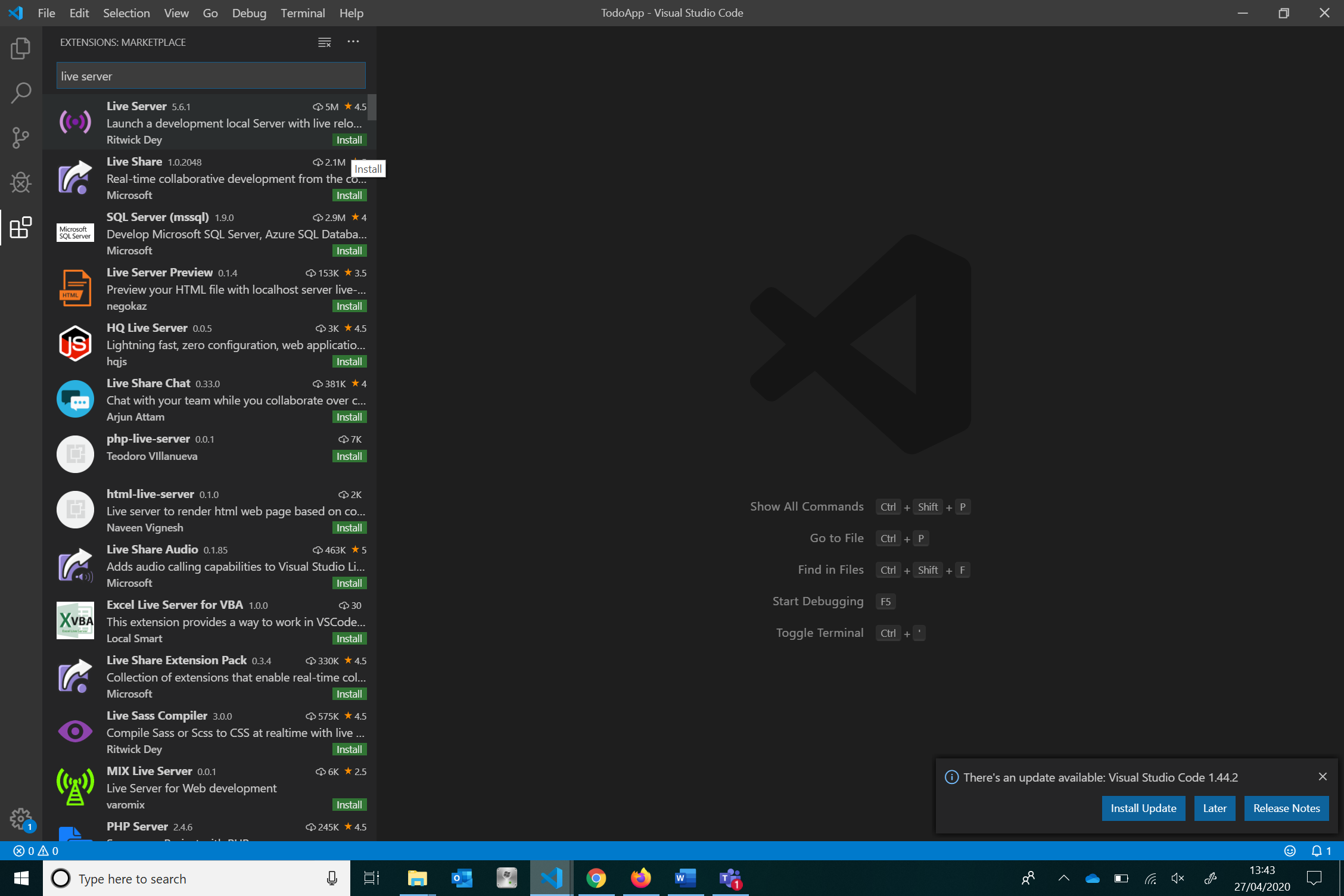
# **React Basics Lab Instructions**

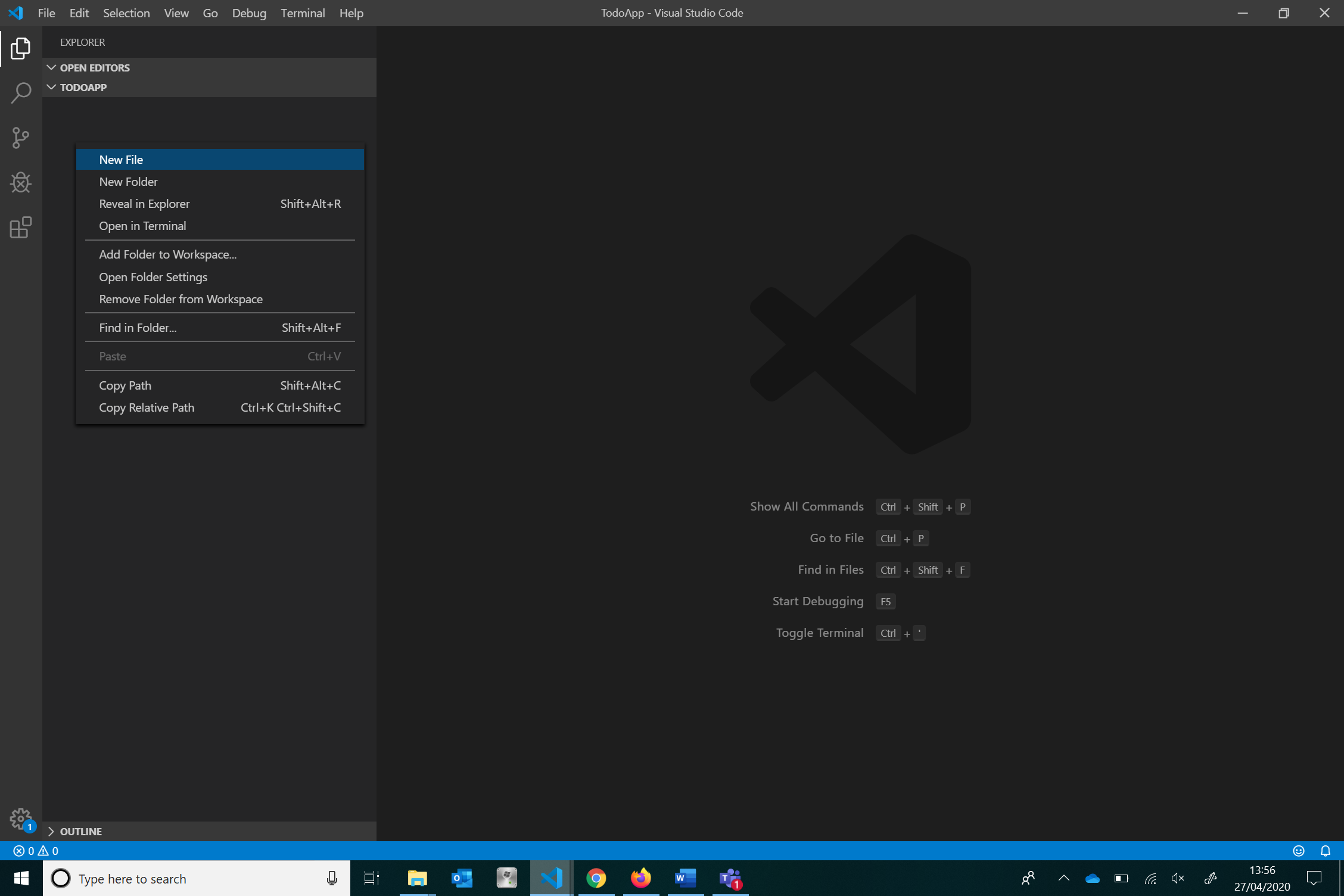
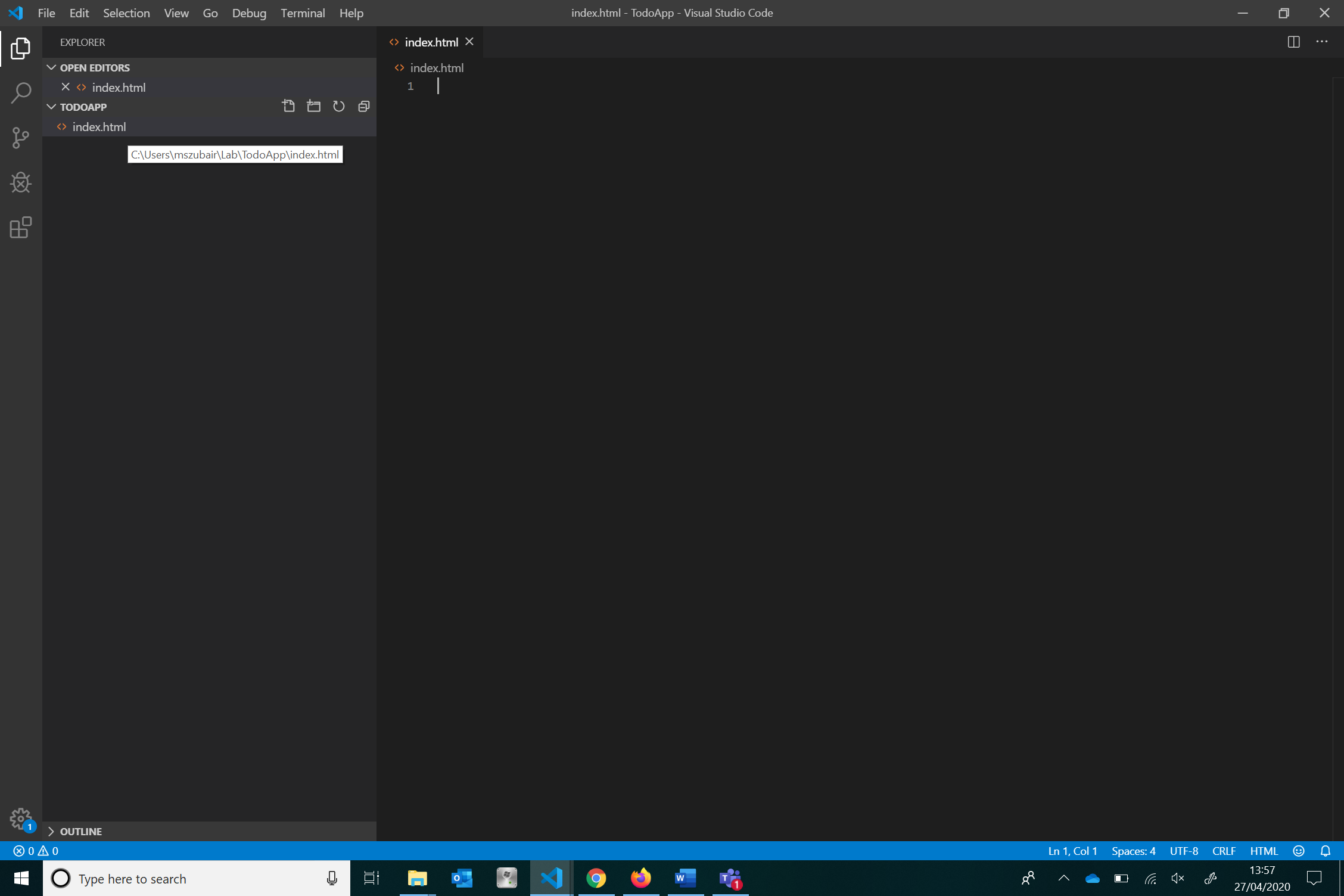
This document provides step by step procedure for quickly setting up a React project and creating a simple Todo List application. The instructions are divided into sections and the end of each section is considered a checkpoint where you can compare your code up to that point with the expected code (which can be downloaded from blackboard). For example, after completing the instructions in Section 2, you can compare your project with the project in section2.zip on blackboard which should contain the result of following all the instructions from the start of the lab instructions up to the end of Section 2.

