PREV
18. In the family

Chapter 19. Uses

React is a powerful interactive UI rendering library, it provides a great way to handle data and user input. It encourages small components that are reusable and easy to unit test. These are all great features that we can apply to other technologies than just the web.

In this chapter we'll look at how to use React for:

- Desktop applications
- Games
- Emails
- Charting

Desktop

With projects like atom-shell or node-webkit we can run a web application on the desktop. The Atom Editor from Github is built with atom-shell and also uses React.

Lets get our SurveyBuilder app working with atom-shell.

First we download and install from https://github.com/atom/atom-shell

Running the atom shell with this desktop script opens up the app in a window.

```
// desktop.js
var app = require('app');
var BrowserWindow = require('browser-window');
// require our SurveyBuilder server and start it.
var server = require('./server/server');
server.listen('8080');
```

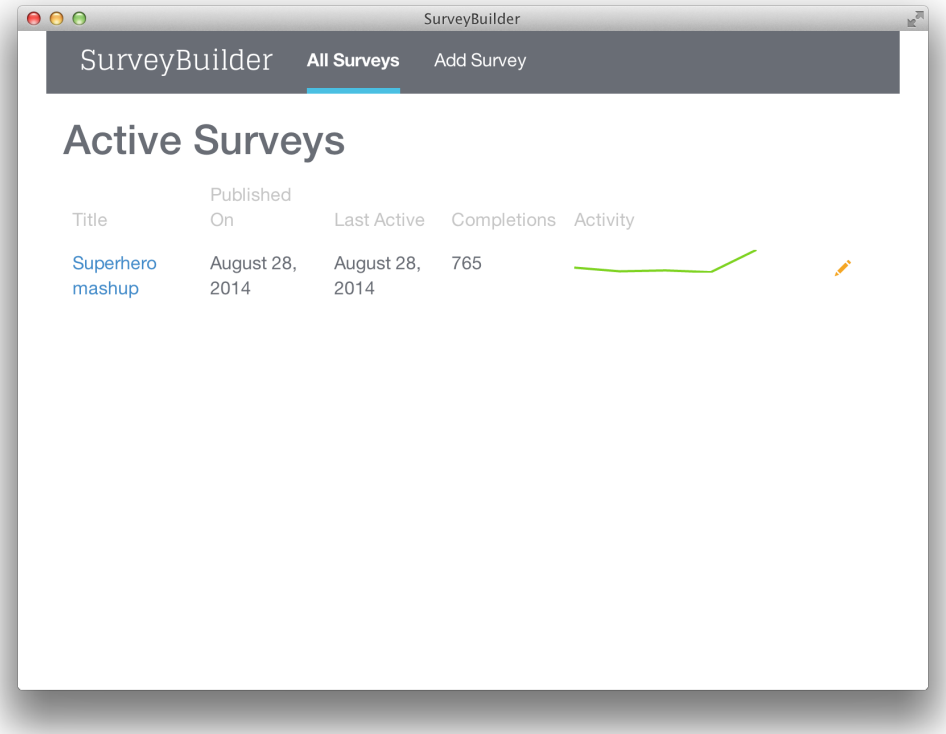
```
// Keep a global reference of the window object, if you don't, the window will
// be closed automatically when the javascript object is GCed.
var mainWindow = null;

// Quit when all windows are closed.
app.on('window-all-closed', function() {
  if (process.platform != 'darwin')
    app.quit();
});

// This method will be called when atom-shell has done everything
// initialization and ready for creating browser windows.
app.on('ready', function() {
  // Create the browser window.
  mainWindow = new BrowserWindow({width: 800, height: 600});

  // and load the index.html of the app.
  //mainWindow.loadUrl('file://' + __dirname + '/index.html');
  mainWindow.loadUrl('http://localhost:8080/');

  // Emitted when the window is closed.
  mainWindow.on('closed', function() {
    // Dereference the window object, usually you would store windows
    // in an array if your app supports multi windows, this is the time
    // when you should delete the corresponding element.
    mainWindow = null;
  });
});
```



With projects like atom-shell and node-webkit we can build desktop applications using the same technologies we use for the web. Just like for the web, React can help you build powerful interactive applications for the Desktop.

Email

Though React is optimized for building interactive UIs for the web, at it's core it renders HTML. This means we can get a lot of the same benefits we'd normally get from writing React application for something as terrible as writing HTML emails.

Building HTML emails requires a series of tables to render correctly in each email client. To write emails you need to turn back the clock a few years and write HTML as if it were 1999.

Successfully rendering emails in a range of email clients is no small feat. To build our design with React we will only touch on a number of challenges you will encounter when building emails, wether they are rendered using React or not.

The core principle of rendering html for emails with React is `React.renderComponentToStaticMarkup`. This function returns an HTML string containing the full component tree, given 1 top level component. The only difference between `React.renderComponentToStaticMarkup` and `React.renderComponentToString` is that `React.renderComponentToStaticMarkup` doesn't create extra DOM attributes like `data-react-id` that React uses client side to keep track of the DOM. Since the email doesn't run client side in the browser - we have no need for those attributes.

Lets build an email with React given this design for desktop and mobile:

Who is your favorite superhero?

3123
Completions

14
Days running

Who is your favorite superhero?

