Learning Objectives

- **Define JSX**
- **Identify when to use double quotes and curly braces with JSX**
- **Explain the purpose of Babel**
- **Replace `React.createElement` with code written in JSX**
- **Reference information passed to a component with `props`**
- **Transform an array of posts into React components**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

JSX Intro

JSX Intro

Having to use the `React.createElement()` function all the time can become tedious, even if we were to alias to a shorter variable like `e`. It also makes it hard to tell at a glance how the element will actually look on the page.

JSX lets us mix HTML and JavaScript in the same file, which makes it both quicker to write and easier to read.

For example, in the last section you created a `<button>` using `React.createElement()`, like this:

```
return React.createElement(
  'button',
  {
    className: 'btn btn-primary mt-2',
    onClick: () => {
      this.handleClick()
    }
  },
  buttonText
)
```

When using JSX, we would build the same `<button>` like this:

```
return <button
  className="btn btn-primary mt-2"
  onClick={
    () => {
      this.handleClick()
    }
  }
>
  {buttonText}
</button>
```

To point out some things of note:

- String properties, like `className`, can be passed in double quotes, like normal HTML attributes are.
- Properties that aren't strings should be passed using curly braces `{}`.

  For example, the function that's passed to `onClick`.
- Variables should be rendered inside curly braces. Notice how we are doing `{buttonText}` not just `buttonText`. Doing the latter would cause the literal text "buttonText" to be rendered, instead of the value of the variable.

We can't just make this change and expect it to work though. What we've written is not valid JavaScript, and would just cause errors. We're going to use a pre-processor called Babel to process the script before it's interpreted.

Babel

Babel is a tool that, among other things, compiles JSX to JavaScript. It can also compile JavaScript that uses new features into JavaScript that's compatible with older browsers, so it's a useful tool to know about.

Like React, there are different levels at which it can be included in a project. If you have a large project with lots of JavaScript files, you could incorporate it as part of your build process to compile all the JavaScript before deployment. This takes a lot of work to set up though. Since we just have a single JavaScript file to compile, we're going to use a simple method. It's not as fast as pre-compiling everything, but for our use case we probably won't notice a difference.

There are two steps to get Babel set up in this way.

First, we need to include the Babel script, before we include any of our own JavaScript that needs to be compiled:

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js">
        </script>
```

Then we need to add the attribute `type="text/babel"` to any `<script>`s that we want to be compiled. For example:

```
<script type="text/babel" src="{% static "blog/blog.js" %}">
        </script>
```

Let's go ahead and convert our `ClickButton` to JSX.

Try It Out

Start inside the `blog/templates/blog/post-table.html` file. Add the Babel script after the React scripts and before `blog.js` is included:

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js">
        </script>
```

Then, add the `type="text/babel"` attribute to the `<script>` tag for `blog.js`.

```
<script type="text/babel" src="{% static "blog/blog.js" %}">
        </script>
```

Now head over to `blog/static/blog/blog.js`. Update the `render()` method so it returns JSX instead of using `React.createElement()`. You can replace your `render()` method with this code:

Open blog.js

```
  render () {
    let buttonText

    if (this.state.wasClicked)
      buttonText = 'Clicked!'
    else
      buttonText = 'Click Me'

    return <button
      className="btn btn-primary mt-2"
      onClick={
        () => {
          this.handleClick()
        }
      }
    >
      {buttonText}
    </button>
  }
```

Go back to your browser and refresh the `/post-table/` page. You should notice that nothing has changed. If you've done everything right, the button should still say *Click Me*, and change to *Clicked!* when clicked.

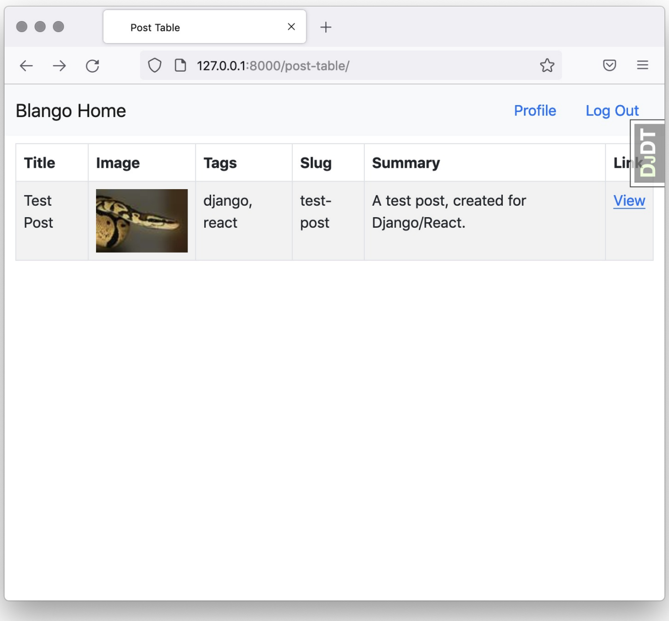
View Blog

Now we have covered all the theory we need to begin the real post listing table component, so let's make a start on that now.

