new resolvers for the `deleteProduct` and `deleteSupplier` mutations and updated the `storeProduct` resolver to reflect the removal of the input type. I also changed the statement that loads the data to use the file created in Listing 25-2.

Listing 25-4. Adding and Updating Resolvers in the resolvers.js File in the src/graphql Folder

```
var data = require("../../graphqlData")();

const mapIdsToProducts = (supplier, nameFilter) =>
    supplier.products.map(id => data.products.find(p => p.id === Number(id)))
        .filter(p => p.name.toLowerCase().includes(nameFilter.toLowerCase()));

let nextId = 100;

module.exports = {

    products: () => data.products,

    product: ({id}) => data.products
        .find(p => p.id === parseInt(id)),

    suppliers: () => data.suppliers.map(s => ({
        ...s, products: ({nameFilter}) => mapIdsToProducts(s, nameFilter)
    })),

    supplier: ({id}) => {
        const result = data.suppliers.find(s => s.id === parseInt(id));
        if (result) {
            return {
                ...result,
                products: ({ nameFilter }) => mapIdsToProducts(result, nameFilter)
            }
        }
    },

    storeProduct(args) {
        const product = { ...args, id: Number(args.id)};
        if (args.id == null || product.id === 0) {
            product.id = nextId++;
            data.products.push(product);
        } else {
            data.products = data.products
                .map(p => p.id === product.id ? product : p);
        }
        return product;
    },

    storeSupplier(args) {
        const supp = { ...args, id: Number(args.id)};
        if (args.id == null) {
            supp.id = nextId++;
            data.suppliers.push(supp)
        } else {
            data.suppliers = data.suppliers.map(s => s.id === supp.id ? supp: s);
        }
        let result = data.suppliers.find(s => s.id === supp.id);
        if (result) {
            return {
                ...result,
                products: ({ nameFilter }) => mapIdsToProducts(result, nameFilter)
            }
        }
    },

    deleteProduct({id}) {
        id = Number(id);
        data.products = data.products.filter(p => p.id !== id);
        data.suppliers = data.suppliers.map(s => {
            s.products = s.products.filter(p => p !== id);
            return s;
        })
        return id;
    },

    deleteSupplier({id}) {
        data.suppliers = data.suppliers.filter(s => s.id !== Number(id));
        return id;
    }
}
```

The new resolvers remove an item from the data arrays and return the value of their id parameters, corresponding to the ID type used in the schema. When a product is removed, any reference to it from a supplier is also removed to avoid errors when subsequently querying for supplier data.

■ **Tip** I have also changed the `storeProduct` and `storeSupplier` functions so they will treat a request whose object has an `id` value of zero the same as if it contained no `id` value at all. This is a useful technique when dealing with form data because it means that all the form values can be sent to the server without needing to remove the `id` property to differentiate between new and modified objects.

Integrating the GraphQL Server with the Development Tools

In Chapter 24, I started the GraphQL tool directly, without using any of the React development tools. For this chapter, I am going to start the GraphQL server automatically, alongside the development HTTP server and the RESTful web service, which the example application is still configured to use. In Listing 25-5, I changed the `scripts` section of the `package.json` file so that the GraphQL server is started as part of the `npm start` command.

Listing 25-5. Configuring the Project Startup in the package.json File in the productapp Folder

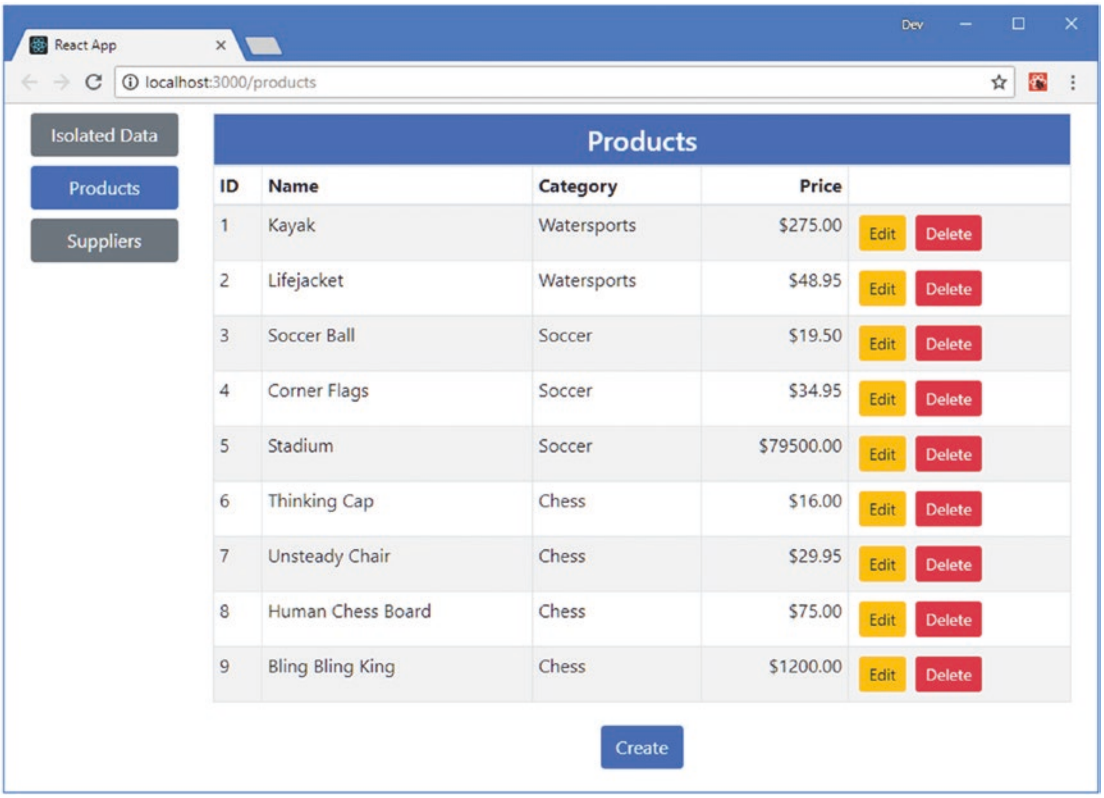
```
...
"scripts": {
  "start": "npm-run-all --parallel reactstart json graphql",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "reactstart": "react-scripts start",
  "json": "json-server --p 3500 -r api.routes.json restData.js",
  "graphql": "node graphqlServer.js"
},
...
```

To start the example application, open a new command prompt, navigate to the `productapp` folder, and run the command shown in Listing 25-6.

Listing 25-6. Running the Example Application

```
npm start
```

The development server, the RESTful web service, and the GraphQL server will all start. A new browser tab will open and display the content shown in Figure 25-1.



Products

ID	Name	Category	Price	
1	Kayak	Watersports	$275.00	Edit Delete
2	Lifejacket	Watersports	$48.95	Edit Delete
3	Soccer Ball	Soccer	$19.50	Edit Delete
4	Corner Flags	Soccer	$34.95	Edit Delete
5	Stadium	Soccer	$79500.00	Edit Delete
6	Thinking Cap	Chess	$16.00	Edit Delete
7	Unsteady Chair	Chess	$29.95	Edit Delete
8	Human Chess Board	Chess	$75.00	Edit Delete
9	Bling Bling King	Chess	$1200.00	Edit Delete

Figure 25-1. *Running the example application*

To ensure that the GraphQL server is running correctly, navigate to `http://localhost:3600/graphql` and enter the query shown in Listing 25-7 into GraphiQL.

