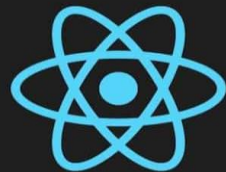


React



What is React?

- JavaScript library for building user interfaces and It is maintained by [Facebook](#) , stable release: **18.x**
- Quick loading, highly performant, mobile enabled interactive applications
- React is simply a client side, we can develop Single Page Applications easily(**SPA**)
- React will need other libraries like [axios](#) to connect backend services
- React uses Virtual DOM, much more performant than the Angular based apps.
- React will efficiently update and render just the right components when your data changes.
- Written React App will does appear faster always to the end user.
- React is Component Based - component logic is written in JavaScript instead of templates, we can easily pass rich data through your app and keep state out of the DOM.

Virtual DOM

- React uses the concept of the Virtual DOM
- it is selectively render the subtree of DOM elements into the rendering of the DOM on state change
- Use different algorithm with the browser DOM tree to identify the changes
- Instead of creating new object, React just identify what change is took place and once identify update that state.
- This way it is creating a virtual DOM and improving the performance too
- Can be render on server and sync on Local

Why React???

s

Fast

Apps made in React can handle complex updates and still feel quick and response

Modular

Instead of writing large files of code, you can write many smaller, reusable files. React's modularity can be a beautiful solution to JavaScript's maintainability problems

Scalable

Large programs that display a lot of changing data are where React performs best

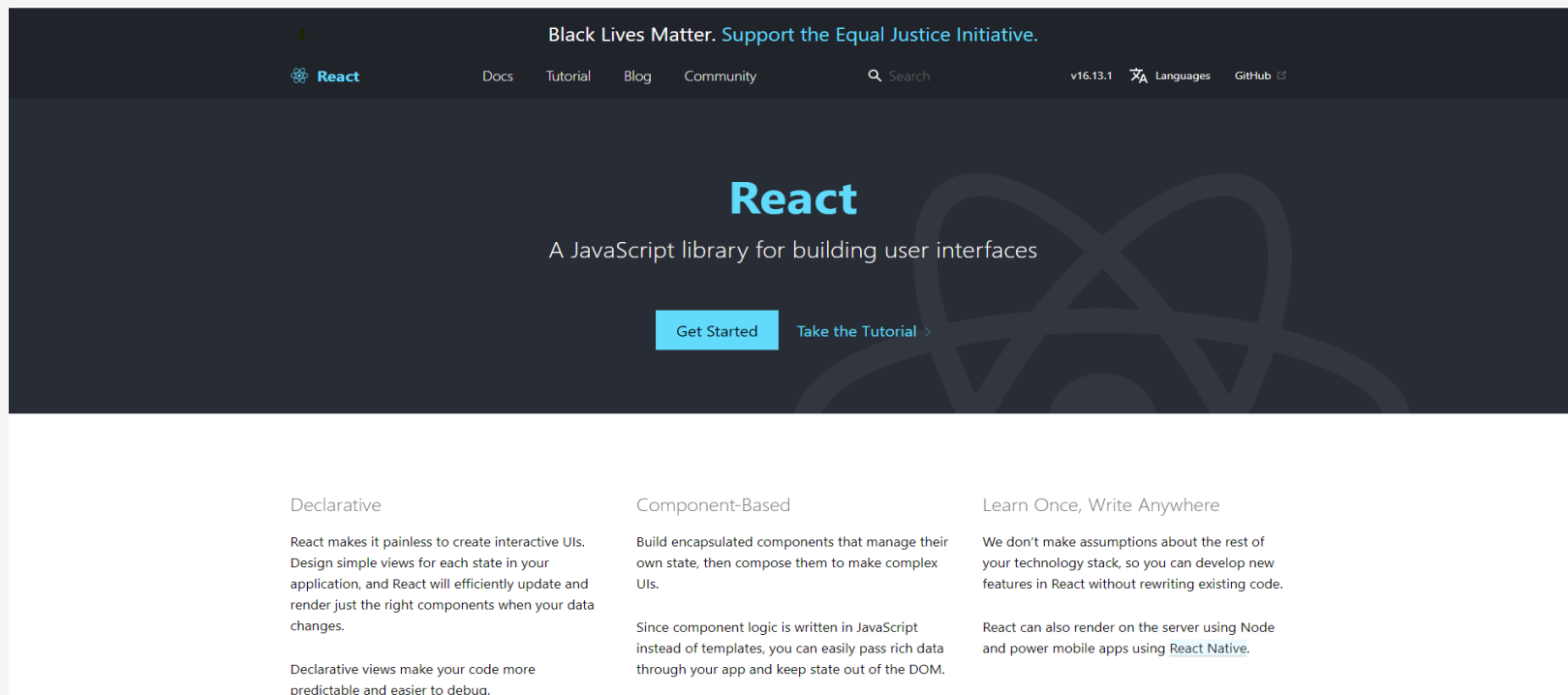
Flexible

Apps built with ReactJS are extremely flexible, This flexibility saves a huge amount of developers' time and money in the long

Popular

Maintained and managed by Facebook

-
- React can also render on the server using Node and power mobile apps using [React Native](#).
 - React apps run in the browser, they do not run on the server by default (Things happen instantly since they happen in the user's browser so do not have to wait for a server response) <https://reactjs.org/>



React Topics

- React installation
- React application flow
- JSX introduction
- Component creation in React
- Adding **reactstrap** to React
- Functional vs class component differences

React Topics

- Reactstrap components-**Navbar,Card,Models,Button,Form,FormGroup,**
- Components communication and passing data –**props & key**
- Routing
- State & Lifecycle
- Handling Events
- **Axios with React**

React Software

- IDE: Visual Studio Code : <https://code.visualstudio.com/download>
- Node.js 16.16.0 <https://nodejs.org/en/>
- Full Internet access is required with no firewall restrictions.

