# Introduction to React

These instructions guide you through a gentle introduction to different React elements. It creates a front-end for the petshop application, assuming that you finished the previous exercises with the back-end and mongodb.

Use the documentation at <https://reactjs.org/docs/getting-started.html> to help you.

## 1. Set-up and initialisation

You need to begin by creating a React application. Open a command prompt or terminal and change to a new directory where you want to store your React files. This should not be the same place as your express work from previous weeks.

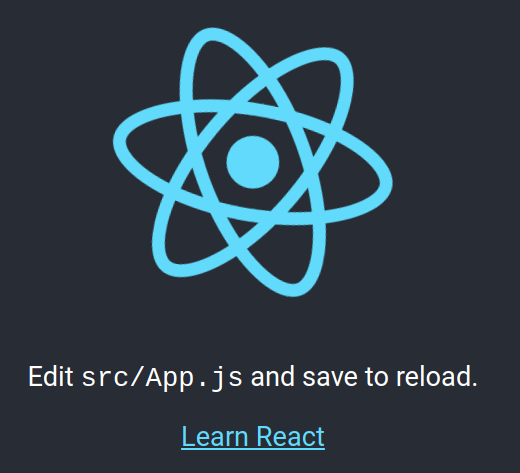
In these instructions we are assuming your React front-end is in the petshop/frontend-react directory.

From the petshop directory type the following instructions:

npx create-react-app frontend-react  
cd frontend-react  
npm start

The npx command should be installed in the same directory as your npm and node executables. You will need to put the full path to it in the command if running on a SHU lab machine.

The framework will launch your default browser once it has started the server. You should see something that looks like the image shown below.



First screen after loading your React application.

## 2. Creating components

We will begin by adding two components using each of the available types: [components as functions](#components-as-functions) and [components as classes](#components-as-classes).

### Components as functions

Our first function is a simple Hello World. Add a new file to the src directory called AnimalFunction.js to your project and put the following code into it.

import React from 'react';  
  
const AnimalFunction = (props) => {  
 return ( <p>This is a simple component</p> );  
}  
  
export default AnimalFunction;

You will now need to import the file into App.js. Put this line at the top of the file

import AnimalFunction from './AnimalFunction';

At the same time edit App.js:

* Remove all of the code that is inside the <div> element in the App() function.
* Put this line of code into the <div> element: <AnimalFunction />

The project will rebuild and restart automatically. You should now see a line of text in the browser window.

### Components as classes

Let’s make a class-based version of the same component. Create a file called AnimalsClass.js with the following code:

import React, { Component } from 'react';  
  
class AnimalsClass extends Component {  
 constructor(props) {  
 super(props);  
 }   
  
 render() {  
 return (  
 <div>  
 <p>A simple class-based component</p>  
 </div>  
 );  
 }  
}  
  
export default AnimalsClass;

Import the file into App.js and add an instance of the class to the <div> element of the App() function.

The project will rebuild and restart automatically. You should now see another line of text in the browser window.

## 3. Using state and props

Now that we have some working components let’s put some data into them. We will modify the class component (AnimalsClass) so that it defines a list, and the function component (AnimalFunction) so that it represents an item within the list. The former will hold data in its (mutable) state and the latter in its immutable props object.

### The function-based component

This is a very simple component. It will receive some data in its props object and will display some (or all) of that data. The data will be passed on the object hierarchy from the dynamic class-based component. Change the return call in AnimalFunction.js to:

const AnimalFunction = (props) => {  
 return (  
 <li>The next animal is {props.name}</li>  
 );  
}

### The class-based component

This component (AnimalsClass) is far richer. Because it has state it will be dynamic. We’ll start with a simple component that instantiates a list and an AnimalFunction but is otherwise static. Notice the import of the AnimalFunction component, that it passes data into that component to hold in its props and that there is a simple constructor.

import React, { Component } from 'react';  
import AnimalFunction from './AnimalFunction';  
  
class AnimalsClass extends Component {  
 constructor(props) {  
 super(props);  
 }   
  
 render() {  
 return (  
 <div>  
 <ul>  
 <AnimalFunction name="Fido" />  
 </ul>  
 </div>  
 );  
 }  
  
