

# React JS

## Lab Guide

### 1. CREATING AND RENDERING AN ELEMENT – `React.createElement()`

Create a div React element using `React.createElement()`. It should have as children the following 3 elements

- A button with text '-'
- A span with id 'counter' and the text '0' as child
- A button with text '+'

Render the React element inside the root div. It should look like shown below.



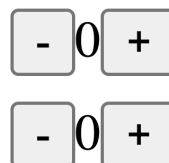
### 2. CREATING AND RENDERING AN ELEMENT – JSX

Include Babel and redo the above exercise by creating the element using JSX.

### 3. RENDERING MULTIPLE ELEMENTS

Render 2 div elements with the same structure as in the previous exercise. Try each of the following 3 ways to do it.

- Render the 2 divs as children of another div
- Add the 2 divs to an array and render the array of React elements
- Enclose the 2 divs in a `React.Fragment` and render



### 4. VIRTUAL DOM, CREATING AND RENDERING AN ELEMENT

Create a React element that represents a registration form (form with id 'registration-form', and action set to '/register'). It should have as children the following elements (in the exact same order)

- A heading within the form
- A label and input element (text) to take username
- A label and input element (password) to take password
- A label and input element (confirm password) to take password
- A submit button

You may also enclose the label and inputs together within a div for styling purpose (this is optional). Render the React element inside the root div. It should look like shown below.

**Register**  

Username

Password

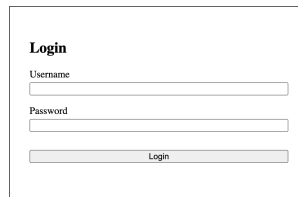
Confirm Password

Register

Create a login form element that represents a login form (form with id 'login-form', and action set to '/login'). It should have as children the following elements (in the exact same order)

- A heading within the form
- A label and input element (text) to take username
- A label and input element (password) to take password
- A submit button

Provide a 20 second gap after registration form re-renders, and replace registration element with login element (use `setTimeout()` to delay the re-rendering)



The image shows a simple login form. It has a heading 'Login' at the top. Below the heading are two input fields: one for 'Username' and one for 'Password'. At the bottom of the form is a button labeled 'Login'.

- Which elements will result in new DOM nodes being created during the re-render?
- Which elements will have their DOM nodes modified?
- Can you use `key` to prevent unnecessary creation of any of the DOM nodes? Which one?

## 5. FUNCTION COMPONENT

Define function components – `RegistrationForm`, and `LoginForm`. They render UI as in the previous exercise. Render a `RegistrationForm` element within the root div, and replace it with a `LoginForm` element after 20 seconds.

## 6. FUNCTION COMPONENT, PROPS, CONDITIONAL RENDERING

Create a function component called `BusinessCard` that renders a business card (visiting card)



It takes as input props the following and shows the details.

- `firstName`, `lastName` – strings
- `lastNameFirst` - boolean
- `designation` - string
- `company` - string
- `contact` - an **object** with **tel** and **emailid** properties (both strings)
- `imageSrc` -string (eg. if the image is in the current folder you may pass './name-of-image.jpg')

If `lastNameFirst` is true the full name is displayed like “`lastName, firstName`”, else it is displayed normally.



**Note:** Except string literals, a props value when included in a JSX element, is enclosed in braces (`{}`).

Pass the props individually first. Also try to gather the props into an object like so, and use props spread operator to add the object's properties as props.

```
const props = {
  firstName: 'John',
  lastName: 'Doe',
  lastNameFirst: true,
  designation: 'CEO',
  company: 'Example Consulting',
  contact: {
    tel: '+001-123-456-7890',
    emailid: 'john.doe@example.com'
  },
  imageSrc: './johndoe.png'
};
```

## 7. COMPOSING COMPONENTS, PASSING DOWN PROPS TO CHILD

Redo the above exercise by defining a new component called `Contact` (child). This takes the contact object as a prop and shows the contact details. Use it in the `BusinessCard` component (parent). Rest of the details are shown in the `BusinessCard`.

**Note:** Make sure to destructure props passed to the `BusinessCard` component, and use the rest operator to gather the contact prop into a separate object. Pass this as a prop to `Contact` element using props spread operator.

