MARCH 25TH, 2015 (/authors/Ben Lewis) By Ben Lewis (/authors/Ben Lewis)
TAGS: Javascript (/categories/javascript)

Integrating React With Backbone ()

There are so many JS frameworks! It can get tiring (http://www.allenpike.com/2015/javascript-framework-fatigue/) to keep up to date with them all.

But like any developer who writes JavaScript, I try to keep abreast of the trends. I like to tinker with new things, and rebuild TodoMVC (http://todomvc.com/) as often as possible.

Joking aside, when it comes to choosing frameworks for a project, emerging frameworks just haven't been battle-tested enough for me to recommend to clients in most cases.

But like much of the community, I feel pretty confident in the future of React (http://facebook.github.io/react/index.html). It's well documented, makes reasoning about data easy, and it's performant.

Since React only provides the view layer of a client-side MVC application (http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller), I still have to find a way to wrap the rest of the application. When it comes to choosing a library that I'm

confident in, I still reach for BackboneJS (http://backbonejs.org/). A company that bets on Backbone won't have trouble finding people who can work on their code base. It's been around for a long time, is unopionated enough to be adaptable to many different situations. And as an added bonus, it plays well with React.

In this post, we'll explore the relationship between Backbone and React, by looking at one way to structure a project that uses them together.

A Note About Setting Up Dependencies

I won't go over setting up all of the package dependencies for the project here, since I've covered this process in a previous post (http://fluxusfrequency.github.io/blog/2015/02/04/setting-up-a-client-side-javascript-project-with-gulp-and-browserify/). For the purposes of this article, you can assume that we're using Browserify (http://browserify.org/).

One package that is worth noting, though, is ReactBackbone (https://github.com/clayallsopp/react.backbone). It will allow us to trigger an automatic update of our React components whenever Backbone model or collection data changes. You can get it with `npm install --save react.backbone`.

We'll also be making use of backbone-route-control (https://www.npmjs.com/package/backbone-route-control) to make it easier to split our URL routes into logically encapsulated controllers. See "caching the controller call" in this article (http://fluxusfrequency.github.io/blog/2014/12/09/caching-asynchronous-queries-in-backbone/) for more information about how to set this package up.

Project Structure

There are many ways to structure the directories for a client-side JS application, and every project lends itself to a slightly different setup. Today we'll be creating our directory structure in a fairly typical fashion for a Backbone project. But we'll also be introducing the concept of *screens* to our application, so we'll also be extending it slightly.

Here's what we'll need:

```
assets/
  |- js/
     |- collections/
     |- components/
     |- controllers/
     |- models/
     |- screens/
     |- vendor/
     |- app.js
     |- base-view.js
     |- index.js
     |- router.js
```

Much of this is standard Backbone boilerplate. The `collections/`, `models/`, and `vendor/`

Much of this is standard Backbone boilerplate. The `collections/`, `models/`, and `vendor/` directories are self-explanatory. We'll store reusable UI components, such as pagination, pills, and toggles, in `components/`.

The heart of our app will live in the `screens/` directory. Here, we'll write React components that will handle the display logic, taking the place of traditional Backbone views and templates. However, we'll still include thin Backbone views to render these components.

We'll talk more about screens in a moment. For now, let's take a look at the how a request will flow through the application, starting from the macro level.

The Application

We'll begin by writing a root-level `index.js file`, which will be the source of the `require` tree that Browserify will use.

```
window.$ = window.jQuery = require('jquery');
var Application = require('./app');

window.app = new Application();
```

What is this `Application`, you may ask? Simply put, it's the function we'll use to bootstrap the entire project. Its purpose is to get all of the dependencies set up, instantiate the controllers and router, kick off Backbone history, and render the main view.

```
var Backbone = require('backbone');

var Router = require('./router');
var MainView = require('./screens/main/index');

var UsersController = require('./controllers/users-controller');

Backbone.$ = $;

var Application = function() {
  this.initialize();
};

Application.prototype.initialize = function() {
  this.controllers = {
    users: new UsersController({ app: this })
  };

  this.router = new Router({
    app: this,
    controllers: this.controllers
  });

  this.mainView = new MainView({
    el: $('#app'),
    router: this.router
  });

  this.showApp();
```

```
};

Application.prototype.showApp = function() {
  this.mainView.render();
  Backbone.history.start({ pushState: true });
};

module.exports = Application;
```