Listing 25-7. Querying the GraphQL Server

```
query {
  product(id: 1) {
    id, name, category, price
  }
}
```

Click the Execute Query button, and you should see the following result:

```
...
{
  "data": {
    "product": {
      "id": "1",
      "name": "Trail Shoes",
      "category": "Running",
      "price": 120
    }
  }
}
...
```

Consuming a GraphQL Service

GraphQL queries are sent to the server using HTTP POST requests, with a JSON request body, like this:

```
...
{"query":"query { product(id: 1) { id, name, category, price }", "variables": null }
...
```

The response is a JSON string containing the results, like this:

```
...
{"data":{"product":{"id":"1","name":"Trail Shoes","category":"Running","price":120}}}
...
```

The use of HTTP and the structure of the request and response make it easy to integrate GraphQL into a React application, following the same pattern I used in Chapter 23 for working with a RESTful web service.

Defining the Queries and Mutations

The starting point when consuming GraphQL is to define the queries and mutations that will be sent to the server. I added a file called `queries.js` to the `src/graphql` folder and added the code shown in Listing 25-8.

Listing 25-8. The Contents of the queries.js File in the src/graphql Folder

```
export const products = {
    getAll: {
        name: "products",
        graphql: `query {
                    products { id, name, category, price}
                }`
    },
    getOne: {
        name: "product",
        graphql: `query ($id: ID!) {
                product(id: $id) {
                    id, name, category, price
                }
            }`
        }
    }

export const suppliers = {
    getAll: {
        name: "suppliers",
        graphql:`query {
              suppliers { id, name, city, products { id, name }}
            }`
    },
    getOne: {
        name: "supplier",
        graphql: `query($id: ID!) {
                supplier(id: $id) {
                    id, name, city, products { id, name }
                }
            }`
    }
}
```

Each query is defined with a GraphQL expression and a name, which the application will use to retrieve the data from the response. The queries rely on the variable feature described in Chapter 24. Next, I added a file called `mutations.js` in the `src/graphql` folder and defined the mutations that the application will need, as shown in Listing 25-9.

■ **Tip** Separating the queries from the mutations isn't required since they are all just strings, but I find it helpful, especially for applications that make heavy use of GraphQL.

Listing 25-9. The Contents of the mutations.js File in the src/graphql Folder

```
export const products = {
    store: {
        name: "storeProduct",
        graphql: `mutation ($id: ID, $name: String!,
                    $category: String!, $price: Float!) {

                    storeProduct(id : $id, name: $name,
                        category: $category, price: $price) {
                            id, name, category, price
                        }
                    }`
    },
    delete: {
        name: "deleteProduct",
        graphql: `mutation ($id: ID!) { deleteProduct(id: $id) }`
    }
}

export const suppliers = {
    store: {
        name: "storeSupplier",
        graphql: `mutation ($id: ID, $name: String!,
                    $city: String!, $products: [Int]) {

                    storeSupplier(id : $id, name: $name,
                        city: $city, products: $products) {
                            id, name, city, products { name }
                    }
                }`
    },
    delete: {
        name: "deleteSupplier",
        graphql: `mutation ($id: ID!) { deleteSupplier(id: $id) }`
    }
}
```

There are mutations for storing and deleting product and supplier objects, and the name of each mutation is used in the name property, following the same pattern established for the queries.

Defining the Data Source

I am going to use the same queries and mutations in different ways in this chapter, and I want to hide the details of how data is handled from the rest of the application while following the same broad pattern I used for working with a RESTful web service. To provide a data source that will use GraphQL to perform data operations, I added a file called `GraphQLDataSource.js` to the `src/graphql` folder and used it to define the class shown in Listing 25-10.

Listing 25-10. The Contents of the GraphQLDataSource.js File in the src/graphql Folder

```
import Axios from "axios";
import * as allQueries from "./queries";
import * as allMutations from "./mutations";

export class GraphQLDataSource {

    constructor(dataType, errorCallback) {
        this.GRAPHQL_URL = "http://localhost:3600/graphql";
        this.queries = allQueries[dataType];
        this.mutations = allMutations[dataType];
        this.handleError = errorCallback;
    }

    GetData(callback) {
        this.SendRequest(callback, this.queries.getAll);
    }

    GetOne(id, callback) {
        this.SendRequest(callback, this.queries.getOne, { id });
    }

    Store(data, callback) {
        this.SendRequest(callback, this.mutations.store, { ...data });
    }
    Update(data, callback) {
        this.Store(data, callback);
    }

    Delete(data, callback) {
        this.SendRequest(callback, this.mutations.delete, { id: data.id });
    }

    async SendRequest(callback, query, data) {
        try {
            let payload = {
                query: query.graphql,
                variables: data == null ? null : { ...data }
            }
            callback((await Axios.post(this.GRAPHQL_URL,
                payload)).data.data[query.name]);
        } catch(err) {
            this.handleError("Operation Failed: Network Error");
        }
    }
}
```

The class defines the same methods as the REST data source created in Chapter 23, which isn't a requirement but helps show how the different types of service differ. To configure the data source for a specific type of data, the constructor receives a data type string, which is used to select the queries and mutations. When making a request, the GraphQL is sent to the user, along with a `variables` object. The result includes the name of the query of mutation that was performed, which is retrieved from the response using the value of the `name` property.

```
...
callback((await Axios.post(this.GRAPHQL_URL, payload)).data.data[query.name]);
...
```

Configuring the Isolated Components

Using the same API as the REST data source from Chapter 23 has simplified the process of integrating the GraphQL data into the application by changing the data source in the components that consume data. In Listing 25-11, I have changed the data source used by the `IsolatedTable` component.

Listing 25-11. Changing the Data Source in the IsolatedTable.js File in the src Folder

```
import React, { Component } from "react";
//import { RestDataSource } from "./webservice/RestDataSource";
import { Link } from "react-router-dom";
import { GraphQLDataSource } from "./graphql/GraphQLDataSource";
import { PRODUCTS } from "./store/dataTypes";

export class IsolatedTable extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: []
        }
        this.dataSource = new GraphQLDataSource(PRODUCTS,
            (err) => this.props.history.push(`/error/${err}`));
    }

    // ...methods omitted for brevity...
}
```

