PREV
16. Build Tools

NEXT
18. In the family

Chapter 17. Architectural Patterns

React renders HTML. You can consider it as the V in MVC, but it doesn't make any decision if you use simple Ajax requests in your components directly like so:

```
var TakeSurvey = React.createClass({
  getInitialData: function () {
    return {
      survey: null
    };
  },
  componentDidMount: function () {
    $.getJSON('/survey/' + this.props.id, function (json) {
      this.setState({survey: json});
    });
  },
  render: function () {
    if (!this.state.survey) return null;

    return <div>{this.state.survey.title}</div>;
  }
});
```

Or if you use an MVC framework, and thus it can easily be integrated in most application architectures.

In this chapter we will go over a few approaches or libraries you could use with React to help structure your application.

MVC

Will be added in the next version.

Routing

Routers are what directs urls in a single page application to specific handlers. You can imagine that for the url `/surveys` you'd want to run a function that loads the users from the serve r and renders a `<ListSurveys />` component.

There are many different kinds of Routers. They exists on the server as well. Some routers work both on the client and on the server.

React is simply a rendering library and does not come with a Router, but there are many options that work great with React. In this section we'll cover a few.

pushState and hash routing

Will be added in the next version.

Backbone.Router

Backbone is Single Page Application library, which employs the Model-View-Whatever architecture. The "Whatever" is a replacement for "Controller" and usually refers to a Router. This is true in Backbone's case as well.

Backbone is modular and you can choose to only use the Router. It works great with React.

Consider our /surveys example from before with Backbone.Router:

```
var SurveysRouter = Backbone.Router.extend({
  routes: {
    "surveys": "list"
  },
  list: function() {
    React.renderComponent(
      <ListSurveys />,
      document.querySelector('body')
    );
  }
});
```

Routers need to be able to handle dynamic segments in the url or the queryString. Backbone.Router supports this like so:

```
// surveys_router.js
var SurveysRouter = Backbone.Router.extend({
  routes: {
    "surveys": "list",
    "surveys/:filter": "list"
  },
  list: function(filter) {
    React.renderComponent(
      <ListSurveys filter={filter}/>,
      document.querySelector('body')
```

```
  }
});
```

In the above example, given a url like so: ‘/surveys/active` the `filter` argument to `SurveysRouter#list` would be `”active”`.

Learn more about Backbone.Router and download it here: http://backbonejs.org/#Router

Aviator

Unlike `Backbone.Router`, Aviator is a standalone routing library.

With Aviator you define your routes separate from your “RouteTargets”. Aviator doesn’t concern itself with how a `RouteTarget` looks or acts, it just tries to call the assigned functions on it.

Imagine a `RouteTarget` like so:

```
// surveys_route_target.js
var SurveysRouteTarget = {
  list: function () {</div><div>    React.renderComponent(</div><div>&lt;ListS
```

Along with that object, a route configuration (there can be only 1 configuration for the entire application) is required. This configuration is usually in a separate file.

```
// routes.js
Aviator.setRoutes({
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list'
  }
});
// Tell Aviator to start dispatching urls to targets:
Aviator.dispatch();
```

To setup `RouteTarget`s for handling params:

```
// routes.js
Aviator.setRoutes({
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list',
    '/:filter': 'list'
  }
});
```

```
// surveys_route_target.js
```

```
    React.renderComponent(
      <ListSurveys filter={request.params.filter}/>,
      document.querySelector('body')
    );
  }
};
```

One of the coolest features in Aviator is multiple targets.

Consider this routes definition:

```
// routes.js
Aviator.setRoutes({
  target: AppRouteTarget,
  '/*': 'beforeAll',
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list',
    '/:filter': 'list'
  }
});
```

For a URL like `'/surveys/active'` Aviator will call `AppRouteTarget.beforeAll` and then `UsersRouteTarget.list` - There can be any number of matched actions like that. You can provide exit functions for a route as well, which will call the chain of route actions in the reverse outer when the user navigates away from the matched route.

Read more about and download Aviator here: https://github.com/swipely/aviator.

react-router

React-router differs from the other routes in that it is completely made up of React components.

The routes are defined as components and the handlers of the routes are components.