First React Application

- `npm install -g create-react-app`
- `npx create-react-app hello-world`
- `cd hello-world`
- `yarn start`

Note: React will not allow you to create project name with capital letters.

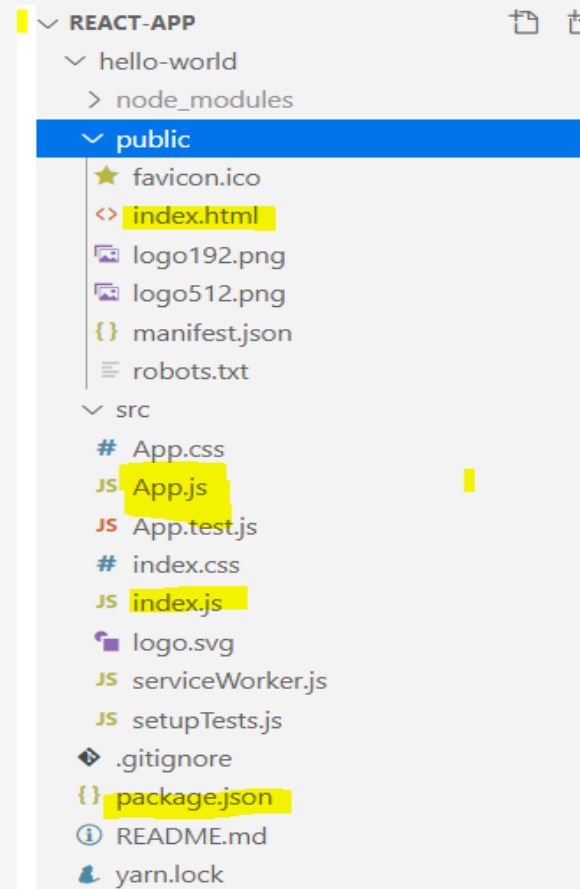
React App Details

➤ **package.json**

➤ **index.js**

➤ **App.js**

➤ **Index.html**



Core Concept of Reactjs

JSX

Components

Unidirectional
Data Flow

Virtual DOM

Introduction of JSX

- JSX, or JavaScript [XML](#) , is an extension to the JavaScript language syntax
- React components are typically written using JSX
- JSX is a syntax extension for JavaScript code looks a lot like HTML(not valid JavaScript web browsers can't read it)
- It was written to be used with React
- If a JS file contains JSX code, then that file will have to be compiled before the file reaches a browser, a JSX compiler will translate any JSX into regular JavaScript.
- Link: <http://magic.reactjs.net/htmltojsx.htm>

JSX

- Default code of App.js component.
- It should render only one ELEMENT.

```
function App() {  
  return (  
    <h1> welcome to ReactJS world!!!</h1>  
  )  
}
```

- It is also type-safe and most of the errors can be caught during compilation

```
return(  
  <div>  
    <p>Rendering</p>  
    <p>Multiple</p>  
    <p>Elements</p>  
  </div>  
);
```

- If we want to return more elements, we need to wrap it with one container element, in this case it is <div>
- JavaScript expressions can be used inside of JSX

ReactStrap

➤ Step 1: Installation:

```
Npm install -g bootstrap reactstrap
```

Step 2: Add this to `index.js`:

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

➤ Some of the components: `Card, CardGroup, Container, Button`

➤ Link: <https://reactstrap.github.io/components/>

React Component

- React code is made of entities called components.
- Components can be rendered to a particular element in the [DOM](#) using the React DOM library.

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

- App is the React Component and that will be rendered in the html(inde.html)element whose name is 'root'.
- 2 types of Components basically.

➤ **Functional component**

➤ **Class-based component**

Functional component

- Functional components are declared with a `function` that then returns some JSX
- Example:

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

`welcome` is a valid React component

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
const element = <Welcome name="Sachin" />;  
ReactDOM.render(element, document.getElementById('root'));
```

Prints:
Hello, Sachin

Functional Component

- The only “state” that a **functional component** effectively has access to are the props passed to it from its parent **component**.
- There are a number of benefits to choosing a functional approach:
 - No side-effects allowed, that is, the operation is stateless
 - Always returns the same output for a given input
 - It gives solution without using state
 - There is no render method used in functional components
 - Component lifecycle method do not exist in functional component because we cannot use `setState()` method inside component



State Hook

- Hooks are functions that let you “hook into” React state and lifecycle features from **function components**.
- React 16.8 feature
- Hooks don’t work inside class components
- **useEffect**: serves the same purpose as **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** in React classes
- **useState**: returns a pair: the **current state value** and a function that lets you update it.

Class component

- It is regular ES6 classes that extends component class from react library

- **Stateful** because they implement logic and **state**.

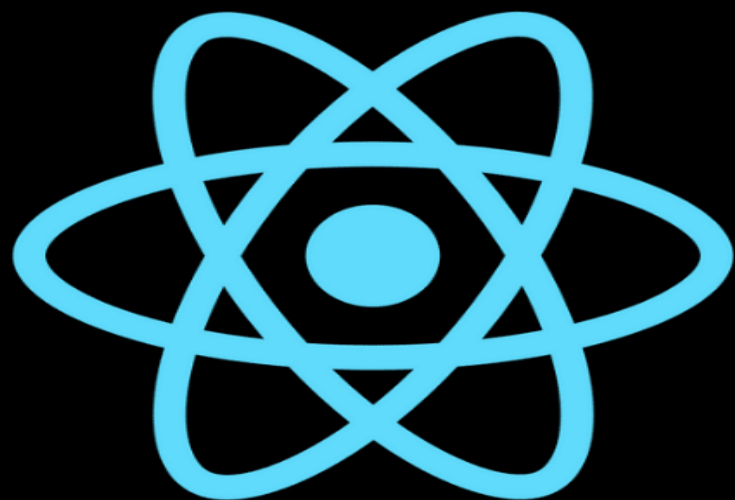
- Lifecycle hooks are available.

- It must have **render()** method returning html

```
export default class Players extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      playersList: [],  
    };  
  }  
  
  componentWillMount() {  
    getPlayers().then(res => {  
      this.setState({playersList: res.data.players})  
      console.log(res);  
    })  
  }  
  
  render() {  
    return (  

```

props



React JS

*What is
PROPS?*

props

➤ “**props**” -Stands for **properties** and is being used for passing data from one component to another

➤ props data is read-only

```
const PlayersList = [
  { name: 'Virat', id: 1, country: 'India' },
  { name: 'Rahul', id: 2, country: 'India' },
  { name: 'Rohit', id: 3, country: 'India' },
]
const Players = PlayersList.map((plr, index) =>
  <Player key={index} player={plr} />
);
const Example = (props) => {
  return (
    <CardGroup>
      {Players}
    </CardGroup>
  )
}
```

Prints 3 Players(Virat,Rahul,Rohit) info in Card Component

```
const Example = (props) => {
  return (
    <div>
      <Card>
        <CardBody>
          <CardTitle>{props.player.name}</CardTitle>
          <CardSubtitle>{props.player.id}</CardSubtitle>
          <CardText>{props.player.category},
            {props.player.country}
          </CardText>
          <Button>Button</Button>
        </CardBody>
      </Card>
    </div>
  );
};
export default Example;
```

props.children

- `props.children` refers to any elements and this includes plain text between the opening and closing tag of component

```
App.js
class App extends Component {
  render() {
    return (
      <div className="App">
        <NavBar/>
        <Emp name="Sachin">onsite</Emp>
        <Emp name="Dhoni">offshore</Emp>
      </div>
    );
  }
}
```

```
Emp.js
const Example =(props) =>{
  return (
    <h1>Emp Name: {props.name},
    location: {props.children}</h1>
  )
}
export default Example;
```

Output:
Emp Name: Sachin, location: onsite
Emp Name: Dhoni, location: offshore

React keys

- A “key” is a special string attribute you need to include when creating lists of elements

```
function NumberList(props) {  
  const numbers = props.nos;  
  const listItems = numbers.map((number) =>  
    <li>{number}</li>  
  );  
  return (  
    <ul>{listItems}</ul>  
  );  
}
```

```
const numbers = [10, 20, 30, 40, 50];
```

```
ReactDOM.render(<NumberList nos={numbers} />, document.getElementById('root'))  
);
```

✖ ▶ Warning: Each child in a list should have a unique "key" prop.

Check the render method of `NumberList`. See <https://fb.me/react-warning-keys> for more information.
in li (at src/index.js:11)
in NumberList (at src/index.js:21)

index.js

React Keys

```
function NumberList(props) {
  const numbers = props.nos;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [10, 20, 30, 40, 50];
ReactDOM.render(<NumberList nos={numbers} />,
  document.getElementById('root'));
```

When you don't have stable IDs for rendered items, you may use the item index as a key

```
function NumberList(props) {
  const numbers = props.nos;
  const todoItems = numbers.map((no, index) =>
    // Only do this if items have no stable IDs
    <li key={index}>
      {no}
    </li>
  );
  return (
    <ul>{todoItems}</ul>
  );
}
```

- Keys help React identify which items have changed, are added, or are removed.
- Keys should be given to the elements inside the array to give the elements a stable identity

Conditional Rendering

```
const Bye = props => <h1> Bye {props.user}</h1>
```

```
const Hello = props => <h1> Hello {props.user}</h1>
```

```
const Greeting = (props) => (props.in)?<Hello user={props.name}/>:<Bye user={props.name}/>
```