ROUTER

Once the application has been booted up, we'll want to be able to accept requests. When one comes in, our app will need to be able to take a look at the URL path in the navigation bar and decide what to do. This is where the router comes in. It's a pretty standard part of any Backbone project, so it probably won't look too out of the ordinary, especially if you've used `backbone-route-control` before.

```
var Backbone = require('backbone');
var BackboneRouteControl = require('backbone-route-control');

var Router = BackboneRouteControl.extend({
  routes: {
    '':          'users#index',
    'users':     'users#index',
    'users/:id': 'users#show'
  }
});

module.exports = Router;
```

When one of these routes is hit, the router will take a look at the controllers we passed into it during app initialization, find the controller with the name to the left of the **#** and try to call the method name to the right of the **#** in the string defined for that route.

CONTROLLERS

Now that the request has been routed through one of the routes, the router will take a look in the matching controller for the method that is to be called. Note that these controllers are not a part of Backbone, but Plain Old JavaScript Objects.

For the purposes of this post, we'll just have a `UsersController` with two actions.

```
var UsersCollection = require('../collections/users-collection');
var UserModel = require('../models/user');
var UsersIndexView = require('../screens/users/index');
var UserShowView = require('../screens/users/show');

var UsersController = function(options) {
  var app = options.app;

  return {
    index: function() {
      var usersCollection = new UsersCollection();

      usersCollection.fetch().done(function() {
        var usersView = new UsersIndexView({
          users: usersCollection
        });
        app.mainView.pageRender(usersView);
      });
    },

    show: function(id) {
      var user = new UserModel({
        id: id
      });

      user.fetch().done(function() {
        var userView = new UserShowView({
          user: user
        });
        app.mainView.pageRender(userView);
      });
    }
  };
};

module.exports = UsersController;
```

This controller loads the `User` model and collection, and uses them to display the user index and show screens. It instantiates a Backbone collection or model, depending on the route, fetches its data from the server, loads it into the screen (which we'll get to momentarily), then shows that screen in the app's `mainView` container.

Screens

At this point, we've accepted a request, routed it through a controller action, decided what

kind of collection or model we are dealing with, and fetched the data from the server. We're ready to render a Backbone view. In this case, it will do little more than pass the data on to the

React component.

THE BASE VIEW

Since there's going to be a lot of repeated boilerplate in our Backbone views, it makes sense to abstract it out into a `BaseView`, which child views will extend from.

```
var React = require('react');
var Backbone = require('backbone');

var BaseView = Backbone.View.extend({
  initialize: function (options) {
    this.options = options || {};
  },

  component: function () {
    return null;
  },

  render: function () {
    React.renderComponent(this.component(), this.el);
    return this;
  }
});

module.exports = BaseView;
```

This base view sets any options passed in as properties on itself, and defines a `render()` method that renders whatever React component is defined in the `component()` method.

THE MAIN VIEW

In order to switch between screens without doing a page re-render, we'll wrap all of our screens in an outer screen called the `mainView`. This view acts as a sort of "picture frame" for the other screens in the app, displaying, hiding, and cleaning them up.

As with all of our screens, it will consist of two parts: a Backbone view, defined in `screens/main/index.js`, and a React component, defined in `screens/main/component.js`.

Backbone View

```
var Backbone = require('backbone');
var BaseView = require('../../base-view');
var MainComponent = require('./component');

var MainView = BaseView.extend({
```

```
  component: function () {
    return new MainComponent({
      router: this.options.router
    });
  },

  pageRender: function (view) {
    this.$('#main-container').html(view.render().$el);
  }
});

module.exports = MainView;
```

Since we passed `#app` as the element for this view to attach to back in `app.js`, it will render itself there. Thinking through what `render` actually means, we know that it will call the code defined in the `BaseView`, which means it will render the whatever's returned by the `component()` function. In this case, it's the React `MainComponent`. We'll take a look at that in a moment.

The other special thing this view does is to render any subviews passed to `pageRender` in the `#main-container` element found within `#app`. As I said, it's basically just a frame for whatever else is going to happen.

