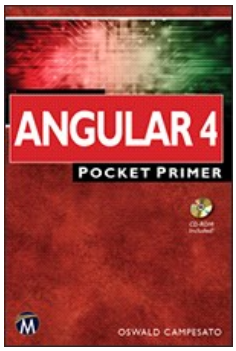

Angular 4 Pocket Primer

by Oswald Campesato

Chapter 7: Flux, Redux, GraphQL, Apollo, and Relay

Overview

This chapter contains a broad introduction to the JavaScript-based technologies `Flux`, `Redux`, `GraphQL`, `Apollo`, and `Relay`. Except for Apollo (which is under the aegis of Meteor), Facebook developed the other technologies in the preceding list. Although these technologies are used in ReactJS-based Web applications, they can also be used in Angular applications as well (and hence this chapter).

Because you can create Angular applications without any of the material in this chapter, the main purpose is to explain some of the concepts in the preceding technologies. Moreover, there is only one complete code sample in this chapter (see the section that discussed Apollo). Although there are various online code samples that combine Angular and Redux, keep in mind that many of them were written for earlier versions of Angular and they might require some modification to work with Angular 4.0.0.

The first section of this chapter describes the `Flux` architecture, designed by Facebook, which has many implementations (including `Redux` and `Relay`). The `Flux` architecture was initially created for developing client-side Web applications. Because `Flux` is language agnostic, you can use the `Flux` pattern in React-based applications, Angular applications, and others. Note that you will also see the `Flux` architecture described as the `Flux` pattern, perhaps in the same sense that model–view–controller (MVC) is also a pattern.

The second section discusses `Redux`, which is a toolkit whose purpose is to store application state outside the application. Interestingly, this approach for storing application state has some advantages, as you will see later in this chapter. You will see a nice example that illustrates how to use `Redux` in an Angular application for keeping track of items.

The third section describes `GraphQL`, which is a JavaScript toolkit that receives `Relay` requests. `GraphQL` processes those requests by retrieving the matching data from a data store, which can be a relational data store or a `NoSQL` data store. This section contains a basic Angular application that uses `GraphQL`.

The fourth section discusses `Apollo`, which is client-side JavaScript that sends data requests to a `GraphQL` toolkit that resides on a server. Apollo has recently gained traction because of its advantages over `Relay`.

One point to remember is that medium-sized and large Web applications benefit from toolkits such as `Redux` or `GraphQL` more so than small Web applications (where MobX might be more suitable). Consequently, if you are currently working on basic Web applications, portions of this chapter might be optional for you right now.

What Is Flux?

The `Flux` pattern is based on other design patterns (such as `Observer` and `Command Query Responsibility Segregation` [CQRS]). `Flux` provides a good foundation for developing sophisticated applications. The primary purpose of `Flux` is its support for one-way data flow. Watch this video for Flux tips:

https://angular-university.io/course/angular2-ngrx

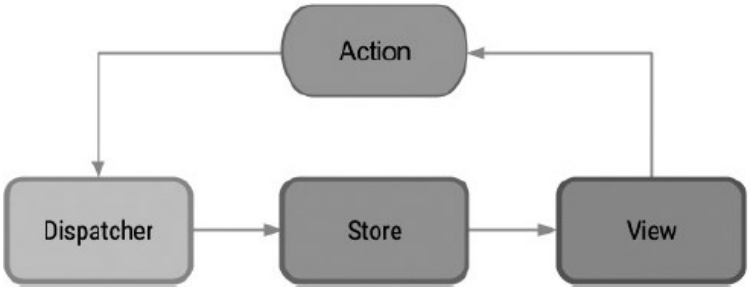

Figure 7.1: The elements of the Flux pattern.

The components of the `Flux` pattern are `Action`, `Dispatcher`, and `Store`. The relationships among these components is depicted here:

```
Action --> Dispatcher --> Store --> View (and back to the
Action)
```

The diagram in Figure 7.1 displays the Flux architecture (copied from the GitHub repository at https://github.com/lgvalle/android-flux-todo-app).

