Intro to ReactJS and Firebase

We will get an inside into ReactJs and the main basic concepts through building a todo list app with some hard-coded input data. We are going to explain everything step by step with details and you can really understand what is happening and concepts like components, lifecycles, states, props, events etc. Then we are going to change the way we obtain the data, it will come from a json placeholder, which is a fake Rest API that acts as a back-end server that feeds this data. Later we are going to use a database (Firebase) to store and get the data.

1. What is React?

ReactJS is a Javascript library created by Facebook and is used for building user interfaces (UIs) and front-end applications.When I say front-end i mean in the browser and it is not the same as when you build something with php or python because your code runs on the server whether is a dedicated server, shared accountor cloud hosting. The code in React runs in the client and there are benefits to this including certain performance bumps in certain areas.

 It makes it easier for developers to manipulate how UI behaves and add more interactive functionality that feels responsive. Many of the concepts introduced by React are against conventional wisdom, but they've worked exceptionally well. Another plus is that you do not have to reload your page for everything to happen, just certain parts.

Is React a library or framework? When you say library you think about something like jQuery where you pull certain features out and use them. For this reason React is often called a framework because of its behaviour and capabilities. React is basically the V in the MVC which stands for Model View Controller, so it's the entire application view.

React is the most popular framework in the industry for now based on the poll and it's a pretty good idea to learn React if you are trying to get a job as a front-end developer. It does the same what other frameworks do, like **VueJs** and **Angular**, with the only difference that React is very small/light and you have to install things separately, while Angular includes everything, like router, HTTP client etc.

2. Why use React?

Vanilla is much more code to do the same thing you could do with a framework, e.g. dynamic routing - **Reinvent the wheel**

ReactJs is a very well **organized** framework and uses self-contained independent **components**.

One big monolithic code base that is very confusing for multiple people to work on.

It's also good for teams because it's organized and everyone can work on a specific task and component.

Javascript does allow us to interact with the DOM and create a dynamic user interface, but you have to do every little task with the DOM manually.

ReactsJs you have something called Virtual Dom and it updates only what is needed, so if you have a blog page it updates only the new post component, not the whole page.
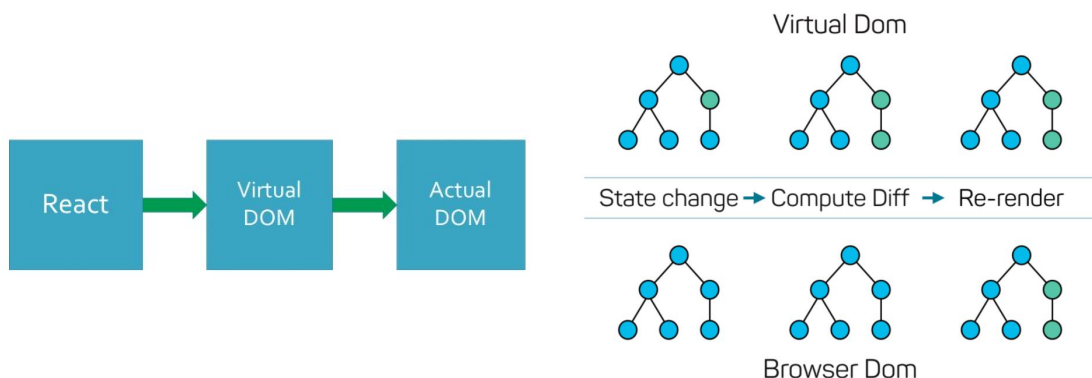
3. JSX

Consider this variable declaration:

```
const element = <h1 className="greeting">Hello, world!</h1>;
```

This syntax is *neither a string nor HTML*. It is called JSX, and it is a syntax extension to JavaScript.

React uses JSX and in the beginning most developers don't like it because they are used to dividing concepts but after some time they find beauty in it. It is easy to create user interfaces and encapsulate each part

into the UI. It also allows React to show more useful error and warning messages.

Under the hood, it gets compiled down to JavaScript-only expression:

```
const element = React.createElement(
  'h1',
  { className: 'greeting' },
  'Hello World'
);
```

The element then will be represented by an object:

```
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

These objects are called "React elements". You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

```
<div id="root"></div>
```

We call this a "root" DOM node because everything inside it will be managed by React DOM.Applications built with just React usually have a single root DOM node. To render a React element into a root DOM node, pass both to ReactDOM.render():

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

4. Components

Instead of separating *technologies* by putting markup and logic in separate files, React separates *concerns* with units called components.

Components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. You can create components in two ways and both ways are equivalent from React's point of view. Still React embraces, class-less programming model, heavily relying on functional components.

Functional Component

```
function helloDPS() {
  return (
    <div>
      'Hello DPS'
    </div>
  );
}
```

Class Component

```
class HelloDPS extends React.Component {
  render() {
    return (
      <div>
        'Hello DPS'
      </div>
    );
  }
}
```

Previously, we rendered an element into DOM, but it was composed only of standard HTML tags, but React can render the same way also Components, for instance:

```
const element = <HelloDPS />;

ReactDOM.render(
  element,
  document.getElementById('root'));
```

For Functional Components, React would render the return value of the function. For Class Components, React would render the return value of the render() method.

Functional Components can only use Hooks, which are very new. Class Components can use lifecycle methods.

5. Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a

form, a dialog, a screen: in React apps, all those are commonly expressed as components.

```
function HelloDPS() {
  return <div> 'Hello DPS Participants' </div>;
}

function App() {
  return (
    <div>
      <HelloDPS />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

6. Props

In a React component, props are variables passed to it by its parent component. They are used to pass data down from your **top level component** and not the other way around.

Props are **immutable** which lets React do fast reference checks and have a better performance.

```
function Hello(props) {
  return <h1>Hello, {props.name}</h1>;
}

function HelloDPS() {
  return <Hello name="DPS">
}

function App() {
  return (
    <div>
      <HelloDPS />
    </div>
  );
}
```

```
ReactDOM.render(
    <App />,
  document.getElementById('root'));
```

Let's recap what happens in this example:

- We call ReactDOM.render() with the App element and the App returns another component called HelloDPS
- The component HelloDPS calls another component Hello with {name: 'DPS'} as the props.
- React DOM updates the DOM to match <h1>Hello, DPS</h1>

7. State

In a React Component, a state on the other hand is still a variable, but directly initialized and managed by the component itself. States are **mutable**, they have worse performance and they can not be accessed from other components.

Functional Component	Class Component

```
import React, { useState } from
'react';

function Example() {
 // Declare a new state
variable:
 const [count, setCount] =
 useState(0);

 return (
   <div>
     <p>Clicked {count}
        times</p>
     <button onClick={() =>
      setCount(count + 1)}>
       Click me
     </button>
   </div>
 );
```

```
class Example extends React.Component {
    constructor(props) {
     super(props);
    this.state = {
      count: 0
    };}
   render() {
    return (
      <div>
        <p>You clicked
         {this.state.count} times</p>
        <button onClick={() =>
         this.setState({ count:
         this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );}}
```

8. Lifecycle Methods

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are:

1. *Mounting* - means putting elements into the DOM
2. *Updating* - a component is updated whenever there is a change in the component's state or props
3. *Unmounting* - a component is removed from the DOM

In the picture below you can find the most important built-in methods you can use depending on the phase. The render() method is required and will always be called in Mounting and Updating, the others are optional and will be called if you define them.
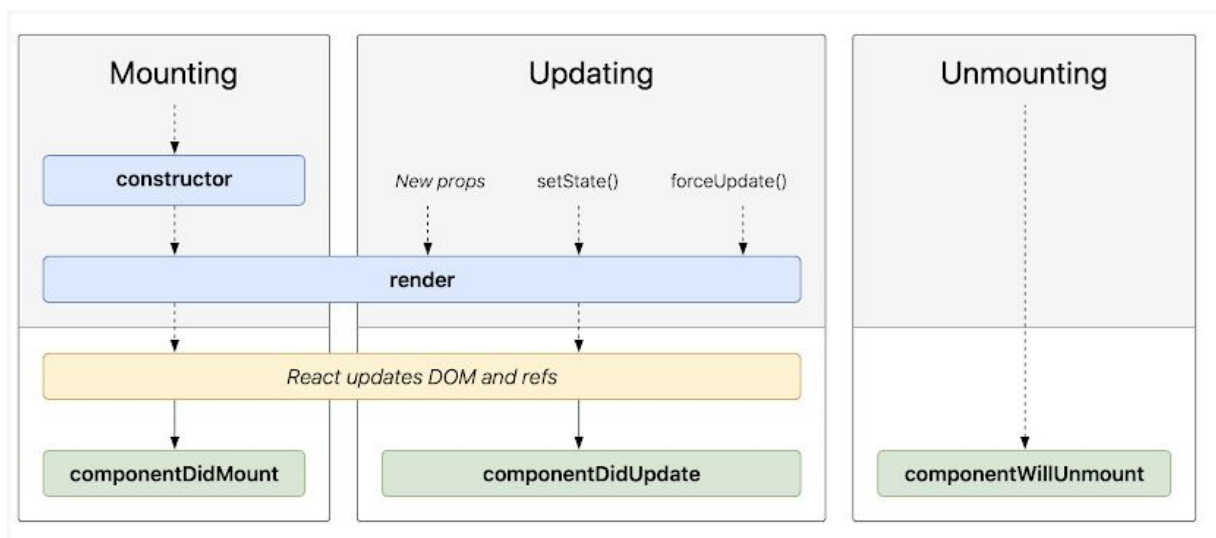