}  
  
export default AnimalsClass

Modify the constructor so that it holds some state. Add this code after the super(props) call.

this.state = {  
 animals: [],  
 error: null  
};

And now we need to use the state object. Let’s change the render method again:

render() {  
 let data = this.state.animals || {};  
 return (  
 <div>  
 <p>Here are the animals</p>  
 <ul>  
 {data}  
 </ul>  
 </div>  
 );  
}

Now when the page refreshes you won’t see anything but you have the infrastructure. Once the state.animals array is populated you will see new list items in the page. Notice that we initialise data with either of two values. The first time that the page loads state.animals will be an empty array, we’ll dynamically add data to it later. When the data held in state changes the page will reload automatically and will display the data.

We will change the state of the AnimalsClass component in response to it successfully loading. This code will go into a method named componentDidMount that is called once the page has finished loading the component.

Add the following method inside your AnimalsClass class:

componentDidMount() {  
 let animals = [  
 { id:"1", name:"Logan", age:"5", breed:"Border Collie", colour:"Black, white, tan" },   
 { id:"2", name:"Ralph", age:"2", breed:"Cocker Spaniel", colour:"Ginger" },  
 { id:"3", name:"Gryphon", age:"9", breed:"Staffie X", colour:"grey" }  
 ];  
 let temp = [];  
  
 animals.forEach(item => {  
 temp.push(<AnimalFunction  
 key={item.id}  
 name={item.name}  
 age={item.age}  
 breed={item.breed}  
 colour={item.colour}  
 />);  
 });  
 this.setState( {animals: temp} );  
 }

Here we:

* Create an array of animals
* Iterate across the array, create an AnimalFunction component from each element and add the component to an array. The braces around the call to get the value of each property within the forEach loop are important.
* Change the state of the component by adding the array of animals to it. Notice the use of the setState() method to do this.
* We set a property called key on each of the AnimalFunction. This is used by react to identify components that undergo change.

You should now see the names of the animals in the initial array shown in your Web page.

A reference solution up to this point can be found in solutions/petshop/frontend-react-no-backend.

## 4. Connecting the UI to the backend

Our final stage is to wire the React frontend to the express backend. We’ll do this by changing the way in which the componentDidMount method fills the state.animals array. Here is some code that uses the fetch API (<https://alligator.io/js/fetch-api/>) to connect to a remote resource:

componentDidMount() {  
 fetch('/petshop/animals')  
 .then(data => data.json())  
 .then(res => {  
 if (res.error)  
 this.setState({ error: res.message });  
  
 let temp = [];  
 res.forEach(item => {  
 temp.push(  
 <AnimalFunction  
 key = {item.\_id}  
 name = {item.name}  
 age = {item.age}  
 breed = {item.breed}  
 colour = {item.colour}  
 />  
 );  
 });  
 this.setState( {animals: temp} );  
 });  
 }

The React project will reload but you won’t see any data. This is because of a cross-site page load security violation. We can avoid this by configuring a proxy with the React application. Open the package.json file of the React application and add the following item below the scripts section (items in the file need to be comma separated):

"proxy": "http://localhost:3050"

Now everything ought to work. You must restart the React application for the change to be loaded, which will open it in your browser, where you can see it connected to the database.

Remember:

* You need to have the database running in a terminal.
* You must be running the backend in a second terminal.
* Run the frontend project from a third terminal.

### 5. Exercises with React

1. Modify the application so that the displayed data is in a table rather than a list.
2. Create a React form that can be used to send new pets to the backend. Store these in the database.
3. Add some error handling to the application so that invalid or meaningless data (or incomplete pets) cannot be added to the system. You may want to do this using an HTML5 form (<https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation>).
4. Implement the ability of visualising only one pet. You can do this by using the \_id field that MongoDB generates for each document.
5. Now implement the ability to edit each item. You can do this by using the \_id field that MongoDB generates for each document and a simple form.
6. Add error handling code for database calls.
7. Add additional code to return HTTP status errors if resources (including routes) are unavailable.
8. Add some style to the page. Look at Bootstrap (<https://getbootstrap.com/docs/3.3/css/>) or Google’s Material design (<https://material.io/>) for inspiration or code.
9. Modify the application so that each pet can be represented by a small avatar. Display the avatar as a circular image.