React Component

Now let's take a look at that `MainComponent`. It's a very simple React component that does nothing more than render the "container" into the DOM.

```
/** @jsx React.DOM */
var React = require('react');
var ReactBackbone = require('react.backbone');

var MainComponent = React.createBackboneClass({
  render: function () {
    return (
      <div>
        <div id="main-container"></div>
      </div>
    );
  }
});

module.exports = MainComponent;
```

That's it, the whole `MainView`. Since it's so simple, it makes a good introduction to how we can render components in this project.

Now let's take a look at something a little more advanced.

USER SHOW VIEW

We'll start by taking a look at how we might write a React component for a user show page.

Backbone View

First, we'll define the `UserShowView` we referenced back in the `UsersController`. It should live at `screens/users/show/index.js`.

```
var BaseView = require('../../../base-view');
var UserScreen = require('./component');

var UserView = BaseView.extend({
  component: function () {
    return new UserScreen({
      user: this.options.user
    });
  }
});

module.exports = UsersView;
```

That's it. Mostly just boilerplate. In fact, pretty much all of our Backbone views will look like this. A simple extension of `BaseView` that defines a `component()` method. That method instantiates a React component and returns it to the `render()` method in the `BaseView`, which in turn is called by the `mainView`'s `pageRender()` method.

React Component

Now, let's dig into the meat of user show screen: the `UserScreen` component. It will live at `screens/users/show/component.js`.

We'll imagine that we can "like" users. We want to be able to increment a user's `likes` attribute by clicking a button. Here's how we'd write this component to handle that behavior.

```
/** @jsx React.DOM */
var React = require('react');
var Backbone = require('backbone');
var ReactBackbone = require('react.backbone');
var User = require('../../../models/user');

var UserShowScreen = React.createBackboneClass({
  mixins: [
     React.BackboneMixin('user', 'change'),
  ],

  getInitialState: function() {
    return {
      liked: false
    }
  },

  handleLike: function(e) {
    e.preventDefault();
    var currentLikes = this.props.user.get('likesCount');
    this.props.user.save({ likesCount: currentLikes + 1 });
  },

  render: function() {
    var user = this.props.user;
    var username = user.get('username');
    var avatar = user.get('avatar').url;
    var likesCount = user.get('likesCount');

    return (
      <div className="user-container">
        <h1>{username}'s Profile</h1>
        <img src={avatar} alt={username} />
        <p>{likesCount} likes</p>
        <button className="like-button" onClick={this.handleLike}>
```

```
          Like
        </button>
      </div>
    );
  }
});

module.exports = UserShowScreeen;
```

You may have noticed that curious `mixins` property. What is that? `react.backbone` gives us some niceties here, since we're calling `React.createBackboneClass` instead of `React.createClass`. Whenever the `user` prop that was passed into this component fires a `change` event, the component's `render()` method will be called. For more information, take a look at the package on GitHub (https://github.com/clayallsopp/react.backbone).

When we click that like button, we're incrementing the `likesCount` attribute on the user, and saving it to the server with our `save()` call. When the result of that `sync` comes back, our view will automatically re-render, and the likes count indication will update! Pretty sweet!

USERS INDEX SCREEN

Before we conclude this post, lets take a look at one more case: the index screen. Here, we'll see how using React can make it easier to render repetitive subcomponents.

Backbone View

The view for this screen will live at `/screens/users/index/index.js`, and look similar to the `UserShowView`.

```
var BaseView = require('../../../base-view');
var UsersIndexScreen = require('./component');

var UsersIndexScreen = BaseView.extend({
  component: function () {
    return new UsersIndexScreen({
      users: this.options.users
    });
  }
});

module.exports = UsersIndexView;
```

Backbone Component

The `UsersIndexScreen` component will also be fairly similar to the `UserShowScreen` one, but with one key difference: since we're going to be rendering the same DOM elements repeatedly, we can leverage subcomponents.

