

Integrating with Other Libraries

React can be used in any web application. It can be embedded in other applications and, with a little care, other applications can be embedded in React. This guide will examine some of the more common use cases, focusing on integration with [jQuery](#) and [Backbone](#), but the same ideas can be applied to integrating components with any existing code.

Integrating with DOM Manipulation Plugins

React is unaware of changes made to the DOM outside of React. It determines updates based on its own internal representation, and if the same DOM nodes are manipulated by another library, React gets confused and has no way to recover.

This does not mean it is impossible or even necessarily difficult to combine React with other ways of affecting the DOM, you just have to be mindful of what each is doing.

The easiest way to avoid conflicts is to prevent the React component from updating. You can do this by rendering elements that React has no reason to update, like an empty `<div />`.

How to Approach the Problem

To demonstrate this, let's sketch out a wrapper for a generic jQuery plugin.

We will attach a `ref` to the root DOM element. Inside `componentDidMount`, we will get a reference to it so we can pass it to the jQuery plugin.

To prevent React from touching the DOM after mounting, we will return an empty `<div />` from the `render()` method. The `<div />` element has no properties or children, so React has no reason to update it, leaving the jQuery plugin free to manage that part of the DOM:

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

Note that we defined both `componentDidMount` and `componentWillUnmount` lifecycle methods. Many jQuery plugins attach event listeners to the DOM so it's important to detach them in `componentWillUnmount`. If the plugin does not provide a method for cleanup, you will probably have to provide your own, remembering to remove any event listeners the plugin registered to prevent memory leaks.

Integrating with jQuery Chosen Plugin

For a more concrete example of these concepts, let's write a minimal wrapper for the plugin Chosen, which augments `<select>` inputs.

Note:

Just because it's possible, doesn't mean that it's the best approach for React apps. We encourage you to use React components when you can. React components are easier to reuse in React applications, and often provide more control over their behavior and appearance.

First, let's look at what Chosen does to the DOM.

If you call it on a `<select>` DOM node, it reads the attributes off of the original DOM node, hides it with an inline style, and then appends a separate DOM node with its own visual representation right after the `<select>`. Then it fires jQuery events to notify us about the changes.

Let's say that this is the API we're striving for with our `<Chosen>` wrapper React component:

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

We will implement it as an uncontrolled component for simplicity.

First, we will create an empty component with a `render()` method where we return `<select>` wrapped in a `<div>`:

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

Notice how we wrapped `<select>` in an extra `<div>`. This is necessary because `Chosen` will append another DOM element right after the `<select>` node we passed to it. However, as far as React is concerned, `<div>` always only has a single child. This is how we ensure that React updates won't conflict with the extra DOM node appended by `Chosen`. It is important that if you modify the DOM outside of React flow, you must ensure React doesn't have a reason to touch those DOM nodes.

Next, we will implement the lifecycle methods. We need to initialize `Chosen` with the ref to the `<select>` node in `componentDidMount`, and tear it down in `componentWillUnmount`:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
```

```
this.$el.chosen('destroy');
}
```

Try it on CodePen

Note that React assigns no special meaning to the `this.el` field. It only works because we have previously assigned this field from a `ref` in the `render()` method:

```
<select className="Chosen-select" ref={el => this.el = el}>
```

This is enough to get our component to render, but we also want to be notified about the value changes. To do this, we will subscribe to the jQuery `change` event on the `<select>` managed by Chosen.

We won't pass `this.props.onChange` directly to Chosen because component's props might change over time, and that includes event handlers. Instead, we will declare a `handleChange()` method that calls `this.props.onChange`, and subscribe it to the jQuery `change` event:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}
```

Try it on CodePen

Finally, there is one more thing left to do. In React, props can change over time. For example, the `<Chosen>` component can get different children if parent component's state changes.



means that at integration points it is important that we manually update the DOM in response to prop updates, since we no longer let React manage the DOM for us.

Chosen's documentation suggests that we can use jQuery `trigger()` API to notify it about changes to the original DOM element. We will let React take care of updating `this.props.children` inside `<select>`, but we will also add a `componentDidUpdate()` lifecycle method that notifies Chosen about changes in the children list:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

This way, Chosen will know to update its DOM element when the `<select>` children managed by React change.

The complete implementation of the `Chosen` component looks like this:

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (

```



```
<div>
  <select className="Chosen-select" ref={el => this.el = el}>
    {this.props.children}
  </select>
</div>
);
}
}
```

[Try it on CodePen](#)

Integrating with Other View Libraries

React can be embedded into other applications thanks to the flexibility of `ReactDOM.render()`.

Although React is commonly used at startup to load a single root React component into the DOM, `ReactDOM.render()` can also be called multiple times for independent parts of the UI which can be as small as a button, or as large as an app.

In fact, this is exactly how React is used at Facebook. This lets us write applications in React piece by piece, and combine them with our existing server-generated templates and other client-side code.

Replacing String-Based Rendering with React

A common pattern in older web applications is to describe chunks of the DOM as a string and insert it into the DOM like so: `$el.html(htmlString)`. These points in a codebase are perfect for introducing React. Just rewrite the string based rendering as a React component.

So the following jQuery implementation...

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```



...could be rewritten using a React component:

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

ReactDOM.render(
  <Button />,
  document.getElementById('container'),
  function() {
    $('#btn').click(function() {
      alert('Hello!');
    });
  }
);
```