Here's a one-paragraph description of the `Flux` pattern: When users perform a gesture (e.g., click a button) in a Web application, that gesture is converted into an `Action` that "models" a state change via (1) a *type* that describes the type of action and (2) a *value* that contains a new value. The `Action` is sent to a `Dispatcher`, which in turn sends the `Action` and the current `Store` to the appropriate `reducer` function to process the `Action`. The `reducer` function contains a `switch` statement and conditional logic to determine which `case` statement matches the `Action` type, after which the code in the matching `case` statement is executed, and then an updated `Store` (sometimes the `Store` does not change) is returned from the `reducer` function. Next, the `View` objects that are registered to detect state-related changes will update the contents of the Web page accordingly. This process repeats every time users perform a gesture in the Web application. The

more general scenario involves multiple reducers and multiple stores, which means that additional logic is required in the `Dispatcher`. In addition, an `Action` can originate from a server as well as a user-initiated event.

Note Application components in a Flux-based Web application do not communicate directly with each other.

Notice that the preceding flow of data is *always* unidirectional. Moreover, data can *only* be updated in one `case` statement in a `reducer`, which in turn simplifies the debugging process.

What Is an Action?

An `Action` is a JavaScript object that contains `JSON`-based data, with a mandatory property called `type`. The purpose of an `Action` is to specify what needs to be modified in the application state to create a new state. Think of an `Action` as a "message" containing information or instructions for updating the state of an application (such as adding a new user, deleting a user, and so forth). A key point: Instead of modifying the current state, a new object is created as a result of "applying" an action to the current state.

In brief, an application involves many `Actions` (performed as asynchronous operations): one `Dispatcher`, one or more `Stores`, and one or more `View` components. `Actions` typically contain data that is associated with user-initiated events (such as user input, key strokes, mouse-related events, and so forth) that occur in a `View` component.

Although an `Action` can be a string or an array, an `Action` is often a `JSON`-based object literal. For example, an `Action` with an `ADD` operation involving a new user would look something like this:

```
{ type: 'ADD_USER', value: 'John Smith' }
```

The preceding `Action` is forwarded (via a dispatcher) to a reducer function that adds the new user to an external data store. You will see more details for this use case later in the chapter.

Another point to keep in mind is that the properties of an `Action` can have different names. For example, the following `Action` is equivalent to the earlier code snippet:

```
{ kind: 'ADD_USER', data: 'John Smith' }
```

Note that you will sometimes see a JavaScript "helper" function (called an "Action Creator") that returns an `Action`, an example of which is shown here:

```
function actionCreator(type, value) {
   return { type, value };
}
```

What Is a Reducer?

A `Reducer` is a JavaScript function that has no side effects (also called a "pure function"), and its purpose is to transform the current state of an application into a new state, based on the contents of an `Action` object. In general, one function is associated with each action type. `Reducers` are JavaScript functions that change the state of an application.

The JavaScript `Array` object has a `reduce()` method, an example of which is shown here:

```
function sum(a,b) { return a + b }
[1,2,3,4].reduce(sum,0);
```

`Redux` leverages the concept of the JavaScript `reduce()` method as follows: A `reducer()` method has a `state` argument and an `action` argument, with conditional logic that returns a new state.

The Store and Application State

A `Store` is a container that holds the global state tree (a JavaScript object) and dispatches actions. A store also holds the reducers and provides subscriptions to changes of state.

Keep in mind that the Flux-based Web applications maintain the application state *external* to the application, and that application components do not communicate with each other when the application state changes; instead, they obtain the application state from the contents of the `Store`. Because cross communication among multiple components can introduce bugs that are difficult to find, the Flux architecture removes this source of errors (think of what can happen in an application that contains hundreds of components that have cross-communication).

Because the reducing function is the only place where the `Store` can be updated, you know where to look in the event that a data-related error occurs.

What Is Redux?

`Redux` is an open source toolkit created by Dan Abramov that implements the `Flux` pattern (the latter was created by Facebook); its home page is located here:

https://github.com/reactjs/react-redux

Redux consists of Actions (for messages), Reducers (for changing state), and a Store (for holding the global state of an application), and together these components implement the Flux pattern.