```
ReactDOM.render(<Greeting in={false} name="Sachin" />,document.getElementById('root'));
```

Change the value to `true` and check the output

Component Lifecycle

- Lifecycle methods as the series of events that happen from the birth of a React component to its death.
- 3 cycles of phases: birth, growth, and death
- Mouning
- Update
- Unmount

|

Component Lifecycle Functions

- ***render()*** : Component state can not be modified within the *render()*
- ***componentWillUnmount()***: is executed before rendering
- ***componentDidMount()***: is executed after the first render only on the client side
- ***componentWillReceiveProps()***: is invoked as soon as the props are updated before another render is called
- ***componentWillUpdate()***: is called just before rendering

Home Players Options ▼



After clicking Players link, Players component has to call **getPlayers()** and store all players into **playersList** array



```
componentWillMount() {  
  getPlayers().then(res => {  
    this.setState({playersList: res.data.players})  
    console.log(res);  
  })  
}
```

State

- A state is primarily used when you make changes that only make sense within the component.
- Component state can be modified over time in response to user actions, network responses, and anything.
- **Class-based** implementation is required (By extending `React.Component`)
- Unlike `props`, `components cannot pass data with state`, but they can create and manage it `internally`.
- State should be modified with a special method called `setState()`



props vs State?

- Components receive data from outside with props, whereas they can create and manage their own data with state
- Props are used to pass data, whereas state is for managing data
- Data from props is read-only, and cannot be modified by a component that is receiving it from outside
- State data can be modified by its own component, but is private (cannot be accessed from outside)
- Props can only be passed from parent component to child (unidirectional flow)
- Modifying state should happen with the `setState()` method

Functional Component vs Class Component

Functional Component

- Less code than class component
- Re-usability
- Stateless by default, but by Hooks –Stateful(React 16.8)
- Recommended option

```
function Follower(props) {  
  const showMessage = () => {  
    alert('Followed ' + props.user);  
  };  
  
  const handleClick = () => {  
    setTimeout(showMessage, 3000);  
  };  
  
  return (  
    <button onClick={handleClick}>Follow</button>  
  );  
}
```

Class Component

- Require to write more code
 - Less re-usable
 - Stateful

```
class Follower extends React.Component {  
  constructor(props) {  
    super(props);  
    this.showMessage = this.showMessage.bind(this);  
    this.handleClick = this.handleClick.bind(this);  
  }  
  
  showMessage() {  
    alert('Followed ' + this.props.user);  
  }  
  
  handleClick() {  
    setTimeout(this.showMessage, 3000);  
  }  
  
  render() {  
    return <button onClick={this.handleClick}>Follow</button>;  
  }  
}
```

React Events

- Handling events with React elements is very similar to handling events on DOM elements
- React events are named using camelCase, rather than lowercase- `onClick`, `onChange`, `onSubmit`
- With JSX you pass a function as the event handler, rather than a string.
- Cricket Application Sample Events:
 - `<Form onSubmit={this.SavePlayer}>`
 - `<Input type="text" onChange={this.handleChange}></Input>`
 - `<Button onClick={()=>modal()} >Edit Player</Button>`

Axios-promise-based HTTP client

➤ **Axios** is an easy to use promise-based HTTP client for the browser and Node. **Js**

➤ **Axios** is promise-based(asynchronous)

➤ `npm install -g axios`

➤ Some of the methods

➤ It is recommending to use this from `service`.

- `axios.get(url[, config])`
- `axios.delete(url[, config])`
- `axios.head(url[, config])`
- `axios.options(url[, config])`
- `axios.post(url[, data[, config]])`
- `axios.put(url[, data[, config]])`
- `axios.patch(url[, data[, config]])`

The **response** object

When the request is successful, your `then()` callback will receive a response object with the following properties:

data: the payload returned from the server. By default, Axios expects JSON

status: the HTTP code returned from the server.

statusText: the HTTP status message returned by the server.

headers: all the headers sent back by the server.

config: the original request configuration.

request: the actual XMLHttpRequest object (when running in a browser).

Get All Players Flow

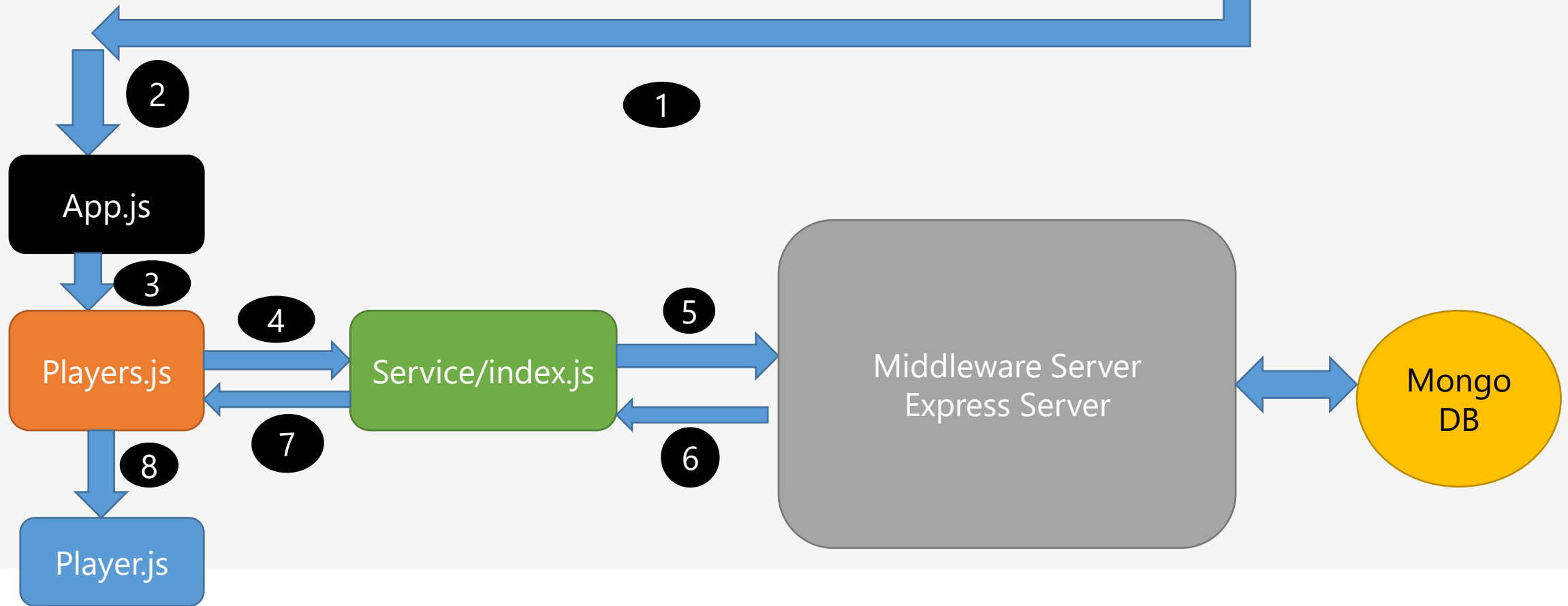


Cricket App

Home

Players

Options ▼



Get All Players Flow- Internals

App.js

Check route

