Tutorial | React

We'll be building a simple but realistic comments box that you can drop into a blog, a basic version of the realtime comments offered by Disqus, LiveFyre or Facebook comments.

We'll provide:

- A view of all of the comments
- A form to submit a comment
- Hooks for you to provide a custom backend

It'll also have a few neat features:

- **Optimistic commenting:** comments appear in the list before they're saved on the server so it feels fast.
- **Live updates:** other users' comments are popped into the comment view in real time.
- **Markdown formatting:** users can use Markdown to format their text.

Running a server

In order to start this tutorial, we're going to require a running server. This will serve purely as an API endpoint which we'll use for getting and saving data. In order to make this as easy as possible, we've created a simple server in a number of scripting languages that does exactly what we need it to do. **You can view the source or download a zip file containing everything needed to get started.**

For sake of simplicity, the server we will run uses a `JSON` file as a database. You would not run this in production but it makes it easy to simulate what you might do when consuming an API. Once you start the server, it will support our API endpoint and it will also serve the static pages we need.

Getting started

For this tutorial, we're going to make it as easy as possible. Included in the server package discussed above is an HTML file which we'll work in. Open up `public/index.html` in your favorite editor. It should look something like this:

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
```

```
    <script src="https://unpkg.com/react@15.3.1/dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@15.3.1/dist/react-dom.js"></script>
    <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
    <script src="https://unpkg.com/jquery@3.1.0/dist/jquery.min.js"></script>
    <script src="https://unpkg.com/remarkable@1.6.2/dist/remarkable.min.js">
</script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
    <script type="text/babel">
      // To get started with this tutorial running your own code, simply remove
      // the script tag loading scripts/example.js and start writing code here.
    </script>
  </body>
</html>
```

For the remainder of this tutorial, we'll be writing our JavaScript code in this script tag. We don't have any advanced live-reloading so you'll need to refresh your browser to see updates after saving. Follow your progress by opening `http://localhost:3000` in your browser (after starting the server). When you load this for the first time without any changes, you'll see the finished product of what we're going to build. When you're ready to start working, just delete the preceding `<script>` tag and then you can continue.

> Note:
>
> We included jQuery here because we want to simplify the code of our future AJAX calls, but it's **NOT** mandatory for React to work.

Your first component

React is all about modular, composable components. For our comment box example, we'll have the following component structure:

```
- CommentBox
  - CommentList
    - Comment
  - CommentForm
```

Let's build the `CommentBox` component, which is just a simple `<div>`:

```
// tutorial1.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        Hello, world! I am a CommentBox.
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
  document.getElementById('content')
);
```

Note that native HTML element names start with a lowercase letter, while custom React class names begin with an uppercase letter.

JSX Syntax

The first thing you'll notice is the XML-ish syntax in your JavaScript. We have a simple precompiler that translates the syntactic sugar to this plain JavaScript:

```
// tutorial1-raw.js
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am a CommentBox."
      )
    );
  }
});
ReactDOM.render(
  React.createElement(CommentBox, null),
  document.getElementById('content')
);
```

Its use is optional but we've found JSX syntax easier to use than plain JavaScript. Read more on the JSX Syntax article.

What's going on

We pass some methods in a JavaScript object to `React.createClass()` to create a new React component. The most important of these methods is called `render` which returns a tree of React components that will eventually render to HTML.

The `<div>` tags are not actual DOM nodes; they are instantiations of React `div` components. You can think of these as markers or pieces of data that React knows how to handle. React is **safe**. We are not generating HTML strings so XSS protection is the default.

You do not have to return basic HTML. You can return a tree of components that you (or someone else) built. This is what makes React **composable**: a key tenet of maintainable frontends.

`ReactDOM.render()` instantiates the root component, starts the framework, and injects the markup into a raw DOM element, provided as the second argument.

The `ReactDOM` module exposes DOM-specific methods, while `React` has the core tools shared by React on different platforms (e.g., React Native).

It is important that `ReactDOM.render` remain at the bottom of the script for this tutorial. `ReactDOM.render` should only be called after the composite components have been defined.

Composing components

Let's build skeletons for `CommentList` and `CommentForm` which will, again, be simple `<div>`s. Add these two components to your file, keeping the existing `CommentBox` declaration and `ReactDOM.render` call:

```
// tutorial2.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = React.createClass({
  render: function() {
    return (
      <div className="commentForm">
```

```
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Next, update the `CommentBox` component to use these new components:

```
// tutorial3.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList />
        <CommentForm />
      </div>
    );
  }
});
```

Notice how we're mixing HTML tags and components we've built. HTML components are regular React components, just like the ones you define, with one difference. The JSX compiler will automatically rewrite HTML tags to `React.createElement(tagName)` expressions and leave everything else alone. This is to prevent the pollution of the global namespace.

Using props

Let's create the `Comment` component, which will depend on data passed in from its parent, `CommentList`. Data passed in from a parent component is available as a 'property' on the child component. These 'properties' are accessed through `this.props`. Using props, we will be able to read the data passed to the `Comment` from the `CommentList`, and render some markup:

```
// tutorial4.js
var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
```

```
        {this.props.children}
      </div>
    );
  }
});
```

By surrounding a JavaScript expression in braces inside JSX (as either an attribute or child), you can drop text or React components into the tree. We access named attributes passed to the component as keys on `this.props` and any nested elements as `this.props.children`.

Component Properties

Now that we have defined the `Comment` component, we will want to pass it the author name and comment text. This allows us to reuse the same code for each unique comment. Now let's add some comments within our `CommentList`:

```
// tutorial5.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        <Comment author="Pete Hunt">This is one comment</Comment>
        <Comment author="Jordan Walke">This is *another* comment</Comment>
      </div>
    );
  }
});
```

Note that we have passed some data from the parent `CommentList` component to the child `Comment` components. For example, we passed *Pete Hunt* (via the `author` attribute) and *This is one comment* (via an XML-like child node) to the first `Comment`. As noted above, the `Comment` component will access these 'properties' through `this.props.author`, and `this.props.children`.

Adding Markdown

Markdown is a simple way to format your text inline. For example, surrounding text with asterisks will make it emphasized.

In this tutorial we use a third-party library **remarkable** which takes Markdown text and converts it to raw HTML. We already included this library with the original markup for the page, so we can just start using it. Let's convert the comment text to Markdown and output it:

```
// tutorial6.js
var Comment = React.createClass({
  render: function() {
    var md = new Remarkable();
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {md.render(this.props.children.toString())}
      </div>
    );
  }
});
```

All we're doing here is calling the remarkable library. We need to convert `this.props.children` from React's wrapped text to a raw string that remarkable will understand so we explicitly call `toString()`.

But there's a problem! Our rendered comments look like this in the browser: "`<p>`This is `<em>`another`</em>` comment`</p>`". We want those tags to actually render as HTML.

That's React protecting you from an XSS attack. There's a way to get around it but the framework warns you not to use it:

```
// tutorial7.js
var Comment = React.createClass({
  rawMarkup: function() {
    var md = new Remarkable();
    var rawMarkup = md.render(this.props.children.toString());
    return { __html: rawMarkup };
  },

  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        <span dangerouslySetInnerHTML={this.rawMarkup()} />
      </div>
    );
```

```
  }
});
```

This is a special API that intentionally makes it difficult to insert raw HTML, but for remarkable we'll take advantage of this backdoor.

Remember: by using this feature you're relying on remarkable to be secure. In this case, remarkable automatically strips HTML markup and insecure links from the output.