One simple use case that illustrates the role of Redux is a Web application that displays a list of names and also allows users to add new names. The flow of data involves the creation of an action (for adding a user), dispatching the action to a method that does the actual adding of a new user, followed by updating a store with the new state of the application, and finally updating the Web page so that it displays the updated list of names. DevTools for Redux with hot reloading, action replay, and customizable user interface (UI) is downloadable here:

https://github.com/gaearon/redux-devtools

Data Structures for Application Data

In addition to storing the application state in a store, there is also the question of which data structures to use for application data. Data structures that work well for server-side code or for the view component do not necessarily work best for the structure of the data in the store. In other words, you cannot simply mimic the same data structures (as tempting as this probably is) for the data in the store and expect the application to be performant. One article that discusses this point is located here:

https://hackernoon.com/avoiding-accidental-complexity- when-structuring-your-app-state-6e6d22ad5e2a#.jzmngm3cf

When Should You Use Redux?

Although answers to this question vary, Dan Abramov himself says that people often start using Redux too soon. Perhaps the right time to use Redux is when the complexity of an application warrants the use of Redux. However, this response raises a new question: What is the right complexity?

Fortunately, there are some guidelines that you can follow to make a determination regarding the use of Redux. The following link also addresses the use of Redux:

http://jamesknelson.com/5-types-react-application-state/

This article is definitely worth the time to read about the details of the preceding numbered list.

There are some additional considerations as well. For example, when you hot reload components, the state is removed from the existing component tree. Hence, if the state of an application is maintained externally, then that state can be reloaded along with the updated components, thereby maintaining a consistent state. Other benefits include better testing facility, centralized logic, time travel debugging, and predictable state updates.

Incidentally, in addition to Redux, there are many implementations of Flux, so you do have options (so far, Redux is the most popular one). One alternative is Mobx, which is discussed later in this chapter.

Simple Examples of Reducers

As you will see in this section, the custom code in reducers often contains a switch statement with multiple case statements (and a default case). The following code block is a generic example of a reducer that takes a state argument (initialized as an empty object) followed by an action argument:

```
const myReducer=  (state = {}, action) = > {
  switch (action.type) {
    case 'ADD':    return { ... }
    case 'DELETE': return { ... }
    case 'UPDATE': return { ... }
    default:       return state
  }
}
```

The details of the case statements are obviously application specific. The following subsections illustrate reducers whose state is a numeric counter, along with an example where the state is an array of strings. Note that some of the code samples use the "spread" operator in JavaScript.

A Reducer to Add/Subtract Numeric Values

In an HTML Web page, a JavaScript variable can keep track of a numeric counter. Now let's learn how to use a reducer function to keep track of such a counter. The "state" of the counter is simply its current value, which will be "stored" separately from the JavaScript code in the Web page.

Suppose we have a Web page that contains two buttons: one button subtracts 1 from a counter and the other button adds 1 to a counter (the initial value of the counter is 0).

The ACTION elements {type:'ADD'} and {type:'SUBTRACT'} correspond to the add button and the subtract button, respectively.

The reducer for this Web page consists of a `switch` statement that contains three `case` statements that handle `ADD`, `SUBTRACT`, and a default operation, as shown here:

```
// two operations: add or subtract
constCountReducer=(state=0,action)=>{
   switch(action.type) {

               case 'ADD':      return state + 1;
               case 'SUBTRACT': return state - 1;
               default:         return state;
           }
      };
```

Note Reducers handle state changes synchronously.

A generalized version of the preceding example involves an `ACTION` element that contains an `amt` field whose numeric value is the amount to add or to subtract from the counter. In this scenario, a typical `ACTION` is `{type:'ADD', amt:3}` or `{pe:'SUBTRACT', amt:5}`, where the value of `amt` is populated elsewhere. Furthermore, the modified version of the `CountReducer` code block is here:

```
constCountReducer=(state=0,action)=>{
   switch(action.type) {
      case 'ADD':      return state + action.amt;
      case 'SUBTRACT': return state - action.amt;
      default:         return state;
   }
};
```

A reducer in a real application is similar to the preceding `CountReducer`: The differences involve the specific `case` statements and the code that is executed in each case statement.