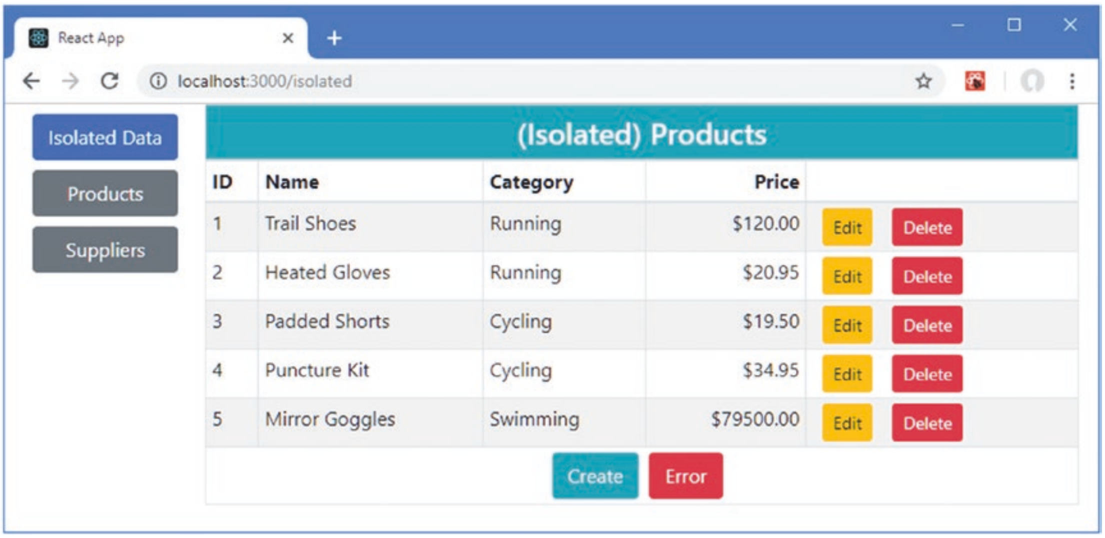
Listing 25-12 makes the corresponding change to the IsolatedEditor component.

Listing 25-12. Changing the Data Source in the IsolatedEditor.js File in the src Folder

```
import React, { Component } from "react";
//import { RestDataSource } from "./webservice/RestDataSource";
import { ProductEditor } from "./ProductEditor";
import { GraphQLDataSource } from "./graphql/GraphQLDataSource";
import { PRODUCTS } from "./store/dataTypes";

export class IsolatedEditor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            dataItem: {}
        };
        this.dataSource = new GraphQLDataSource(PRODUCTS,
            (err) => this.props.history.push(`/error/${err}`));
    }

    save = (data) => {
        data = { ...data, price: Number(data.price)}
        const callback = () => this.props.history.push("/isolated");
        if (data.id === "") {
            this.dataSource.Store(data, callback);
        } else {
            this.dataSource.Update(data, callback);
        }
    }

    cancel = () => this.props.history.push("/isolated");

    render() {
        return <ProductEditor key={ this.state.dataItem.id }
            product={ this.state.dataItem } saveCallback={ this.save }
            cancelCallback={ this.cancel } />
    }

    componentDidMount() {
        if (this.props.match.params.mode === "edit") {
            this.dataSource.GetOne(this.props.match.params.id,
                data => this.setState({ dataItem: data}));
        }
    }
}
```

Notice that I parse the value of the `price` property into a `Number` before the data is sent to the server. The GraphQL server checks the data it receives against the types defined in the schema and will reject string values, which are what form elements typically produce, if another type, such as `Float`, is required.

When you save the changes to the `IsolatedTable` and `IsolatedEditor` components, the application will update, and clicking the Isolated Data button will show the data obtained from the GraphQL server, as shown in Figure 25-2.



ID	Name	Category	Price	
1	Trail Shoes	Running	$120.00	Edit Delete
2	Heated Gloves	Running	$20.95	Edit Delete
3	Padded Shorts	Cycling	$19.50	Edit Delete
4	Puncture Kit	Cycling	$34.95	Edit Delete
5	Mirror Goggles	Swimming	$79500.00	Edit Delete

Figure 25-2. *Using GraphQL data*

Using GraphQL with a Data Store

The process for using the GraphQL with a data store is similar to the process I followed in Chapter 23 for RESTful data, using middleware to intercept actions and trigger requests to the server. I added a file called `GraphQLMiddleware.js` to the `src/graphql` folder and used it to define the Redux middleware shown in Listing 25-13.

Listing 25-13. The Contents of the GraphQLMiddleware.js File in the src/graphql Folder

```
import { STORE, UPDATE, DELETE} from "../store/modelActionTypes";
import { PRODUCTS, SUPPLIERS } from "../store/dataTypes";
import { GraphQLDataSource } from "./GraphQLDataSource";

export const GET_DATA = "graphql_get_data";
export const getData = (dataType) => {
    return {
        type: GET_DATA,
        dataType: dataType
    }
}

export const createGraphQLMiddleware = () => {

    const dataSources = {
        [PRODUCTS]: new GraphQLDataSource(PRODUCTS, () => {}),
        [SUPPLIERS]: new GraphQLDataSource(SUPPLIERS, () => {})
    }

    return ({dispatch, getState}) => next => action => {
        switch (action.type) {
            case GET_DATA:
                if (getState().modelData[action.dataType].length === 0) {
                    dataSources[action.dataType].GetData((data) =>
                        data.forEach(item => next({ type: STORE,
                            dataType: action.dataType, payload: item})));
                }
                break;
            case STORE:
                action.payload.id = null;
                dataSources[action.dataType].Store(action.payload, data =>
                    next({ ...action, payload: data }))
                break;
            case UPDATE:
                dataSources[action.dataType].Update(action.payload, data =>
                     next({ ...action, payload: data }))
                break;
            case DELETE:
                dataSources[action.dataType].Delete({id: action.payload },
                    () => next(action));
                break;
            default:
                next(action);
        }
    }
}
```

This middleware intercepts the actions that are dispatched by the rest of the application and uses the data source class to send a query or mutation to the GraphQL server. In Listing 25-14, I have replaced the REST middleware with the GraphQL code.

Listing 25-14. Enabling the GraphQL Middleware in the index.js File in the src/store Folder

```
import { createStore, combineReducers, applyMiddleware, compose } from "redux";
import modelReducer from "./modelReducer";
import stateReducer from "./stateReducer";
import { customReducerEnhancer } from "./customReducerEnhancer";
import { multiActions } from "./multiActionMiddleware";
import { asyncEnhancer } from "./asyncEnhancer";
//import { createRestMiddleware } from "../webservice/RestMiddleware";
import { createGraphQLMiddleware } from "../graphql/GraphQLMiddleware";

const enhancedReducer = customReducerEnhancer(
    combineReducers(
        {
            modelData: modelReducer,
            stateData: stateReducer
        })
);

// const restMiddleware = createRestMiddleware(
//     "http://localhost:3500/api/products",
//     "http://localhost:3500/api/suppliers");

export default createStore(enhancedReducer,
    compose(applyMiddleware(multiActions),
        applyMiddleware(createGraphQLMiddleware()),
        asyncEnhancer(2000)));

export { saveProduct, saveSupplier, deleteProduct, deleteSupplier }
    from "./modelActionCreators";
```

Adjusting to the GraphQL Data Format

The data format returned by the GraphQL queries for the supplier data includes the related product data, which means that a separate request for the product data is no longer required and that the components that display the related data must be adapted to the new format. In Listing 25-15, I disabled the automatic query for product data when the application requires supplier data.

Listing 25-15. Disabling the Related Data Query in the DataGetter.js File in the src Folder

```
import React, { Component } from "react";
//import { PRODUCTS, SUPPLIERS } from "./store/dataTypes";

export const DataGetter = (dataType, WrappedComponent) => {

    return class extends Component {
        render() {
            return <WrappedComponent { ...this.props } />
        }

        componentDidMount() {
            // this.props.getData(PRODUCTS);
            // if (dataType === SUPPLIERS) {
            //     this.props.getData(SUPPLIERS);
            // }
            this.props.getData(dataType);
        }
    }
}
```

In Listing 25-16, I have commented out the code in the TableConnector component that processed the supplier data to incorporate the names of the related products. This information will be directly available to the components now that the data is coming from the GraphQL server. TableConnector also triggers the data request.