9. Thinking in React Components

Now we are going to start with creating our app, but before i want to let you know what basics you need for this tutorial and in general for ReactJs:

- You should know basic javascript programming, including objects, arrays, conditionals, functions, because you can   not learn javascript and ReactJs at the same time.
- You should understand classes, destructuring, high order array methods (forEach, map, filter), arrow functions, fetch API and promises that are included in ES6, because they are heavily used in React. You could learn this stuff while you are learning React but I think that you may have some trouble distinguishing what is actually React and what is simple javascript.

! Break The UI Into A Component Hierarchy

As I said, ReactJs puts everything needed in the user interface into self-contained components which make the application really well-organized. We are going to end up with something like this, a todo list application and the reason why I am showing this now is because I want you to get a structure into your head. It has a bunch of components. You can see the different components that are outlined. We have TodoItems for every item in the list, a Todos that wraps the TodoItems, an AddTodo that is a form for adding new TodoItem, Header, Search for searching into the Todos and an About Page just to show how the routing works.
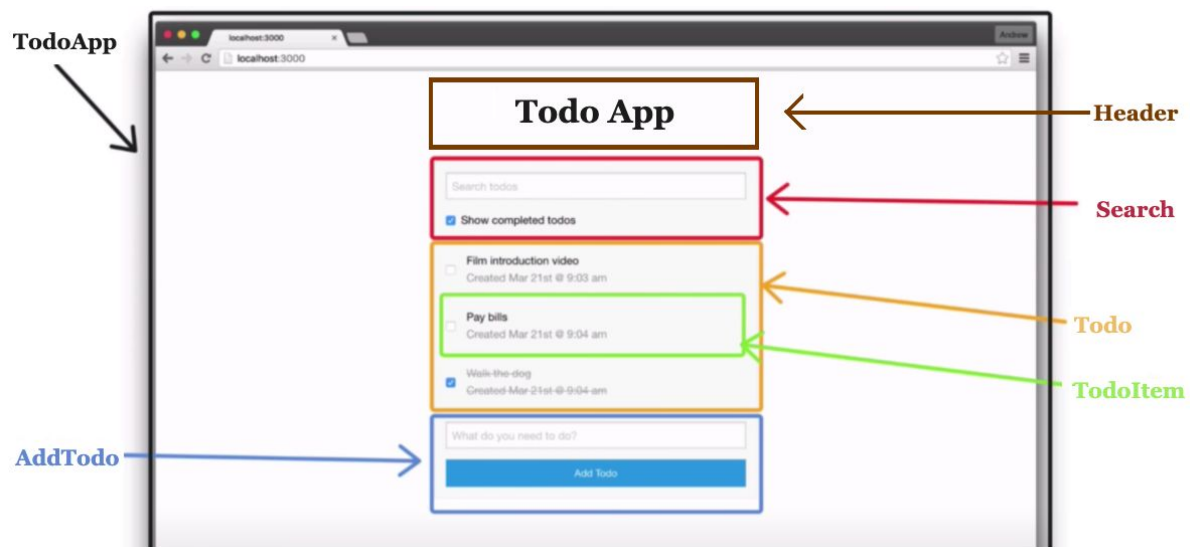
10. Starting the App

We are going to use custom CSS and not Bootstrap. I know it's not the best looking app but using a framework will distract us from ReactJs and I also want to show how to style with basic components.

- Download NodeJS if you already do not have it on your pc, because we need to use the node package manager (npm).

● Download the React Developer Tools for Chrome because with this tool you can see the states and props from the browser.
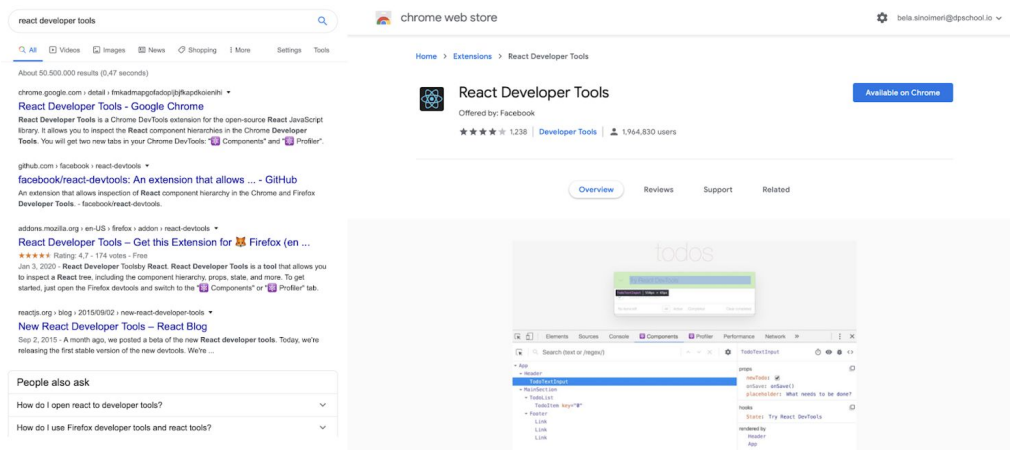
Now I am going to use a cli tool for creating the app create-react-app. It will be using a webpack which is a bundler but needs no configuration from us, it comes bundled with a dev server with hot reload. It will auto reload as we save.

Open Visual Studio, create an empty folder with the name TodoApp and in the terminal run **npx create-react-app .**

The dot means i want to generate a React app inside this empty folder. Let's explain a little bit what are the files we get after running this command:

Package.json - It is a manifest file which has all the dependencies and packages. Main package includes 3 parts: React, React-Dom that is used for loading components into the browser and React-Scripts that have to do with the dev server and being able to compile and run tests. For scripts we have react-scripts start, build (to compile so the browser can read), test, eject (dont touch only if you want to customize the webpack file).

```json
{} package.json ×
{} package.json > ...
1    {
2      "name": "todoapp",
3      "version": "0.1.0",
4      "private": true,
5      "dependencies": {
6        "@testing-library/jest-dom": "^4.2.4",
7        "@testing-library/react": "^9.5.0",
8        "@testing-library/user-event": "^7.2.1",
9        "react": "^16.13.1",
10       "react-dom": "^16.13.1",
11       "react-scripts": "3.4.1"
12     },
13     "scripts": {
14       "start": "react-scripts start",
15       "build": "react-scripts build",
16       "test": "react-scripts test",
17       "eject": "react-scripts eject"
18     },
```

Index.html - its important to understand that React is a Single Page app and everything is put inside the id of root. This is an extremely important concept of ReactJs because it's where your entire application runs. If you want to use Bootstrap CDN or any other CDN you can put it in this file.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Index.js - We can see how the components are put inside the root. It is rendering the App into the root with getElementById. The service worker is more used for PWA and for offline working apps, so we can remove it.

```js
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

App.js - It includes a class component. You can either import Component either use extends React.Component and both ways are doing the exact same thing only expressed differently. Render is the lifecycle method and it returns something in JSX. You can include js into brackets like for example the logo. You can not use the class attribute in React, you have to use className.

