**Tutorial (ASP.NET Core)**

This tutorial covers the end-to-end process of creating a brand new ASP.NET MVC website and adding a React component in it. We will start from scratch and end with a fully functioning component. It assumes you have basic knowledge of ASP.NET MVC and using Visual Studio.

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

Simple server-side in-memory storage for comments

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.

**Getting started**

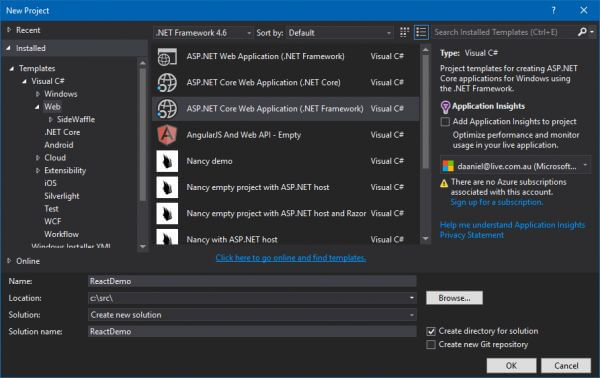
For this tutorial we'll be using Visual Studio 2015. If you do not already have a copy of Visual Studio, [the Community version](https://www.visualstudio.com/vs/community/) is free. We will be using ASP.NET Core MVC.

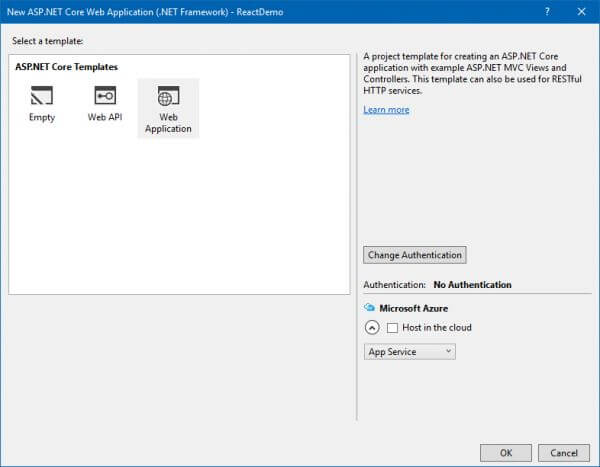
**New Project**

Start by creating a new ASP.NET Core MVC project:

File → New → Project

Ensure ".NET Framework 4.6" is selected in the dropdown list at the top

Go to Templates → Visual C# → Web and select the "ASP.NET Core Web Application (.NET Framework)" template. Call it "ReactDemo" [](https://reactjs.net/img/tutorial/newproject_core.png)

In the "New ASP.NET Core Web Application" dialog, select the Web Application template. Also, click "Change Authentication" and select "No Authentication" [](https://reactjs.net/img/tutorial/new_webapp.png)

Note: We are using .NET Framework in this tutorial, but you can instead use .NET Core if you want to be able to run your site on Linux or Mac OS. Currently .NET Core is missing some of the functionality provided by .NET Framework, so it is recommended to use .NET Framework unless you have a reason to use .NET Core specifically (eg. cross-platform support).

**Remove example content**

The default Web Application template includes some example content that we don't need. Delete the following files:

Controllers\HomeController.cs

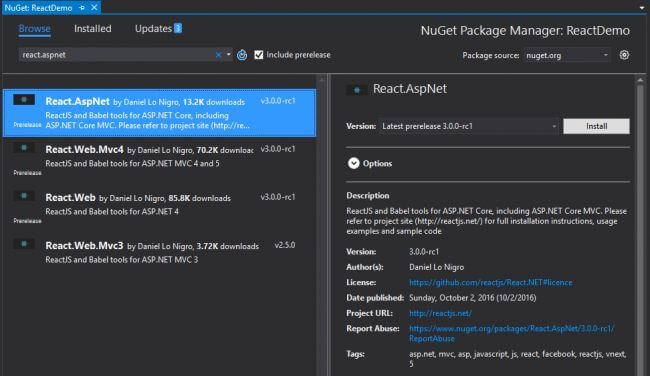
Views\Home and Views\Shared folders

bundleconfig.json

Project\_Readme.html

**Install ReactJS.NET**

We need to install ReactJS.NET to the newly-created project. This is accomplished using NuGet, a package manager for .NET. Right-click on the "ReactDemo" project in the Solution Explorer and select "Manage NuGet Packages". Click the "Browse" tab, search for "React.AspNet", and install the **React.AspNet** package.

