# Piggy Bank Application

**Duration: 90 minutes**

### Learning Objectives

* Be able to create a single component application
* Be able to use the JSX syntax
* Use props and state to managing data flow in the application
* Install and be able to use React Dev Tools

## Introduction

In this lesson we are going to build a Piggy Bank application that displays a total and allows a user to deposit and withdraw an amount, which will update the total. We want the the total to change in response to a user’s interaction and we will use the React framework to do what it does so well - manipulate the DOM easily and efficiently.

Previously we have been dividing up our view logic into separate view files to keep our applications modular, and working with React is no different. Throughout the week you will be separating your UI into separate view files, which in React are called Components. A Component is part of the React framework, and its role is to display a section of our user interface. Well-designed React applications have many small components with a single responsibility. However, in this lesson we will be focussing on learning the syntax used by React, so we will only have one component that describe the UI.

The component in our application will be responsible for displaying and keeping track of the total, as well as displaying a ‘deposit’ and ‘withdraw’ button that will update the state of the component with clicked, updating the displayed total. The React Components has some methods that will help us do this.

## Create Piggy Bank Application

Let’s begin by using create-react-app to give us a start-point.

npx create-react-app piggy\_bank

cd piggy\_bank

npm start

And we will delete the boilerplate code we don’t need:

1. the logo file
2. the logo input in App.js
3. the JSX from App.js’ return method

rm src/logo.svg

*// src/App.js*

**import** React **from** 'react';

*// DELETED*

**import** './App.css';

**function** App() {

**return** (

*// ...*

);

}

**export** **default** App;

If we look at what the component returns from the render function, we can see there is some new syntax there. It looks like HTML, but we know that it’s not because it is inside a JavaScript file. This syntax is called JSX and is what we can use in React to describe what we want to be rendered to the page. We are going to be looking more at JSX next. For now let’s delete the boilerplate code, so we can write our own.

**import** React, { Component } **from** 'react';

**import** './App.css';

**function** App() {

**return** (

*// DELETED*

);

}

**export** **default** App;

And let’s render “Hello World!” to the page to ensure we don’t have any errors at this stage.

**class** App **extends** Component {

render() {

**return** (

**<**h1 className**=**"title"**>**Hello World**!<**/h1> /**/** NEW

);

}

}

We have seen how to render to the page using React. Let’s create a PiggyBank component that is going to be responsible for rendering the UI. Our component rendering hierarchy is going to be App renders PiggyBank.

### Creating a Component

When we create a React Component we use a few es6 language features:

1. Destructuring

When we import from ‘react’, we are importing the whole React object, as well as extracting out a particular part of the package, the Component object. Doing this means we don’t need to type React.Component when we want use it, instead we can just type Component.

1. Class Syntax with Inheritance

We create components using the Class syntax. We will create a PiggyBank class which inherits from React’s Component using the extend keyword, and calling super in the constructor. This gives us access to all of the React Components properties and methods, while rewriting our own custom components.

Note: Under the hood this is still prototypal inheritance, but using the new Class syntax we can inherit from React’s Component to create our own custom Components.

Every React Component must implement a render method that returns a DOM element, an array of DOM elements or null. (We will be looking more at this new syntax inside the render method in a moment.)

touch src/PiggyBank.js

*// src/PiggyBank.js*

**import** React, { Component } **from** 'react';

**class** PiggyBank **extends** Component {

}

**export** **default** PiggyBank;

What is the only method that our Component has to implement? render, which must return a DOM element, an array of DOM elements or null.

**class** PiggyBank **extends** Component {

render() {

**return** (

**<**h1**>**Hello World**!** I am a PiggyBank**<**/h1>

)

}

}

This will not be rendered to the page, until is it rendered by another component. To add it to the rendering hierarchy, we need to tell the App component to render it.

Next, we need to import and use our PiggyBank into our App.js file.

*// src/App.js*

**import** React, { Component } **from** 'react';

**import** PiggyBank **from** './PiggyBank'; *// NEW*

**class** App **extends** Component {

render() {

**return** (

**<**PiggyBank **/>** *// MODIFIED*

);

}

}

**export** **default** App;

Note: all JSX tags must be closed, either with a separate closing tag or self-closed.