## **Section 1: Setting Up**

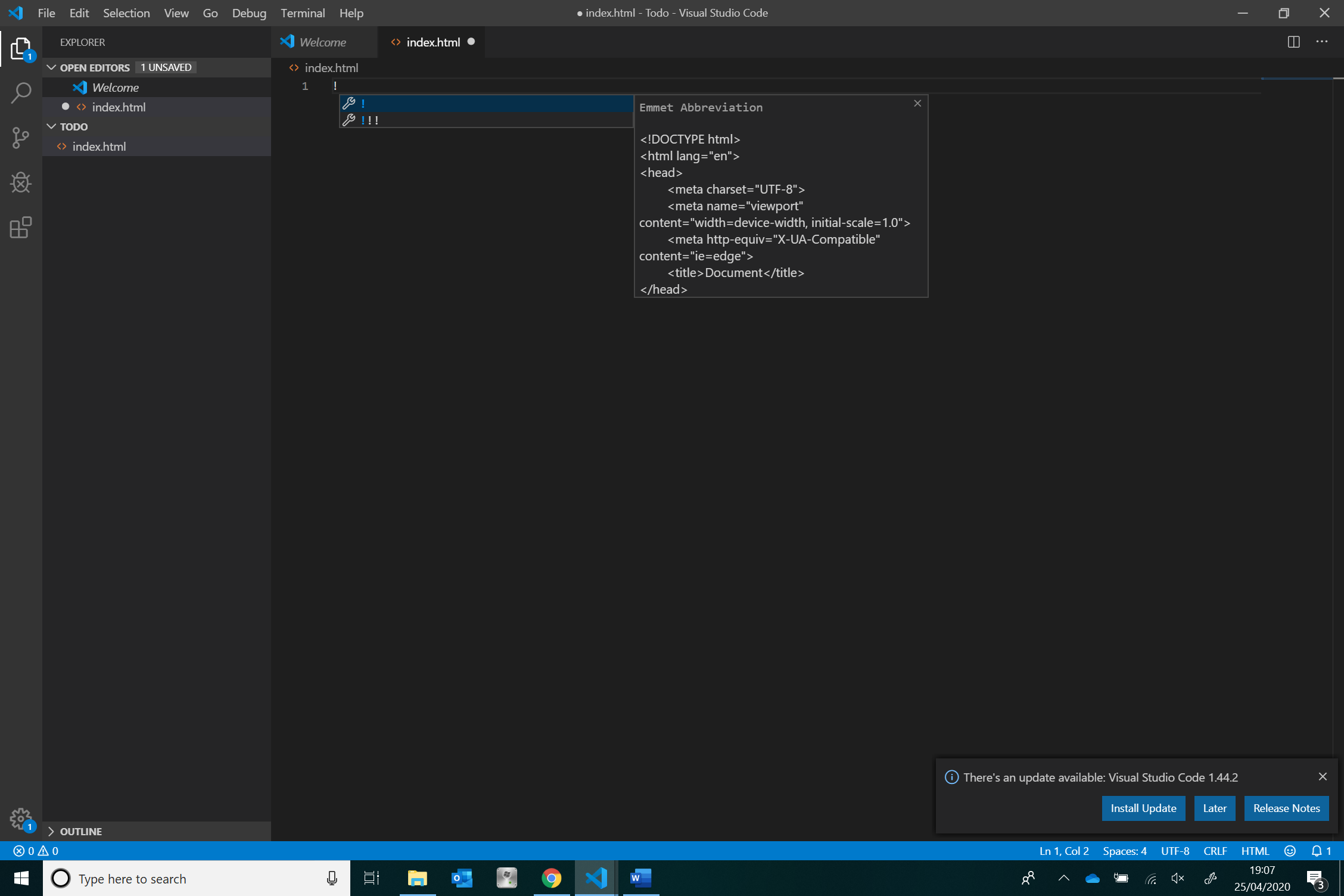
1. Create an empty folder and name it TodoApp.
2. Open VSCode, and from the ‘File’ menu select ‘Open Folder’. Navigate to the location of your ‘Todo’ folder in the opened file explorer window, select the folder and click ‘Select Folder’.
3. React requires a server to run, therefore lets start by installing VSCode Live Server extension which we will use to serve our project. Press Control + Shift + X or click the extensions icon on the left navigation pane of VSCode to display VSCode extensions.
4. In the extensions search menu, search ‘Live Server’ and click install on the result that’s has the description ‘Launch a development local Server with live reload feature for static & dynamic pages’.



1. Once it is installed press Control + Shift + E or click on the file explorer icon on the left navigation pane of VSCode.
2. Right click within the file explorer area and select New File to create a new file, name it ‘index.html’.

1. Create two other files and name them index.js and styles.css.
2. Open up ‘index.html’ and add a html document template by typing ‘!’ and then pressing enter when VSCode gives you the option to insert a template.



1. Change the title of the html document from ‘Document’ to ‘Todo App’ by modifying the title tag in the head of the document.
2. Add the two script tags below for loading React, ReactDom and Babel in the head section of your html document.

<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>

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

1. Add the following code to the head of the html document link styles.css file.

<link rel="stylesheet" href="./styles.css"/>

1. Add the following code just above the closing body tag of the html document to link the index.js file. Note that the type attribute is set to ‘text/babel’ because we want abbel to transpile the JSX that we will be writing in index.js.

<script type="text/babel" src="./index.js"></script>

1. Everything is now set for you to start wiring React code. To test if its all working, lets create and display a simple component that returns a heading with the text ‘Todo App’. Add a div tag in the body of your html document, give it an id of ‘root’. We will be rendering our React app in this div.