[](https://reactjs.net/img/tutorial/nuget_core.png)

We also need to modify the Startup.cs file to initialize ReactJS.NET. You can learn more about this on the [Getting Started on ASP.NET Core](https://reactjs.net/getting-started/aspnetcore.html) page. Open Startup.cs and perform the following changes:

At the top of the file, add:

using Microsoft.AspNetCore.Http;

using React.AspNet;

Directly above:

// Add framework services.

services.AddMvc();

Add:

services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

services.AddReact();

Directly **above**:

app.UseStaticFiles();

Add:

// Initialise ReactJS.NET. Must be before static files.

app.UseReact(config =>

{

// If you want to use server-side rendering of React components,

// add all the necessary JavaScript files here. This includes

// your components as well as all of their dependencies.

// See http://reactjs.net/ for more information. Example:

//config

// .AddScript("~/Scripts/First.jsx")

// .AddScript("~/Scripts/Second.jsx");

// If you use an external build too (for example, Babel, Webpack,

// Browserify or Gulp), you can improve performance by disabling

// ReactJS.NET's version of Babel and loading the pre-transpiled

// scripts. Example:

//config

// .SetLoadBabel(false)

// .AddScriptWithoutTransform("~/Scripts/bundle.server.js");

});

Finally, add this to Views\\_ViewImports.cshtml:

@using React.AspNet

**Create basic controller and view**

Since this tutorial focuses mainly on ReactJS.NET itself, we will not cover creation of an MVC controller in much detail. To learn more about ASP.NET MVC, refer to [its official website](http://www.asp.net/mvc).

Right-click on the Controllers folder and select Add → New Item

Select .NET Core → ASP.NET → MVC Controller Class

Name the file HomeController.cs

Once the controller has been created, we also need to create a view 1. Right-click on the Views folder, click "New Folder", and create a "Home" folder 2. Right-click on the Views\Home folder and select Add → New Item 3. Select .NET Core → ASP.NET → MVC View Page 4. Name the file Index.cshtml

Replace the contents of the new view file with the following:

@{

Layout = null;

}

<html>

<head>

<title>Hello React</title>

</head>

<body>

<div id="content"></div>

<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react-dom.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/remarkable/1.7.1/remarkable.min.js"></script>

<script src="@Url.Content("~/js/tutorial.jsx")"></script>

</body>

</html>

*Note: In a real ASP.NET MVC site, you'd use a layout. However, to keep this tutorial simple, we will keep all HTML in the one view file.*

We also need to create the referenced JavaScript file (tutorial.jsx). Right-click on wwwroot\js and select Add → New Item. Select .NET Core → Client-side → JavaScript File, enter "tutorial.jsx" as the file name, and click "Add".

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

**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>. Add this code to Tutorial.jsx:

var CommentBox = React.createClass({

render: function() {

return (

<div className="commentBox">

Hello, world! I am a CommentBox.

</div>

);

}

});

ReactDOM.render(

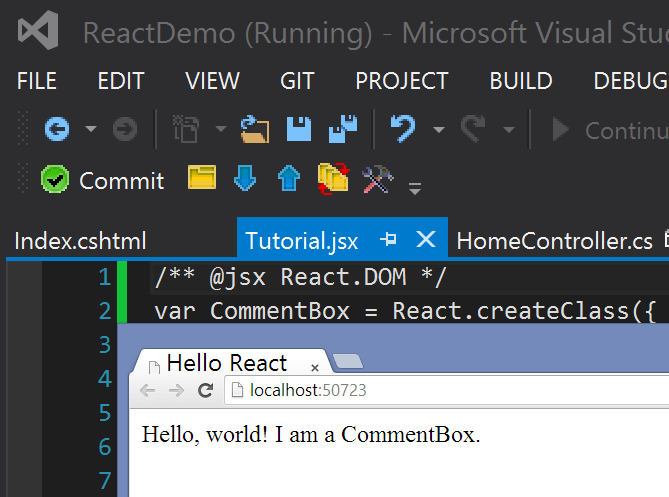
<CommentBox />,

document.getElementById('content')

);

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

At this point, run your application by clicking the "Play" button in Visual Studio. If successful, your default browser should start and you should see "Hello, world! I am a CommentBox."