From here you could start moving more logic into the component and begin adopting more common React practices. For example, in components it is best not to rely on IDs because the same component can be rendered multiple times. Instead, we will use the [React event system](#) and register the click handler directly on the React `<button>` element:

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}

ReactDOM.render(
  <HelloButton />,
  document.getElementById('container')
);
```

Try it on CodePen

You can have as many such isolated components as you like, and use `ReactDOM.render()` to render them to different DOM containers. Gradually, as you convert more of your app to React, you will be able to combine them into larger components, and move some of the `ReactDOM.render()` calls up the hierarchy.



Embedding React in a Backbone View

Backbone views typically use HTML strings, or string-producing template functions, to create the content for their DOM elements. This process, too, can be replaced with rendering a React component.

Below, we will create a Backbone view called `ParagraphView`. It will override Backbone's `render()` function to render a React `<Paragraph>` component into the DOM element provided by Backbone (`this.el`). Here, too, we are using `ReactDOM.render()`:

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  render() {
    const text = this.model.get('text');
    ReactDOM.render(<Paragraph text={text} />, this.el);
    return this;
  },
  remove() {
    ReactDOM.unmountComponentAtNode(this.el);
    Backbone.View.prototype.remove.call(this);
  }
});
```

Try it on CodePen

It is important that we also call `ReactDOM.unmountComponentAtNode()` in the `remove` method so that React unregisters event handlers and other resources associated with the component tree when it is detached.

When a component is removed *from within* a React tree, the cleanup is performed automatically, but because we are removing the entire tree by hand, we must call this method.

Integrating with Model Layers

While it is generally recommended to use unidirectional data flow such as React state, Flux, or Redux, React components can use a model layer from other frameworks and libraries.



Using Backbone Models in React Components

The simplest way to consume Backbone models and collections from a React component is to listen to the various change events and manually force an update.

Components responsible for rendering models would listen to 'change' events, while components responsible for rendering collections would listen for 'add' and 'remove' events. In both cases, call this.forceUpdate() to rerender the component with the new data.

In the example below, the `List` component renders a Backbone collection, using the `Item` component to render individual items.

```
class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }
}
```



```

componentWillUnmount() {
  this.props.collection.off('add', 'remove', this.handleChange);
}

render() {
  return (
    <ul>
      {this.props.collection.map(model => (
        <Item key={model.cid} model={model} />
      ))}
    </ul>
  );
}
}

```

[Try it on CodePen](#)

Extracting Data from Backbone Models

The approach above requires your React components to be aware of the Backbone models and collections. If you later plan to migrate to another data management solution, you might want to concentrate the knowledge about Backbone in as few parts of the code as possible.

One solution to this is to extract the model's attributes as plain data whenever it changes, and keep this logic in a single place. The following is a [higher-order component](#) that extracts all attributes of a Backbone model into state, passing the data to the wrapped component.

This way, only the higher-order component needs to know about Backbone model internals, and most components in the app can stay agnostic of Backbone.

In the example below, we will make a copy of the model's attributes to form the initial state. We subscribe to the `change` event (and unsubscribe on unmounting), and when it happens, we update the state with the model's current attributes. Finally, we make sure that if the `model` prop itself changes, we don't forget to unsubscribe from the old model, and subscribe to the new one.

Note that this example is not meant to be exhaustive with regards to working with Backbone, but it should give you an idea for how to approach this in a generic way:

```

function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {

```

```

super(props);
this.state = Object.assign({}, props.model.attributes);
this.handleChange = this.handleChange.bind(this);
}

componentDidMount() {
  this.props.model.on('change', this.handleChange);
}

componentWillReceiveProps(nextProps) {
  this.setState(Object.assign({}, nextProps.model.attributes));
  if (nextProps.model !== this.props.model) {
    this.props.model.off('change', this.handleChange);
    nextProps.model.on('change', this.handleChange);
  }
}

componentWillUnmount() {
  this.props.model.off('change', this.handleChange);
}

handleChange(model) {
  this.setState(model.changedAttributes());
}

render() {
  const propsExceptModel = Object.assign({}, this.props);
  delete propsExceptModel.model;
  return <WrappedComponent {...propsExceptModel} {...this.state} />;
}
}
}

```

To demonstrate how to use it, we will connect a `NameInput` React component to a Backbone model, and update its `firstName` attribute every time the input changes:

```

function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      My name is {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {

```

```
function handleChange(e) {
  props.model.set('firstName', e.target.value);
}

return (
  <BackboneNameInput
    model={props.model}
    handleChange={handleChange}
  />
);

};

const model = new Backbone.Model({ firstName: 'Frodo' });
ReactDOM.render(
  <Example model={model} />,
  document.getElementById('root')
);
```

Try it on CodePen

This technique is not limited to Backbone. You can use React with any model library by subscribing to its changes in the lifecycle methods and, optionally, copying the data into the local React state.

Is this page useful?  

[Edit this page](#)