Listing 25-16. Disabling Data Processing in the TableConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
//import { startEditingProduct, startEditingSupplier } from "./stateActions";
import { deleteProduct, deleteSupplier } from "./modelActionCreators";
import { PRODUCTS, SUPPLIERS } from "./dataTypes";
import { withRouter } from "react-router-dom";
//import { getData } from "../webservice/RestMiddleware";
import { getData } from "../graphql/GraphQLMiddleware";
import { DataGetter } from "../DataGetter";

export const TableConnector = (dataType, presentationComponent) => {

    const mapStateToProps = (storeData, ownProps) => {
        if (dataType === PRODUCTS) {
            return { products: storeData.modelData[PRODUCTS] };
        } else {
            return { suppliers: storeData.modelData[SUPPLIERS] };
                // suppliers: storeData.modelData[SUPPLIERS].map(supp => ({
                //     ...supp,
                //     products: supp.products.map(id =>
                //         storeData.modelData[PRODUCTS]
                //             .find(p => p.id === Number(id)) || id)
                //             .map(val => val.name || val)
                //     }))
        }
    }

    const mapDispatchToProps = (dispatch, ownProps) => {
        return {
            getData: (type) => dispatch(getData(type)),
            deleteCallback: dataType === PRODUCTS
                ? (...args) => dispatch(deleteProduct(...args))
                : (...args) => dispatch(deleteSupplier(...args))
        }
    }

    const mergeProps = (dataProps, functionProps, ownProps) => {
        let routedDispatchers = {
            editCallback: (target) => {
                ownProps.history.push(`/${dataType}/edit/${target.id}`);
            },
            deleteCallback: functionProps.deleteCallback,
            getData: functionProps.getData

        }
        return Object.assign({}, dataProps, routedDispatchers, ownProps);
    }

    return withRouter(connect(mapStateToProps,
        mapDispatchToProps, mergeProps)(DataGetter(dataType,
            presentationComponent)));
}
```

To locate objects by ID, I changed the `EditorConnector` so that it doesn't parse the URL parameter to a `Number`, as shown in Listing 25-17.

Listing 25-17. Changing ID Matching in the EditorConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
//import { endEditing } from "./stateActions";
import { PRODUCTS, SUPPLIERS  } from "./dataTypes";
import { saveAndEndEditing } from "./multiActionCreators";
import { withRouter } from "react-router-dom";

export const EditorConnector = (dataType, presentationComponent) => {

    const mapStateToProps = (storeData, ownProps) => {
        const mode = ownProps.match.params.mode;
        const id = ownProps.match.params.id;
        return {
            editing: mode === "edit" || mode === "create",
            product: (storeData.modelData[PRODUCTS].find(p => p.id === id)) || {},
            supplier:(storeData.modelData[SUPPLIERS].find(s => s.id === id)) || {}
        }
    }

    const mapDispatchToProps = {
        //cancelCallback: endEditing,
        saveCallback: (data) => saveAndEndEditing(data, dataType)
    }

    const mergeProps = (dataProps, functionProps, ownProps) => {
        let routedDispatchers = {
            cancelCallback: () => ownProps.history.push(`/${dataType}`),
            saveCallback: (data) => {
                functionProps.saveCallback(data);
                ownProps.history.push(`/${dataType}`);
            }
        }
        return Object.assign({}, dataProps, routedDispatchers, ownProps);
    }
    return withRouter(connect(mapStateToProps,
        mapDispatchToProps, mergeProps)(presentationComponent));
}
```

To display the names of the products, I made the change shown in Listing 25-18 to the SupplierTableRow component.

Listing 25-18. Selecting Product Names in the SupplierTableRow.js File in the src Folder

```
import React, { Component } from "react";

export class SupplierTableRow extends Component {

    render() {
        let s = this.props.supplier;
        return <tr>
            <td>{ s.id }</td>
            <td>{ s.name }</td>
            <td>{ s.city}</td>
            <td>{ s.products != null ?
                    s.products.map(p => p.name).join(", ") : "" }</td>
            <td>
                <button className="btn btn-sm btn-warning m-1"
                    onClick={ () => this.props.editCallback(s) }>
                        Edit
                </button>
                <button className="btn btn-sm btn-danger m-1"
                    onClick={ () => this.props.deleteCallback(s) }>
                        Delete
                    </button>
            </td>
        </tr>
    }
}
```

The next change is to accommodate the new data format when editing supplier data, ensuring that the id values of the related products area displayed to the user, as shown in Listing 25-19.

Listing 25-19. Selecting Product IDs in the SupplierEditor.js File in the src Folder

```
import React, { Component } from "react";

export class SupplierEditor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            formData: {
                id: props.supplier.id || "",
                name: props.supplier.name || "",
                city: props.supplier.city || "",
                products: props.supplier.products != null
```

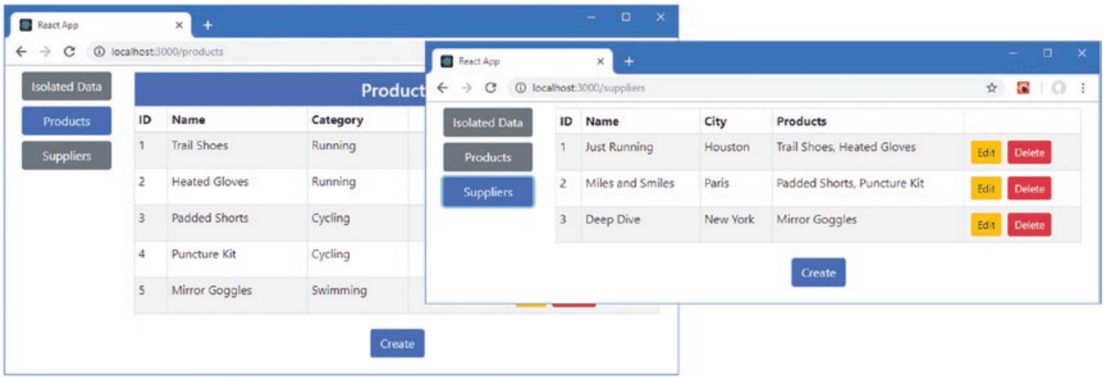
```
                    ? props.supplier.products.map(p => p.id) : [],
            }
        }
    }

    // ...methods omitted for brevity...
}
```

The final change is to parse the string value obtained from the form element for the product `price` property into a number, which ensures that the data sent to the server matches the `Float` type specified in the schema, as shown in Listing 25-20.

Listing 25-20. Parsing the Price Value in the ProductEditor.js File in the src Folder

```
...
handleClick = () => {
    this.props.saveCallback(
        {
            ...this.state.formData,
            price: Number(this.state.formData.price)
        });
}
...
```

Save all the changes, and the application will work entirely with data obtained from the GraphQL server and placed in the Redux data store, as shown in Figure 25-3.