Hook up the data model

So far we've been inserting the comments directly in the source code. Instead, let's render a blob of JSON data into the comment list. Eventually this will come from the server, but for now, write it in your source:

```
// tutorial8.js
var data = [
  {id: 1, author: "Pete Hunt", text: "This is one comment"},
  {id: 2, author: "Jordan Walke", text: "This is *another* comment"}
];
```

We need to get this data into `CommentList` in a modular way. Modify `CommentBox` and the `ReactDOM.render()` call to pass this data into the `CommentList` via props:

```
// tutorial9.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.props.data} />
        <CommentForm />
      </div>
    );
  }
});

ReactDOM.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

Now that the data is available in the `CommentList`, let's render the comments dynamically:

```
// tutorial10.js
var CommentList = React.createClass({
  render: function() {
    var commentNodes = this.props.data.map(function(comment) {
      return (
        <Comment author={comment.author} key={comment.id}>
          {comment.text}
        </Comment>
      );
    });
    return (
      <div className="commentList">
        {commentNodes}
      </div>
    );
  }
});
```

That's it!

Fetching from the server

Let's replace the hard-coded data with some dynamic data from the server. We will remove the data prop and replace it with a URL to fetch:

```
// tutorial11.js
ReactDOM.render(
  <CommentBox url="/api/comments" />,
  document.getElementById('content')
);
```

This component is different from the prior components because it will have to re-render itself. The component won't have any data until the request from the server comes back, at which point the component may need to render some new comments.

Note: the code will not be working at this step.

Reactive state

So far, based on its props, each component has rendered itself once. `props` are immutable: they are

passed from the parent and are "owned" by the parent. To implement interactions, we introduce mutable **state** to the component. `this.state` is private to the component and can be changed by calling `this.setState()`. When the state updates, the component re-renders itself.

`render()` methods are written declaratively as functions of `this.props` and `this.state`. The framework guarantees the UI is always consistent with the inputs.

When the server fetches data, we will be changing the comment data we have. Let's add an array of comment data to the `CommentBox` component as its state:

```
// tutorial12.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

`getInitialState()` executes exactly once during the lifecycle of the component and sets up the initial state of the component.

Updating state

When the component is first created, we want to GET some JSON from the server and update the state to reflect the latest data. We're going to use jQuery to make an asynchronous request to the server we started earlier to fetch the data we need. The data is already included in the server you started (based on the `comments.json` file), so once it's fetched, `this.state.data` will look something like this:

```
[
  {"id": "1", "author": "Pete Hunt", "text": "This is one comment"},
  {"id": "2", "author": "Jordan Walke", "text": "This is *another* comment"}
]
```

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

Here, `componentDidMount` is a method called automatically by React after a component is rendered for the first time. The key to dynamic updates is the call to `this.setState()`. We replace the old array of comments with the new one from the server and the UI automatically updates itself. Because of this reactivity, it is only a minor change to add live updates. We will use simple polling here but you could easily use WebSockets or other technologies.

```
// tutorial14.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
```

```
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});

ReactDOM.render(
  <CommentBox url="/api/comments" pollInterval={2000} />,
  document.getElementById('content')
);
```

All we have done here is move the AJAX call to a separate method and call it when the component is first loaded and every 2 seconds after that. Try running this in your browser and changing the `comments.json` file (in the same directory as your server); within 2 seconds, the changes will show!

Adding new comments

Now it's time to build the form. Our `CommentForm` component should ask the user for their name and comment text and send a request to the server to save the comment.

```
// tutorial15.js
var CommentForm = React.createClass({
  render: function() {
    return (
      <form className="commentForm">
        <input type="text" placeholder="Your name" />
        <input type="text" placeholder="Say something..." />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

Controlled components

With the traditional DOM, `input` elements are rendered and the browser manages the state (its rendered value). As a result, the state of the actual DOM will differ from that of the component. This is not ideal as the state of the view will differ from that of the component. In React, components should always represent the state of the view and not only at the point of initialization.

Hence, we will be using `this.state` to save the user's input as it is entered. We define an initial `state` with two properties `author` and `text` and set them to be empty strings. In our `<input>` elements, we set the `value` prop to reflect the `state` of the component and attach `onChange` handlers to them. These `<input>` elements with a `value` set are called controlled components. Read more about controlled components on the Forms article.