3123
Completions

14
Days running

To render our email we have made a small script that outputs HTML that can be used to send an email:

```
// render_email.js
var React = require('react');
var SurveyEmail = require('survey_email');
var survey = {};

console.log(
  React.renderComponentToStaticMarkup(
    <SurveyEmail survey={survey}/>
  )
);
```

Lets get the core structure of SurveyEmail going

First lets build an Email component:

```
var Email = React.createClass({
  render: function () {
    return (
      <html>
        <body>
          {this.prop.children}
        </body>
      </html>
    );
  }
});
```

The `<SurveyEmail/>` component uses `<Email/>`

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    return (
      <Email>
        <h2>{survey.title}</h2>
      </Email>
    );
  }
});
```

Next, per the design we want to render 2 KPIs next to each other on desktop clients and stacked on a mobile device. Each KPI look similar in structure so they could share the same component:

```
var SurveyEmail = React.createClass({
  render: function () {
```

```
        <tr>
          <td>{this.props.kpi}</td>
        </tr>
        <tr>
          <td>{this.props.label}</td>
        </tr>
      </table>
    );
  }
});
```

Let's add them to the `<SurveyEmail/>` component:

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    var completions = survey.activity.reduce(function (memo, ac) {
      return memo + a;
    }, 0);
    var daysRunning = survey.activity.length;

    return (
      <Email>
       <h2>{survey.title}</h2>
       <KPI kpi={completions} label='Completions'/>
       <KPI kpi={daysRunning} label='Days running'/>
     </Email>
    );
  }
});
```

This stacks our KPIs, but our design had them next to each other for desktop. The challenge now is to both have this work for desktop and mobile, and to solve this there are a few gotchas we have to cover first.

Lets augment `<Email/>` with a way of adding a CSS file:

```
var fs = require('fs');
var Email = React.createClass({
  propTypes: {
    responsiveCSSFile: React.PropTypes.string
  },
  render: function () {
    var responsiveCSSFile = this.props.responsiveCSSFile;
    var styles;
    if (responsiveCSSFile) {
      styles = <style>{fs.readFileSync(responsiveCSSFile)}</style>;
    }
    return (
      <html>
        <body>
```

```
        </body>
      </html>
    );
  }
});
```

The complete `<SurveyEmail/>` looks like this:

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },

  render: function () {
    var survey = this.props.survey;
    var completions = survey.activity.reduce(function (memo, ac) {
      return memo + a;
    }, 0);

    var daysRunning = survey.activity.length;

    return (
      <Email responsiveCSS='path/to/mobile.css'>
        <h2>{survey.title}</h2>
        <table className='for-desktop'>
          <tr>
            <td>
              <KPI kpi={completions} label='Completions'/>
            </td>
            <td>
              <KPI kpi={daysRunning} label='Days running'/>
            </td>
          </tr>
        </table>
        <div className='for-mobile'>
          <KPI kpi={completions} label='Completions'/>
          <KPI kpi={daysRunning} label='Days running'/>
        </div>
      </Email>
    );
  }
});
```

We grouped the Email into ‘for-desktop’ and ‘for-mobile’. Sadly we can’t use something like float: left in emails since that isn’t supported by most browsers and The HTML spec calls out the align and valign properties as being obsolete and therefor React doesn’t support those properties, thought they could have provided a similar implementation to floating 2 divs. Instead we are left with 2 groups which we can target with responsive style sheets to hide or show depending on the screen size.

Even thought we have to use tables it’s clear that using React for rendering emails gives us a lot of the same benefits from writing interactive UIs for a browser: Reusable, composable and testable components.

Charting

For our demo application we want to chart the number of completions of a survey pr day. We want this represented as a simple Sparkline in our table of surveys to at a glance show the attendance of our survey.

React has support for SVG tags and thus making a simple SVG becomes trivial.

To render a Sparkline, we need only a `<Path/>` with a set of instructions.

The complete example looks like this:

```
var Sparkline = React.createClass({
  propTypes: {
    points: React.PropTypes.arrayOf(React.PropTypes.number).isRequired
  },
  render: function () {
    var width = 200;
    var height = 20;
    var path = this.generatePath(width, height, this.props.points);

    return (
      <svg width={width} height={height}>
        <path d={path} stroke='#7ED321' strokeWidth='2' fill='none'/>
      </svg>
    );
  },

  generatePath: function (width, height, points) {
    var maxHeight = arrMax(points);
    var maxWidth = points.length;

    return points.map(function (p, i) {
      var xPct = i / maxWidth * 100;
      var x = (width / 100) * xPct;
      var yPct = 100 - (p / maxHeight * 100);
      var y = (height / 100) * yPct;

      if (i === 0) {
        return 'M0,' + y;
      }
      else {
        return 'L' + x + ',' + y;
      }
    }).join(' ');
  }
});
```

The Sparkline component above requires an array of numbers that represents the points. It then builds a simple svg with a path.

The interesting part is in the generatePath function which computes where each point should be rendered and returns an svg path description.

It returns a string like so “M0,30 L10,20 L20,50”. SVG paths translates this into drawing commands. Each command is separated by a blank space. “M0,30” means Move cursor to x0 and y30. Then “L10,20” means draw a line from the current cursor to x10 and y20 and so on.

It can be tedious to writing scale functions like this for larger charts, but its quite simple to drop in libraries like d3 and use the scale functions d3 provides instead of manual creating the path.