## 8. DEFAULT PROPS, PROP TYPES

Add default values for the following props of the `BusinessCard` component

- `lastNameFirst` - false
- `imageSrc` - `'./profile.png'`

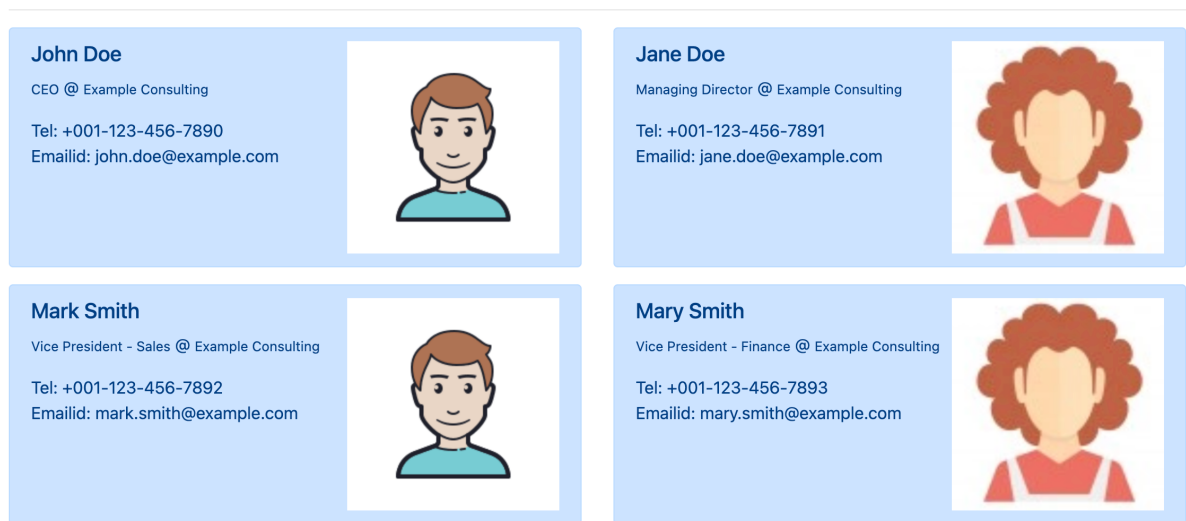
Define data types for all the props of the `BusinessCard` and `Contact` components.

## 9. RENDERING LIST

Create a function component called `Team` that is passed an array of objects of the form in Exercise 4 (each object represents details of a team member). The component displays each

team member's details in a `BusinessCard` component like so. Use the email id as the value of key while rendering the list of `BusinessCard` elements.

## Our team



### 10. CLASS COMPONENT AND STATE, `setState()`, `componentDidMount()`

Create a class component `FibonacciSequence` that shows numbers in the Fibonacci sequence. Initially 0, 1 are shown. With the passing of every second a new number should be added to the list of numbers shown. For example, after a second, "0, 1, 1" is displayed, then after one more second, "0, 1, 1, 2" etc. What will you maintain in the state of this component?

### 11. CLASS COMPONENT AND STATE, `setState()`, `componentDidMount()`, `componentDidUpdate()`, `componentWillUnmount()`

Create a class component called `FluCounter` that shows the number of Flu cases. In state, maintain the following (at least – see note below)

- Number of cases – initially this is 1
- Multiplication factor (the factor by which the cases multiply in 3 seconds) – initially this is 2 (indicating that the number of cases doubles in 3 seconds)

Thus, the component shows 1 initially, and after 3 seconds it shows 2, after 6 seconds 4, after 9 seconds 8, etc.

This continues 10 times (i.e. for 30 seconds), and then the multiplication factor changes to 0.5. Thus, the number now halves every 3 seconds. When it reaches 1 again, it should start doubling again, and the wave continues.

You must use `componentDidUpdate()` to check if 10 changes have been made with same multiplication factor, and change it there (you can definitely achieve it in other ways, but you can do this way for practice).

Render a `FluCounter` instance inside the root div. After 1 minute render some other element within the root element in place of the `FluCounter` instance (use `setTimeout` to schedule the re-rendering, and render a simple div for example). When this happens, the `FluCounter`

instance should log a message – ‘My time is up!’ before disappearing from the page. Use `componentWillUnmount()` to do this.

### Note

- You may need to add more properties in the state than mentioned above, to solve this problem.
- Make use of the right flavour of `setState` (object/function as 1<sup>st</sup> argument) based on whether a state change is independent of current state or not.

### Hints

- `setInterval()` takes in a function as the first argument and schedules it for execution periodically (the second argument is that time period provided in milliseconds).  
<https://javascript.info/settimeout-setinterval>
- `setInterval()` returns an interval id (a number). This is passed to `clearInterval()` to stop further execution of the function. You may store this id in the `FluCounter` component instance (as “this.id” for example) so that it can be used across multiple lifecycle methods.