This is how our router looks with react-router:

```
var appRouter = (
  <Routes location="history">
    <Route title="SurveyBuilder" handler={App}>
      <Route name="list" path="/" handler={ListSurveys} />
      <Route title="Add Survey to SurveyBuilder" name="add" path="/add_survey" handler={
      <Route name="edit" path="/surveys/:surveyId/edit" handler={EditSurvey} />
      <Route name="take" path="/surveys/:surveyId" handler={TakeSurveyCtrl} />
      <NotFound title="Page Not Found" handler={NotFoundHandler}/>
    </Route>
  </Routes>
);
```

Each handler is a component for that specific page. To get the router to run, you render it as usual

```
React.renderComponent(
  appRouter,
  document.querySelector('body')
);
```

Like the other Routers, react-router has a similar concept of params. For instance the route `'/surveys/:surveyId'` will pass in the surveyId property to the `TakeSurveyCtrl` component.

One of the cool features of react-router is that at provides a Link component which you can use for navigation and it will map it to the routes. It will even automatically mark the current page as active by adding the 'active' class to the link.

Here's how our `<MainNav/>` component looks with react-router's Link component:

```
var MainNav = React.createClass({
  render: function () {
    return (
      <nav className='main-nav' role='navigation'>
        <ul className='nav navbar-nav'>
          <li><Link to="list">All Surveys</Link></li>
          <li><Link to="add">Add Survey</Link></li>
        </ul>
      </nav>
    );
  }
});
```

Read more about and download react-router here: https://github.com/rackt/react-router.

Om (ClojureScript)

Om is a very popular ClojureScript interface to React.
With persistent data structures from ClojureScript Om can re-render from the root of your application very fast. Each rendition is trivial to snapshot and use for something like undo.

Here's how an Om component looks:

```
(ns example
  (:require [om.core :as om :include-macros true]
            [om.dom :as dom :include-macros true]))

(defn App [data owner]
  (reify
    om/IRender
    (render [this]
      (dom/h1 nil (:text data)))))

(om/root App {:text "Survey Builder"}
  {:target (. js/document (querySelector "body"))})
```

This renders: `<h1>Survey Builder</h1>`

Flux

The previous architectural patterns have all emerged around React since it became open-source. However, from the beginning Flux has been the pattern React's original authors intended it to be used with.

Flux is an architectural pattern introduced by Facebook. It compliments React with a uni-directional data-flow that's easy to reason about and requires very little scaffolding to get up and running.

Flux is made of three main parts: the store, the dispatcher, and the views (which are React components.) Actions, which are helper methods to create a semantic interface to the dispatcher, can be thought of as a fourth part of the Flux pattern.

You can think of your top-most React component as a Controller-View. The Controller-View component interfaces with the store and facilitates communication with its children view components. This is not unlike a ViewController in the world of iOS.

Each node in the Flux pattern is independent, enforcing strict separation of concerns and keeping each easily testable in isolation.

Data Flow

A unique aspect of Flux is its one-way information flow. In a world of modern MVC frameworks this can seem strange, but it comes with some distinct benefits. Since Flux does not make use of two-way data binding your app becomes easier to reason about. State is easier to track since it is maintained by the stores (which own the data.) Stores deliver the data through their change methods. This triggers views to render. User input causes actions to dispatch through the dispatcher, which delivers the payload to the stores that care about the particular action.