Here's the main component, which lives at `screens/users/index/component.js`:

```
/** @jsx React.DOM */
var React = require('react');
var ReactBackbone = require('react.backbone');
var UserBlock = require('.user-block');

var UsersIndexScreen = React.createBackboneClass({
  mixins: [
     React.BackboneMixin('users', 'change')
  ],

  render: function() {
    var userBlocks = this.props.users.map(function(user) {
      return <UserBlock user={user} />
    });

    return (
      <div className="users-container">
        <h1>Users</h1>
        {userBlocks}
      </div>
    );
  }
});

module.exports = UsersIndexScreen;
```

We're just looping through the users that were passed into the component, and wrapping each one in a `UserBlock` React component. This component can be defined in a file that lives right alongside `index.js` and `component.js`.

```
/** @jsx React.DOM */
var React = require('react');
```

```
var Backbone = require('backbone');
var ReactBackbone = require('react.backbone');

var MemberBlock = React.createBackboneClass({
  render: function () {
    var user = this.props.user;
    var username = user.get('username');
    var avatar = user.get('avatar').url;
    var link = '/users/' + user.get('id');

    return (
      <div className="user-block">
        <a href={link}>
          <h2>{username}</h2>
          <img src={avatar} alt={username} />
        </a>
      </div>
    );
  }
});

module.exports = UserBlock;
```

Voila! An index view at `/users` that shows all of our users' beautiful faces and links to their show pages. It was pretty painless, thanks to React!

Wrapping Up

We've now traced the entire series of events that happens when someone loads up our application and requests a route. After going through the router and controller, the fetched data is injected through a Backbone view into a React component, which is then rendered by the app's `mainView`.

We only barely scratched the surface of what React is capable of here.

If you haven't checked out the React component API docs (http://facebook.github.io/react/docs/component-api.html), I highly suggest doing so. Once I began to fully harness the power it gave me, I found my projects' view layers much cleaner.

Plus, I get all of the React performance benefits (http://facebook.github.io/react/docs/advanced-performance.html) for free!

I hope that this post has helped make it more obvious how to get started with integrating React into a Backbone app. To me, it always seemed like a good idea, but I didn't know where to begin. Once I got a sense of the pattern, though, it became pretty easy to do.

P.S. Do you have a different pattern for using React in your Backbone app? Want to talk about using React in Ember or Angular? Leave us a note in the comments!

Share your thoughts with @engineyard on Twitter (https://www.twitter.com/intent/tweet?text=@engineyard%20&related=engineyard&url=https://blog.engineyard.com/2015/integrating-react-with-backbone)

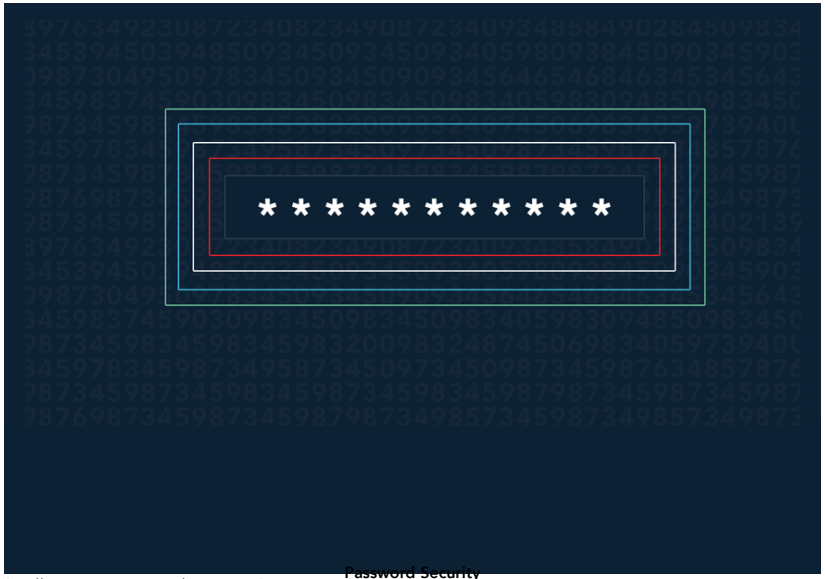
OR

Talk about it on reddit (http://www.reddit.com/r/javascript/comments/30amaa/integrating_react_with_backbone/)