```
<Route path="/players" exact component={Players}/>
```

Players.js

1) State inside constructor
playersList: []

2) componentWillMount()
Call service/index.js `getPlayers()`
`this.setState({playersList: res.data.players})`

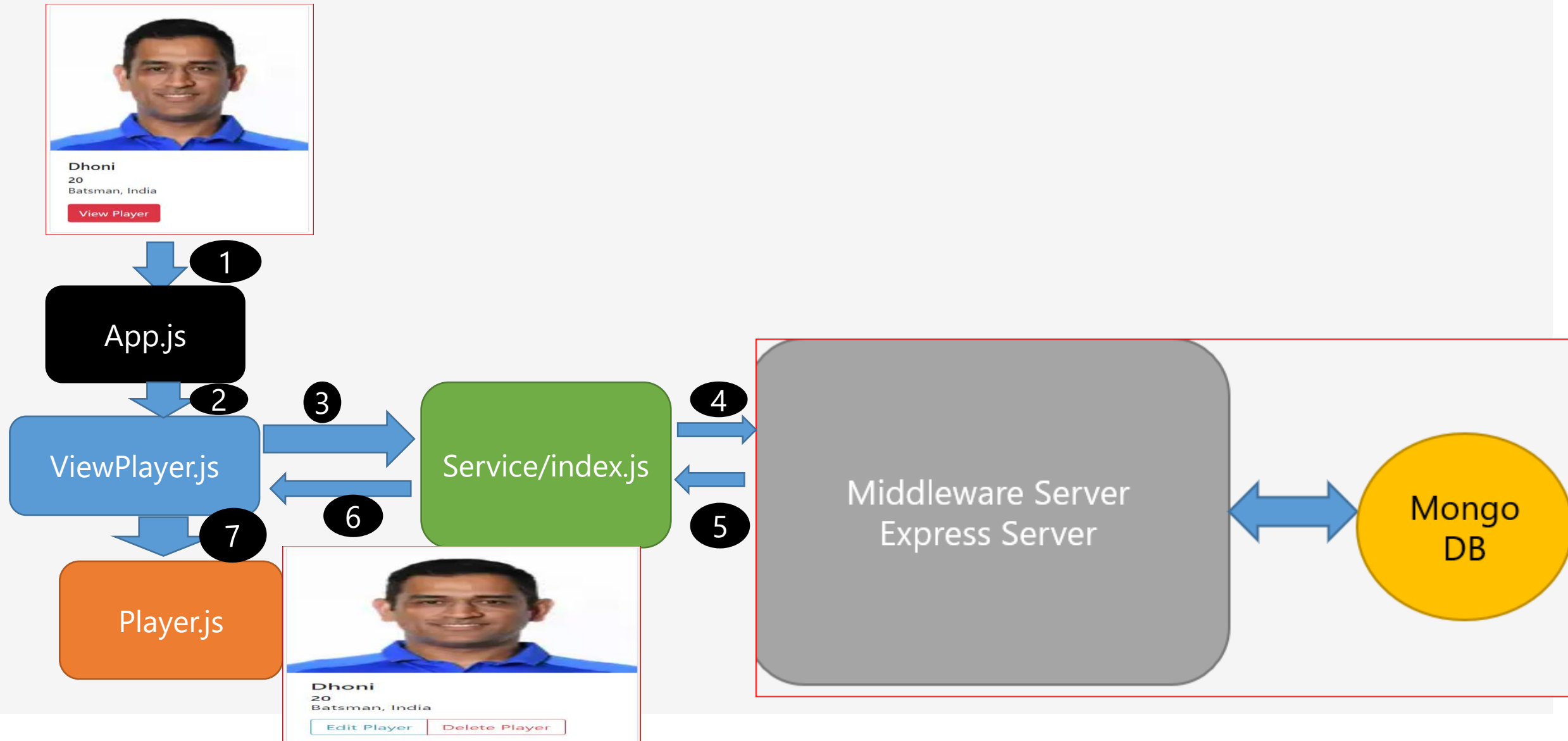
3) render()
send each player to Player.js component

Player.js

1) Print each Player in Card
2) And check weather the request from Players.js or from ViewPlayer.js component
3) If the request comes from Players.js component display **View Player** Button or else

Edit Player, **Delete Player** Buttons

View Single Player Flow



View Single Player Flow-Internals



App.js
Check the route
`<Route path="/players/:id" component={ViewPlayer}/>`

ViewPlayer.js
constructor

1) inside constructor