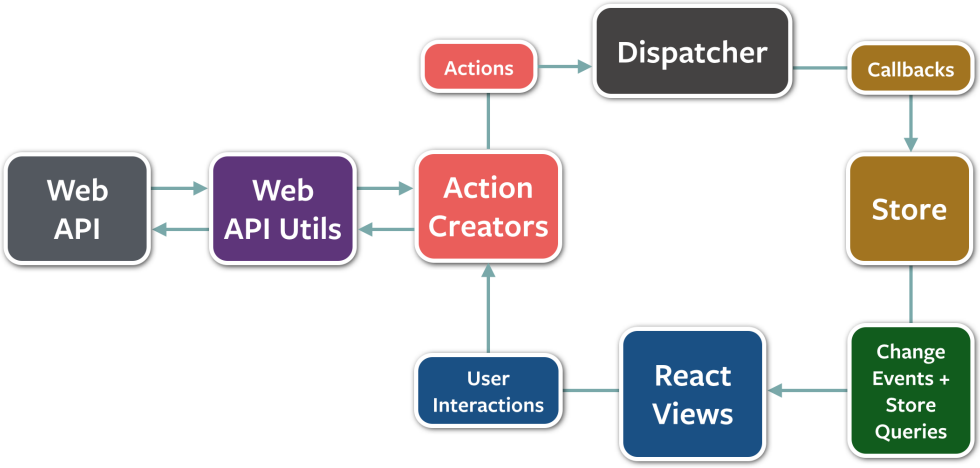

Figure 17-1. Flux enforces a one-way data flow

Flux Parts

Flux is comprised of distinct parts with specific responsibilities. Within the uni-directional flow of information each Flux part takes the input from downstream, processes it, and sends it's output upstream.

- Dispatcher — a central hub within your app
- Actions — the domain-specific-language of your app
- Store — business-logic and data-interactions
- Views — the component tree for rendering the app

Below we discuss each part, what it's responsible for, and how to use it effectively within your React apps.

DISPATCHER

We've chosen to start with the dispatcher because it is the central hub through which all user interaction and data flows. The dispatcher is a singleton within the Flux pattern.

The dispatcher is responsible for registering callbacks on the stores and managing the dependencies between them. User actions flow into the dispatcher. A data payload flows out into the stores that have registered for it.

Our sample app contains a fairly simple but effective dispatcher for managing a single store. However, as you grow your app you'll inevitably run into situations where you need to manage multiple stores and their dependencies upon each other. We'll discuss this case below.

From your user's perspective this is where Flux starts. Every action they take through the UI creates an action that is sent to the dispatcher.

While actions are not a formal part of the Flux pattern, they do constitute the domain-specific-language of your app. User actions are translated into meaningful dispatcher actions — statements that the store can act on.

Our survey-taking app has three distinct actions the user can take:

1. Saving a new survey definition
2. Deleting an existing survey
3. Recording survey results

We've captured these actions within our `SurveyActions` object.

```
var SurveyActions = {
  save: function(survey) {...},
  delete: function(id) {...},
  record: function(results) {...}
};
```

For example, consider a user taking one of our surveys — after they've filled out all the required form fields they click "Save". This "click" action is translated into an "onSave" method call.

```
var TakeSurvey = React.createClass({
  ...
  handleClick: function() {
    this.props.onSave(this.state.results);
  },
  render:function(){
    return (
      <div className="survey">
        ...
        <button ... onClick={this.handleClick}>Save</button>
      </div>
    );
  }
});
```

The `TakeSurveyCtrl` Controller-View handles the `onSave` call, translating the "save" action from the user to the store's "record" action.

```
var TakeSurveyCtrl = React.createClass({
  ...
  handleSurveySave: function(results) {
    SurveyActions.record(results);
  },
  render:function () {
    var props = merge({}, this.state.survey, {
      onSave: this.handleSurveySave
```

```
  }
});
```

In this way user actions flow through the components into the dispatcher. In the process user actions are translated into the domain-specific-language of the Flux app.

STORE

The store is responsible for encapsulating business logic and interactions with your app's data. The store chooses which actions to respond to from the dispatcher by registering for them. Stores send their data into the React component hierarchy through their change event.

It's important to keep strict separation of concerns with the store. No other part of your app should know how to interact with the data. All updates flow through the dispatcher into the store. Fresh data flows back into the app through the store's change event.

Following our example above of a user recording their survey results, the dispatcher sends the "record" action to the store.

```
Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    ...
    case SurveyConstants.RECORD_SURVEY:
      SurveyStore.recordSurvey(payload.results);
      break;
  }
});
```

And the store receives the action, performs the work of saving the results, and when done emits the change event.

```
SurveyStore.prototype.recordSurvey = function(results) {
  // handle saving the results here
  this.emitChange();
}
```

This change event is handled by the main Controller-View defined in `app.js`, with the new state flowing through the React component hierarchy, and the React components will re-render as needed.