ID	Name	Category
1	Trail Shoes	Running
2	Heated Gloves	Running
3	Padded Shorts	Cycling
4	Puncture Kit	Cycling
5	Mirror Goggles	Swimming

ID	Name	City	Products
1	Just Running	Houston	Trail Shoes, Heated Gloves
2	Miles and Smiles	Paris	Padded Shorts, Puncture Kit
3	Deep Dive	New York	Mirror Goggles

Figure 25-3. *Using GraphQL data in the data store*

■ **Tip** If you encounter errors, stop the development tools and start them again using `npm start`. This will reset the data used by the GraphQL server and undo the effects of changes made in previous sections.

Using a GraphQL Client Framework

I demonstrated how GraphQL can be used with a Redux data store in the previous section because it shows how easily the data can be used and provides a comparison against working with a RESTful web service.

There is a different approach, which is to use a package that replaces the data store and provides GraphQL data directly to the components that need it while caching data to avoid the kind of repeated HTTP requests that navigation between isolated components can lead to.

The package that I use in this section is called Apollo Client, which is the client-side package from the same developers as the Apollo GraphQL server that I mentioned in Chapter 24 (but that works with any GraphQL server). Full documentation for the Apollo Client is available at `https://www.apollographql.com/docs/react`).

■ **Note** The examples in this chapter rely on the packages installed at the start of the chapter.

Configuring the Client

The first step is to configure the client so that it knows where to send the GraphQL requests and mutations. Listing 25-21 shows the configuration statements I added to the App component.

Listing 25-21. Configuring Apollo Client in the App.js File in the src Folder

```
import React, { Component } from "react";
// import { Provider } from "react-redux";
// import dataStore from "./store";
import { Selector } from "./Selector";
//import { PRODUCTS, SUPPLIERS } from "./store/dataTypes";
import ApolloClient from "apollo-boost";
import { ApolloProvider } from "react-apollo";

const client = new ApolloClient({
    uri: "http://localhost:3600/graphql"
});

export default class App extends Component {

    render() {
        return  <ApolloProvider client={ client }>
                    <Selector />
                </ApolloProvider>
    }
}
```

A new `ApolloClient` object is created to manage the relationship to the GraphQL server, and its constructor accepts a configuration object. The `uri` property of the configuration object specifies the URL for GraphQL requests. There are other configuration options, but the defaults are suitable for most projects (see `www.apollographql.com` for details).

The `ApolloProvider` component is used to integrate GraphQL features with React, and the `client` prop is assigned the `ApolloClient` object.

To simplify the example, I have removed the content contained within the `Selector` component and the data store. I will define the components displayed by the selector directly shortly.

■ **Tip** I have removed the Redux data store from the application only to simplify the example application. Apollo Client and Redux can be used in the same application, although you should be careful not to store data in Redux that is being managed by Apollo Client because the two will easily get out of sync.

Creating a GraphQL Component

The next step is to create components that will act as a bridge between GraphQL and the content displayed to the user. This is the same basic approach that I have been using throughout this part of the book, and it means that the content rendered to the user is produced by components that do not depend directly on a specific mechanism for data, whether it be the Redux data store, a RESTful web service, or a GraphQL client. I added a file called `GraphQLTable.js` to the `src/graphql` folder and added the code shown in Listing 25-22.

CHOOSING AN ALTERNATIVE GRAPHQL CLIENT

I picked Apollo Client because it is flexible and easy to get on with. The main alternative is Relay (`https://facebook.github.io/relay`), which is developed by Facebook. Relay is more difficult to get started with and only works with GraphQL schemas that follow a particular structure.

There are also packages that offer a subset of the features provided by Apollo Client or Relay, such as Adrenaline (`https://github.com/gyzerok/adrenaline`).

Listing 25-22. The Contents of the GraphQLTable.js File in the src/graphql Folder

```
import React, { Component } from "react";
import { Query } from "react-apollo";
import gql from "graphql-tag";
import * as queries from "./queries";
import { ProductTable } from "../ProductTable";

export const GraphQLTable = () => {

    const getAll = gql(queries.products.getAll.graphql);

    return class extends Component {

        constructor(props) {
            super(props);
            this.editCallback = (item) => this.props.history
                .push(`/products/edit/${item.id}`);
        }
        render() {
            return <Query query={ getAll }>
                {({loading, data, refetch }) => {
                    if (loading) {
                        return <h5
                            className="bg-info text-white text-center m-2 p-2">
                                Loading...
                        </h5>
                    } else {
                        return <React.Fragment>
                            <ProductTable products={data.products}
                                editCallback= { this.editCallback }
                                deleteCallback={ () => {} } />
                            <div className="text-center">
                                <button className="btn btn-primary"
                                        onClick={ () => refetch() }>
                                    Reload Data
                                </button>
                            </div>
                        </React.Fragment>
                    }
                }}
            </Query>
        }
    }
}
```

There is a lot going on in Listing 25-22, and it is worth breaking down the component in detail to understand each part. Like the other packages used in this part of the book, Apollo Client relies on higher-order components to provide features. In this case, the GraphQLTable provides features to the example application's ProductTable component. I start by setting up the GraphQL query that will obtain the data, like this:

```
...
const getAll = gql(queries.products.getAll.graphql);
...
```

The gql function accepts a query expressed as a string and processes it so that it can be used by Apollo Client. I already defined the queries required by the application and organized them by data type. The gql function can also be used directly with template strings, which allows queries to be defined like this:

```
...
const getAll = gql`query { products {
    id, name, category, price
}}`
...
```

The Query component provides a component with access to the data returned by a query, which is specified by the query prop, like this:

```
...
return <Query query={ getAll }>
...
```

The query is sent to the server as soon as the Query component is rendered, and it uses a render prop function to provide its features through an object that defines properties describing the outcome of the query, the most useful of which are described in Table 25-2.

Table 25-2. *Useful Apollo Client Render Prop Object Properties*

Name	Description
data	This property returns the data produced by the query.
loading	This property returns true when the query is being processed.
error	This property returns details of any query errors.
variables	This property returns the variables used for the query.
refetch	This property returns a function that can be used to resend the query, optionally with new variables.

For the query in Listing 25-22, I use the loading, data, and refetch properties.

```
...
{({loading, error, refetch}) => {
...
```

When the component is first rendered and the request is sent to the GraphQL server, the loading value is true. When the request has completed, the component updates, with the loading value as false, and provides the result through the data property.

The query is executed when the Query component is rendered, but the results are cached, which means that no query is sent to the server the next time the data is required. The refetch property provides a function that sends the query again when it is invoked, refreshes the data in case, and updates the component. The function assigned to the refresh property accepts an object that can be used to provide new variables for the query. This is a useful feature, but it means that you must make sure not to invoke the function directly when using an event handler, like this:

```
...
<button className="btn btn-primary" onClick={ () => refetch() }>
...
```

If you don't specify an inline function, as shown, then the event object will be passed to the refetch function, which will attempt to use it as the source of variables for the query and encounter an error.