Post Table Components

Post Table Components

To show what we’re aiming for at the end of this section, here’s what the post table will look like:



Title	Image	Tags	Slug	Summary	Link
Test Post		django, react	test-post	A test post, created for Django/React.	View

post table

It’s made up of two `Component` classes: `PostRow` and `PostTable`.

`PostRow` represents one row in the table. To build it, we’ll be using `props` for the first time. This is an object that contains all the properties/attributes that were passed to the component.

The `props` object will contain just one item: `post`, which is a `Post` object fetched from the `Post` API.

We implement just the `render` method, and don’t need to use any `state`, as the `render()` method can just refer to `this.props.post` to get the `Post` object that was passed in.

Here’s the `PostRow` class in its entirety:

```
class PostRow extends React.Component {
  render () {
    const post = this.props.post

    let thumbnail

    if (post.hero_image.thumbnail) {
      thumbnail = <img src={post.hero_image.thumbnail}/>
    } else {
      thumbnail = '-'
    }

    return <tr>
      <td>{post.title}</td>
      <td>
        { thumbnail }
      </td>
      <td>{post.tags.join(', ')}</td>
      <td>{post.slug}</td>
      <td>{post.summary}</td>
      <td><a href={'/post/' + post.slug + '/'}>View</a></td>
    </tr>
  }
}
```

It should be pretty clear what's going on here. We're returning a `<tr>` that contains a number of `<td>`s with the `Post` data. Like Django, React will escape any HTML that it outputs, so we don't have to worry about doing that manually. We're able to call JavaScript code inside curly braces, to output the `tags` array joined by `,`. Another example is how we can concatenate strings, such as to build the `href` attribute.

Another thing to note is that when we're using JSX we can assign HTML templates to variables. We assign an `<img>` element to the `thumbnail` variable if the `hero_image` is present. Otherwise, we just assign a `-` to be output instead.

Next let's look at `PostTable`. We intend for `PostTable` to be responsible for fetching the data, and then passing each `Post` that's received to a `PostRow` component to render inside a table.

Since we haven't yet covered fetching data in JavaScript, we'll be hardcoding it in the right format for now. It will live inside the component's `state`.

Since `PostTable` is quite a long class, we'll look at it piece by piece. First, the `state` setup. Remember this is hardcoded for testing:

```
class PostTable extends React.Component {
  state = {
    dataLoaded: true,
    data: {
      results: [
        {
          id: 1,
          tags: [
            'django', 'react'
          ],
          'hero_image': {
            'thumbnail': '/media/__sized__/hero_images/snake-
419043_1920-thumbnail-100x100-70.jpg',
            'full_size': '/media/hero_images/snake-
419043_1920.jpg'
          },
          title: 'Test Post',
          slug: 'test-post',
          summary: 'A test post, created for Django/React.'
        }
      ]
    }
  }

  // other methods to come
}
```

The `state` has two items in it:

- `dataLoaded` would normally start out `false` and then be set `true` once data is loaded from the API. We're just setting it to `true` straight away since data is hardcoded.
- `data` matches the format of a Post list response. It is expected to have an array of `results`.

We've omitted some of the `Post` data that we won't be using, but it won't matter that it will be there in the real API response.

Next we'll look at the first part of the `render()` method. It builds the variable `rows`, which will be rendered inside a `<table>`:

```
class PostTable extends React.Component {
  render () {
    let rows
    if (this.state.dataLoaded) {
      if (this.state.data.results.length) {
        rows = this.state.data.results.map(post => <PostRow
        post={post}/>)
      } else {
        rows = <tr>
          <td colSpan="6">No results found.</td>
        </tr>
      }
    } else {
      rows = <tr>
        <td colSpan="6">Loading&hellip;</td>
      </tr>
    }

    // the rest of the method to come
  }
}
```

`rows` will end up being in one of three forms.