CONTROLLER-VIEW

Your app's component hierarchy will typically have a top-level component responsible for interacting with the store. Simple apps will only have one. More complex apps might have multiple controller-views.

Within our sample app the Controller View `App` is defined in the `app.js` file. The process of wiring

1. When the component mounts, add the change listener
2. As the store changes, request the new data and process accordingly
3. When the component unmounts, clean up the change listener

Here is the snipped from `app.js` for handling store interactions.

```
var App = React.createClass({
  handleChange: function() {
    SurveyStore.listSurveys(function(surveys) {
      // handle the survey data
    });
  },
  componentDidMount: function() {
    SurveyStore.addChangeListener(this.handleChange);
  },
  componentWillUnmount: function() {
    SurveyStore.removeChangeListener(this.handleChange);
  },
  ...
});
```

Managing Multiple Stores

Our simple survey app only requires one store, but inevitably apps grow to need multiple stores. This becomes tricky when one store depends upon another, requiring the second store to complete its actions before the first can perform its own actions.

For example, maybe we create a separate store to maintain a survey-results summary, tallying the results of all survey respondents. This summary store requires the main store to complete it's "record" action before we can safely update the summary store.

This will require a few changes:

- The Dispatcher must be updated so it can enforce the action queue.
- We need the ability to tell the Dispatcher to wait for an action to complete.
- The callbacks registered on the dispatcher must define which actions they wait upon.

Our stores should not be responsible for any of this. It's the dispatcher that enforces the action queue, and the methods we register with the dispatcher that control the flow of calls to our stores. Therefore it's the functions we register with the dispatcher who own this work.

While a full refactor of the Dispatcher is beyond the scope of this discussion, here we'll discuss the main points concerning multiple stores. A complete solution can be found at github.com/facebook/flux.

Updating the Dispatcher

Our existing Dispatcher simply pushes new callbacks into an array. However, to enforce order we need to track each callback. Therefore we refactor our Dispatcher to assign each registered callback an id. This id is the token other callbacks can use when telling the dispatcher they need to wait on another store.

```
Dispatcher.prototype.register = function(callback) {
  var id = uniqueId('ID-');
  this.handlers[id] = {
    isPending: false,
    isHandled: false,
    callback: callback
  };
  return id;
};
```

Next we must add a `waitFor` method so we can tell the store to invoke certain callbacks before proceeding. An example of the `waitFor` function might look like this, where we pass in a list of ids returned from the `register` function and make sure they are invoked before proceeding.

```
Dispatcher.prototype.waitFor = function(ids) {
  for (var i = 0; i < ids.length; i++) {
    var id = ids[i];
    if(!this.isPending[id] && !this.isHandled[id]) {
      this.invokeCallback(id);
    }
  }
};
```

REGISTERING FOR DEPENDENT ACTIONS

Now we have two stores concerned about the `RECORD_SURVEY` action.

- The `SurveyStore` needs to record the survey results
- The `SurveySummaryStore` needs to re-tally all the results.

This means that `SurveySummaryStore` is dependent upon the `SurveyStore` completing the `RECORD_SURVEY` action before it can complete its work.

When we register the SurveyStore at the top of our App, we store a reference to the dispatcher token.

```
// Wire up the SurveyStore with the action dispatcher
SurveyStore.dispatchToken = Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    ...
    case SurveyConstants.RECORD_SURVEY:
      SurveyStore.recordSurvey(payload.results);
      break;
    ...
```

Then we register our new `SurveySummaryStore`, placing the call to `waitFor` before we access the `SurveyStore` data.

```
SurveySummaryStore.dispatchToken = Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    case SurveyConstants.RECORD_SURVEY:
      Dispatcher.waitFor(SurveyStore.dispatchToken);
      // At this point it's guaranteed the `SurveyStore` callback has been run
      // and we can safely access its data to summarize it.
      SurveySummaryStore.summarize(SurveyStore.listSurveys());
      break;
    }
  }
});
```

By now you've seen how React can be used with many modern architectural patterns. From integrating React into existing projects using traditional MVC to starting fresh with a new pattern like Flux, React has proven to be quite adaptable.

Beyond adapting to many architectural patterns there are other libraries and tools that work well with React. Up next you can read about other tools in the family that can both support and enhance your React p