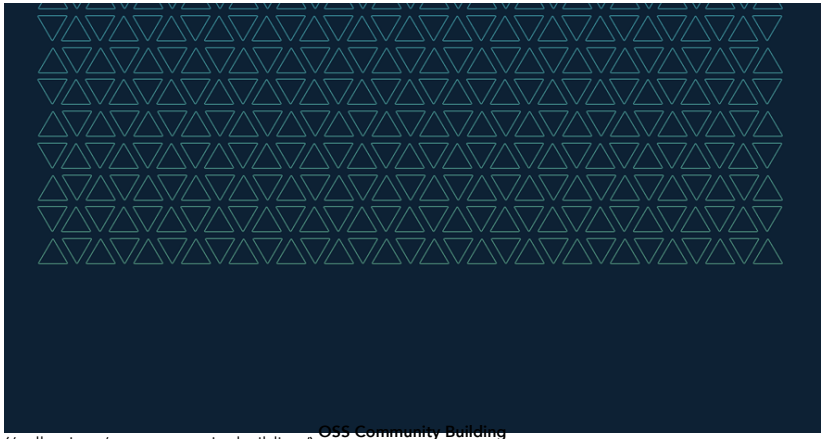
About Ben Lewis

Ben, a Colorado native and a third-generation programmer, lived many lives before following the footsteps of his forebears. After several years as an organic farm worker, gardener, and elementary school music teacher, he fell in love with coding one spring break and never looked back. A graduate of the second gSchool class, he prides himself in a dedication to Test-Driven Development, and specializes in Single Page Apps and Service-Oriented Architecture. He lives in Boulder with his partner Monica and their daughter Lumin, and enjoys cooking, gardening, hiking, and riding his bike. He continues to pursue music as a regional bluegrass fiddle performer. Ben tweets as @fluxusfrequency (http://twitter.com/fluxusfrequency) and works for Quick Left (http://quickleft.com).

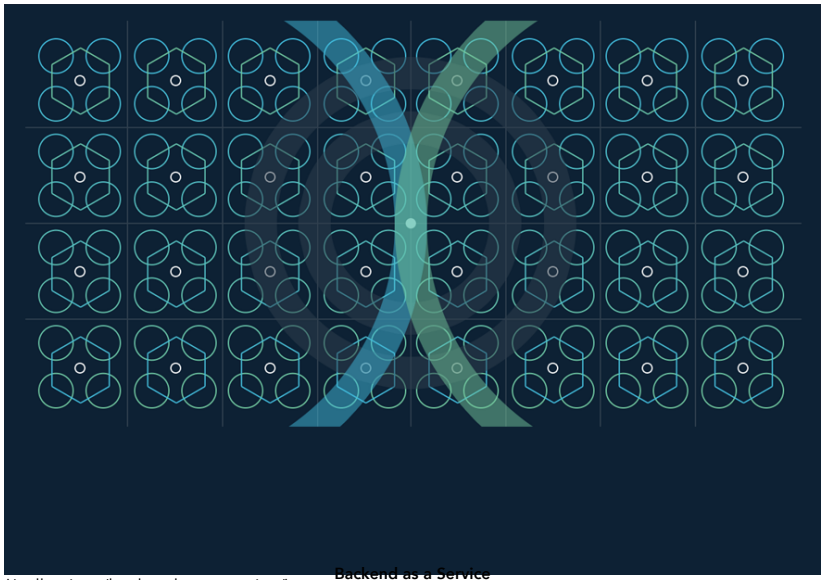
Extending MySQL with PHP's MySQLnd
(/collections/extending-mysql/)



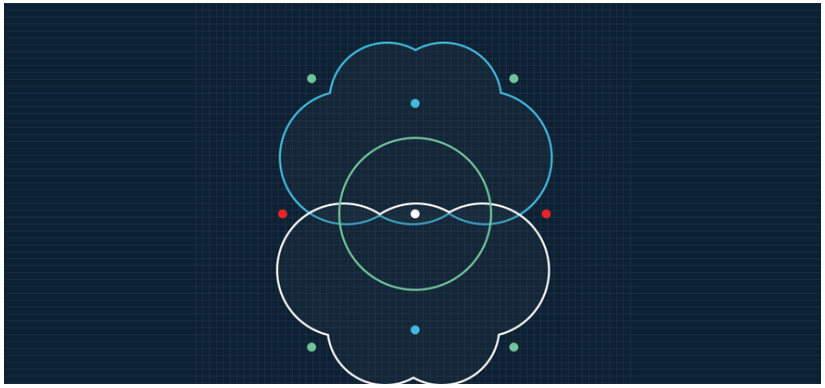
Password Security
(/collections/password-security/)

