Tutorial

We'll be building a simple, but realistic comments box that you can drop into a blog, similar to 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:** as other users comment we'll pop them 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.

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

```
<!-- template.html -->
<html>
  <head>
    <title>Hello React</title>
    <script src="http://fb.me/react-0.4.1.js"></script>
```

```
  <body>
    <div id="content"></div>
    <script type="text/jsx">
      /**
       * @jsx React.DOM
       */
      // Your code here
    </script>
  </body>
</html>
```

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

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 class="commentBox">
        Hello, world! I am a CommentBox.
```

```
});
React.renderComponent(
  <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({
  render: function() {
    return (
      React.DOM.div({
        className: 'commentBox',
        children: 'Hello, world! I am a CommentBox.'
      })
    );
  }
});
React.renderComponent(
  CommentBox({}),
  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

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.renderComponent()` 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 class="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

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

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

Next, update the `CommentBox` component to use its new friends:

Code

```
// tutorial3.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div class="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.DOM.tagName" expressions and leave everything else alone. This is to prevent the pollution of the global namespace.

Component Properties

Let's create our third component, `Comment`. We will want to pass it the author name and comment text so we can reuse the same code for each unique comment. First let's add some comments to the `CommentList`:

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div class="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` component as both XML-like children and attributes. Data passed from parent to child is called **props**, short for properties.

Using props

Let's create the Comment component. It will read the data passed to it from the CommentList and render some markup:

Code

```
// tutorial5.js
var Comment = React.createClass({
  render: function() {
    return (
      <div class="comment">
        <h2 class="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`.

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).

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 class="comment">
        <h2 class="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

But there's a problem! Our rendered comments look like this in the browser: " This is another comment ". 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 class="comment">
        <h2 class="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:

```
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 `renderComponent()` call to pass this data into the `CommentList` via props:

Code

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

React.renderComponent(
  <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({
```

```
      });
      return (
        <div class="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.renderComponent(
  <CommentBox url="comments.json" />,
  document.getElementById('example')
);
```

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

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 class="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:

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

We will use jQuery 1.5 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. The easiest way to do this is to run `python -m SimpleHTTPServer` in your application's directory.

Code

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    $.ajax({
      url: 'comments.json',
      dataType: 'json',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
    return {data: []};
  },
  render: function() {
    return (
      <div class="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
```

The key 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 trivial 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',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentWillMount: function() {
    this.loadCommentsFromServer();
    setInterval(
      this.loadCommentsFromServer.bind(this),
      this.props.pollInterval
    );
  },
  render: function() {
    return (
      <div class="commentBox">
        <h1>Comments</h1>
```

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

React.renderComponent(
  <CommentBox url="comments.json" pollInterval={5000} />,
  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 5 seconds after that. Try running this in your browser and changing the `comments.json` file; within 5 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 class="commentForm">
        <input type="text" placeholder="Your name" />
        <input type="text" placeholder="Say something..." />
        <input type="submit" />
      </form>
    );
  }
});
```

event and clear it.

Code

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

Events

with valid input.

We always return `false` from the event handler to prevent the browser's default action of submitting the form. (If you prefer, you can instead take the event as an argument and call `preventDefault()` on it.)

Refs

We use the `ref` attribute to assign a name to a child component and `this.refs` to reference the component. We can call `getDOMNode()` 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 to its parent. We do this by passing a `callback` in props from parent to child:

Code

```
// tutorial17.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    // TODO: submit to the server and refresh the list
```

```
    return {data: []};
  },
  componentWillMount: function() {
    this.loadCommentsFromServer();
    setInterval(
      this.loadCommentsFromServer.bind(this),
      this.props.pollInterval
    );
  },
  render: function() {
    return (
      <div class="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() {
    var author = this.refs.author.getDOMNode().value.trim();
    var text = this.refs.text.getDOMNode().value.trim();
    this.props.onCommentSubmit({author: author, text: text});
    this.refs.author.getDOMNode().value = '';
    this.refs.text.getDOMNode().value = '';
```

```
    render: function() {
      return (
        <form class="commentForm" onSubmit={this.handleSubmit}>
          <input type="text" placeholder="Your name" ref="author" />
          <input
            type="text"
            placeholder="Say something..."
            ref="text"
          />
          <input type="submit" />
        </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',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    $.ajax({
```

```
      dataType: 'json',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentWillMount: function() {
    this.loadCommentsFromServer();
    setInterval(
      this.loadCommentsFromServer.bind(this),
      this.props.pollInterval
    );
  },
  render: function() {
    return (
      <div class="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

Code

```
// tutorial20.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    var comments = this.state.data;
    comments.push(comment);
    this.setState({data: comments});
    $.ajax({
      url: this.props.url,
      data: comment,
      dataType: 'json',
      mimeType: 'textPlain',
      success: function(data) {
        this.setState({data: data});
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentWillMount: function() {
    this.loadCommentsFromServer();
```

```
      this.props.pollInterval
    );
  },
  render: function() {
    return (
      <div class="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 React in the reference or start hacking! Good luck!

Alexandre ScartrakkBeatz Juca · Follow · Chief Technology Officer at M.I.D Technologies

Great stuff

Reply · 2 · Like · Follow Post · July 31 at 11:14am

Fabio Zendhi Nagao · Follow · CTO at LojComm Internet

Great stuff xD congrats Tom, Jordan and FB team!

Also, here are some minor notes I made while reading this tutorial:

// tutorial11.js

- document.getElementById('example')
+ document.getElementById('content')

// tutorial15.js

Should also highlight ` </form>`.

// tutorial16.js
... See More

Reply · 1 · Like · Follow Post · July 24 at 10:59am

Ribhararnus Pracutiar · Follow · Top Commenter

You forgot type: "POST" jquery ajax in handleCommentSubmit.

Reply · 1 · Like · Follow Post · July 21 at 6:12am

Jinsu Mathew · Follow · Chief Technology Officer (CTO) at MindHelix Technologies

In the last section:

var comments = this.state.data;.

comments.push(comment);.

this.setState({data: comments});.

$.ajax({.... See More

Reply · Like · Follow Post · August 10 at 8:43am