## 12. FUNCTION COMPONENT AND STATE, `useState()`, `useEffect()`, `useRef()`

Create a function component – Counter, with UI as per Exercise 2. In addition add an input text box where user can enter a number. If ‘+’ is clicked, the counter value increases by the number specified in the text box. For click on ‘-’, the counter value decreases by that number. Maintain the count in a state variable (`useState`). Assume initial value is 0. Use either `ref` (`useRef`) or controlled components pattern (suggest trying both ways) to read the number in the input.

Next write a side-effect function (passed to `useEffect`) that stores the count value in `localStorage` whenever it changes (i.e. on state change). Write another effect that loads the count value from `localStorage` when the component first renders (so it shows the current value stored in `localStorage` if one exists, else it shows 0).

## 13. FUNCTION COMPONENT AND STATE, `useState()`, `useRef()`

Create a function component - Calculator. It has 2 input boxes that take user inputs (operand), and a dropdown that has 4 basic arithmetic operations - +, -, \*, /. There is a button that user clicks after entering the operand values and selecting the operation. The complete expression, and the result should be displayed in the component. Use refs to get hold of input elements values.

## 14. API ACCESS, AJAX, STATE, SIDE-EFFECTS

Before attempting this exercise, sign up at <https://openweathermap.org/>, and view the API key at [https://home.openweathermap.org/api\\_keys](https://home.openweathermap.org/api_keys). You will need to use the current weather API - <https://openweathermap.org/current> (this is free with usage limits).

Define a function component called `WeatherReport` that shows details of Wind, Cloudiness, Pressure, Humidity, Sunrise, Sunset and Geographical coordinates for a given city. Get some details for your city from <https://openweathermap.org/city>. These details are to be

passed as props. You can also find details of weather by going to this URL - substitute city name with your city's name. Also substitute your api key.

```
https://api.openweathermap.org/data/2.5/weather?q={city
name}&appid={api key}
```

The weather data also contains a description of the weather and the name of the icon representing the weather condition - use this to display an image of the weather condition within the component. You can find details of how to get the URL of the icon (i.e. image) from the incoming data, here

<https://openweathermap.org/weather-conditions>

Also define a function component called SunriseSunset that shows Sunrise and Sunset timings. Use this within WeatherReport to show the details of sunrise and sunset. The props need to be gathered within WeatherReport using rest operator, and passed to an element of SunriseSunset as props (using props spread operator).

**Note:**

- a. The API key may take a few hours to generate. You will not require the API till you start working with the backend to fetch weather data and use it (will be done in forthcoming exercises). For a guide on using the API check <https://openweathermap.org/guide>

If you are able to get the API key, you can try making an Ajax call using Axios to fetch the current weather data for your city. Once the call is successful, you can render the WeatherReport element within the root (passing it the data you obtained using the API call as props).

## 15. API ACCESS, AJAX, STATE, SIDE-EFFECTS

Redo the above exercise using class components.

## 16. STYLING, CLASSES

Add styles to the WeatherReport component by showing the details inside a box with background color dictated by weather condition codes (group codes) -

<https://openweathermap.org/weather-conditions>

For example,

- If the weather is clear, show the details in lightblue background
- If the weather is rainy/drizzle - use dark blue

Similarly choose some colors for the other weather conditions.

## 17. RENDERING LIST

Create an CitiesWeatherReport component that let's one view weather details for a group of cities. It takes as input (props) the weather details for a group of cities. It should display the weather details for each city. When doing so, use the WeatherReport component created in a previous exercise. Also make sure to give a unique key prop value for each WeatherReport element rendered.

```

const weatherDetails = [
  {
    "name": "New York",
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "01n"
      }
    ],
    "main": {
      "temp": 281.52,
      "feels_like": 278.99,
      "temp_min": 280.15,
      "temp_max": 283.71,
      "pressure": 1016,
      "humidity": 93
    },
    ...
  },
  {
    "name": "New York",
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "01n"
      }
    ],
    "main": {
      "temp": 281.52,
      "feels_like": 278.99,
      "temp_min": 280.15,
      "temp_max": 283.71,
      "pressure": 1016,
      "humidity": 93
    },
    ...
  },
  ...
]

```

**Note:** You can get weather details for multiple cities in one API call - check <https://openweathermap.org/current#severalid>. Extra credits if you make an Ajax call to fetch the weather data and then pass them as props to the CitiesWeatherReport component when rendering it.