<body>

    <div id="root">

    </div>

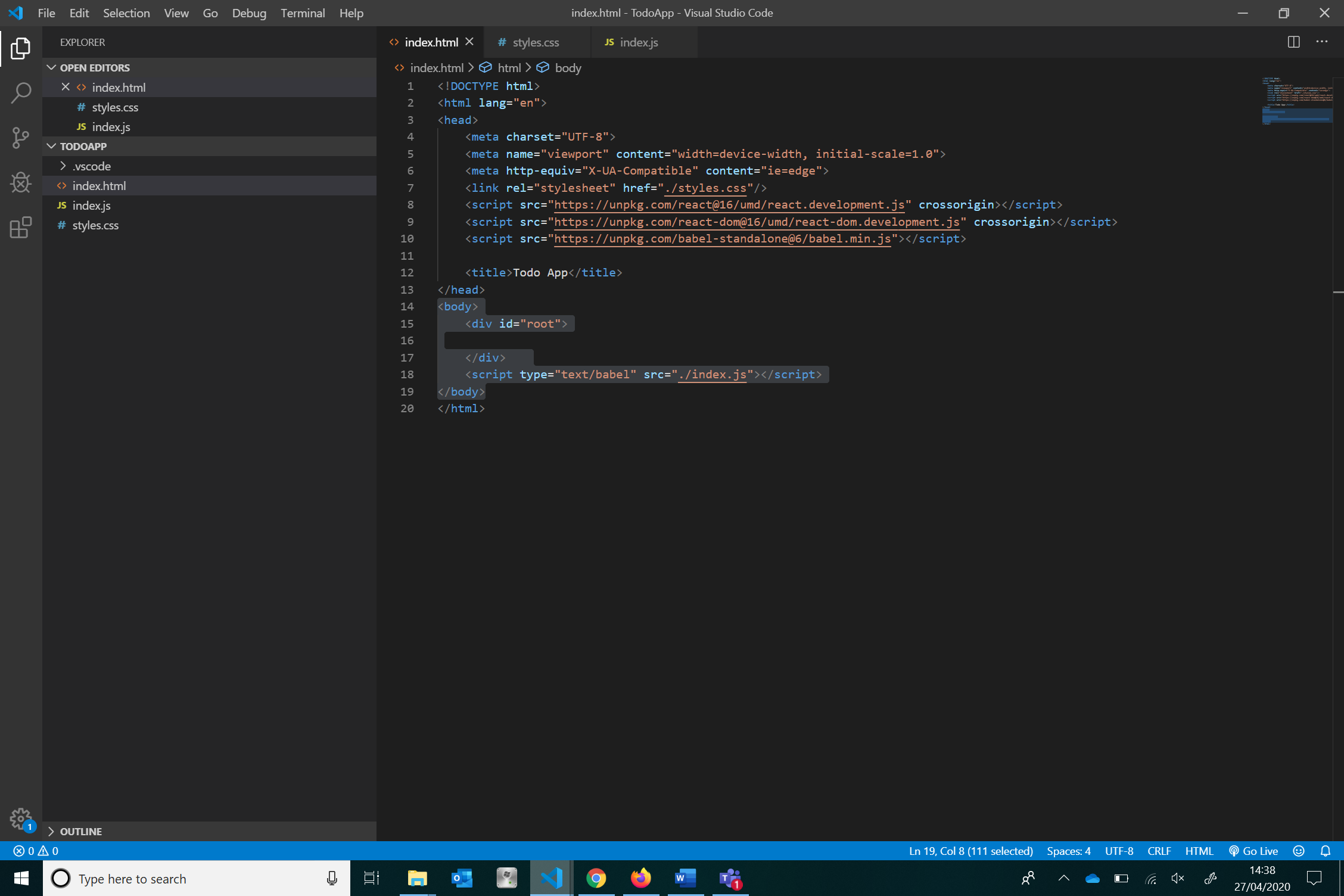
    <script type="text/babel" src="./index.js"></script>

</body>

1. At this point we want our React app to just render the heading ‘Todo App’ i.e. the component <h1> Todo App </h1>. To do this, we use ReactDOM’s ‘render’ method, which takes as its parameters the component to render and the location for it to be rendered which in our case is the div with id ‘root’. Open up the index.js file and paste in the code below:

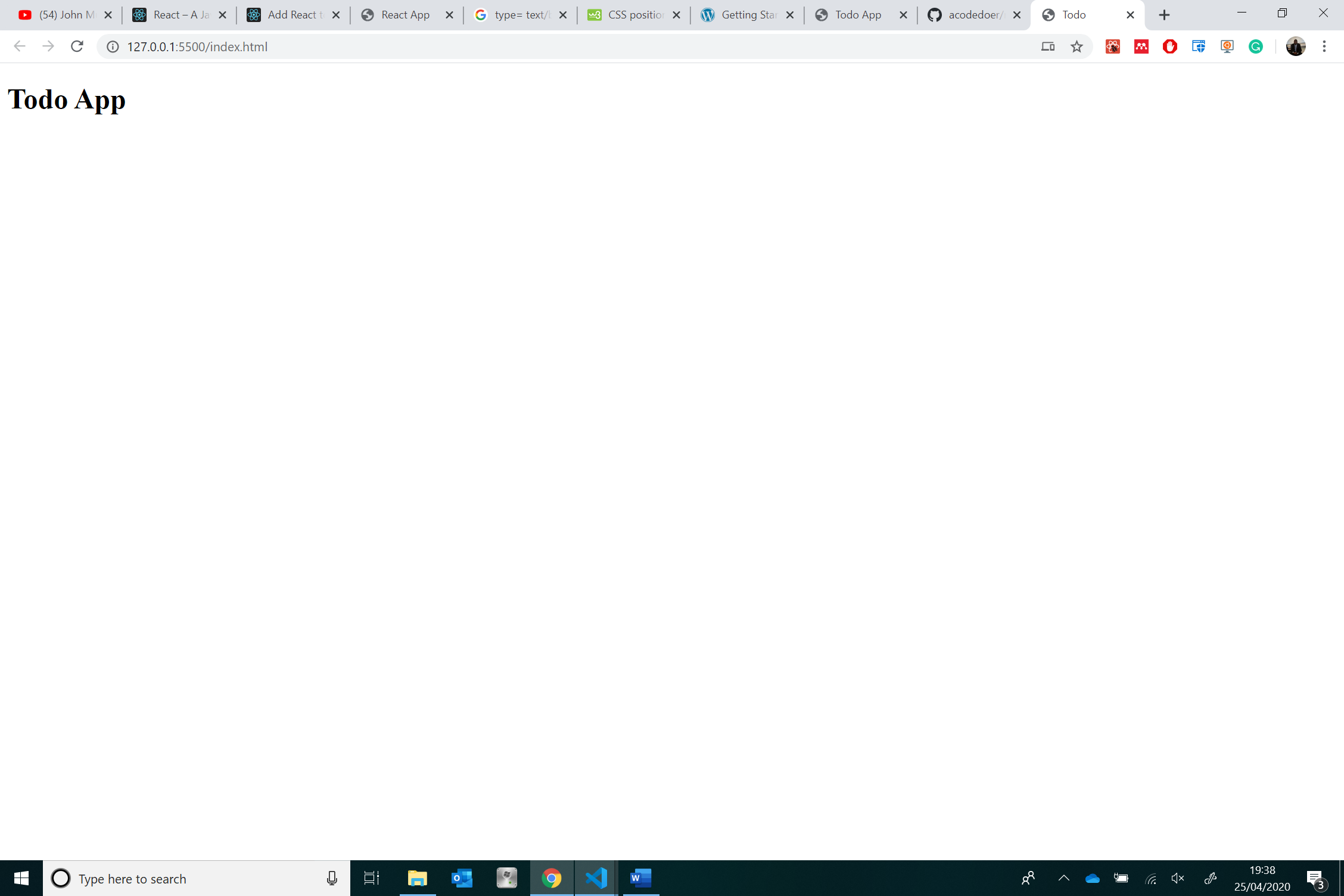
ReactDOM.render(<h1>Todo App</h1>,document.getElementById("root"))

1. Now let us run the project using VSCode’s LiveServer. On the bottom right of the screen you should see a clickable ‘Go Live’ text, click that to start your server.



If the ‘Go Live’ is not visible, press Control + Shift + P to a dropdown list, scroll through or search ‘Live Server: Open with Live Server’ and the click that to start the server and run your project. If you are having problems with this step, make sure you have done Step 4 correctly.

1. Once you choose to open your project with live server, the project should be loaded on a new browser tab and you should see a page similar to the one below.



Liver Server will automatically update and rerun your code whenever you make changes within VSCode, so you don’t have to worry about doing that manually.

1. Now that we have React setup, we can move to writing more complex components. If you are having difficulties completing this section, you can have a look at this section’s checkpoint source code on blackboard.

## **Section 2: Creating your First Class Component**

1. Although it is cool to be able to display a heading, we want to create a real Todo app. Let us start by creating a class component. Our class component is made up of a constructor which takes in props and a render method which returns what is to be displayed on the screen. To begin with, let us have the class component return our previous header for rendering. Add the code below to the top of your index.js file.

class Todo extends React.Component{

    constructor(props){

        super(props)

    }

    render(){

        return <h1>Todo App</h1>

    }

}

1. The next step is to update our RecatDOM.render method to render the Todo component as shown in the code below.

ReactDOM.render(<Todo/>,document.getElementById("root"))

1. On updating the ReactDOM.render method, view your updated Todo App in your browser, you will notice that it looks exactly the same as we are rendering the same heading using the class Component. Try changing the JSX in the class component’s return to see how that is reflected in your Todo App (e.g. change it to <h1>Hello from the App</h1>).
2. We now have our basic Todo class component. This component is going to be a container of sorts i.e. it is going to store to-do items in its state, it is going to render each to-do item as a separate component, and also render a Header component. In the next section we will create the Header component.
3. If you are having difficulties completing this section, you can have a look at this section’s checkpoint source code on blackboard.

## **Section 3: Creating the Header Component**

1. Let us start by creating the Header component as a functional component (since it does not need state to handle data). After the Todo class component, add the code below:

function Header(props){

    return(

    )

}

1. Right now, the component is not returning anything to be rendered. However, we want it to return a heading similar to the one we have been displaying, a form to type in to-do items, and a button to add to-do items (note that when returning more than one element, a single parent element such as a div should be used to wrap elements). Update the Header component as shown below:

function Header(props){

    return(

<div className="Header">

            <h1>Todo App</h1>

            <div>

                <input type ="text"></input>

                <button>+Item</button>

            </div>

        </div>

    )

}

1. Before the Header component is visible in the Todo App, it has to be rendered by the Todo component instead of the older header (<h1>Todo App</h1>). Update the Todo component’s render method as shown below and save the changes.