We should now see that the PiggyBank is being rendered on the page.

## State and Props

We are now going to look at two mechanisms by which React controls data flow in the application. Components have two type attributes that they can use to store and receive data: state and props.

### Props (or Properties)

Props (or properties) are attributes that are given to a component by their parent component. Props cannot be changed, they are immutable. Props are the way we can pass data from one component to another. Let’s pass a title attribute from the App component to PiggyBank component, so that PiggyBank can display it.

We define the prop in JSX like we set attributes on an HTML component, but we get to define the name of the prop (we will use ‘title’). We can either pass the prop as a string or use the braces ({}) to pass in other JavaScript values.

*// src/App.js*

**import** React, { Component } **from** 'react';

**import** PiggyBank **from** './PiggyBank';

**function** App() {

**return** (

**<**PiggyBank title**=**"Savings Pig" **/>**

);

}

**export** **default** App;

We can now access this ‘title’ prop inside the PiggyBank component from the props object using ‘this.props’. Let’s display it on the page inside the h1. When we are writing JSX, we have to wrap the JavaScript in braces, so that it can be differentiated from the JSX.

*// src/PiggyBank.js*

**class** PiggyBank **extends** React.Component {

render() {

**return** (

**<**div className**=**"bank-box"**>**

**<**h1**>**{**this**.props.title}**<**/h1>

**<**/div>

);

}

}

## State: Displaying a total

State is where a component can store data. State, unlike props, is defined in the component itself and can be changed. When state is changed, React renders the page. Changing (or setting) the state to cause a rerender of the page is the way we can quickly and easily update the page.

We are going to use state to store the Piggy Banks total. The total is a value that will be changed and when it is changed (with a user clicking deposit or withdraw) we want to rerender the page to display the new value.

Let’s set up our initial state, which will be a property of the class called state and will be an object. We want a total property that starts at zero. We can then display this. Our initial state is defined in the constructor of our component class.

*// src/PiggyBank.js*

**class** PiggyBank **extends** Component {

**constructor**(props) { *// NEW*

**super**(props);

**this**.state **=** {

total: 0

}

}

*// ...*

}

We can display this on the page, accessing the total with this.state.total. Because a React Component must either return one DOM element, or an array of DOM elements, we need to wrap our two separate elements in a <Fragment> tag, which we can get from React by using destructuring.

**import** React, { Component, Fragment } **from** 'react'; *// MODIFIED*

**class** PiggyBank **extends** Component {

*// ...*

render() {

**return** (

**<**Fragment**>** *// NEW*

**<**h1**>**{**this**.props.title}**<**/h1>

**<**p**>**Total: £{**this**.state.total}**<**/p> /**/** NEW

**<**/Fragment> /**/** NEW

);

}

}

## Updating State

Now we want to add a button that will increase the total when we deposit money into the piggy bank. Let’s start by adding a ‘Deposit’ button to the render method.

render() {

**return** (

**<**Fragment**>**

**<**h1**>**{**this**.props.title}**<**/h1>

**<**p**>**Total: £{**this**.state.total}**<**/p>

**<**button**>**Deposit**<**/button>

**<**/Fragment>

);

}

When a user clicks on this button, we want an amount to be added to the total, and for this change to be reflected on the page. We can trigger this rerendering of the page by updating the state with React’s method, setState. Every time we call setState render will be called and the page will be updated to reflect the changes.

It might seem inefficient to render every time, but remember React uses the virtual DOM to only update the DOM elements that have changed, keeping it highly performant. There is where we start to see the real power of React. Let’s write the method that is going to be responsible for incrementing the total. It’s going to do this by calling setState.