18. Implement a WeatherTracker component that displays the weather for a list of cities. It render a CityInput component that accepts a city name and adds it to a list, as child. It also displays a CitiesWeatherReport element that shows weather reports for cities in the cities list that is maintained.

First decide what needs to be maintained in state. Which components need to work with the list of cities? Where will this state be maintained - WeatherTracker (parent), CityInput, or CitiesWeatherReport? Where will you define the method to add a city to the list of cities? How will CityInput be able to access this method?

**Note:**

- a. The componentDidMount() method of CitiesWeatherReport needs to make an Ajax call to fetch weather details for every city in the list, every time it is rendered.
  - b. Fetching data for a city may be redundant, as previous Ajax calls may have fetched the data for existing cities. Try creating a data caching utility which maintains the data fetched for a particular city. Once an Ajax call to fetch data for a city is made frshly, the cached data is updated. If another call goes out for the same city within 15 mintes (say) on the same date, then the cached data may instead be used. This will take some amount of involved design and implementation. The data cache may be part of the state maintained by a component. This is completely optional.
19. We shall add filter functionality in the WeatherTracker component. Add a CityFilter component with a search input. User can type within this input and all cities that have the search term as a substring of the city name are displayed. For example, if "New" is typed, cities like "New York", "New Orleans" are displayed (assuming they exist in the list of added cities).
  20. Add a delete button against each city list item in CitiesWeatherReport component. It should remove the city list item when clicked.
  21. Add propTypes and defaultProps for all components in the Weather tracker application. Set appropriate data types for props, and also default values for the props where it makes sense.
  22. Add loading, and error state UIs for the Weather tracker application apart from the loaded state UI which already exists. You can display a loading message for CitiesWeatherReport as it fetches the data.
  23. Use Create React App to create the Weather tracker application. Modularize and organize the code you have created in earlier exercises using a standalone HTML file. Create folders for components, services (that has methods to fetch data from OpenWeatherMap API, and data caching layer in case you have implemented that), models (like CurrentWeather - that models the response from the API), actions etc. You app should work like before.

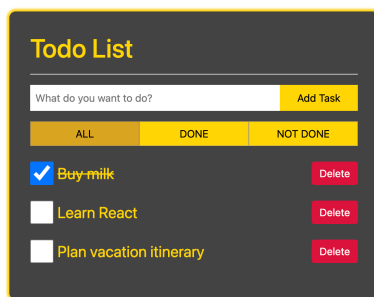


24. We shall now break down the application into 3 screens.
- On top of each screen is a Navigation Menu - it has 2 links - "All Cities", and " All Reports". Clicking "All Cities" takes user to the landing page (<http://localhost:3000/>). Clicking "All Reports" shows the page displaying weather reports for all cities (<http://localhost:3000/all>)
  - **<http://localhost:3000/>**. It shows a list of cities that have been added in the application. The CityInput component accepts the name of a new city. A new component called CitiesList is displayed below it, that shows the names of cities added to the application.
  - **<http://localhost:3000/:city>**, where :city is a placeholder for a city's name. Thus an actual route could be <http://localhost:3000/new+york> or <http://localhost:3000/sunnyvale> etc. The component displays the weather details for the specified city only.
  - **<http://localhost:3000/all>** - The CitiesWeatherReport component that displays weather report of all added cities is displayed (only limited weather information is displayed) Clicking any one city within this list, displays the detailed weather report for that city (<http://localhost:3000/:city>)
25. Introduce Redux into the Weather tracker application.
- Store the cities list and weather data fetched for cities in the Redux store.
  - Define a reducer to take care of making state changes for fetch of weather data for a particular city, and weather data for a list of cities.
  - Define the weather data for cities in a "normalized" form for easy data access.
  - When making a call to fetch data for a list of cities, check whether each city's weather data is already available in the store. If so, exclude that city automatically - a good place to do this is in the Saga middleware functions.