```js
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Now we run the command *npm start* at the terminal and this command will start the development server. This is the template we always get when we create a new React app. We need to clean it by deleting

index.css/logo and remove them from all imports. Also in the App.css we remove the default css and we put core css like:

```css
* {
box-sizing: border-box;
margin: 0;
padding: 0;}

body {
font-family: Arial, Helvetica, sans-serif;
line-height: 1.4;}

a {
color: #333;
text-decoration: none;}
```

The only thing I want to show from the App it's a div with className App and a h1 App.

```jsx
import React, {Component} from 'react';
import './App.css';

class App extends Component {
 render() {
 return (
   <div className="App">
   <h1> App </h1>
   </div>
 );
 }
 }
export default App;
```
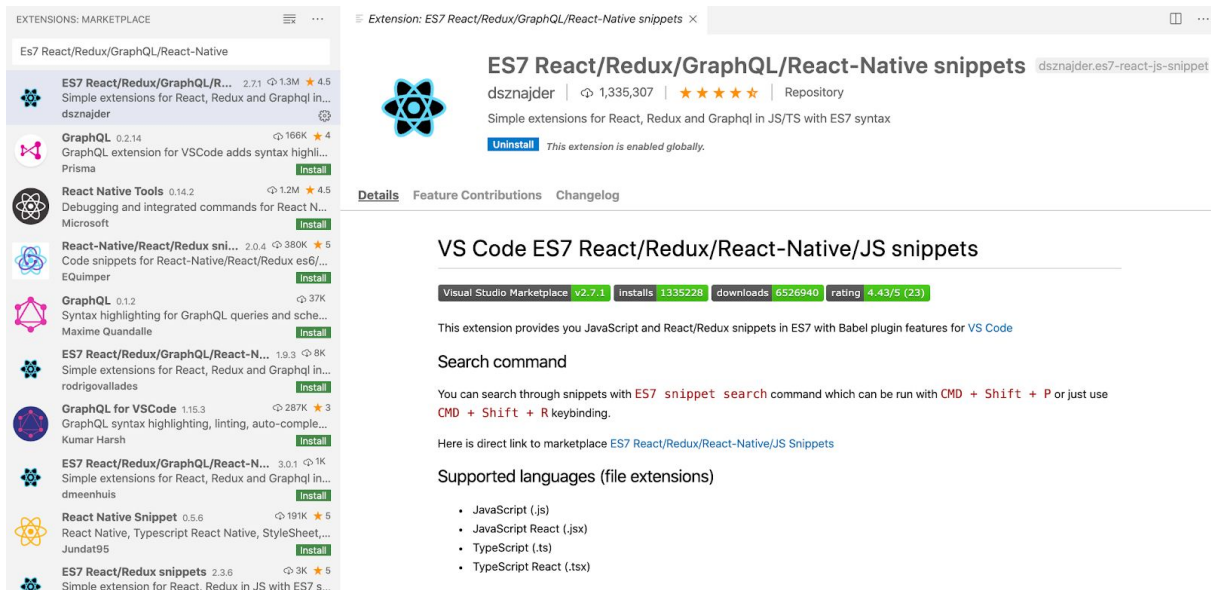

Now we have to create a class component that wraps all the TodoItems, so lets create a component folder and inside it a file with the name Todos. Be careful because components should always be uppercase.

I usually use a lot of extensions and one of my favorites is called *Es7 React/Redux/GraphQL/React-Native snippets*. This one I would highly recommend to you for React Development. It will allow you to generate components very quickly with just writing 3 letters:

- functional - **rfc**
- class **-rcc**

It generates the className from the file name.This can speed things up.

This will be the autogenerated file. We just add the h1 to show that the component works.

```jsx
import React, { Component } from 'react'

export default class Todos extends Component {
    render() {
        return (
            <div>
                <h1>Todos</h1>
            </div>
        )
    }
}
```

To be able to see the component we need to import it at **App.js**.

```jsx
import React, { Component } from 'react';
import './App.css';
import Todos from "./component/Todos";

class App extends Component {
 render() {
 return (
    <div className="App">
    <Todos/>
    </div>
 );
}
}
export default App;
```
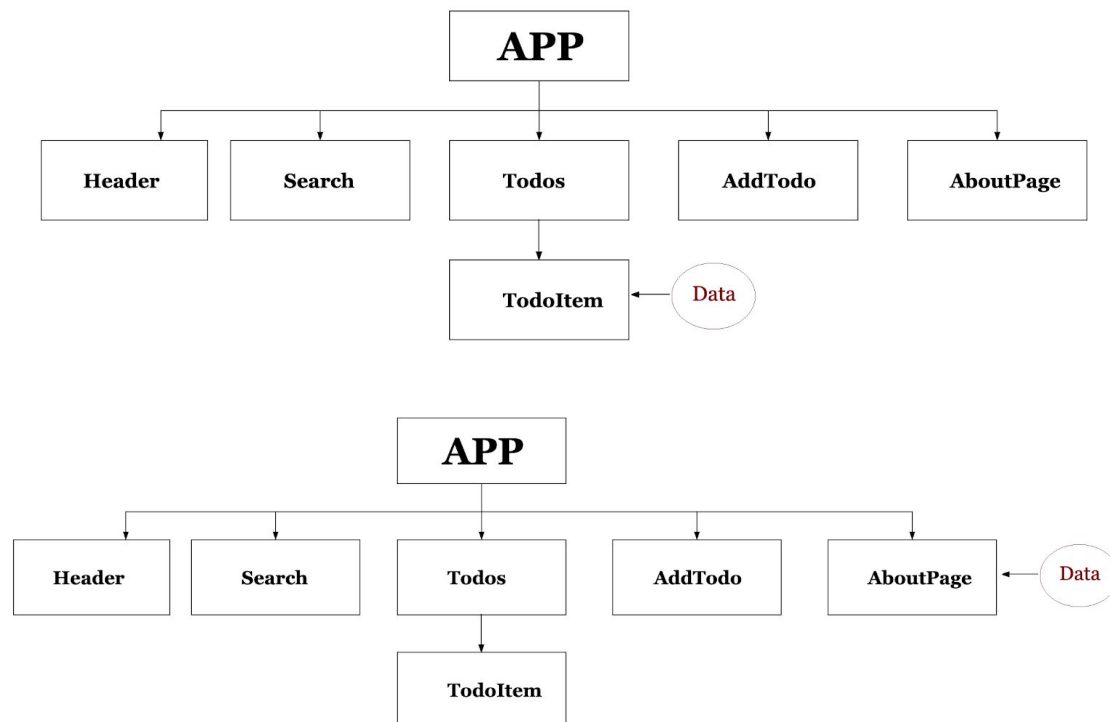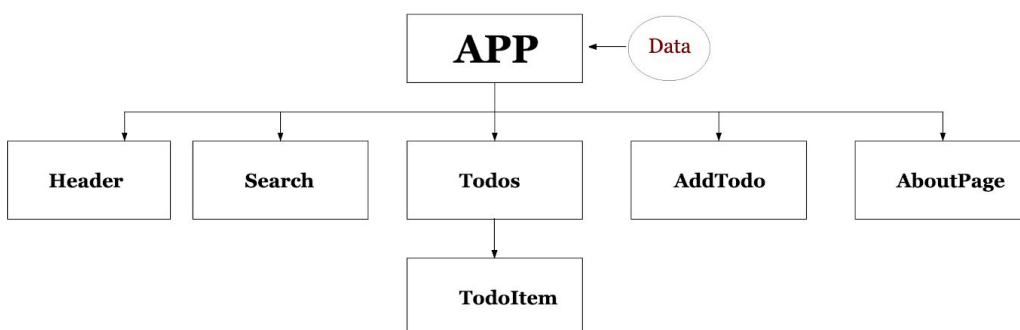
11. Deciding about the state

One important thing is to decide where we should **put the state**, also based on the documentation "Thinking in React". Every component can have state which is an object that determines how that component renders and behaves. Sometimes in the app we need to have an **"application level" state** that you want to share between multiple components by using a state manager like Redux or context API enabled from React, without the need to use third party packages.

```
                          APP
        ┌────────┬────────┼────────┬────────┐
        ▼        ▼        ▼        ▼        ▼
     Header   Search    Todos   AddTodo  AboutPage
                          │
                          ▼
                      TodoItem  ◄──  Data
```

```
                          APP
        ┌────────┬────────┼────────┬────────┐
        ▼        ▼        ▼        ▼        ▼
     Header   Search    Todos   AddTodo  AboutPage ◄── Data
                          │
                          ▼
                      TodoItem
```

We are not going to use none of them, but we are going to put all the data (state) into the upper level, in the App.js. In this way every component can get access to the state.

```
                          APP  ◄── Data
        ┌────────┬────────┼────────┬────────┐
        ▼        ▼        ▼        ▼        ▼
     Header   Search    Todos   AddTodo  AboutPage
                          │
                          ▼
                      TodoItem
```

The first way we are going to obtain data is by hard-coding a javascript object, more specifically an array of objects like below:

```
state = {
    todos: [
        {
            id: 1,
            title: "Take out the trash",
            completed: false
        },]
};
```

Let's copy it and add it two more times.

```
state = {
    todos: [
        {
            id: 1,
            title: "Take out the trash",
            completed: false
        }, {
            id: 2,
            title: "Going for shopping",
            completed: false
        },{
            id: 3,
            title: "Saving Money",
            completed: false
        }]
};
```