render(){

    return (

        <div className="Todo">

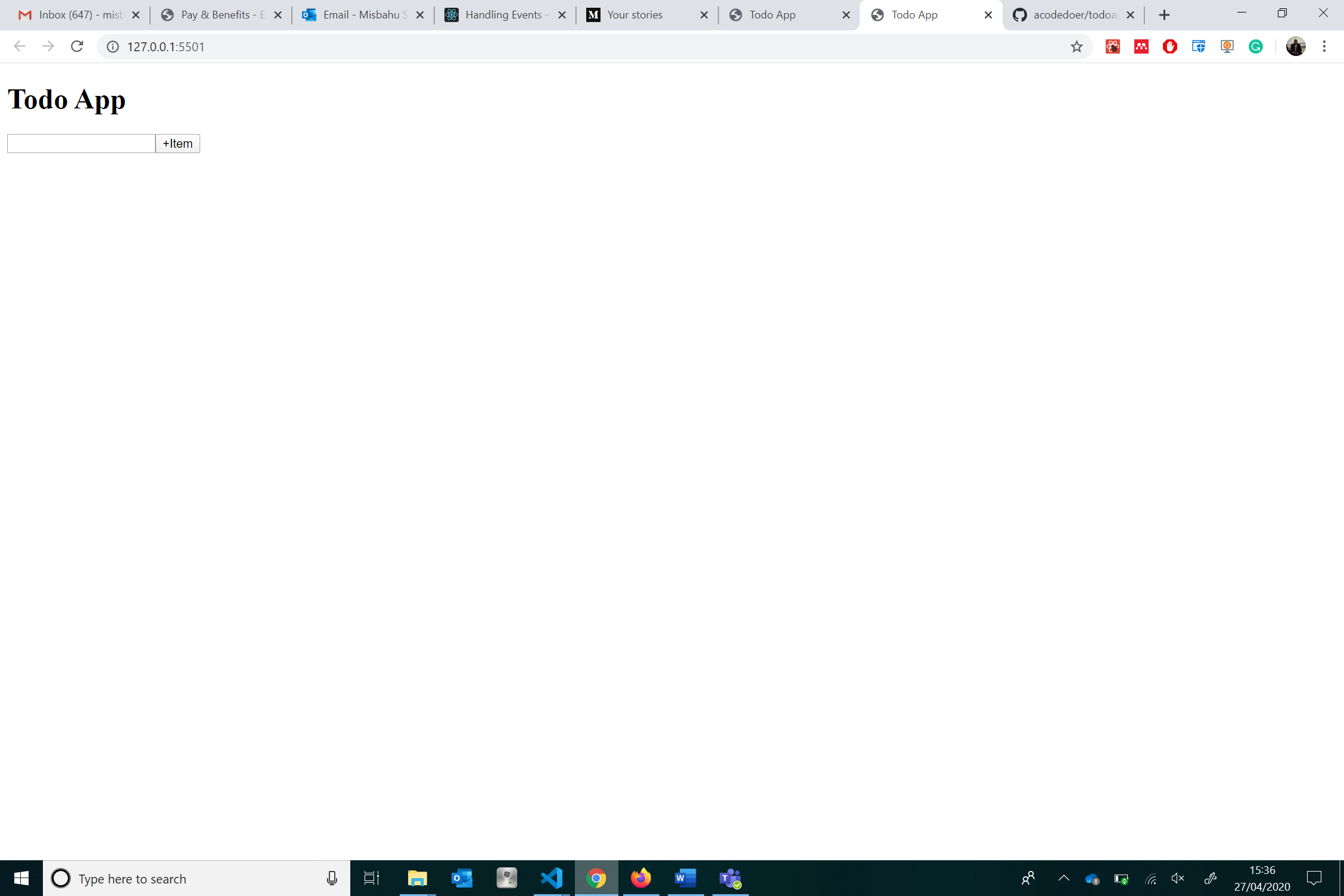
            <Header/>

        </div>

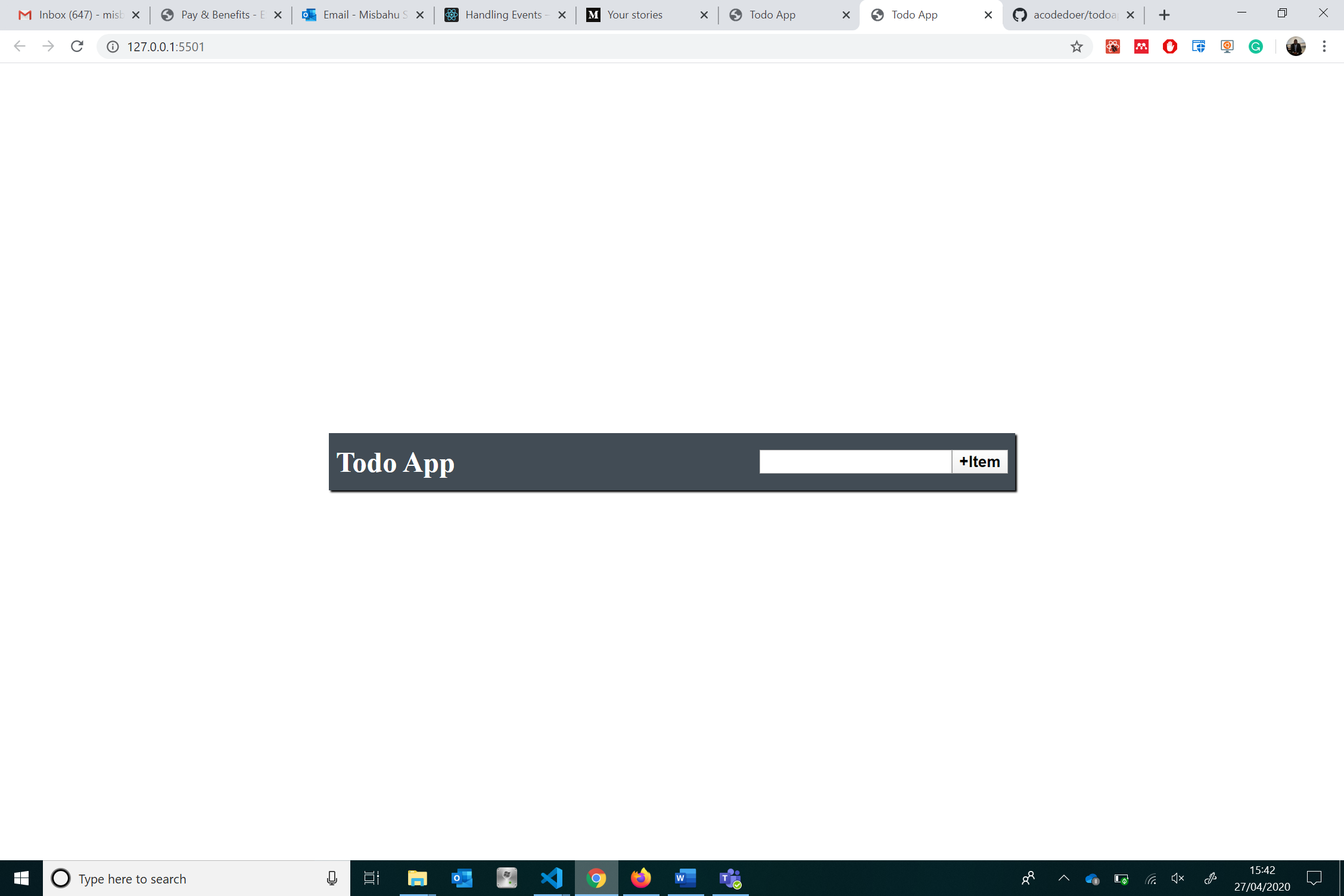
    )