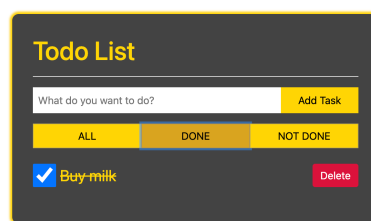
# Creating a Todo List

Create a Todo List application that lets you maintain a list of things to do, and track if they have been completed or not. Specifically,

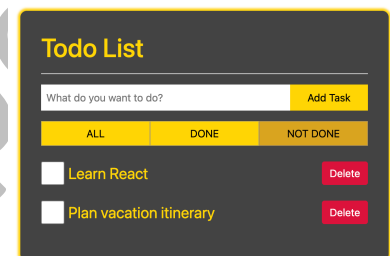
1. Entering the task to do in the input, and clicking “Add Task” adds the task to the end of the list.
2. Any task can be deleted by clicking the delete button for it.
3. The tasks are assumed to be incomplete (NOT DONE) when they are added. Checking a checkbox next to the task toggles it between complete (DONE) and incomplete (NOT DONE). An incomplete task is marked as struck out.
4. The three buttons on top (ALL, DONE, NOT DONE) help to view all tasks, the ones completed, or the ones that are incomplete. The selected filter is evident from the darker shade for the button.



All tasks to do (ALL)



Completed tasks (DONE)



Incomplete tasks (NOT DONE)

## Step-by-step hints

1. **Design** the component tree for the application. See which pieces of the application’s UI forms a logical whole.

### Suggested hierarchy

- **App**, a top-level wrapper component
  - o **TaskInput** to receive user input for new task (input, Add Task button)
  - o **TaskFilter** to filter based on completion status (3 buttons)
  - o **TaskList** to list the tasks to do along with check box and Delete button.

This hierarchy is assumed and is used to explain further steps. If you design it differently, you would need to interpret the hints that follow appropriately.

## 2. COMPONENTS

Create the **TaskInput** component (function component is suggested). Enclose the input and the button inside a form element and return the form element. Render an element of TaskInput, to see if it works fine.

Add Task

### 3. PASSING PROPS, RENDERING LIST

Create the **TaskList** component. Let it accept the items to show (say, **tasks**) as a prop. A sample for this prop is shown below.

```
const tasks = [  
  { id: uuidv4(), name: 'Buy milk', completed: true },  
  { id: uuidv4(), name: 'Learn React', completed: false },  
];
```

The TaskList renders the tasks within an ordered list (ol, li). Each li should have a checkbox, name of the task and a delete button. Make sure to set the **key** using the id.

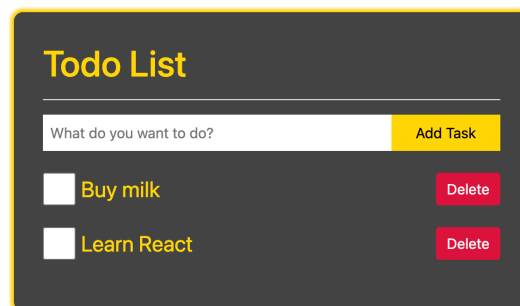
**Note:** Here the JS library – **uuid**, is used to generate a unique id for every task. It can be included in HTML with script source <https://cdnjs.cloudflare.com/ajax/libs/uuid/8.3.2/uuidv4.min.js>. For usage as a Node package check <https://github.com/uuidjs/uuid>

Render an element of TaskList, passing it tasks as props, to see if it works fine.



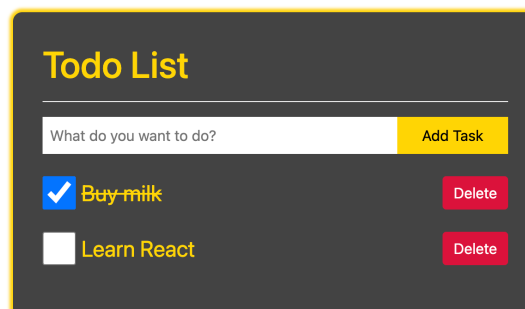
### 4. COMPOSING COMPONENTS, PASSING DOWN PROPS TO CHILD

Create the **App** component that has a **TaskInput** element and **TaskList** element as children. The app takes the tasks array as prop and passes it down to TaskList component. Render an element of App, passing it tasks as props, to see if it works fine.



### 5. CONDITIONALLY APPLYING STYLES / CSS CLASSES

Add styles so that completed tasks are struck out - try using both inline as well as CSS classes that are applied conditionally.



## 6. MAINTAINING STATE, PASSING FUNCTION PROPS, EVENT HANDLING, REFS, CONTROLLED COMPONENT

Implement the **Add Task feature**. Since the list of tasks to show would change dynamically, we would need to maintain the tasks to show in some component's state.

Ask yourself these questions before you begin.