Even though we are hard-coding it for now, we need to keep the same logic as when we work with databases/api. For this reason we assign a unique id for each task. Try it out with *console.log(this.state.todos)* or directly check the state from the React Developer tool at **App**. You can see that at Todos tab the state is empty.

```
App
    Todos

props
    new prop: ""

state
    ▼ todos: [{…}, {…}, {…}]
        ▶ 0: {completed: false, id: 1, title: "Take out the tras…}
        ▶ 1: {completed: false, id: 2, title: "Going for shoppin…}
        ▶ 2: {completed: false, id: 3, title: "Saving Money"}
```

Now i want to pass them into the Todos component into App.js like below:

```
<Todos todos={this.state.todos} />
```


Go in the Todos.js and between render and return add:

```
console.log(this.props.todos)
```

Why props? So let's focus on what we learned from the theoretical part, it comes from the App.js and it is passed to the Todos file as props (from parent to child).
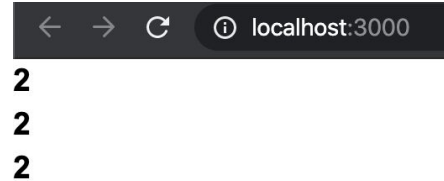

We need to loop through the Todos and **return the title** for each one of them. We need to use the map method which is a high order array method and it's used for different things. It can return an array from an array but we are using it to return a JSX from an array. The idea is that we loop through every object in the Todos array and return something for e.g. 2, so we will see 2 for 3 times. After we checked that the logic works we need to replace 2's with todo.title's. You can use a function inside the map, but I will prefer sticking with arrow function during this tutorial.

```
return (
this.props.todos.map((todo) => (
<h3> {1 + 1} </h3>
)
)
)
```
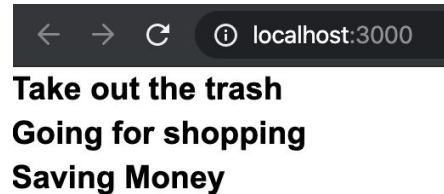
localhost:3000

2
2
2

```
return (
this.props.todos.map((todo) => (
<h3> {todo.title} </h3>
)
)
)
```

localhost:3000

Take out the trash
Going for shopping
Saving Money

We will get a warning. This is a very common warning in React. We need to ignore it for now, because we are going to deal with it later.

```
⊗ ▶Warning: Each child in a list should have a unique "key"    index.js:1
   prop. See https://fb.me/react-warning-keys for more information.
      in Todos (at App.js:25)
      in div (at App.js:24)
      in App (at src/index.js:7)
      in StrictMode (at src/index.js:6)
```

I do not want to load only the title, but i want to load a **whole** new component with the name TodoItem.

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

export class TodoItem extends Component {
 render() {
   return (
     <div>
      <p>Hello</p>
     </div>
   )
 }
}

export default TodoItem
```
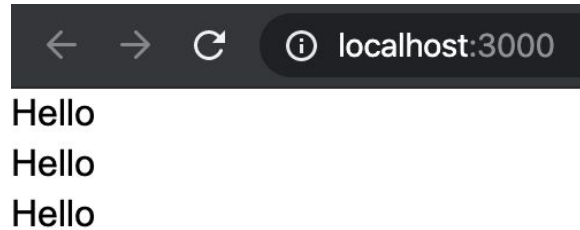
Let's go back to Todos.js, import TodoItem, remove the h3 and output the Todo Item.We will get three times Hello, because we have three Todos and we map through them. We do not want Hello, but that particular item.

```
return (
this.props.todos.map((todo)    =>
(
<TodoItem/>
)
)
)
```
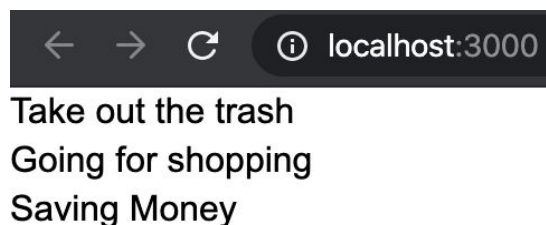
We are going to add todo into the TodoItem Component at Todos.js:

```
<TodoItem todo={todo}/>
```

This **todo** takes every todo as a prop and passes them into TodoItem by simply adding `{this.props.todo.title}` instead of Hello. Now we are seeing each todo title, based on a new component TodoItem.js:

Now we can easily fix the warning that says that each child in an array or iterator should have a unique key prop. This warning does not break the app but let's fix it by putting an unique key using the id we set before.

```
<TodoItem  key={todo.id} todo={todo}/>
```

12. Styling Tips

It's time to look a little bit into the style of TodoItem. There are different ways we can style in React:

1. *Directly as inline coding* - Not the best method

```
<div style ={{backgroundColor: "#f4f4f4"}}>
```

We do not use background-color like in CSS, but we always use camelCase, backgroundColor. This is the same for every kind of styling we want to do!

2. Styled-components

```
<div style ={itemStyle}>

const itemStyle = {
    backgroundColor: '#f4f4f4'
  }
```

This is more useful when we want to do several and different styling inside one div. It is way better to use this, than just directly inline styling.

3. CSS stylesheet

Create a stylesheet for every component only if you have a lot of styling included. Simply import the css file wherever you want to use it. We are not going to work with stylesheets because it is not the focus of our app.

I want to put my **style in a function** above the render because I want my style to change based on if the todo is completed or not. If it's completed I want to pass a line through. To check if this styling works change the second object in todos into completed:true.

```
<div style={this.getStyle()}>
```

```
getStyle = () => {
if (this.props.todo.completed) {
return {
textDecoration: "line-through"
        }
} else {
return {
textDecoration: "none"
        }
    }
}
```

Take out the trash
~~Going for shopping~~
Saving Money

```
getStyle = () => {
return {
background: "#f4f4f4",
padding: "10px",
borderBottom: "1px #ccc dotted",
textDecoration:
this.props.todo.completed ?
"line-through" : "none"
        }
    }
}
```

Take out the trash

~~Going for shopping~~

Saving Money
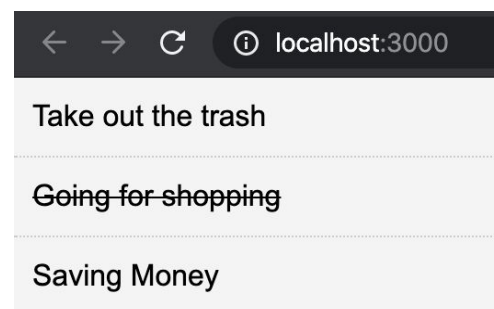
This is a good example of how to create dynamic styling and how to make the code cleaner with simple actions. Next thing i want to do is to put a checkbox near the paragraph:

```
<p>
<input type="checkbox" /> {' '}
{this.props.todo.title} </p>
```
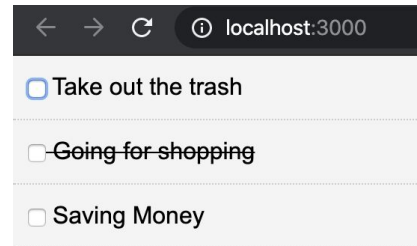

13. Handling events

We need to add an event to the input if we want this checkbox to do something. You can read more about handling events in React here.

```
onChange={this.markComplete}

markComplete(e) {
 console.log(this.props)
}
```

When we click the checkbox we get this error because the function is a custom one and we created it, so we have to bind it to remove the undefined warning.

This is a very common error when working with functions. There are two ways to fix it:

1. this.markComplete.bind(this)

2. Use arrow function

```
markComplete= (e) =>{
    console.log(this.props)
    }
```
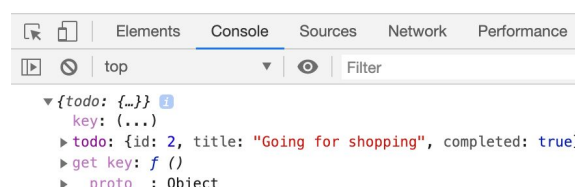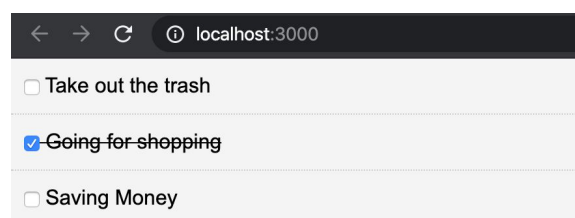
14. Prop Drilling

This is one of the toughest things to understand in React, because we are not using Redux or contextAPI for state management and we have to basically CLIMB the tree. This is usually known as prop or component drilling.

14.1 Checkbox

We basically have our state in App.js, so we can not change for e.g. the completed true or false with setState directly from TodoItem but we have to climb from TodoItem to Todos to App. We do this with props:
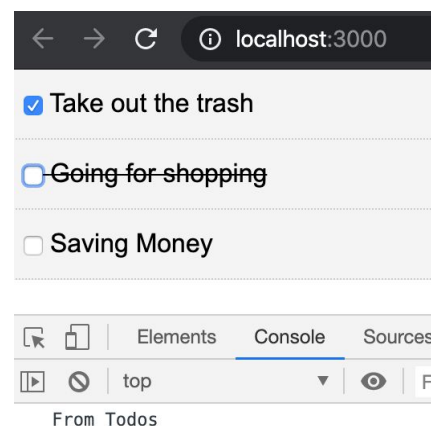
From TodoItem to Todos

We remove the function from TodoItem and we add props to the onChange. Later we add the function into Todos with console.log "From Todos" and we attach markComplete into the TodoItem component.