If you see this, congratulations! You've just built your first React component. You can leave the application running while you continue this tutorial. Simply change the JSX file and refresh to see your changes.

**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:

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](http://facebook.github.io/react/docs/jsx-in-depth.html).

**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 divcomponents. 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](http://facebook.github.io/react-native/)).

**Composing components**

Let's build skeletons for CommentList and CommentForm which will, again, be simple <div>s. Add these two components to the top of your file, just before the existing CommentBoxdeclaration and ReactDOM.render call:

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:

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 our CommentList component. Data passed in from the CommentList component is available as a 'property' on our Comment 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:

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:

var CommentList = React.createClass({

render: function() {

return (

<div className="commentList">

<Comment author="Daniel Lo Nigro">Hello ReactJS.NET World!</Comment>

<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 *Daniel Lo Nigro* (via the author attribute) and *Hello ReactJS.NET World* (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:

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](https://en.wikipedia.org/wiki/Cross-site_scripting). There's a way to get around it but the framework warns you not to use it:

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:

var data = [

{ id: 1, author: "Daniel Lo Nigro", text: "Hello ReactJS.NET World!" },

{ id: 2, author: "Pete Hunt", text: "This is one comment" },

{ id: 3, 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:

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:

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!

**Server-side Data**

Let's return some data from the server. To do so, we need to first create a C# class to represent our comments. Right-click on ReactDemo and select Add → New Folder and name the folder "Models". Once the models folder has been created, right click on it, select Add → Class, and enter "CommentModel.cs" as the file name. We'll create a basic comment model:

namespace ReactDemo.Models

{

public class CommentModel

{

public int Id { get; set; }

public string Author { get; set; }

public string Text { get; set; }

}

}

In a real application, you'd use the repository pattern here, and retrieve the comments from a database. For simplicity, we'll just modify our controller to have a hard-coded list of comments.

using System.Collections.Generic;

using Microsoft.AspNetCore.Mvc;

using ReactDemo.Models;

namespace ReactDemo.Controllers

{

public class HomeController : Controller

{

private static readonly IList<CommentModel> \_comments;

static HomeController()

{

\_comments = new List<CommentModel>

{

new CommentModel

{

Id = 1,

Author = "Daniel Lo Nigro",

Text = "Hello ReactJS.NET World!"

},

new CommentModel

{

Id = 2,

Author = "Pete Hunt",

Text = "This is one comment"

},

new CommentModel

{

Id = 3,

Author = "Jordan Walke",

Text = "This is \*another\* comment"

},

};

}

public ActionResult Index()

{

return View();

}

}

}

Let's also add a new controller action to return the list of comments:

[Route("comments")]

[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]

public ActionResult Comments()

{

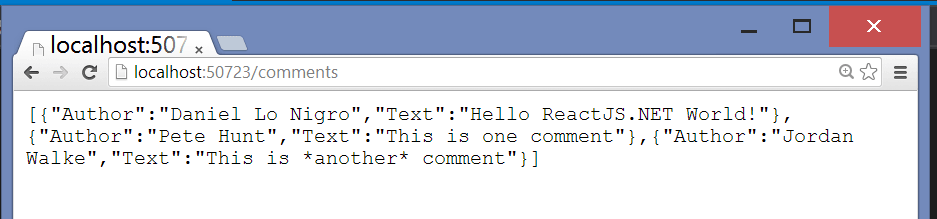
return Json(\_comments);

}

The Route attribute specifies that this action should be used when /comments is loaded. This method of defining URL routes is known as "attribute routing".

The ResponseCache attribute is used here to prevent browsers from caching the response. When designing a real world API, caching of API requests should be considered more carefully. For this tutorial it is easiest to simply disable caching.