A Reducer to Add/Remove Strings from an Array

The following code block defines a reducer that can add an item (`ADD_ITEM`) and remove an item (`DEL_ITEM`) from an array:

```
export const arrReducer = (state = [], action) = > {
  switch.action(type) {
     case 'ADD_ITEM':return[...state,action.payload];
     case 'DEL_ITEM':
       return state.filter(n=>n.val != action.payload.val);
     default:        return state;
   }
};
```

In the preceding code block a new item is added by creating a new array that contains the current state, with the item appended to that array. Notice how the `spread` operator "…" provides a compact way to list the items in the current state.

An existing item is deleted by conditional logic in a `filter()` method: Everything in the current state, except for the item in question, is returned.

Redux Reducers Are Synchronous

Keep in mind the following points about `Redux` reducers. First, the `arrReducer()` method in the preceding section is a pure function because no mutation occurs in this function; no external variables are required, and a new state is created instead of using the array `push()` method (which is a mutator).

Second, reducers are synchronous; however, you can use redux-observable if you need a reducer that performs calculations asynchronously. Third, data logic in a `reducer()` is separate from the view layer. Fourth, multiple reducers can be defined in an application.

RxFlux is a Flux implementation based on RxJS, and its home page is located here:

https://github.com/fdecampredon/rx-flux

A very good introduction to `Redux` is this video series created by Dan Abramov:

https://egghead.io/series/getting-started-with-redux

The Redux Store

The `Redux Store` maintains the state of an application, which is represented as a global state tree. The `Store` provides two functions: `getState()` and `dispatch()`.

The `getState()` function allows different parts of an application to access the state-related information.

There are two other points to keep in mind regarding the `Store`. First, the store holds the reducers and invokes them in a "broadcast" fashion whenever actions are dispatched. Second, the store provides a subscription mechanism to notify portions of the application (such as the UI) when the state tree has been modified.

The following code block shows you how to create a store:

```
import { createStore } from 'redux'
import RootReducer from '··/Reducers/'
const store = createStore(RootReducer)
```

Middleware

The `Redux Store` can also be extended by plugins in the form of middleware. Various types of third-party middleware are available to perform an assortment of tasks, such as persistence, logging, and flow control.

To summarize, Redux works as follows:

- The application state is encapsulated in a JavaScript object called "state."
- The state is held in a store.
- The store is immutable and never directly changed.
- User interactions fire actions that describe the event and encapsulate the data.

A function called a reducer combines the old state and the action to create the new version of the state, which is kept in the store. Redux simplifies the task of state management by separating the functional code from the presentational code. Instead of using Redux in Angular applications, it's easier to use an implementation of Redux, such as ngrx-store.

https://github.com/mgechev/angular-seed

An extensive description of ngrx-store that also reinforces material from the first part of this chapter can be found here:

https://gist.github.com/btroncone/a6e4347326749f938510

A free 10-minute video about `ngrx-store` (as this book goes to print) can be found here:

https://egghead.io/lessons/angular-2-ngrx-store-in-10- minutes

This concludes the section regarding Angular applications and `Redux`. If you decide that `Redux` is too complex for your needs, one alternative to `Redux` is `MobX`, which is simpler than working with `Redux`. `MobX` is considered one of the popular (and simpler) alternatives to `Redux`, and its home page is located here:

https://github.com/mobxjs/mobx

What Is GraphQL?

`GraphQL` is a data query language and runtime from Facebook that can send data to mobile and Web applications; its home page is located here:

http://graphql.org/

GraphQL is a specification, which means that it can be used with any platform and any language. Facebook maintains its reference implementation, which is written in JavaScript. The `GraphQL` specification is located here:

http://facebook.github.io/graphql/

A `GraphQL` schema acts as a "wrapper" around a data store that can include NoSQL data and relational data. For example, the following link explains how to use `GraphQL` with Mongo:

https://www.compose.io/articles/using-graphql-with-mongodb/

Facebook's StarWars schema is here: https://goo.gl/oCrK7F

The following are aspects of GraphQL:

- Query language
- Query semantics

- Query variables
- Mutations
- Fragments

The GraphQL query language is a major part of GraphQL, examples of which you will see in a subsequent section. GraphQL query variables enable you to pass values to GraphQL queries, which is obviously better than using hard-coded values. Mutations allow you to change the dataset behind GraphQL. A mutation is very similar to a field in a GraphQL query, but GraphQL assumes a mutation has side effects and changes the dataset behind the schema. GraphQL fragments provide a mechanism for grouping commonly used fields and reusing them. Some of the programming languages that have implemented `GraphQL` are located here:

http://graphql.org/code/

Companies that use `GraphQL` include Coursera, Intuit, Pinterest, and Shopify. In addition, GitHub supports `GraphQL` (starting from 2016), and has released its public `GraphQL` API:

http://githubengineering.com/the-github-graphql-api/

GraphQL versus REST

As you will see in subsequent sections, `GraphQL` enables you to specify fine-grained queries that return only the data that is required by a client application. Keep in mind the following point about applications that involve `GraphQL`: Data fetching details are made on the client, whereas data fetching details are made on the server in applications that use Representational State Transfer (REST).

There are several advantages to this approach:

- No redundant data is sent to the client.
- Adding new data fields on the server does not affect queries.
- Only one network round trip is required.

The preceding advantages of `GraphQL` are particularly important for mobile applications, where the cost of retrieving data can be significant. In addition, the `GraphQL` query is unaffected by the addition of new fields to the customer object (on the server).

On the other hand, a REST-based request is more coarse-grained and involves "overfetching": Such a request returns 100% of the fields in a customer object, in which no distinction is made between fields that are required and fields that are not required. Moreover, the addition of new fields to a customer object increases the size of the payload returned to the client.

GraphQL Queries

A `GraphQL` query is a string interpreted by a server that returns data in a specified format. Here is an example of a very simple `GraphQL` query:

```
{
    emp {
      fname
    }
}
```

The preceding query can be read as "give me the `fname` attribute of the `emp` entity." As you can see, the `emp` entity is followed by a pair of curly braces that contain a single attribute called `fname`.

The next query is considerably more complex, yet follows the same mechanism as the previous query:

```
{
   user(id: 30000) {
     id,
     name,
     isViewerFriend,
     profilePicture(size:50){
       uri,

       width,
       height
     }
   }
}
```

The preceding query requests the attributes `id`, `name`, and `isViewerFriend` of the user whose `id` is `30000`. In addition, the query

requests the attributes `uri`, `width`, and `height` of the `profilePicture` (50 × 50 pixel size) of the same user.

The response to the preceding query is shown here:

```
{
  "user" : {
    "id":30000,
    "name":"JohnSmith",
    "isViewerFriend":true,
    "profilePicture":{
      "uri":  "http://www.acme.com/johnsmith.jpg",
      "width":  50,
      "height": 50
    }
  }
}
```

`GraphQL` queries can be very complex, and sometimes the data that is returned contains duplicate "subtrees" of data. The open source project Apollo (discussed later in this chapter) removes the duplicate subtrees from the data that is returned by the server.

Defining a Type System in GraphQL

This section contains an example of a type system that models the JavaScript Object Notation (JSON)-based data in the file `employees.json`.

The first part of the type system defines an `Employee` interface, and the second part of the type system defines a `Query` type, as shown here:

```
interface Employee {
  id: String!
  fname: String
  lname: String
}

type Query {
  emp: Employee

}
```

For this section, let's make the initial simplifying assumption that there is only one employee named "John Smith." The following query specifies the `fname` field of an employee:

```
{
    emp {
      fname
    }
}
```

The result of the preceding query is shown here:

```
{
    "data": {
      "emp": {
        "fname": "John"
      }
    }
}
```

The following query is a more verbose way of specifying the `fname` field of an employee:

```
queryEmpNameQuery{
  emp {
    fname
  }
}
```

The result of the preceding query is shown here:

```
{
    "data": {
      "emp": {
        "fname": "John"
      }
    }
```

```
 }
```

A query that contains the query keyword and an operation name (such as `EmpNameQuery`) is required to specify multiple fields. For example, the following query specifies the `fname` and the `lname` fields:

```
queryEmpNameQuery{
  emp {
    fname
    lname
  }
}
```