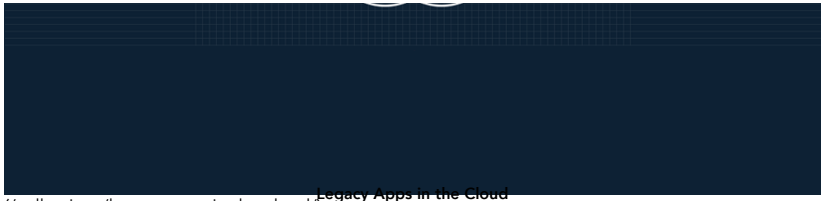


OSS Community Building
(/collections/oss-community-building/)

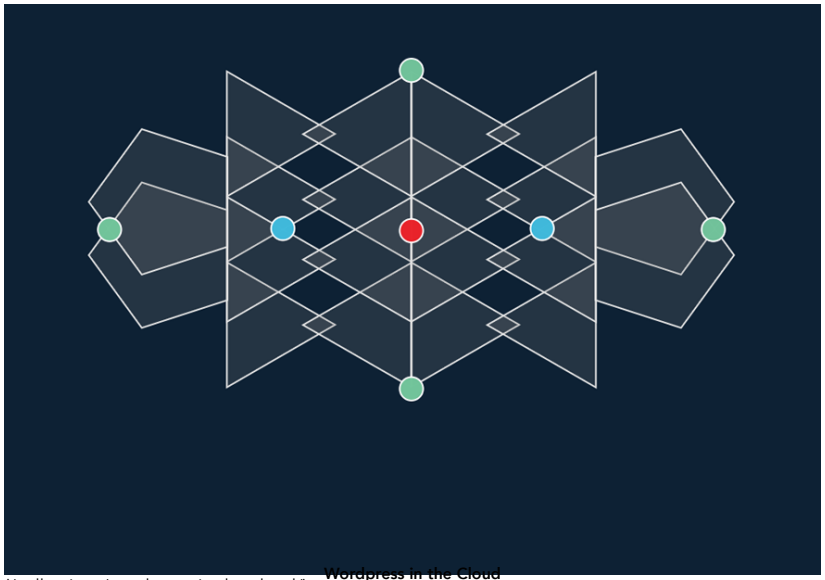


Backend as a Service
(/collections/backend-as-a-service/)

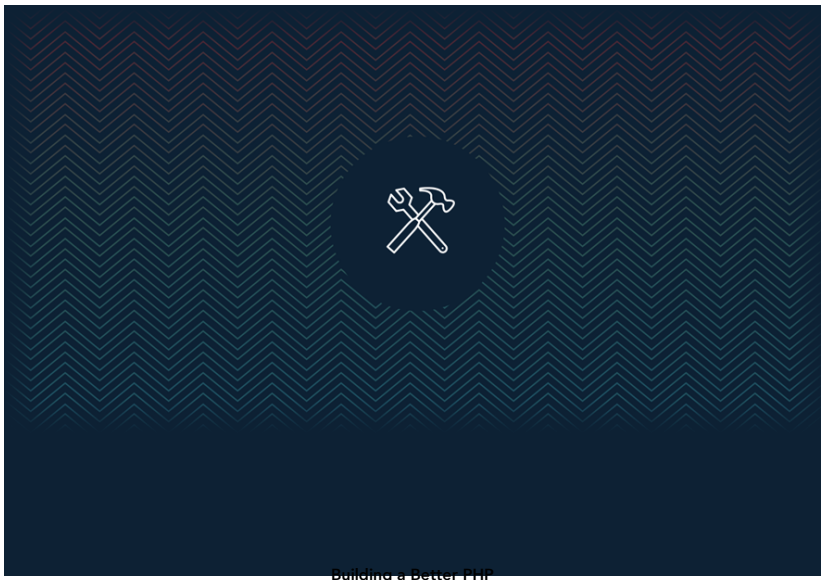




Legacy Apps in the Cloud
(/collections/legacy-apps-in-the-cloud/)



Wordpress in the Cloud
(/collections/wordpress-in-the-cloud/)



Building a Better PHP
(/collections/building-a-better-php/)