```
onChange={this.props.markComplete}


markComplete= () =>{
        console.log("From Todos")
        }


markComplete={this.markComplete}
```
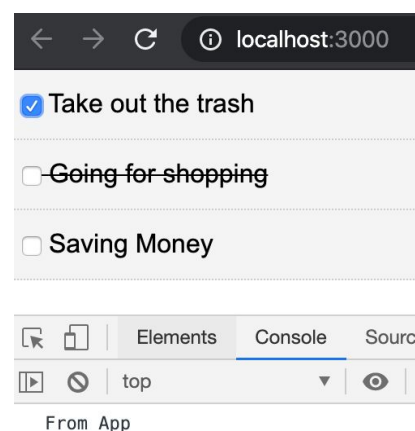
From Todos to App

Again we have to go up one level because the state is not here but in the App.js and we do the same steps.

```
markComplete= () =>{
      console.log("From App")
        }
```

We need to align the line through with the checked checkbox, for this reason we add checked at the input:

```
checked={this.props.todo.completed}
```

We need to know which one we are clicking and marking complete based on the id. We can do this in two option base on React documentation, we choose the second one with arrow function:

```
onChange={(e) => this.props.markComplete(this.props.todo.id, e)}
onChange={this.props.markComplete.bind(this, this.props.todo.id)}
```

Let's also destruct to make the code cleaner, so go above the return and put:

```
const { id, title, completed } = this.props.todo
```

From now on we can just pass `id`, `completed` and `title`.

Now in the markComplete function we add id as argument and we log it. It will return just the id of the TodoItem we clicked and will not do anything else.

```
markComplete= (id) =>{
    console.log(id)
    }
```



```
markComplete = (id) => {
this.setState({todos:
this.state.todos.map(todo => {
if (todo.id === id) {
todo.completed = !todo.completed;}
        return todo;
    })
    });
  };
```
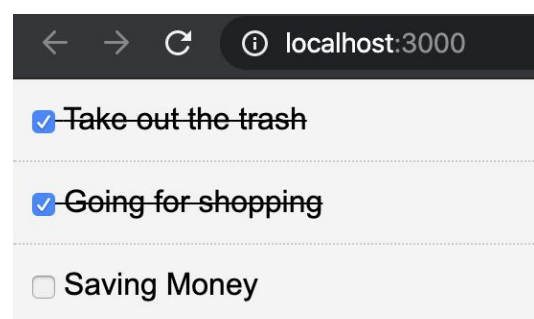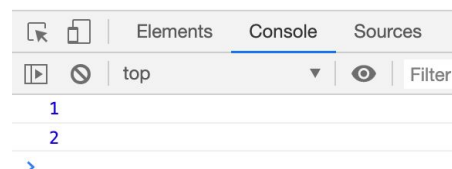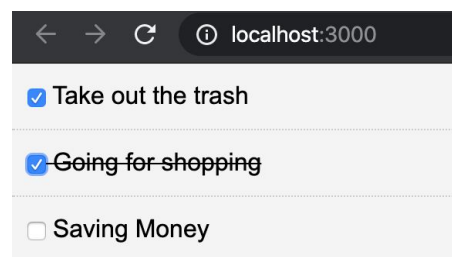
We have to check if the id of the object we are looping in todos (todo) is the same with the id of the one we click in the checkbox and if this condition is true we toggle the completed from true to false and the other way. We are using toggle, because if we just set it to true it will stay like that.

14.2 Delete

Now i want to add a delete button. In order to do this we are going to use the same logic to create a prop and send it up to App.js. At the TodoItem.js we add the button below the title inside the paragraph:

```
<button style={btnStyle}>x</button>
```

And we style it a little bit:

```
const btnStyle = {
 background: '#ffc0cb',
 color: '#fff',
 border: 'none',
 padding: '5px 9px',
 borderRadius: '50%',
 cursor: 'pointer',
 float: 'right'
}
```

We need to add an event to this button so when we click something happens.

```
onClick={(e) => this.props.deleteTodo(id, e)}
```

We need to add the props into TodoItem component at Todos.js:

```
deleteTodo={this.props.deleteTodo}
```

We need to add it also into Todos component at App.js, with no props because we are actually into where the state is saved:

```
deleteTodo={this.deleteTodo}
```

Now we need to create a function to manipulate the state and delete the TodoItem that we want to select. We are going to use another high order

array method, filter which will loop through the array and based on the condition will return another array.

```javascript
deleteTodo = (id) => {
  this.setState({
    todos: [...this.state.todos.filter(todo => todo.id !== id)]
    })
  };
```

We basically need to copy everything that is already in the state so we can use the spread operator and then we create an array with all the elements except the one with the id we choose to delete.

! They are going to come back after we reload because we do not have a backend - *Remember React is about UI.*

14.3 AddTodo Component

Let's create the next class component AddTodo.js:

```javascript
import React, { Component } from 'react';

export class AddTodo extends Component {
 render() {
   return (
     <form style={{ display: 'flex' }}>
       <input
         type="text"
         name="title"
         style={{ flex: '1', padding: '5px' }}
         placeholder="--- Add Todo ---"
       />
       <input
         type="submit"
         value="Submit"
         className="btn"
         style={btnStyle}
       />
```

```
        </form>
    )
  }
}

const btnStyle = {
    background: '#ffc0cb',
    color: '#fff',
    border: 'none',
    padding: '5px 9px',
    cursor: 'pointer',
    fontSize: '16px',
    float: 'right'
  }

export default AddTodo
```

For the styling i also added this at App.css:

```
::placeholder {
font-size: 16px
}

input, textarea{
 font-size: 16px
}
```

We need to import this component at App.js and put it before Todos component. After we run the app something like this will show in localhost:

Now we want to add the functionality. We add a state value and set it to nothing at the AddTodo.js

```
    state = {
        title: ''
      }
```

Then we add the value at the input:

```
value={this.state.title}
```

This will show us a warning because the idea is that when you type something that fires an event and changes the state, so we must have an handler called onChange:

```
⊗ ▸Warning: Failed prop type: You provided a `value` prop to a form field without an `onChange` handler. This will render a read-
  only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.
     in input (at AddTodo.js:11)
     in form (at AddTodo.js:10)
     in AddTodo (at App.js:47)
     in div (at App.js:46)
     in App (at src/index.js:7)
     in StrictMode (at src/index.js:6)
```

```
onChange = {(e) => this.setState({ title: e.target.value })}
```

After adding the onChange we can check the React Developer Tool and log this.state.title to check it from console:
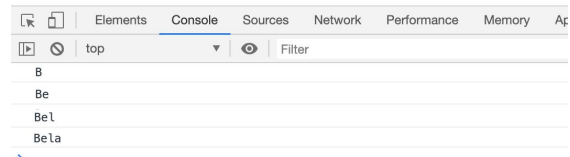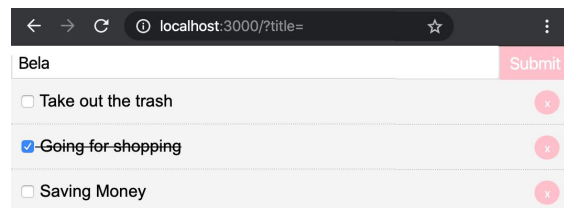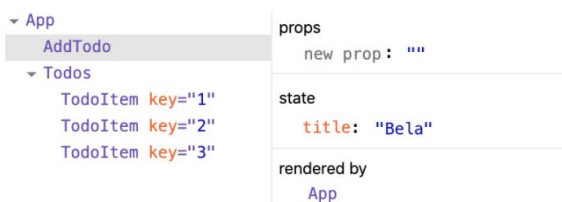
The state is being changed, but what if we had different fields like email, password and we don't want to change them one by one. How to make our code more efficient? So the thing we can do:

```
onChange={(e) => this.setState({ [e.target.name]: e.target.value
})}
```

We put this into the form:

```
onSubmit={this.Submit}

Submit = (e) => {
      e.preventDefault();
      this.props.addTodo(this.state.title);
```
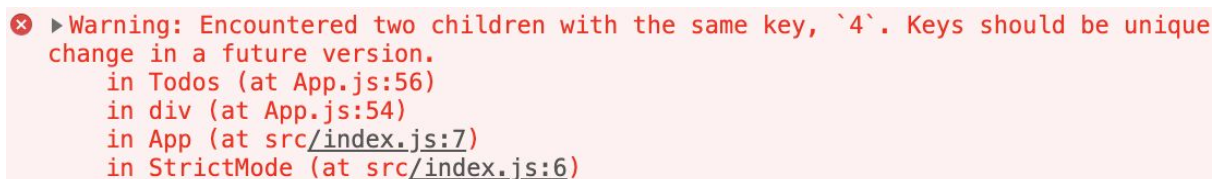
```
        this.setState({ title: ' ' });}
```
We clear after submitting and set the state again to empty. Now we need to add the addTodo prop into the AddTodo component the same way we did the prop drilling for other features:

```
<AddTodo addTodo={this.addTodo}/>
```

```
addTodo = (title) => {
const newTodo = {
id:4,
title,
completed: false
};
this.setState({ todos: [...this.state.todos, newTodo] });
};
```

We will get an error after the second time we submit because in this way we are always going to add todo objects with the same id 4. For this reason we have to find a way to generate unique id's.