The result of the preceding query is shown here:

```
{
  "data": {
    "emp": {
      "fname":"John",
      "lname": "Smith"
    }
  }
}
```

Now let's suppose that our data set consists of the following employees:

```
[
  {"fname":"Jane","lname":"Jones","city":"SanFrancisco"},
  {"fname":"John","lname":"Smith","city":"NewYork"},
  {"fname":"Dave","lname":"Stone","city":"Seattle"},
]
```

Let's look at the earlier query again:

```
queryEmpNameQuery{
  emp {
    fname
    lname
  }
}
```

Now the result of the preceding query involves the `fname` and `lname` fields of three employees, as shown here:

```
{
  "data": {
    {
      "fname":"Jane",
      "lname": "Jones"
    },
    {
      "fname":"John",
      "lname": "Smith"
    },
    {
      "fname":"Dave",
      "lname": "Stone"
    }
  }
}
```

The preceding code samples provide a very basic introduction to the types of queries that you can define in `GraphQL`. You can also define parameterized queries as well as queries that return a hierarchical dataset.

For example, the following parameterized query specifies the `fname`, `lname`, and `city` fields for the employee whose first name is `Jane`:

```
{
    Employee(fname: "Jane") {
      fname
      lname
      city
    }
}
```

The preceding query returns the following data:

```
 {
```

```
  "data": {
    "emp": {
      "fname": "Jane",
      "lname": "Jones"
      "city":  "San Francisco"
    }
  }
}
```

Useful GraphQL Links

`Reindex` is a `GraphQL` generator and its home page is located here:

https://www.reindex.io/

Scaphold.io is an online service for creating `GraphQL` schemas and its home page is located here:

www.scaphold.io

A list of tools and integrations with various languages and data storage engines for `GraphQL` is located here:

https://www.npmjs.com/search?q=graphql

If you work with React applications, you can use `GraphiQL`, which is an in-browser integrated development environment (IDE) that is downloadable here:

https://github.com/graphql/graphiql

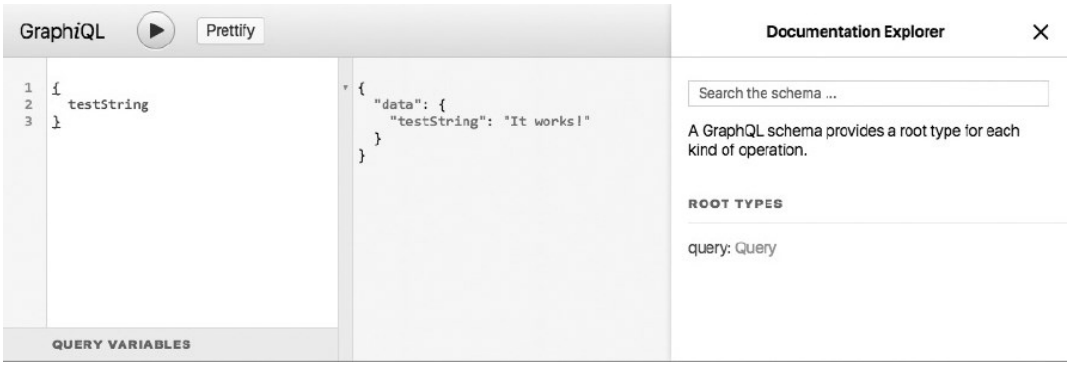

Figure 7.2: GraphiQL in a Chrome rrowser.

The following link contains documentation and a code sample for GraphQL:

http://graphql.org/graphql-js/basic-types/

Figure 7.2 displays an example of GraphiQL in a Chrome browser.

The next section discusses Apollo, followed by an application that combines Apollo and Angular.

What Is Apollo?

`Apollo` is an open source project that is designed to work with `GraphQL`, and its home page is located here:

http://www.apollodata.com/

An interactive tutorial (with a video) for Apollo is located here:

https://www.learnapollo.com/introduction/get-started/

The code sample in this section consists of a server component and an Angular-based client component, where the former requires some configuration steps.

Note You must launch the Angular client after you launch the Apollo server, as described in the next two subsections.