If you hit /comments in your browser, you should now see the data encoded as JSON:



**Fetching from the server**

Now that we have a data source, we can replace the hard-coded data with the dynamic data from the server. We will remove the data prop and replace it with a URL to fetch:

ReactDOM.render(

<CommentBox url="/comments" />,

document.getElementById('content')

);

Note that in a real app, you should generate the URL server-side (via Url.Action call) and pass it down, or use [RouteJs](http://dan.cx/projects/routejs) rather than hard-coding it. This tutorial hard-codes it for simplicity.

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, 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:

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'll use the standard XMLHttpRequest API to retrieve the data. If you need support for old browsers (mainly old Internet Explorer), you can use an AJAX library or a multipurpose library such as jQuery. componentWillMount()executes immediately and only once before the rendering occurs. In the following example, componentWillMount() loads the data from our XMLHttpRequest and assigns it to the datavariable. Finally, it sets the data variable in state, using setState().

var CommentBox = React.createClass({

getInitialState: function() {

return {data: []};

},

componentWillMount: function() {

var xhr = new XMLHttpRequest();

xhr.open('get', this.props.url, true);

xhr.onload = function() {

var data = JSON.parse(xhr.responseText);

this.setState({ data: data });

}.bind(this);

xhr.send();

},

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. So, by moving the XMLHttpRequest call from componentWillMount(), which is executed only once *before* rendering, to a function called loadCommentsFromServer(), we can then call it multiple times from componentDidMount() at a set interval to check for any updates to the comments. 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 [SignalR](http://signalr.net/) or other technologies.

var CommentBox = React.createClass({

loadCommentsFromServer: function() {

var xhr = new XMLHttpRequest();

xhr.open('get', this.props.url, true);

xhr.onload = function() {

var data = JSON.parse(xhr.responseText);

this.setState({ data: data });

}.bind(this);

xhr.send();

},

getInitialState: function() {

return {data: []};

},

componentDidMount: function() {

this.loadCommentsFromServer();

window.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="/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.

**Adding new comments**

To accept new comments, we need to first add a controller action to handle it. This will just be some simple C# code that appends the new comment to the static list of comments:

[Route("comments/new")]

[HttpPost]

public ActionResult AddComment(CommentModel comment)

{

// Create a fake ID for this comment

comment.Id = \_comments.Count + 1;

\_comments.Add(comment);

return Content("Success :)");

}

**The Form**

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.

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](http://facebook.github.io/react/docs/forms.html#controlled-components).

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](https://reactjs.net/react/docs/interactivity-and-dynamic-uis.html#under-the-hood-autobinding-and-event-delegation) 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.

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:

var CommentBox = React.createClass({

loadCommentsFromServer: function() {

var xhr = new XMLHttpRequest();

xhr.open('get', this.props.url, true);

xhr.onload = function() {

var data = JSON.parse(xhr.responseText);

this.setState({ data: data });

}.bind(this);

xhr.send();

},

handleCommentSubmit: function(comment) {

// TODO: submit to the server and refresh the list

},

getInitialState: function() {

return {data: []};

},

componentDidMount: function() {

this.loadCommentsFromServer();

window.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:

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:

var CommentBox = React.createClass({

loadCommentsFromServer: function() {

var xhr = new XMLHttpRequest();

xhr.open('get', this.props.url, true);

xhr.onload = function() {

var data = JSON.parse(xhr.responseText);

this.setState({ data: data });

}.bind(this);

xhr.send();

},

handleCommentSubmit: function(comment) {

var data = new FormData();

data.append('author', comment.author);

data.append('text', comment.text);

var xhr = new XMLHttpRequest();

xhr.open('post', this.props.submitUrl, true);

xhr.onload = function() {

this.loadCommentsFromServer();

}.bind(this);

xhr.send(data);

},

getInitialState: function() {

return {data: []};

},

componentDidMount: function() {

this.loadCommentsFromServer();

window.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>

);

}

});

ReactDOM.render(

<CommentBox url="/comments" submitUrl="/comments/new" pollInterval={2000} />,

document.getElementById('content')

);