```
⊗ ▶Warning: Encountered two children with the same key, `4`. Keys should be unique
   change in a future version.
       in Todos (at App.js:56)
       in div (at App.js:54)
       in App (at src/index.js:7)
       in StrictMode (at src/index.js:6)
```

We are going to install a package called uuid:

npm install uuid

The first step is to:
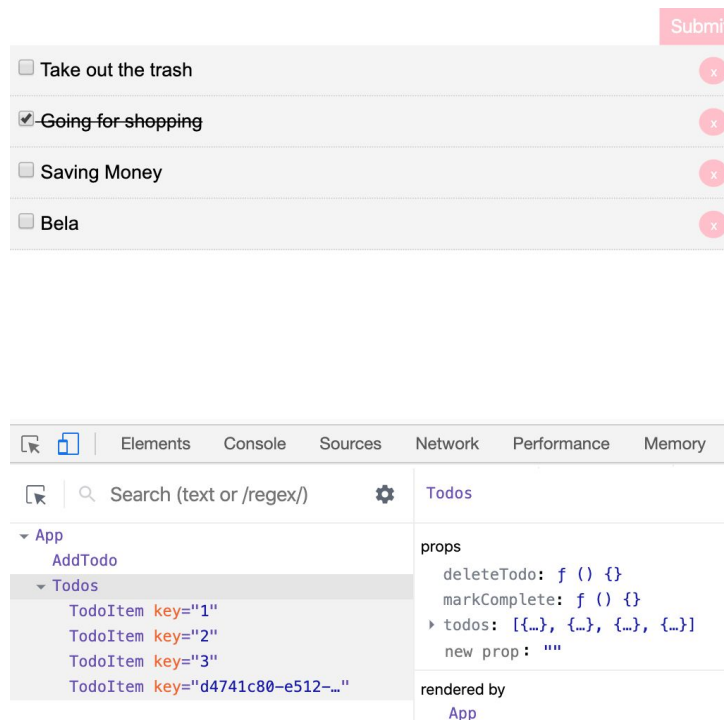
```
import { v4 as uuidv4 } from 'uuid';
```

Then we need to replace the number 4 with:

```
uuidv4()
```

You can also check the documentation for more information.

In the picture below a new entry was submitted with the name Bela, the TodoItem was created and as you can see in the React Developer Tool the key is different because it is created with this uuid.

14.4 Search

We need to create a new component with name Search.js like below:

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

export class Search extends Component {
 render() {
   return (
 <div align="right" style={{backgroundColor: "rgb(51, 51, 51)"}}>
     <input
        type="text"
        className="input"
        style={{ flex: '1', padding: '5px', marginBottom:"20px" }}
        placeholder="--- Search Todo ---"
        onChange={this.props.searchTodo}
        />
   </div>
   )
 }
}


export default Search
```

We need to import this component at App.js

```
import Search from "./component/Search"
```

The second part of adding a search bar will be to create an array of our filtered todos. We will create a new empty array called filtered at state.

```
filtered: []
```

Our original todos are located in the App component, which is the parent. This state is being passed into the Todos component (child) and then in the TodoItem (child of the child). You may ask, what is the point of telling you this? We need to pass data into our `filtered` state every time the TodoItem gets re-rendered.

```
componentDidMount() {
  this.setState({
    filtered: this.state.todos
  });
}
```

After we update the filtered with the todos, we check the input bar. If the input bar of search is empty, it should display all todos. If there is text in the search bar, it should only show todos that contain that text.

```
searchTodo = (e) => {
  let search = [];
  if (e.target.value !== "") {
    search = this.state.filtered.filter(searchitem => {
      const searchlw= searchitem.title.toLowerCase();
      const targetlw = e.target.value.toLowerCase();
      return searchlw.includes(targetlw)
    });
    this.setState({
      todos: search
    })
  } else {
    this.setState({
      todos: this.state.filtered
    })
  }
}
```

After adding the function we need to add the component in the render before the AddTodo:

```
<Search searchTodo={this.searchTodo} />
```

We also need to add `filtered: this.state.todos` at the addTodo function because we want to search also in the todos that we recently added.

What's the big deal and why are we doing all this? The big deal is that now we have a list we can manipulate without altering the original list. All we have to do is modify our `filtered` state and the items being displayed will also reflect that, but we haven't lost the original list by doing this.

This is so far how it looks. You probably question why I styled it like this? I wanted to match it with my Header.
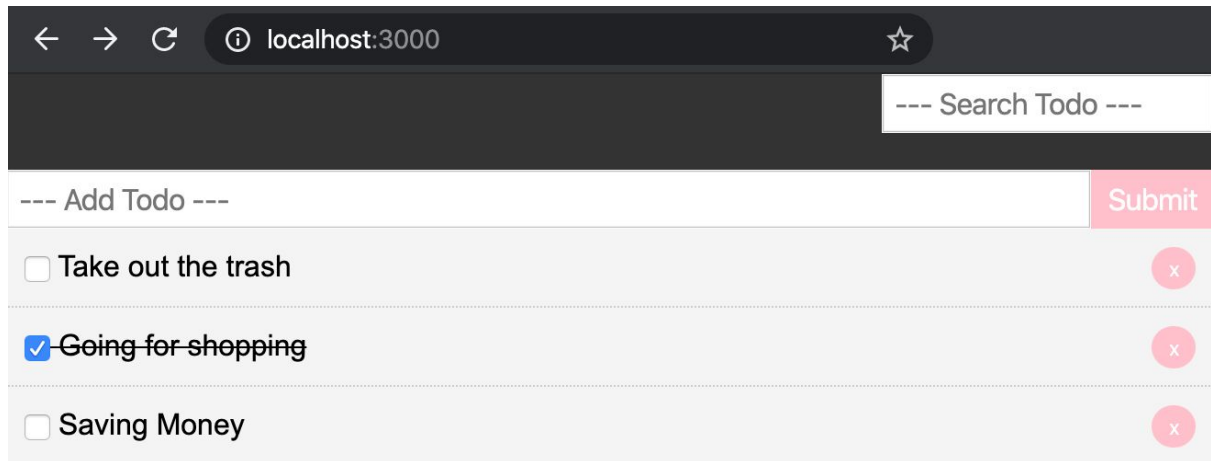
15. Header

Lets create a header as a function based component in a new folder layout. A function component it's like the class component but with only one lifecycle method that is the return. Return acts like the render at class component. The other lifecycles like ComponentDidMount are **not** used and the only way to create them is by React Hooks. Our function is going to be a very basic one like below:

```
import React from 'react';

function Header() {
    return (
        <header style={headerStyle}>
            <h1>TodoList</h1>
        </header>
    )
}

const headerStyle = {
    background: '#333',
    color: 'rgb(255, 192, 203)',
    textAlign: 'center',
```
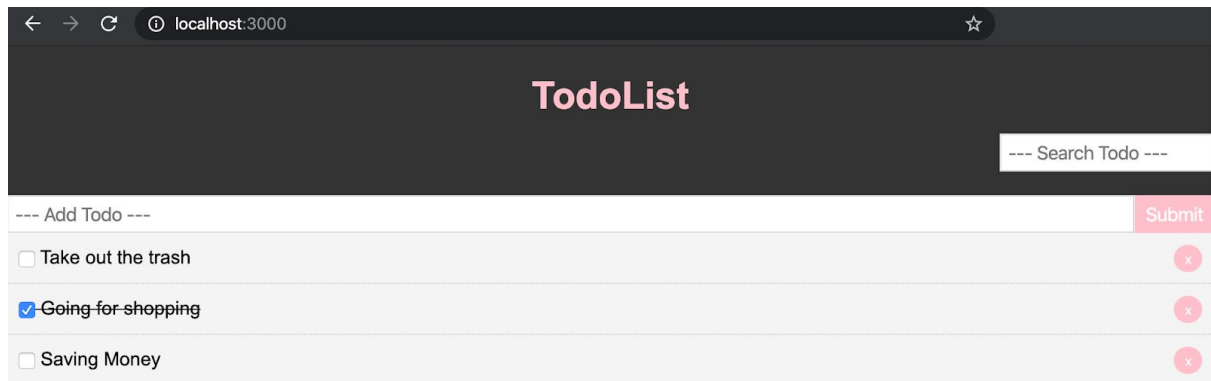
```
    padding: '10px',
    paddingTop: "20px"
}

export default Header;
```