Launching the Apollo Server

Download and uncompress the code from this GitHub repository:

https://github.com/apollographql/GitHunt-API

Perform the setup steps that are described in the `README.md` file, which includes registering an application on GitHub to obtain two keys. Next, install the required modules with this command:

```
npm install
```

Now launch the Apollo server with this command:

```
npm start
```

Figure 7.3 displays the contents that the Apollo server renders (if everything was configured correctly) in a Chrome browser.

Launching the Angular Client

Download and uncompress the Angular client-side code from this GitHub repository:

https://github.com/apollographql/githunt-angular

Next, install the required modules with this command:

```
npm install
```

Now launch the Angular client with this command:

```
npm start
```

Figure 7.4 displays a portion of the contents that the Angular client renders (if everything was configured correctly) in a Chrome browser.

Now let's look at some of the client-side Angular code.

Figure 7.3: The Apollo aerver in a Chrome browser.

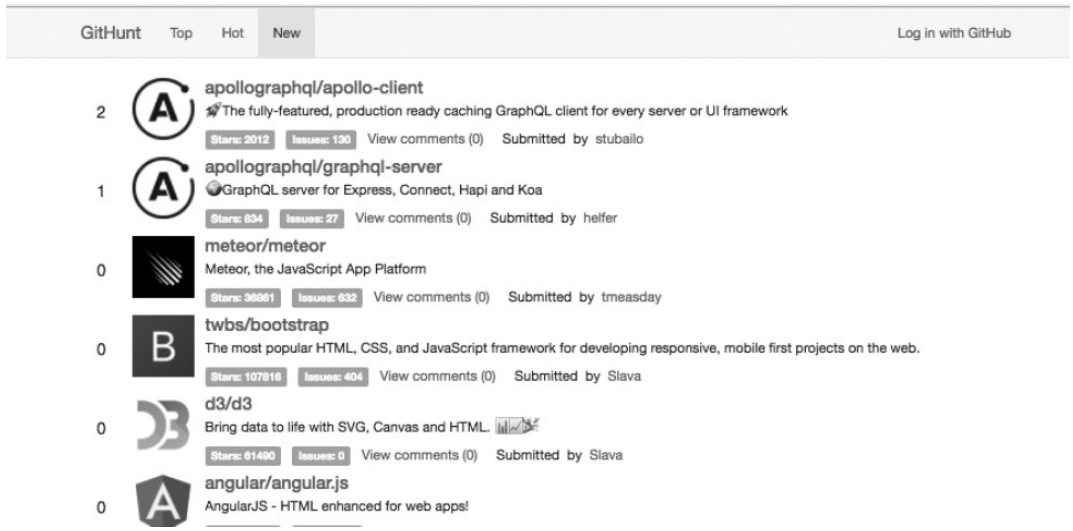

Figure 7.4: The Angular client in a Chrome browser.

Project Structure in the Angular Client

The `app/src` subdirectory of the Angular code in the previous section contains the following files:

```
./app.component.html
./app.component.ts
./app.module.ts
./client.ts
./comments
./comments/comment.component.html
./comments/comment.component.ts
./comments/comments-page.component.html
./comments/comments-page.component.ts
./comments/comments-page.model.ts
./comments/index.ts
./feed/feed-entry.component.html
./feed/feed-entry.component.ts
./feed/feed-entry.model.ts
./feed/feed.component.html
./feed/feed.component.ts
./feed/feed.model.ts
./feed/index.ts
./feed/vote-buttons.component.html
./feed/vote-buttons.component.ts
./feed/vote-buttons.model.ts
./index.ts
./navigation/navigation.component.html
./navigation/navigation.component.ts
./new-entry/new-entry.component.html
./new-entry/new-entry.component.ts
./new-entry/new-entry.model.ts
./profile/profile.component.html
./profile/profile.component.ts
./profile/profile.model.ts
./routes.ts
./shared/index.ts
./shared/info-label.component.ts
./shared/loading.component.ts
./shared/repo-info.component.html
./shared/repo-info.component.ts
./shared/repo-info.model.ts
./subscriptions.ts
```

The file `app.component.ts` contains nothing more than the definition of the `AppComponent` class.