```
this.state = {  
  player: {},  
  modal: false // parent(ViewPlayer) state  
}
```

2) **componentWillMount():**

Call service/index.js and store single player into player object

```
this.setState({player: res.data.player})
```

3) **render() :** send player to Player.js component along with view, callback function toggleModal.

```
<Player modal={this.toggleModal} view player={this.state.player}></Player>
```

4) Keep UpdatePlayer component inside Modal- this will be enabled only if the user clicks Edit Player button on Player.js component

Player.js

- 1) Print each Player in Card
- 2) And check whether the request from Players.js or from ViewPlayer.js component
- 3) In this case request from ViewPlayer.js component, so displays

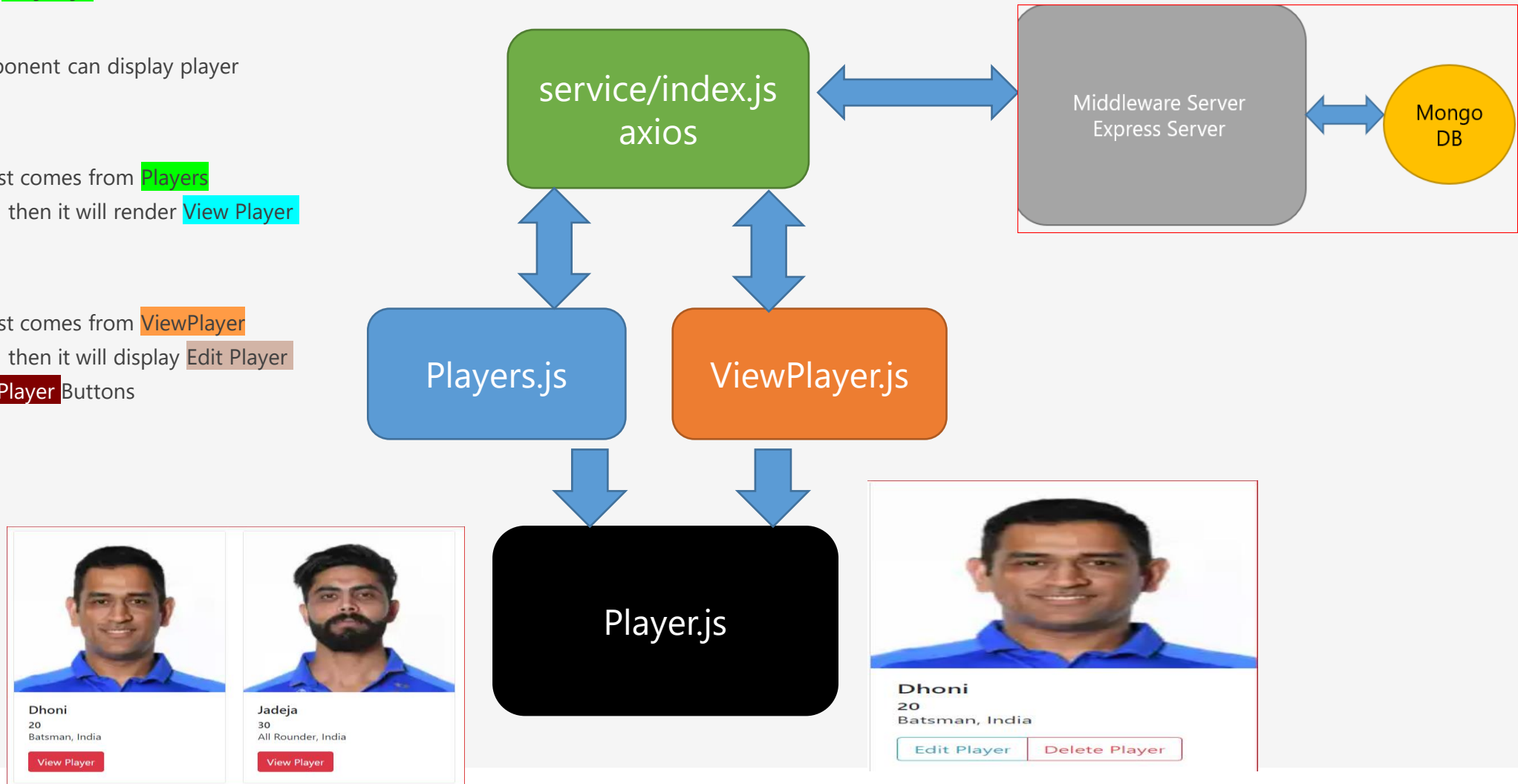
Edit Player, Delete Player Buttons



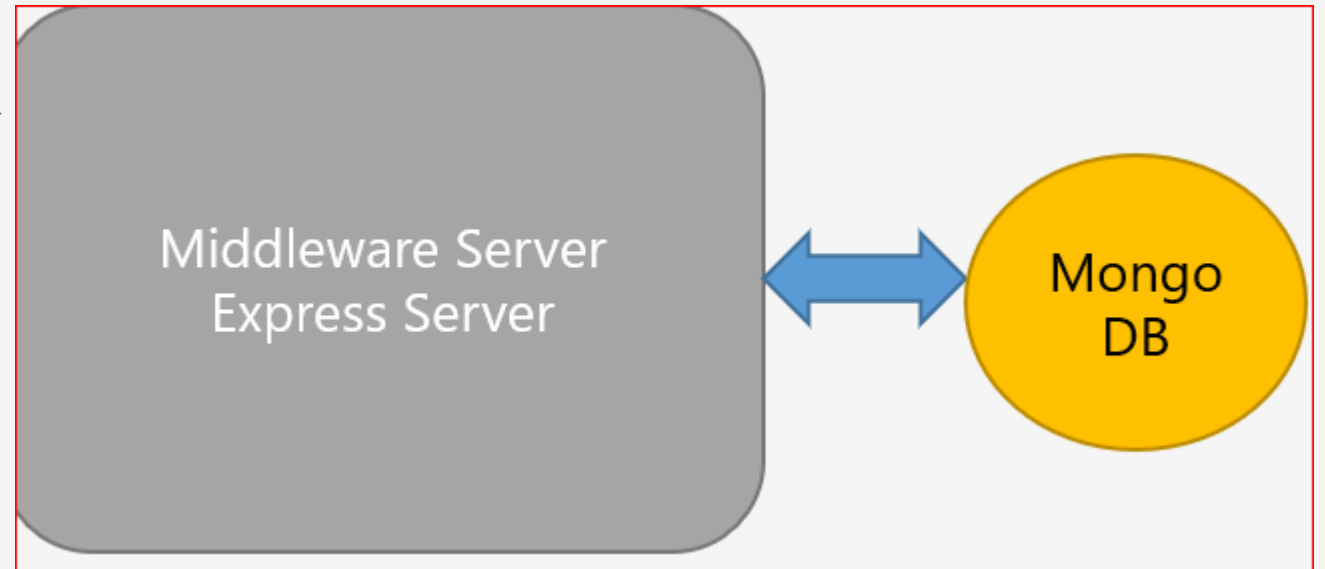
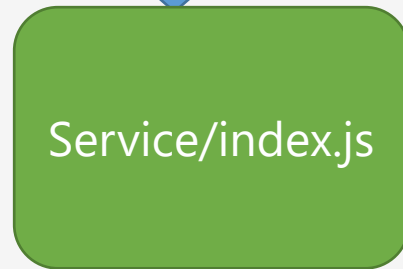
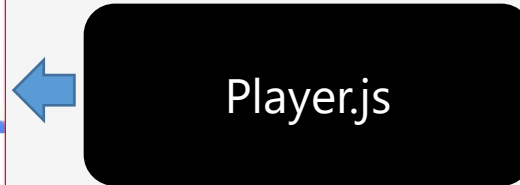
Best practice-Reuse Component-Player

Player.js

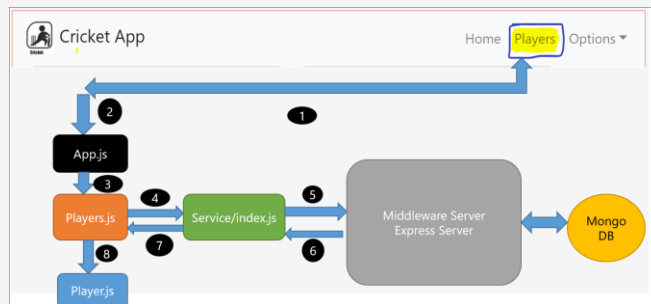
- **Player** component can display player details
