Tutorial

Edit on GitHub

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.

Want to skip all this and just see the source?

It's all on GitHub.

Running a server

While it's not necessary to get started with this tutorial, later on we'll be adding functionality that requires `POST`ing to a running server. If this is something you are intimately familiar with and want to create your own server, please do. For the rest of you who might want to focus on learning about React without having to worry about the server-side aspects, we have written simple servers in a number of languages - JavaScript (using Node.js), Python, Ruby, Go, and PHP. These are all available on GitHub. You can view the source or download a zip file to get started.

To get started using the tutorial, just start editing `public/index.html`.

Getting started

For this tutorial, we'll use prebuilt JavaScript files on a CDN. Open up your favorite editor and create a new HTML document:

Code

```
<!-- index.html -->
<html>
  <head>
    <title>Hello React</title>
    <script src="https://fb.me/react-0.13.1.js"></script>
    <script src="https://fb.me/JSXTransformer-0.13.1.js"></script>
    <script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/jsx">
      // Your code here
    </script>
  </body>
</html>
```

For the remainder of this tutorial, we'll be writing our JavaScript code in this script tag.

> **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:

Code

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

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

Code

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

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:

Code

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

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

Composing components

Let's build skeletons for `CommentList` and `CommentForm` which will, again, be simple `<div>`s:

Code

```
// 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:

Code

```
// 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. 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:

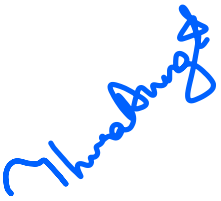
Code

```
// 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`:



Code

```
// 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 an 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.

First, add the third-party **Showdown** library to your application. This is a JavaScript library which takes Markdown text and converts it to raw HTML. This requires a script tag in your head (which we have already included in the React playground):

Code

```
<!-- index.html -->
<head>
  <title>Hello React</title>
  <script src="https://fb.me/react-0.13.1.js"></script>
  <script src="https://fb.me/JSXTransformer-0.13.1.js"></script>
  <script src="https://code.jquery.com/jquery-1.10.0.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/showdown/0.3.1/showdown.min.
</head>
```

Next, let's convert the comment text to Markdown and output it:

Code

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

All we're doing here is calling the Showdown library. We need to convert `this.props.children` from React's wrapped text to a raw string that Showdown 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:

Code

```
// tutorial7.js
var converter = new Showdown.converter();
var Comment = React.createClass({
  render: function() {
    var rawMarkup = converter.makeHtml(this.props.children.toString());
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        <span dangerouslySetInnerHTML={{__html: rawMarkup}} />
      </div>
    );
  }
});
```

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

Remember: by using this feature you're relying on Showdown to be secure.

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:

Code

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

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

Code

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

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

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

Code

```
// tutorial10.js
var CommentList = React.createClass({
  render: function() {
    var commentNodes = this.props.data.map(function (comment) {
      return (
        <Comment author={comment.author}>
          {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:

Code

```
// tutorial11.js
React.render(
  <CommentBox url="comments.json" />,
  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.

Reactive state

So far, each component has rendered itself once based on its props. `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 is updated, 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:

Code

```
// 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. In a real application this would be a dynamic endpoint, but for this example, we will use a static JSON file to keep things simple:

Code

```
// tutorial13.json
[
  {"author": "Pete Hunt", "text": "This is one comment"},
```

```
  {"author": "Jordan Walke", "text": "This is *another* comment"}
]
```

We'll use jQuery to help make an asynchronous request to the server.

Note: because this is becoming an AJAX application you'll need to develop your app using a web server rather than as a file sitting on your file system. As mentioned above, we have provided several servers you can use on GitHub. They provide the functionality you need for the rest of this tutorial.

Code

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      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 when a component is rendered. 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.

Code

```
// tutorial14.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      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>
    );
  }
});

React.render(
  <CommentBox url="comments.json" 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; 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.

Code

```
// 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>
    );
  }
});
```

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.

Code

```
// tutorial16.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = React.findDOMNode(this.refs.author).value.trim();
    var text = React.findDOMNode(this.refs.text).value.trim();
    if (!text || !author) {
      return;
    }
    // TODO: send request to the server
    React.findDOMNode(this.refs.author).value = '';
    React.findDOMNode(this.refs.text).value = '';
    return;
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Your name" ref="author" />
        <input type="text" placeholder="Say something..." ref="text" />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

Events

React attaches event handlers to components using a camelCase naming convention. 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.

Refs

We use the `ref` attribute to assign a name to a child component and `this.refs` to reference the component. We can call `React.findDOMNode(component)` on a component to get the native browser DOM element.

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:

Code

```
// tutorial17.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      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>
    );
  }
});
```

Let's call the callback from the `CommentForm` when the user submits the form:

Code

```
// tutorial18.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = React.findDOMNode(this.refs.author).value.trim();
    var text = React.findDOMNode(this.refs.text).value.trim();
    if (!text || !author) {
      return;
    }
    this.props.onCommentSubmit({author: author, text: text});
    React.findDOMNode(this.refs.author).value = '';
    React.findDOMNode(this.refs.text).value = '';
    return;
  },
  render: function() {
```

```
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Your name" ref="author" />
        <input type="text" placeholder="Say something..." ref="text" />
        <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:

Code

```
// tutorial19.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      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) {
```



```
        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.

Code

```
// tutorial20.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
```

```
      url: this.props.url,
      dataType: 'json',
      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;
    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) {
        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!

← Prev

Next →

A Facebook & Instagram collaboration.
Acknowledgements

© 2013–2015 Facebook Inc.
Documentation licensed under CC BY 4.0.