- If the data has been loaded (`dataLoaded` is `true`) and there is more than one result (`this.state.data.results.length` is not 0) then the results are mapped to a function that returns a `PostRow` for each `Post` in the results. `post` is passed as a property, so it will end up in `this.props` on the `PostRow` component (we've already seen how this is accessed). Each of these is stored in the `rows` variable. We also need to provide the `key` property, which must be unique for each element of the array. We know that the `Post`'s `id` will be unique so we can use that, but you could also use the array index.
- If the data has been loaded, but there are no results returned (`this.state.data.results.length` is 0), then `rows` will be a `<tr>` with a single `<td>`, with the text *No results found.*
- Finally, while the data is loading, `dataLoaded` is `false`, so a single row with the content *Loading…* will be displayed.

Once again, remember that since we have test data, we'll only see the first state at this stage.

Here's the second half of the `render()` method. It renders a table with the `rows` inside the table body:

```
class PostTable extends React.Component {
  render () {
    // let row... etc omitted

    return <table className="table table-striped table-bordered
        mt-2">
      <thead>
      <tr>
        <th>Title</th>
        <th>Image</th>
        <th>Tags</th>
        <th>Slug</th>
        <th>Summary</th>
        <th>Link</th>
      </tr>
      </thead>
      <tbody>
      {rows}
      </tbody>
    </table>
  }
}
```

There's nothing too notable to point out. The classes applied to the `<table>` are the Bootstrap classes that make the table look a bit better. You can check out the Bootstrap table documentation for more info.

Try It Out

Try It Out

You can copy and paste this next code block into `blog.js`, replacing the code you already have there. Notice that the class that's passed to `ReactDOM.render()` function is updated to `PostTable`.

```
class PostRow extends React.Component {
  render () {
    const post = this.props.post

    let thumbnail

    if (post.hero_image.thumbnail) {
      thumbnail = <img src={post.hero_image.thumbnail}/>
    } else {
      thumbnail = '-'
    }

    return <tr>
      <td>{post.title}</td>
      <td>
        {thumbnail}
      </td>
      <td>{post.tags.join(', ')}</td>
      <td>{post.slug}</td>
      <td>{post.summary}</td>
      <td><a href={'/post/' + post.slug + '/'}>View</a></td>
    </tr>
  }
}

class PostTable extends React.Component {
  state = {
    dataLoaded: true,
    data: {
      results: [
        {
          id: 15,
          tags: [
            'django', 'react'
          ],
          'hero_image': {
            'thumbnail': '/media/__sized__/hero_images/snake-
          419043_1920-thumbnail-100x100-70.jpg',
            'full_size': '/media/hero_images/snake-
          419043_1920.jpg'
          },
          title: 'Test Post',
          slug: 'test-post',
          summary: 'A test post, created for Django/React.'
        }
      ]
    }
  }

  render () {
    let rows
    if (this.state.dataLoaded) {
      if (this.state.data.results.length) {
        rows = this.state.data.results.map(post => <PostRow
        post={post} key={post.id}/>)
      } else {
        rows = <tr>
          <td colSpan="6">No results found.</td>
        </tr>
      }
    } else {
      rows = <tr>
        <td colSpan="6">Loading&hellip;</td>
      </tr>
    }

    return <table className="table table-striped table-bordered
        mt-2">
      <thead>
      <tr>
        <th>Title</th>
        <th>Image</th>
        <th>Tags</th>
        <th>Slug</th>
        <th>Summary</th>
        <th>Link</th>
      </tr>
      </thead>
      <tbody>
      {rows}
      </tbody>
    </table>
  }
}
```

```
const domContainer = document.getElementById('react_root')
ReactDOM.render(
  React.createElement(PostTable),
  domContainer
)
```

info

Image Names

In the code sample below,`hero_image` is defined as:

```
'hero_image': {
  'thumbnail': '/media/__sized__/hero_images/snake-
        419043_1920-thumbnail-100x100-70.jpg',
  'full_size': '/media/hero_images/snake-419043_1920.jpg'
},
```

This will only show an image if you have a file called `snake-419043_1920.jpg`. If not, you will see the icon for a broken image link. If you want an image to show, find a file name in the `/media/__sized__/hero_images` and `/media/hero_images` directories. Use them in place of `snake-419043_1920-thumbnail-100x100-70.jpg` and `snake-419043_1920.jpg`.

There's one final change, in `blog/templates/blog/post-table.html`: add a `title` block. This can come just before the `content` block:

Open post-table.html

```
{% block title %}Post Table{% endblock %}
```

Now you're done, load up `/post-table/` in a browser and check that it looks like the screenshot we showed at the start of the section.

View Blog

In the next section, we'll see how to fetch the data from the API and render it in the table.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish JSX"
```

- Push to GitHub:

```
git push
```