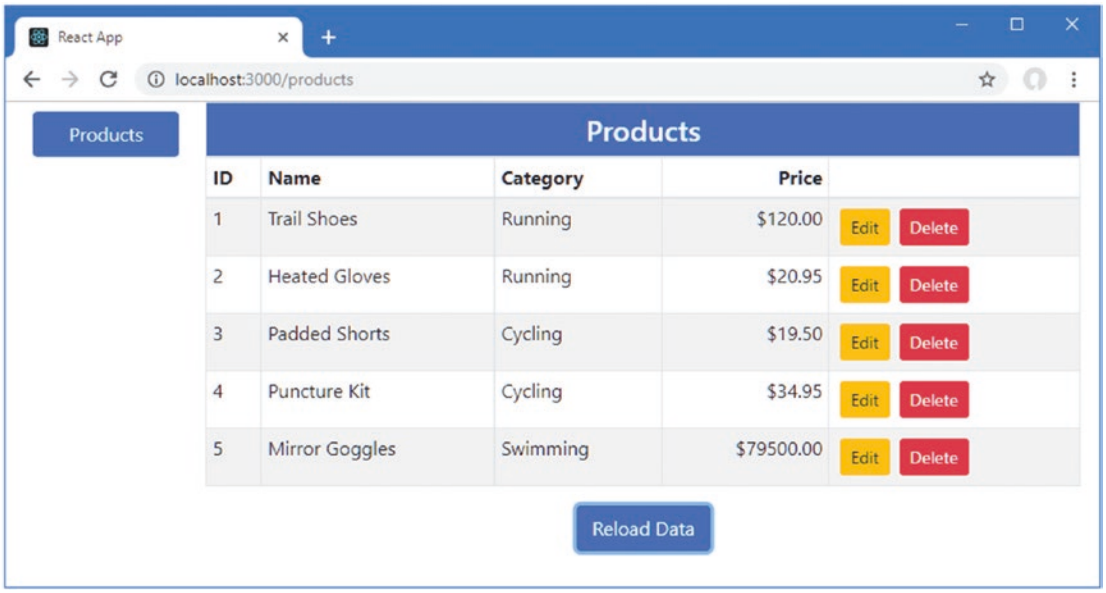
Applying the GraphQL Component

In Listing 25-23, I have replaced the routing components used by the `Selector` component so that the `GraphQLTable` component is used to respond to the `/product` URL.

Listing 25-23. Changing the Routing Configuration in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ToggleLink } from "./routing/ToggleLink";
// import { RoutedDisplay } from "./routing/RoutedDisplay";
// import { IsolatedTable } from "./IsolatedTable";
// import { IsolatedEditor } from "./IsolatedEditor";
// import { RequestError } from "./webservice/RequestError";
import { GraphQLTable } from "./graphql/GraphQLTable";

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/products">Products</ToggleLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" exact={true}
                                component={ GraphQLTable()}  />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

Save the changes, and you will see the GraphQL product and suppliers displayed, as shown in Figure 25-4.



Products

ID	Name	Category	Price	
1	Trail Shoes	Running	$120.00	Edit Delete
2	Heated Gloves	Running	$20.95	Edit Delete
3	Padded Shorts	Cycling	$19.50	Edit Delete
4	Puncture Kit	Cycling	$34.95	Edit Delete
5	Mirror Goggles	Swimming	$79500.00	Edit Delete

Figure 25-4. *Using a GraphQL client package*

■ **Tip** If you encounter errors, stop the development tools and start them again using `npm start`. This will reset the data used by the GraphQL server and undo the effects of changes made in previous sections.

If you use the F12 Developer Tools to see the network requests that the browser makes, you will see that queries for data are sent the first time that the routing buttons are clicked but not when the same table is selected again. Click the Reload Data button to invoke the query's refresh function and trigger a new query.

Using Mutations

The `Mutation` component is used to provide access to GraphQL mutations. In Listing 25-24, I have used `Mutation` to provide access to the mutations that will delete product or supplier objects.

Listing 25-24. Using a Mutation in the GraphQLTable.js File in the src/graphql Folder

```
import React, { Component } from "react";
import { Query, Mutation } from "react-apollo";
import gql from "graphql-tag";
import * as queries from "./queries";
import { ProductTable } from "../ProductTable";
import * as mutations from "./mutations";
```

```
export const GraphQLTable = () => {

    const getAll = gql(queries.products.getAll.graphql);
    const deleteItem = gql(mutations.products.delete.graphql);

    return class extends Component {

        constructor(props) {
            super(props);
            this.editCallback = (item) => this.props.history
                .push(`/products/edit/${item.id}`);
        }

        render() {
            return <Query query={ getAll }>
                {({loading, data, refetch }) => {
                    if (loading) {
                        return <h5
                            className="bg-info text-white text-center m-2 p-2">
                                Loading...
                        </h5>
                    } else {
                        return <Mutation mutation={ deleteItem }
                                refetchQueries={ () => [{query: getAll}]  }>
                            { doDelete =>
                                 <React.Fragment>
                                    <ProductTable products={data.products}
                                        editCallback= { this.editCallback }
                                        deleteCallback={ (p) =>
                                            doDelete({variables: {id: p.id} }) }  />
                                    <div className="text-center">
                                        <button className="btn btn-primary"
                                                onClick={ () => refetch() }>
                                            Reload Data
                                        </button>
                                    </div>
                                </React.Fragment>
                        }
                        </Mutation>
                    }
                }}
            </Query>
        }
    }
}
```

The Mutation component follows a similar pattern to the Query component and relies on a render prop function to provide access to a mutation. The Mutation component is configured using props, the most useful of which are described in Table 25-3.

Table 25-3. *Useful Mutation Props*

Name	Description
`mutation`	This prop specifies the mutation that will be sent to the server.
`variables`	This prop specifies the variables for the mutation. Variables can also be provided when the mutation is performed.
`refetchQueries`	This prop specifies one or more queries to be performed when the mutation has been completed.
`update`	This prop specifies a function that is used to update the cache when the mutation has been completed.
`onCompleted`	This prop specifies a callback function that is invoked when the mutation has completed.

I started by passing the mutation to the `gql` function so that it can be used as the value for the mutation prop on the `Mutation` component, like this:

```
...
return <Mutation mutation={ deleteMutation }
    refetchQueries={ () => [{ query: getAll}]}>
...
```

The cached data at the client will often become out-of-date once a mutation is performed, and the `Mutation` component provides two props that can be used to keep the cache in sync. The `refetchQueries` prop is assigned a function that receives the mutation result and returns an array of objects, each of which has `query` and, optionally, `variables` properties. When the mutation has completed, the queries are sent to the server, and the results are used to update the cache. This is the approach I have taken in Listing 25-24, where I have configured the `Mutation` to update the cache with the results of the `getAll` query:

```
...
return <Mutation mutation={ deleteMutation }
    refetchQueries={ () => [{ query: getAll}]}>
...
```

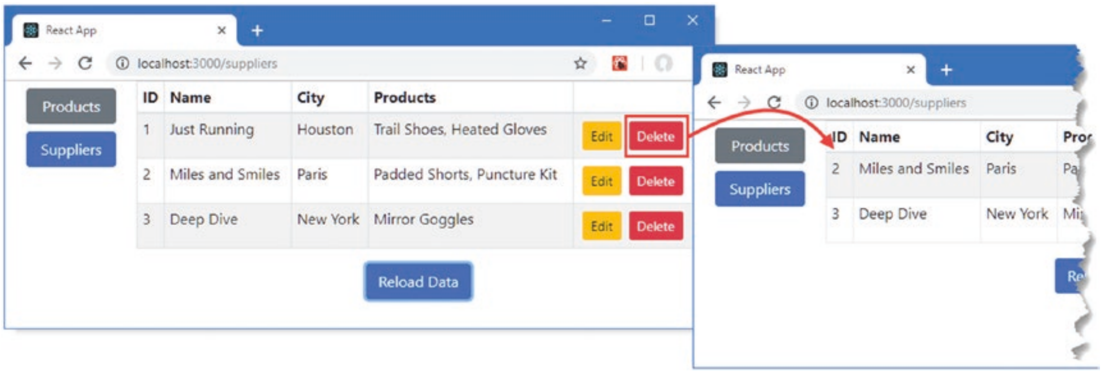
The mutation is provided as an argument to the render prop function.

```
...
return <Mutation mutation={ deleteItem }
    refetchQueries={ () => [{query: getAll}]  }>
        { doDelete => {
...
```

Variables for the mutation can be supplied as a prop on the `Mutation` component or as an argument to the function. I used a function in Listing 25-24 because I receive the object that is to be deleted through a callback.

```
...
deleteCallback={ (p) => doDelete({variables: {id: p.id} }) }
...
```

The effect is that clicking a Delete button invokes the mutation and then sends queries the server for the updated data, which is displayed to the user, as shown in Figure 25-5.