We need to import it at App.js and to include it before the other components. This is what the header will look like:

16. React Router

I decided i wanted to integrate this part because even though it is not necessary for this project, it is a must for learning and using React. We are going to use React Router, which is the standard routing library for React. It keeps your UI in sync with the URL and helps you navigate through the page. We have to install it separately with the command below:

npm i react-router-dom

I am going to create a folder called pages and a file called About.js:

```
import React from 'react'

function About() {
 return (
   <React.Fragment>
     <h1>About</h1>
     <p>This is the TodoList app for DPS Participants</p>
   </React.Fragment>
 )
}


export default About;
```

Fragments let you group a list of children without adding extra nodes to the DOM. So it acts like a ghost element, because it doesn't show at the dom but it perfectly works when you do not need a div.

Now we need to go into the App.js and import:

```
import {BrowserRouter as Router, Route } from 'react-router-dom';
```

We need to wrap everything with the Router. Below you can see before and after we wrap and route everything in the render:

```
<div className="App">
  <Header />
  <Search
      searchTodo={this.searchTodo}/>
  <AddTodo
    addTodo={this.addTodo}/>
  <Todos
   todos={this.state.todos}
   markComplete={this.markComplete}
   deleteTodo={this.deleteTodo} />
</div>
```

```
<Router>
  <div className="App">
    <Header />
    <Route
        path="/"
        render={() => (
     <React.Fragment>
       <Search
          searchTodo={this.searchTodo}/>
    <AddTodo
        addTodo={this.addTodo}/>
    <Todos
        todos={this.state.todos}
        markComplete={this.markComplete}
        deleteTodo={this.deleteTodo} />
     </React.Fragment>
          )}/>
    <Route
        path="/about"
        component={About} />
    </div>
  </Router>
```

This is what we get when we change the URL to /about. It shows both routes at the same time. To fix it we need to add **exact** at path /.
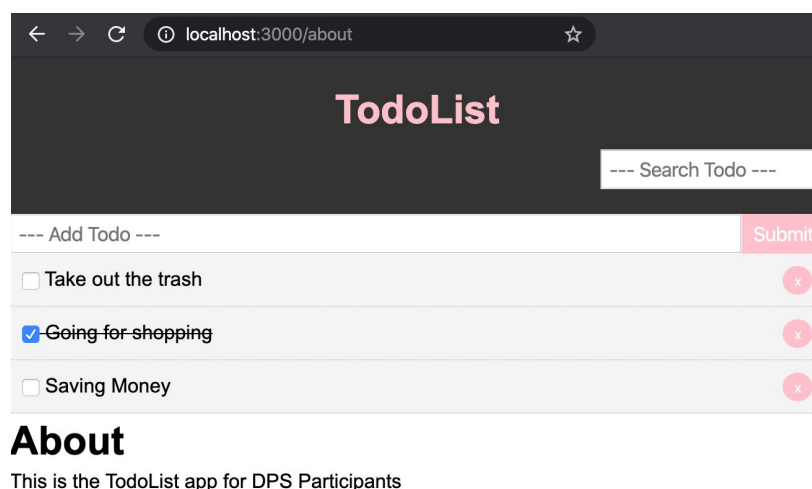
Cool. Now we have two paths and we want to connect them from the UI point of view. To be able to go from one page to another we can not use a href but Link from the router.

So at the header file we import Link from "react-router-dom" and below h1 we add:

```
<Link to="/">Home</Link> | <Link to="/about">About</Link>
```

We can see them because of the style, so we need to add some styling:

```
        <Link style={linkStyle} to="/">Home</Link> | <Link
            style={linkStyle} to="/about">About</Link>

const linkStyle = {
        color: '#fff',
        textDecoration: 'none'}
```

This is the final result for the About Page, including some extra styling i am not going to add here because styling is not the focus of this tutorial. If you want to check it you can find it on the Github repo :)
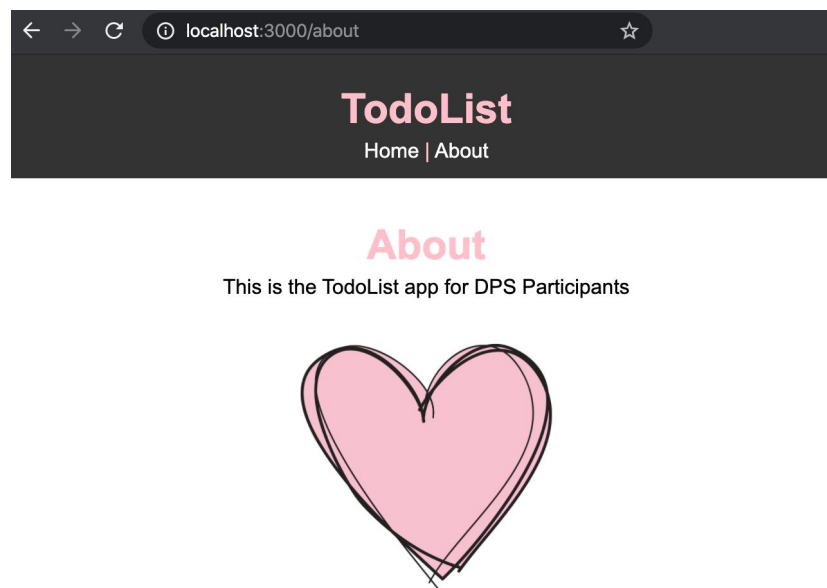
17. Json Placeholder

A lot of developers in the beginning are struggling with one question. How to make HTTP requests? For this reason we are going to obtain the data in another way through creating a json placeholder.

We are going to use todos from this link. We can use normal fetch, but I prefer using the axios library just to give you another hint how to make http requests. Use the command:
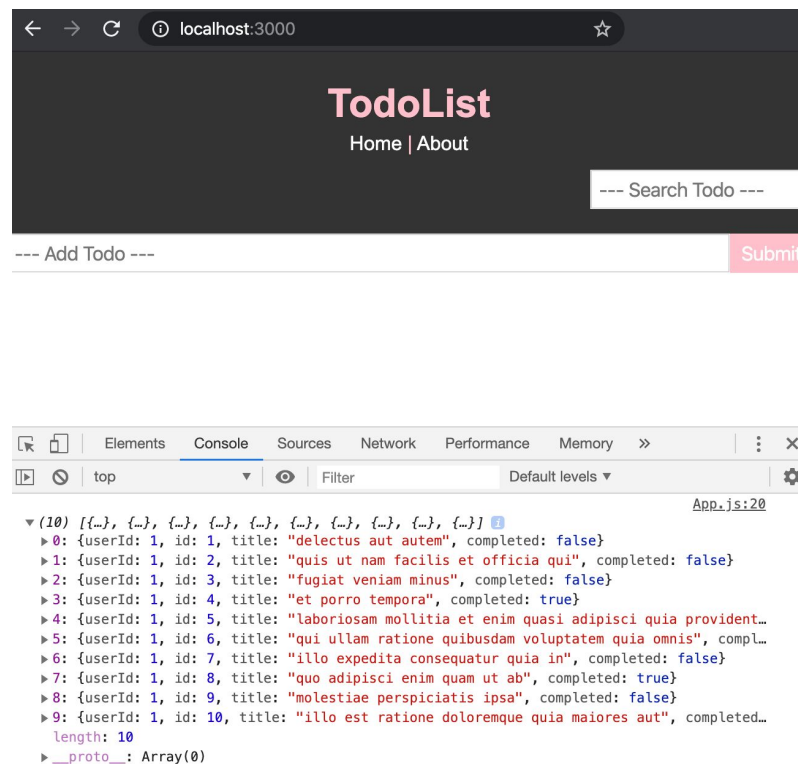
npm i axios

Completely clear todos: `todos: []`

We have to use componentDidMount to obtain the data from the placeholder:

```
componentDidMount() {
  axios
    .get('https://jsonplaceholder.typicode.com/todos?_limit=10')
    .then(res => console.log( res.data ));
}
```