**class** PiggyBank **extends** Component {

**constructor**(props) {

**super**(props);

**this**.state **=** {

total: 0

};

}

deposit() {

**this**.setState();

}

*// ...*

}

setState takes a callback that will be passed the previous state. We can the increment the value from the previous state. We return a new object, which React will then use to update the state. Because the total already has a ‘total’ property, is will assign the new value to it.

deposit() {

**this**.setState(prevState **=>** {

**return** {total: prevState.total **+** 5};

});

}

Now we call set this on the button’s click event in the JSX using the onClick attribute.

render() {

**return** (

**<**Fragment**>**

**<**h1**>**{**this**.props.title}**<**/h1>

**<**p**>**Total: £{**this**.state.total}**<**/p>

**<**button onClick**=**{ **this**.deposit }**>**Deposit**<**/button>

**<**/Fragment>

);

}

If we now try and click the button on the page, we get an error. This is because we have a context problem. The deposit method is being passed as a callback to the button, therefore inside deposit, this isn’t the PiggyBank class. We want it to be the PiggyBank class, so that it can access the state.

### bind

The way React suggest to solve this problem is to the bind method. bind is a function method (in JavaScript, functions are first class objects and therefore can have methods stored on them). It allows us to call it on a function and pass in whatever context we want it to have. It returns a new function with the specified context. React suggests to create this new version of the function as a property of the class.

*// src/PiggyBank.js*

**constructor**(props) {

...

**this**.deposit **=** **this**.deposit.bind(**this**);

}

Note: There are other solution to the problem, for example binding the function as it’s passed to the onClick with <button onClick={ this.deposit.bind(this) }>Add</button>, or to wrap it in an arrow function with <button onClick={ () => { this.deposit() } }>Add</button> but binding in the constructor is the preferred option (and the one recommended in the React documentation) because the bind only has to happen once, when the class is initially set up. If we put it in our render() method, it would happen each time the JSX re-renders.

Now when we click on the button we should see the page being updated, with the total being incremented by 5 each time.

# Chrome Dev Tools

An other advantage of React is that there are powerful development tools in chrome. Download the following Chrome Dev Tool extension and you will have a new ‘React’ tab in the dev tool. (You might need to reopen the Chrome Dev Tools window after installing it). [Link to React-Dev-Tools] (https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en)

By clicking on the different components in the React Dev Tools we can see any props or state that they have. You can also see it being updated live as the state changes.

### Tasks: (15 minutes)

1. Define the amount to be deposited in the App component and pass it has a prop to the PiggyBank component. PiggyBank’s deposit method should then access the amount off it’s props.
2. Add a ‘withdraw’ button which decreases the total by the amount defined and passed down from the App component, preventing it from going below 0.

### Solutions:

1. Define the amount to be deposited in the App component and pass it has a prop to the PiggyBank component. PiggyBank’s deposit method should then access the amount off it’s props.

*// src/App.js*

**<**PiggyBank

title**=**"Savings Pig"

depositAmount**=**{5} *// NEW*

/>

*// src/PiggyBank.js*

deposit() {

**this**.setState(prevState **=>** {

**return** {total: prevState.total **+** **this**.props.depositAmount}; *// NEW*

});

}

1. Add a ‘withdraw’ button which decreases the total by the amount defined and passed down from the App component, preventing it from going below 0.

*// src/PiggyBank.js*

**constructor**(props) {

*// ...*

**this**.withdraw **=** **this**.withdraw.bind(**this**)

}

withdraw() {

**this**.setState(prevState **=>** {

**let** newAmount **=** prevState.total **-** **this**.props.depositAmount;

**if**(newAmount **<** 0){

newAmount **=** 0;

}

**return** {total: newAmount};

});

}

## Recap

 What method does every React component have to implement? Name two differences between props and state? How do we trigger a rerendering in React?

# Conclusion

React allows us to render our UI as components, providing structure to our front end. It’s use of the virtual DOM and rerendering triggered by the setState() method makes updating the DOM really efficient.

We can now create single component applications, using JSX syntax to render information to the screen. We have seen how to trigger a rerender of the UI by updating a component’s state on user interaction. We have also seen how to pass props down from a parent component.

Lastly, React Dev Tools offers a way of inspecting components’ props and state during development.

## Additional Resources

JSX Gotchas - https://shripadk.github.io/react/docs/jsx-gotchas.html