ID	Name	City	Products
1	Just Running	Houston	Trail Shoes, Heated Gloves
2	Miles and Smiles	Paris	Padded Shorts, Puncture Kit
3	Deep Dive	New York	Mirror Goggles

ID	Name	City	Pro
2	Miles and Smiles	Paris	Pa
3	Deep Dive	New York	Mi

Figure 25-5. *Using a mutation*

Updating Cached Data Without a Query

Re-querying the server after a mutation is useful when the application doesn't know what the impact of the changes will be. For simpler operations, querying for new data is excessive because the application knows exactly what the effect of the mutation will be and can apply this change directly to the cached data. In Listing 25-25, I have changed the configuration of the `Mutation` so that it no longer performs a query once an item has been deleted and uses a function to update the data cache instead.

Listing 25-25. Updating Cached Data in the GraphQLTable.js File in the src/graphql Folder

```
import React, { Component } from "react";
import { Query, Mutation } from "react-apollo";
import gql from "graphql-tag";
import * as queries from "./queries";
import { ProductTable } from "../ProductTable";
import * as mutations from "./mutations";

export const GraphQLTable = () => {

    const getAll = gql(queries.products.getAll.graphql);
    const deleteItem = gql(mutations.products.delete.graphql);

    return class extends Component {

        constructor(props) {
            super(props);
            this.editCallback = (item) => this.props.history
                .push(`/products/edit/${item.id}`);
        }
        removeItemFromCache(cache, mutationResult) {
            const deletedId =  mutationResult.data[mutations.products.delete.name];
            const data =
                cache.readQuery({ query: getAll })[queries.products.getAll.name];
            cache.writeQuery({
                query: getAll,
                data: { products: data.filter(item => item.id !== deletedId) }
            });
        }

        render() {
            return <Query query={ getAll }>
                {({loading, data, refetch }) => {
                    if (loading) {
                        return <h5
                            className="bg-info text-white text-center m-2 p-2">
                                Loading...
                        </h5>
                    } else {
                        return <Mutation mutation={ deleteItem }
                                update={ this.removeItemFromCache }>
                            { doDelete =>
                                 <React.Fragment>
                                    <ProductTable products={data.products}
                                        editCallback= { this.editCallback }
                                        deleteCallback={ (p) =>
                                            doDelete({variables: {id: p.id} }) }  />
                                    <div className="text-center">
                                        <button className="btn btn-primary"
                                                onClick={ () => refetch() }>
                                            Reload Data
                                        </button>
                                    </div>
                                </React.Fragment>
                        }
                        </Mutation>
                    }
                }}
            </Query>
        }
    }
}
```

The update prop on a Mutation is used to specify a method that will be invoked when a mutation has completed. The method receives the Apollo Client cache and the results from the mutation and is responsible for updating the cached data using the methods described in Table 25-4.

Table 25-4. *The Apollo Client Cache Methods*

Name	Description
`readQuery`	This method is used to read data from the cache associated with a specific query. An error will be thrown if you try to read data that is not in the cache, which typically occurs if a query has not yet been executed.
`writeQuery`	This method is used to update the data in the cache associated with a specific query.

The `removeItemFromCache` method in Listing 25-25 uses the `readQuery` method to retrieve the cached data associated with the `products` query, filters out the deleted item, and writes the remaining items back to the cache using the `writeQuery` method. The `readQuery` method accepts an object with `query` and an optional `variables` property, and the write query method accepts an object with query, data, and an optional `variables` property. The result is that when you click the Delete button for a product, the object is deleted from the local cache after the mutation has been completed without the need for an additional query.

Adding Support for Supplier Data and Editing

Now that the basic GraphQL client features are in place, I am going to adapt the `GraphQLTable` so that it supports product and supplier data and introduce support for editing data. In Listing 25-26, I have changed `GrpahQLTable` so that it receives the type of data it is working with as a parameter and selects the queries, mutations, and component to display to the user dynamically.

Listing 25-26. Supporting Multiple Data Types in the GraphQLTable.js File in the src/graphql Folder

```
import React, { Component } from "react";
import { Query, Mutation } from "react-apollo";
import gql from "graphql-tag";
import * as queries from "./queries";
import { ProductTable } from "../ProductTable";
import * as mutations from "./mutations";
import { PRODUCTS, SUPPLIERS } from "../store/dataTypes";
import { SupplierTable } from "../SupplierTable";

export const GraphQLTable = (dataType) => {

    const getAll = gql(queries[dataType].getAll.graphql);
    const deleteItem = gql(mutations[dataType].delete.graphql);

    return class extends Component {

        constructor(props) {
            super(props);
            this.editCallback = (item) => this.props.history
                .push(`/${dataType}/edit/${item.id}`);
        }
        removeItemFromCache = (cache, mutationResult) => {

            const deletedId =  mutationResult.data[mutations[dataType].delete.name];
            const data =
                cache.readQuery({ query: getAll })[queries[dataType].getAll.name];
            cache.writeQuery({
                query: getAll,
                data: { [dataType]: data.filter(item => item.id !== deletedId) }
            });
        }

        getRefetchQueries() {
            return dataType === PRODUCTS
                ? [{query: gql(queries[SUPPLIERS].getAll.graphql)}] : []
        }

        render() {
            return <Query query={ getAll }>
                {({loading, data, refetch }) => {
                    if (loading) {
                        return <h5
                            className="bg-info text-white text-center m-2 p-2">
                                Loading...
                        </h5>
                    } else {
                        return <Mutation mutation={ deleteItem }
                                update={ this.removeItemFromCache }
                                refetchQueries={ this.getRefetchQueries }>
                            { doDelete =>
                                 <React.Fragment>
                                     { dataType === PRODUCTS &&
                                        <ProductTable products={data.products}
                                            editCallback= { this.editCallback }
                                            deleteCallback={ (p) =>
                                                doDelete({variables: {id: p.id} }) }
                                        />
                                    }
                                    { dataType === SUPPLIERS &&
                                        <SupplierTable suppliers={data.suppliers}
                                            editCallback= { this.editCallback }
                                            deleteCallback={ (p) =>
                                                doDelete({variables: {id: p.id} }) }
                                        />
                                    }
                                    <div className="text-center">
                                        <button className="btn btn-primary"
                                                onClick={ () => refetch() }>
                                            Reload Data
                                        </button>
                                    </div>
                                </React.Fragment>
                        }
                        </Mutation>
                    }
                }}
            </Query>
        }
    }
}
```

Notice that this example combines the `update` and `refetchQueries` props on the `Mutation`. I need to keep the supplier data consistent when a product is deleted, but using the `readQuery` method for data not in the cache produces an error. To keep the example simple—and not to duplicate too much logic from the GraphQL server's resolvers—I use the `update` prop to perform a simple excision from the cache and the `refetchQueries` prop to get fresh `suppliers` data.