This is the response as promise that we can get from the code above, retrieving 10 items with ?= query parameter:

The data we get from the response are set as state into todos:

```
.then(res => this.setState({ todos: res.data }));
```
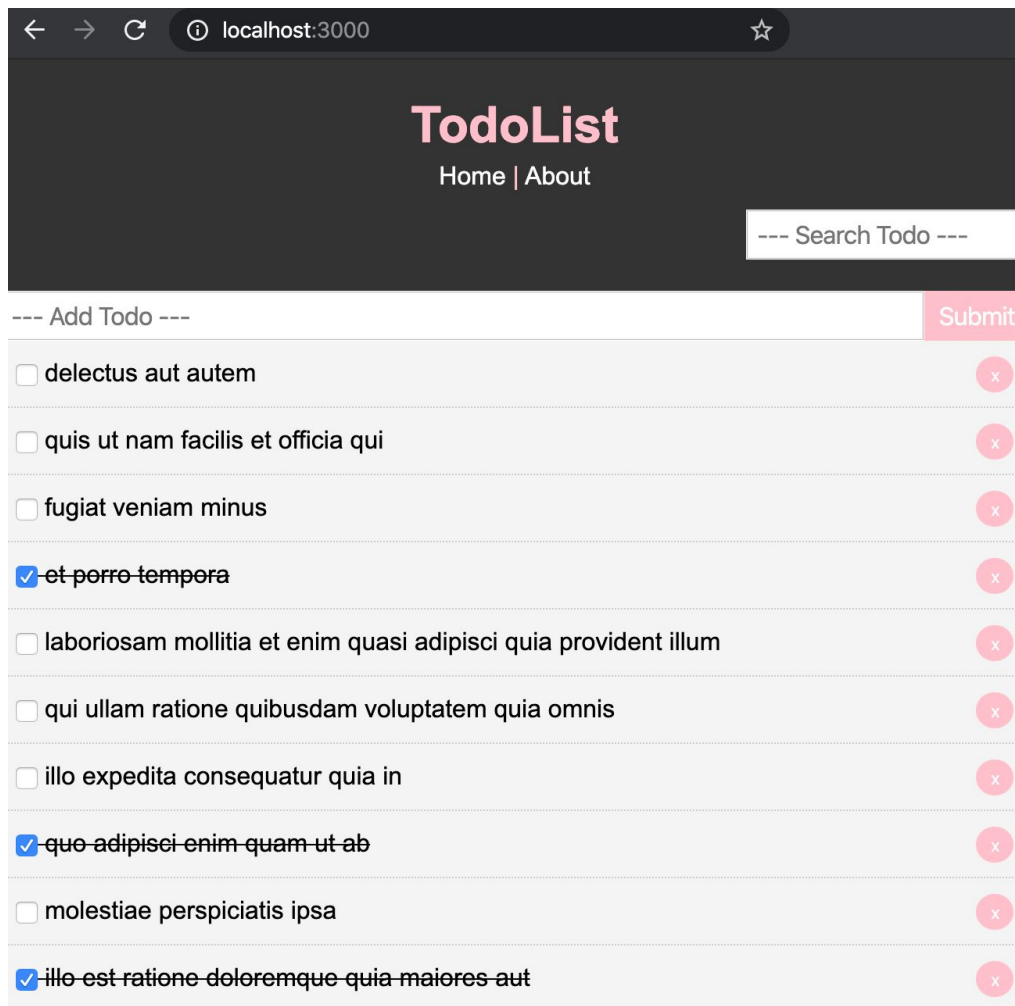
Perfect :) Now we can obtain data from the json placeholder. We have to continue with the other functionalities like addTodo and delete.

Below we can find all the HTTP methods that are supported. We can add a new todo only by making a **post request**.

```
GET        /posts
GET        /posts/1
GET        /posts/1/comments
GET        /comments?postId=1
GET        /posts?userId=1
POST       /posts
PUT        /posts/1
PATCH      /posts/1
DELETE     /posts/1
```

It does not actually change the json placeholder, it just mimics it and gives us a response as a fake back-end. We are just trying to understand the concept of requests with this part.

```
addTodo = (title) => {
  axios
    .post('https://jsonplaceholder.typicode.com/todos', {
      title,
      completed: false
    })
    .then(res => {
      res.data.id = uuidv4();
      this.setState({ todos: [...this.state.todos, res.data],
                 filtered: [...this.state.todos, res.data]  });
    });
};
```

I added Bela and this is what i get in the React Developer Tool. For the functionalities with the json placeholder it takes some seconds to actually appear on the UI.
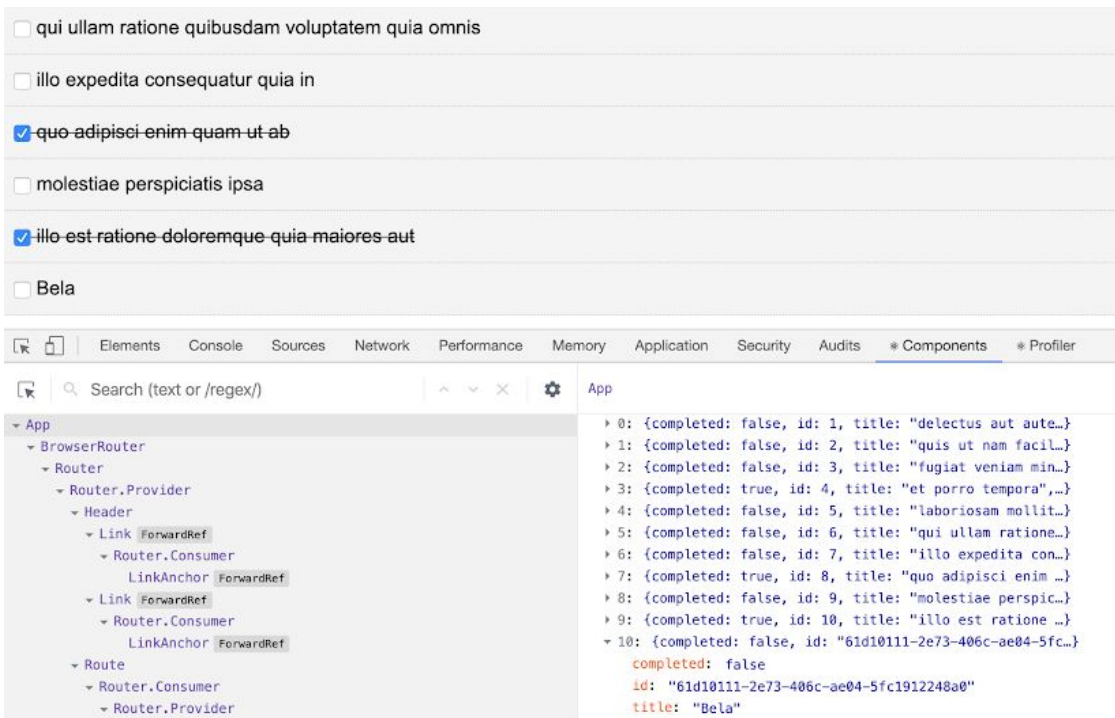
We can delete a todo only by making a **delete request**. This is a little bit different from addTodo, because in order to delete a todo we have to check for its id. For this reason we are using different apostrophes.

```
deleteTodo = (id) => {
axios.delete(`https://jsonplaceholder.typicode.com/todos/${id}`).t
hen(res =>
    this.setState({
    todos: [...this.state.todos.filter(todo => todo.id !== id)]
    })
  )};
```

18. PropTypes - A good practice

PropTypes is sort of a validation for the props that a component should have. We can set the type and if it is required or not.When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, propTypes is only checked in development mode.

So our component Todos have a prop type of todos inside the App.js. Into Todos.js we need to import PropTypes from "prop-types" and we add the following lines before export default:

```
Todos.propTypes = {
   todos: PropTypes.array.isRequired,
   markComplete: PropTypes.func.isRequired,
   deleteTodo: PropTypes.func.isRequired

 }
```

We have to do the same for TodoItem as it has a proptype of todo. Inside todoItem we add the proptype as object because from the mapping we get only one object from the array at one time:

```
TodoItem.propTypes = {
   todo: PropTypes.object.isRequired,
   markComplete: PropTypes.func.isRequired,
   deleteTodo: PropTypes.func.isRequired
 }
```

Also at addTodo and Search:

```
AddTodo.propTypes = {            Search.propTypes = {
addTodo:PropTypes.func.isRequ   searchTodo:PropTypes.func.isRequired
ired                             }
 }
```

19. Homework

Add a pencil icon and if you click it, a new component is called for editing the title of that todo. The title appears at the addTodo input, you can change it and after the submit the title is updated.

Github Link: https://github.com/bsinoimeri/TodoApp-React