```
// tutorial16.js
var CommentForm = React.createClass({
  getInitialState: function() {
    return {author: '', text: ''};
  },
  handleAuthorChange: function(e) {
    this.setState({author: e.target.value});
  },
  handleTextChange: function(e) {
    this.setState({text: e.target.value});
  },
  render: function() {
    return (
      <form className="commentForm">
        <input
```

```
          type="text"
          placeholder="Your name"
          value={this.state.author}
          onChange={this.handleAuthorChange}
        />
        <input
          type="text"
          placeholder="Say something..."
          value={this.state.text}
          onChange={this.handleTextChange}
        />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

Events

React attaches event handlers to components using a camelCase naming convention. We attach `onChange` handlers to the two `<input>` elements. Now, as the user enters text into the `<input>` fields, the attached `onChange` callbacks are fired and the `state` of the component is modified. Subsequently, the rendered value of the `input` element will be updated to reflect the current component `state`.

(The astute reader may be surprised that these event handlers work as described, given that the method references are not explicitly bound to `this`. `React.createClass(...)` automatically binds each method to its component instance, obviating the need for explicit binding.)

Submitting the form

Let's make the form interactive. When the user submits the form, we should clear it, submit a request to the server, and refresh the list of comments. To start, let's listen for the form's submit event and clear it.

```
// tutorial17.js
var CommentForm = React.createClass({
  getInitialState: function() {
    return {author: '', text: ''};
  },
  handleAuthorChange: function(e) {
    this.setState({author: e.target.value});
  },
  handleTextChange: function(e) {
```

```
    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var author = this.state.author.trim();
    var text = this.state.text.trim();
    if (!text || !author) {
      return;
    }
    // TODO: send request to the server
    this.setState({author: '', text: ''});
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input
          type="text"
          placeholder="Your name"
          value={this.state.author}
          onChange={this.handleAuthorChange}
        />
        <input
          type="text"
          placeholder="Say something..."
          value={this.state.text}
          onChange={this.handleTextChange}
        />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

We attach an `onSubmit` handler to the form that clears the form fields when the form is submitted with valid input.

Call `preventDefault()` on the event to prevent the browser's default action of submitting the form.

Callbacks as props

When a user submits a comment, we will need to refresh the list of comments to include the new one. It makes sense to do all of this logic in `CommentBox` since `CommentBox` owns the state that represents the

list of comments.

We need to pass data from the child component back up to its parent. We do this in our parent's `render` method by passing a new callback (`handleCommentSubmit`) into the child, binding it to the child's `onCommentSubmit` event. Whenever the event is triggered, the callback will be invoked:

```
// tutorial18.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    // TODO: submit to the server and refresh the list
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Now that `CommentBox` has made the callback available to `CommentForm` via the `onCommentSubmit` prop, the `CommentForm` can call the callback when the user submits the form:

```
// tutorial19.js
var CommentForm = React.createClass({
  getInitialState: function() {
    return {author: '', text: ''};
  },
  handleAuthorChange: function(e) {
    this.setState({author: e.target.value});
  },
  handleTextChange: function(e) {
    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var author = this.state.author.trim();
    var text = this.state.text.trim();
    if (!text || !author) {
      return;
    }
    this.props.onCommentSubmit({author: author, text: text});
    this.setState({author: '', text: ''});
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input
          type="text"
          placeholder="Your name"
          value={this.state.author}
          onChange={this.handleAuthorChange}
        />
        <input
          type="text"
          placeholder="Say something..."
          value={this.state.text}
          onChange={this.handleTextChange}
        />
        <input type="submit" value="Post" />
      </form>
```

```
    );
  }
});
```

Now that the callbacks are in place, all we have to do is submit to the server and refresh the list:

```
// tutorial20.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
```

```
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Optimization: optimistic updates

Our application is now feature complete but it feels slow to have to wait for the request to complete before your comment appears in the list. We can optimistically add this comment to the list to make the app feel faster.

```
// tutorial21.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    var comments = this.state.data;
    // Optimistically set an id on the new comment. It will be replaced by an
    // id generated by the server. In a production application you would likely
    // not use Date.now() for this and would have a more robust system in place.
    comment.id = Date.now();
    var newComments = comments.concat([comment]);
    this.setState({data: newComments});
```

```
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        this.setState({data: comments});
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

Congrats!

You have just built a comment box in a few simple steps. Learn more about why to use React, or dive into the API reference and start hacking! Good luck!