}

1. The Todo App should now have a different look (see image below).



1. **Optional:** To make the app look more interesting, add the css code in the ***file*** to your styles.css file and save. Make sure you are using the classNames provided in the instructions. Your Todo App should now look like the image below:



1. If you are having difficulties completing this section, you can have a look at this section’s checkpoint source code on blackboard.

## **Section 4: Creating the Item Component**

1. Let us now create a component for displaying each to-do item.
2. Below the Header component, add a new functional component named Item as shown below:

function Item(props){

    return(

    )

}

1. Each item should return the to-do item itself as text, and a delete button that can be used to delete the item. Therefore, update the Item component as shown below:

function Item(props){

    return(

<div className="Item">

             <p>This is what I need to do </p>

             <button>Delete</button>

         </div>

    )

}

1. Just like the Header component was rendered by the Todo component before it became visible, this component also has to be rendered by the Todo component. Therefore, update the render method of the Todo component to render the two components as shown below and save

render(){

    return (

        <div className="Todo">

            <Header/>

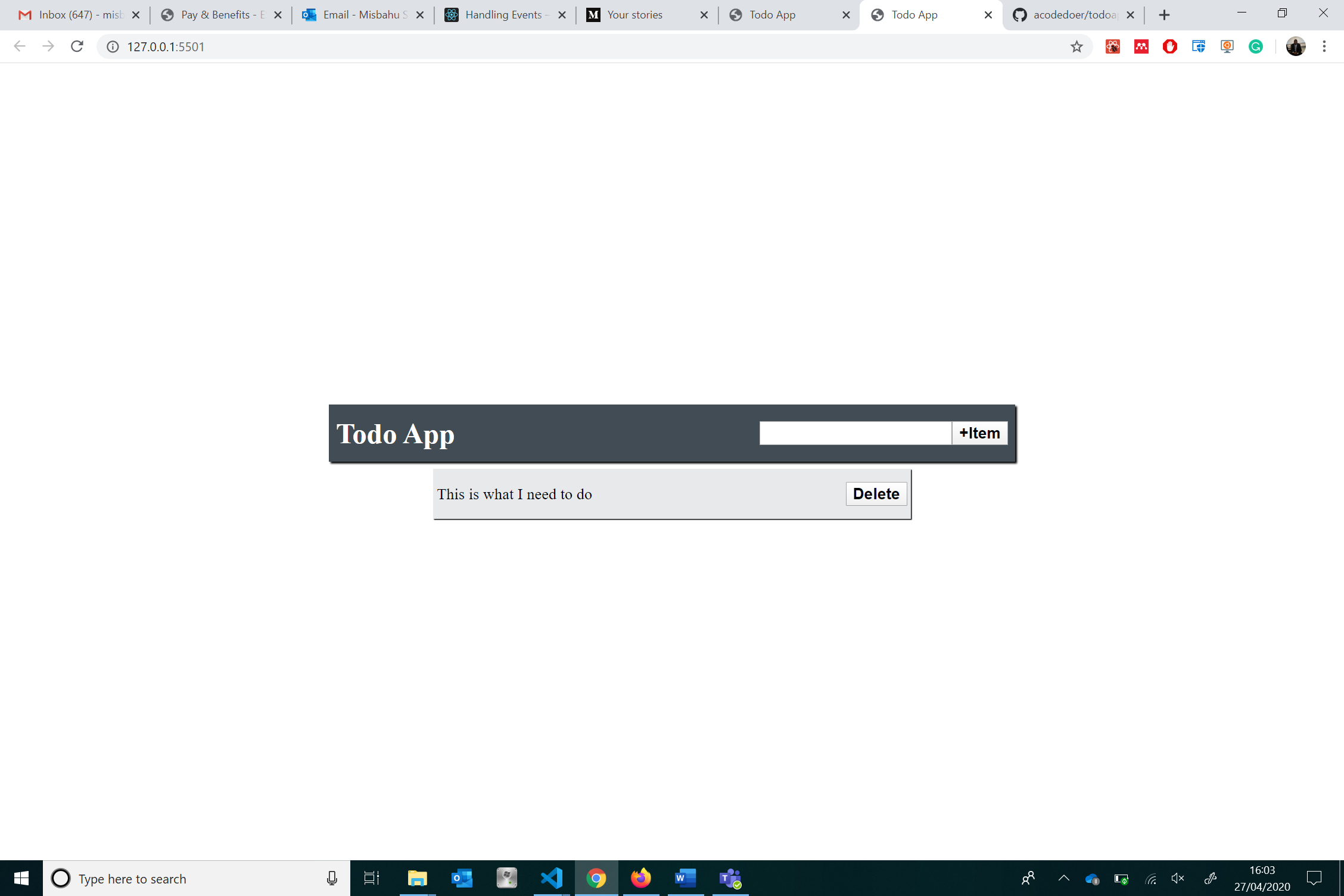
            <Item/>

        </div>

    )

}

1. The Todo App should now have a header and a single to-do item as shown in the image below:



1. In the next section we will be setting up out app to display to-do items dynamically rather than having them hardcoded as done in this section.
2. If you are having difficulties completing this section, you can have a look at this section’s checkpoint source code on blackboard.