- Where would you maintain the state? What would be its initial value?
- Where would you define the function to add a task? What would it receive as an argument? How would it update the tasks state?
- The TaskInput needs to add a task on submit of the form (when Add Task is clicked). How would it get access to the function to add a task?
- You need to prevent form submission. How would you do it?
- How can you read the user input?
- How would you generate a unique id for the new task

### Hints

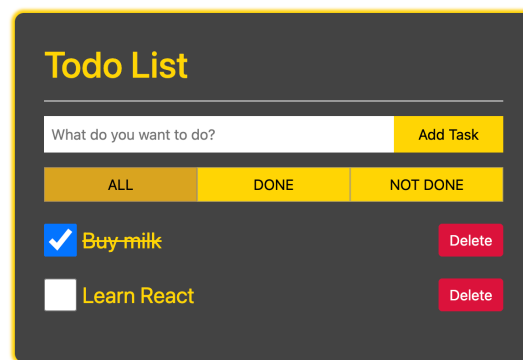
- The list of tasks would need to be added through TaskInput component (adding a task is initiated on click of Add Task button), and is required by the TaskList component, we require a common ancestor component. This leaves us with only one option! Which one?
- You can change the state in the component where it is maintained. This leaves with only one option for the component to define the function to add a task.
- Functions can be passed as prop from a parent component to a child component, and this acts as a way for a child to influence the state of the parent (we can consider this as a form of communication between child and parent).
- When a prop (having non-primitive like array/object) is used to set state (eg. the initial value), make sure to create a copy of the prop, and set the state. This way we do not modify props. The spread operator is a convenient way to create a copy.
- When updating an object or array state, we make sure to copy them over (it makes detecting changes easy as a simple equality check with the old value will help understand if the data changed). Read this article - <https://www.robinwieruch.de/react-state-array-add-update-remove/>
- You can use either **ref** or **controlled components pattern** to get hold of task input provided by the user. In the first approach, use useRef for function component, and createRef for class components - <https://reactjs.org/docs/refs-and-the-dom.html>. For the second approach refer <https://reactjs.org/docs/forms.html#controlled-components>
- The event handler gets an event object whose preventDefault() methods prevents default action of the browser for the event (eg. form submission by browser)
- For generating the unique id, use the uuid library

## 7. PASSING ARGUMENTS TO AN EVENT HANDLER

Implement the **Delete Task feature**

- Where would you define the function that deletes a task? What would it receive as argument(s)? How would it update the tasks state?
- The TaskList should be able to delete tasks. How would it get access to the function to do so?

- How would you enable passing the task, to the function that deletes a task, when the delete button is clicked? You would need to define an `onClick` event handler inline in the JSX markup for the Delete button, and pass the task when calling the function.
8. Implement the **Toggle Completion Status feature**. The completion status of a task has to toggle (change to the opposite value) on checking / unchecking the checkbox.
- Where would you define the function to toggle the completion status for a task? What would it receive as argument(s)? How would it update the tasks state?
  - The `TaskList` component should be able to toggle completion status of tasks. How would it get access to the function to do so?
  - How would you enable passing the task, to the function that toggles the completion status of a task, when the checkbox is checked / unchecked? You would need to define an `onChange` event handler inline in the JSX markup for the checkbox, and pass the task when calling the function.
9. **MAINTAINING STATE, PASSING DOWN PROPS TO CHILD**
- Create the **TaskFilter** component. Render it as a child of the `App` component. In this step we shall render only the UI for the component. In the next step we would implement filter tasks feature.
- The `TaskFilter` needs to know the current filter selection (`ALL/DONE/NOT DONE`) in order to show the button differently. How would you maintain the current filter selection?
  - Where would you maintain that value? What would be the initial value?
  - How will the `TaskFilter` get the value?



## 10. PASSING ARGUMENTS TO AN EVENT HANDLER

Implement the **Filter Tasks feature**.

- Where would you define the function that filters a task? What would this function receive as argument(s)? What state will it change?
- Using the tasks and current filter selection you can derive the set of tasks to be shown (for example, given the tasks, and current filter selection 'DONE' (say) you know the tasks the app needs to show. You can define a function to return the filtered set of tasks for current filter selection – but where would you define this?
- How would `TaskFilter` access the function?
- How would it set up the function to be called with the right argument(s)
- What changes would you make to have the `TaskList` render only the filtered set of tasks instead of all the tasks that it currently does?

## 11. DEFAULT PROPS, PROP TYPES

Add default values for the props of each components (wherever it makes sense). Define data types for all the props each component

www.digdeeper.in