Creating the Editor Component

To allow the user to edit objects, I added a file called `GraphQLEditor.js` in the `src/graphql` folder and used it to define the component shown in Listing 25-27.

Listing 25-27. The Contents of the GraphQLEditor.js File in the src/graphql Folder

```
import React, { Component } from "react";
import gql from "graphql-tag";
import * as queries from "./queries";
import * as mutations from "./mutations";
import { Query, Mutation } from "react-apollo";
import { PRODUCTS } from "../store/dataTypes";
import { ProductEditor } from "../ProductEditor";
import { SupplierEditor } from "../SupplierEditor";

export const GraphQLEditor = () => {

    return class extends Component {

        constructor(props) {
            super(props);
            this.dataType = this.props.match.params.dataType;
            this.id = this.props.match.params.id;
            this.query = gql(queries[this.dataType].getOne.graphql);
            this.variables = { id: this.id };
            this.mutation = gql(mutations[this.dataType].store.graphql);
            this.navigation = () => props.history.push(`/${this.dataType}`);
        }

        render() {
            return <Query query={ this.query} variables={ this.variables }>
            {
                ({loading, data}) => {
                    if (!loading) {
                        return <Mutation mutation={ this.mutation }
                                onCompleted={ this.navigation }>
```

```
                            { (store) => {
                                if (this.dataType === PRODUCTS) {
                                    return <ProductEditor key={ this.id }
                                        product={ data.product }
                                        saveCallback={ (formData) =>
                                            store({variables: formData})}
                                        cancelCallback={ this.navigation } />
                                } else {
                                    return <SupplierEditor key={ this.id }
                                        supplier={ data.supplier }
                                        saveCallback={ (formData =>
                                            store({ variables: formData }))}
                                        cancelCallback={ this.navigation } />
                                }
                            }
                        }
                        </Mutation>
                    } else {
                        return null;
                    }
                }
            }
            </Query>
        }
    }
}
```

This component builds on the same features used for the table. A Query is used to request the data from the GraphQL server, with the variables prop being used to provide the variables required by the query. A Mutation is used to store the data, with the onCompleted prop used to navigate away from the editor once the mutation has completed.

Notice that I have not updated the cached data. Apollo Client has a clever feature where mutations that alter a single object and have responses with an id property and properties that have been changed are automatically pushed into the cache and used by other Query objects. In the case of the editor component, this means that changes to a product or supplier are automatically shown in the data table, without the need to update the cache or re-query for the data.

Updating the Routing Configuration

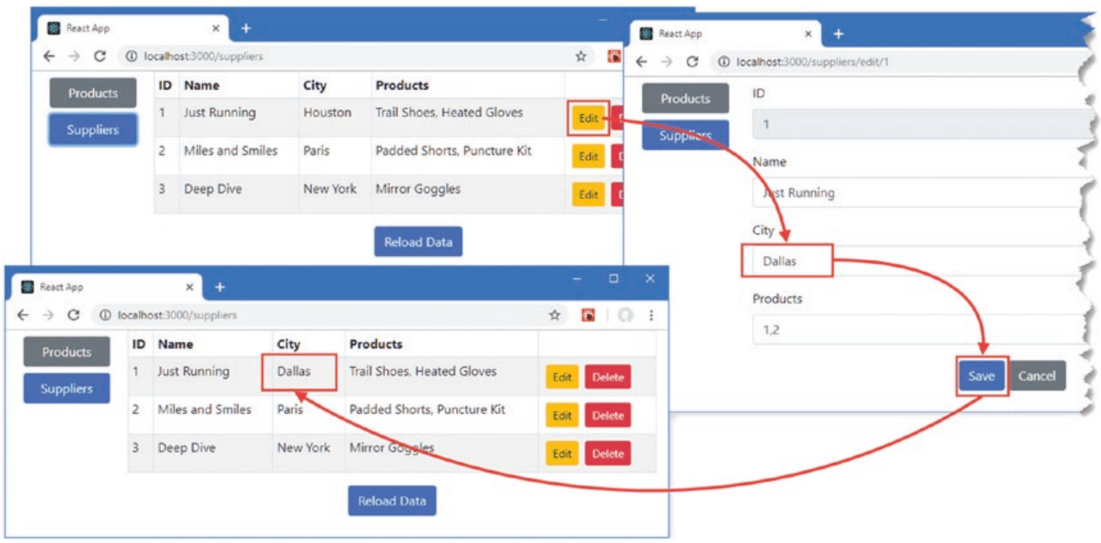
To complete the example, Listing 25-28 updates the routing configuration for the example application to add support for supplier data and for the new editor components.

Listing 25-28. Updating the Routing Configuration in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ToggleLink } from "./routing/ToggleLink";
import { GraphQLTable } from "./graphql/GraphQLTable";
import { PRODUCTS, SUPPLIERS } from "./store/dataTypes";
import { GraphQLEditor } from "./graphql/GraphQLEditor";
```

```
export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/products">Products</ToggleLink>
                        <ToggleLink to="/suppliers">Suppliers</ToggleLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" exact={true}
                                component={ GraphQLTable(PRODUCTS) }  />
                            <Route path="/suppliers" exact={true}
                                component={ GraphQLTable(SUPPLIERS) }  />
                            <Route path="/:dataType/edit/:id"
                                component= { GraphQLEditor() } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

When you save the changes, the application will be updated to support supplier data and editing, as shown in Figure 25-6.



ID	Name	City	Products	
1	Just Running	Houston	Trail Shoes, Heated Gloves	Edit
2	Miles and Smiles	Paris	Padded Shorts, Puncture Kit	Edit
3	Deep Dive	New York	Mirror Goggles	Edit

ID
1
Name
Just Running
City
Dallas
Products
1,2

ID	Name	City	Products		
1	Just Running	Dallas	Trail Shoes, Heated Gloves	Edit	Delete
2	Miles and Smiles	Paris	Padded Shorts, Puncture Kit	Edit	Delete
3	Deep Dive	New York	Mirror Goggles	Edit	Delete

***Figure 25-6.** Supporting supplier data and editing*

■ **Tip** If you encounter errors, stop the development tools and start them again using `npm start`. This will reset the data used by the GraphQL server and undo the effects of changes made in previous sections.

Summary

In this chapter, I showed you the different ways that a React application can consume a GraphQL service. I showed you how to use GraphQL in isolated components, how to intercept data store actions and service them using GraphQL, and how to adopt a GraphQL client that manages the data on behalf of the application.

And that is all I have to teach you about React. I started by creating a simple application and then took you on a comprehensive tour of the different building blocks in the framework, showing you how they can be created, configured, and applied to create web applications.

I wish you every success in your React projects, and I can only hope that you have enjoyed reading this book as much as I enjoyed writing it.