Listing 7.1 displays the contents of the file `app.module.ts` that performs all the work, which differs from the other code samples in this book.

Listing 7.1: app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule  } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule}  from'@angular/
                                                    forms';
import { RouterModule } from '@angular/router';
import { ApolloModule }  from 'apollo-angular';
import { EmojifyModule } from 'angular2-emojify';

import { AppComponent }  from './app.component';
import { NavigationComponent }
         from './navigation/navigation.component';
import { ProfileComponent}
         from'./profile/profile.component';
import { NewEntryComponent }
         from './new-entry/new-entry.component';
import { FEED_DECLARATIONS } from './feed';
import { COMMENTS_DECLARATIONS } from './comments';
import { SHARED_DECLARATIONS } from './shared';
import { routes } from './routes';
import { provideClient } from './client';
```

```
import { InfiniteScrollModule}
         from'angular2-infinite-scroll';

@NgModule({
  declarations: [
    AppComponent,
    NavigationComponent,
    ProfileComponent,
    NewEntryComponent,

    ...FEED_DECLARATIONS,
    ...COMMENTS_DECLARATIONS,
    ...SHARED_DECLARATIONS
  ],
  entryComponents: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule.forRoot(routes),
    ApolloModule.forRoot(provideClient),
    EmojifyModule,
    InfiniteScrollModule
  ],
  bootstrap:[AppComponent],
})
export class AppModule {}
```

Listing 7.1 starts by importing various custom components via the initial import statements, including the Apollo-related `ApolloModule`, which is shown in bold. Next, the `NgModule` decorator contains a `declarations` property that specifies some components, some of which involve the spread operator (which you have not seen in previous examples). The `NgModule` decorator also contains an `imports` property that uses the `ApolloModule.forRoot()` syntax (shown in bold) to reference the exported function `provideClient()` that is defined in `client.ts`. This function handles the details of fetching data from the server.

Various GitHub repositories with examples containing Apollo can be found here:

https://github.com/apollostack

One of the advantages of `Apollo` over `Relay` (discussed later) is its ability to remove duplicate subtrees in the dataset that is returned by a `GraphQL` query. However, Facebook developed Relay, and it's the topic of the next section.

What Is Relay?

Relay is a JavaScript-based technology from Facebook that acts as a "wrapper" around ReactJS components that require data from a server. The Relay home page is located here:

https://github.com/facebook/relay

ReactJS applications can use `Relay` (discussed later in the chapter) to issue data requests to `GraphQL`. In addition, `GraphQL` can provide data to clients (such as Angular clients) that do not use Relay.

Note `GraphQL` can be used independently of `Relay`, whereas `Relay` cannot be used without `GraphQL`.

By way of comparison, `REST`-based requests return data for an entity (such as the data about a customer), whereas `Relay` enables you to request individual fields of an entity (such as the first name and last name of a customer). In simplified terms, you can view `Relay` as a finer-grained alternative to `REST`.

Relay uses a network layer to communicate with a GraphQL server. Relay provides a network layer that is compatible with `express-graphql`, and additional features will be added to the network layer as they are developed.

The following link contains an interactive tutorial for Relay:

https://www.learnrelay.org/

Note GraphQL can be used independently of Relay, whereas Relay cannot be used without GraphQL.

Relay Modern

Facebook released React Fiber in early 2017, which is an extensive rewrite of ReactJS, with the goals of improved performance and extensibility. Facebook also rewrote Relay, which is called Relay Modern. The new features of Relay Modern include static queries, AOT (ahead-of-time) compilation, and built-in garbage collection.

Relay Modern also provides a compatibility API in the event that you are already using an older version of Relay.

Summary

This chapter started with `Flux`, which is a unidirectional pattern for Web applications. You also learned about `Redux`, which is one of the implementations of the `Flux` pattern. In addition, you saw how to use `Flux/Redux` in Angular applications. Next you learned about `Relay` and `GraphQL`, both of which were developed by Facebook. Then you saw an example of an application that uses `GraphQL` to retrieve server-side data that was defined in a `JSON`-based file.