## **Section 5: Setting up and Rendering from State**

1. We have an Item component, but right now it only displays a placeholder item. What we actually want is for it to display to-do items that we save in the Todo component’s state. In this section we will setup state, add sample to-do items to it, and display them in the app.
2. In the Todo component constructor, initialise state below the super method as shown below:

constructor(props){

        super(props)

        this.state = {

        }

    }

1. Next, add an array named items which will be used to store to-do items.

this.state = {

items: []

}

1. For now, add some sample to-do items in the items array. The code below shows a few examples, you can add others if you wish.

this.state = {

    items: ['Buy milk', 'Take out trash', 'Renew Netflix subscription']

}

1. The next step is to display each item in state by lopping through the items array and rendering each to-do item in an Item component. This can be done by replacing the single Item component in the render method of the Todo component, with an array mapping function as shown below:

render(){

    return (

        <div className="Todo">

            <Header/>

            {

                this.state.items.map(function(data,index){

                    return <Item text={data} key={index}/>

                })

            }

        </div>

    )

}

The map function loops through an array and provides an element and the index of that element in each iteration. In each iteration of the above code, an element of the items array is passed as a prop named **text** and the index is passed as a prop named **key** to the Item component. Notice that we can add this javascript code into the return of our component by wrapping it in curly braces, same has to be done to variables being passed as props (e.g. **data** and **index**).

1. Finally, to make sure that the props passed to the Item component are used, update the Item by replacing the placeholder to-do item “This is what I need to do” with “{props.text}”. The updated Item component should look like the code below:

function Item(props){

    return(

      <div className="Item">

             <p>{props.text}</p>

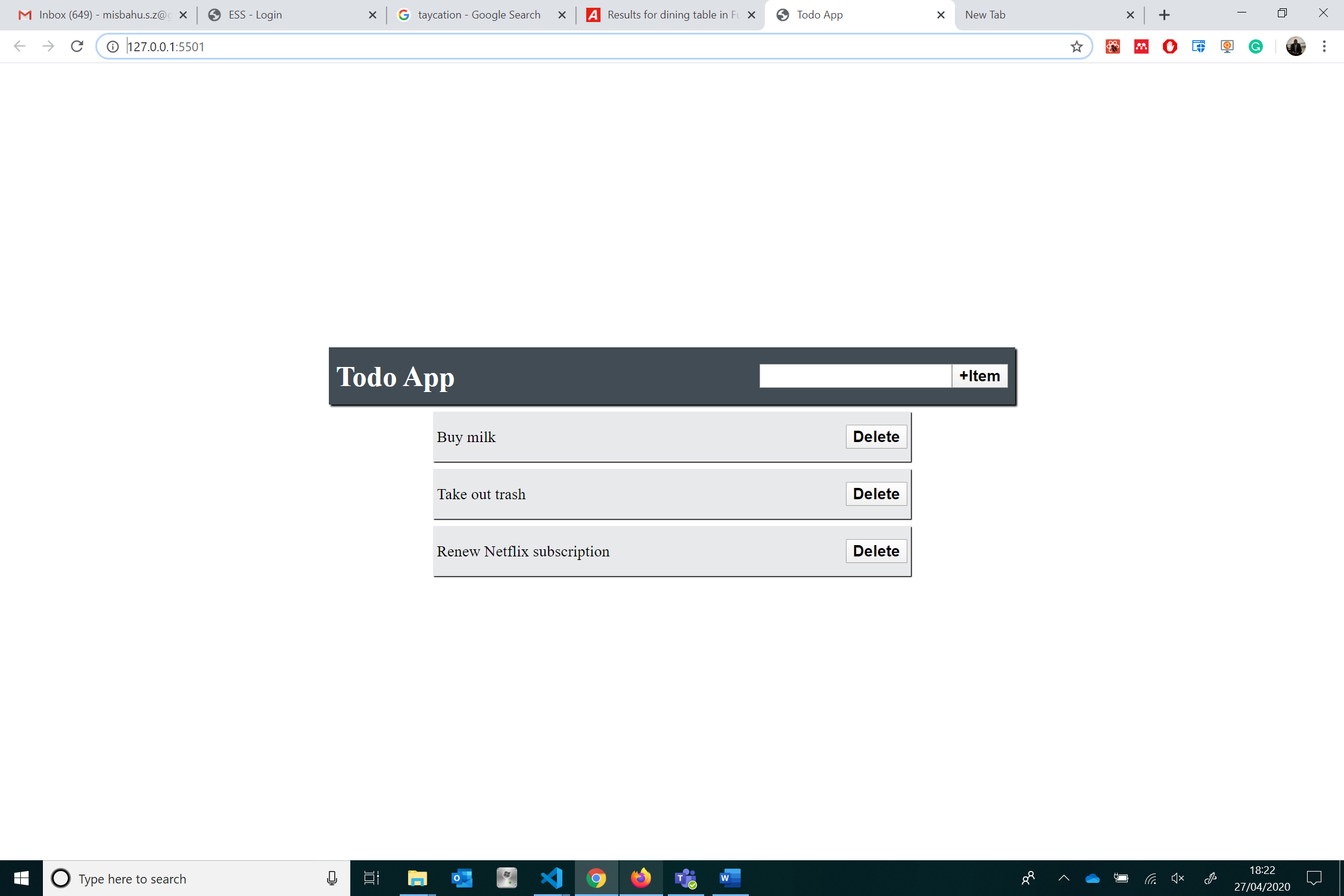
             <button>Delete</button>

         </div>

    )

}

1. Save your code and view the update Todo App, you will notice that the items you added in the items array are now displayed.



Try adding/removing items to the array and watch the Todo App gets updated.

1. If you are having difficulties completing this section, you can have a look at this section’s checkpoint source code on blackboard.

## **Section 6: Creating Items**

## **Section 7: Deleting Items**