- If the request comes from **Players** component, then it will render **View Player** Button
- If the request comes from **ViewPlayer** component, then it will display **Edit Player** and **Delete Player** Buttons



Delete Player Flow



Prints
rest of
the
players



Add Player Flow

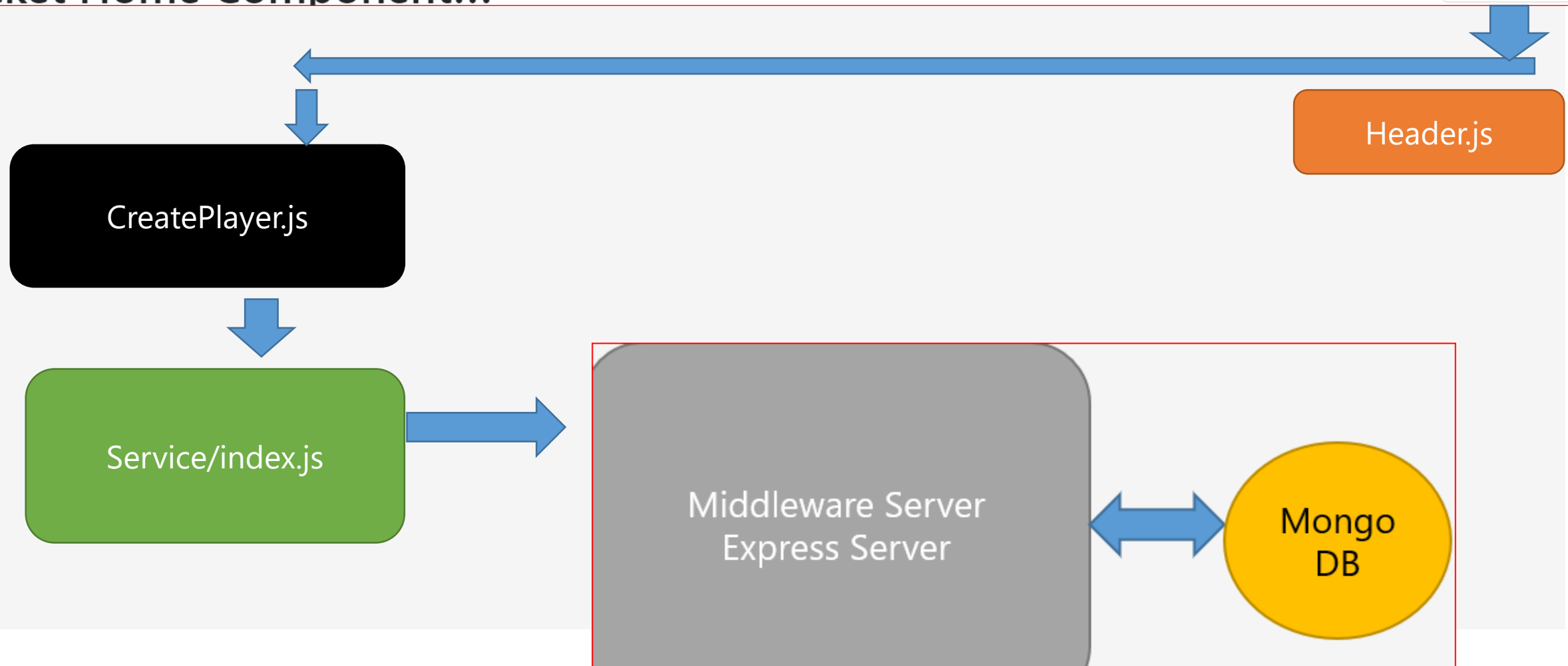


Cricket App

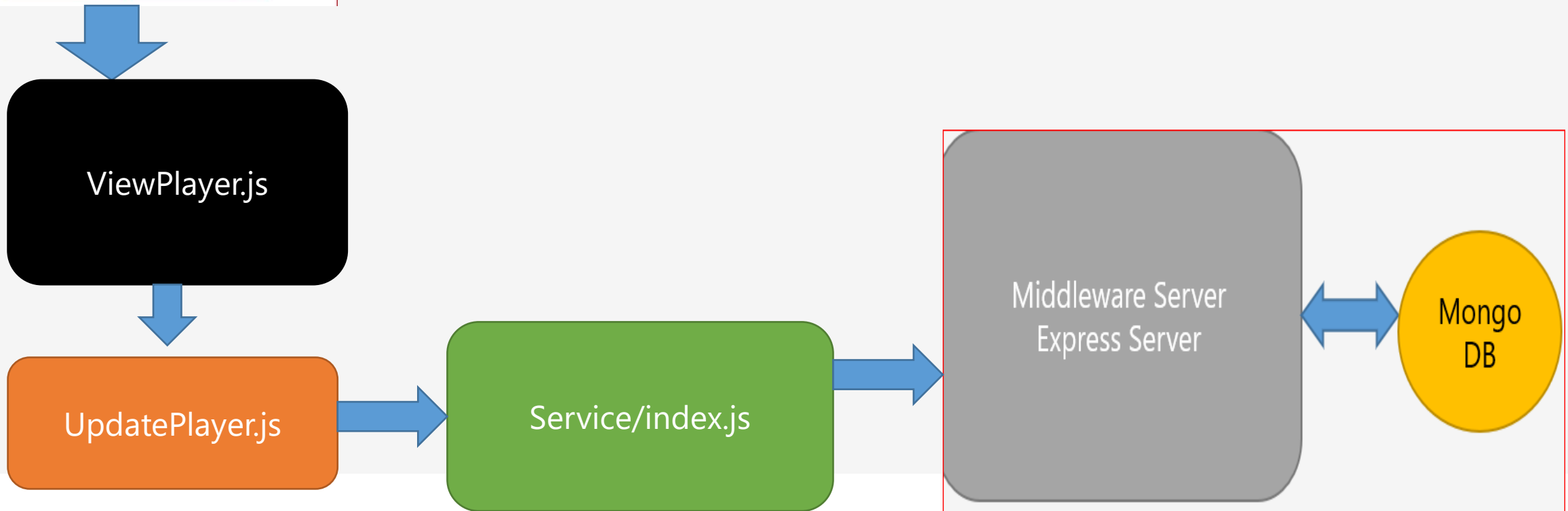
Home Players Options

Cricket Home Component!!!

Add Player



Update Player



Update Player-Internals



After clicking Edit Player button we are calling `toggleModel()` function of `ViewPlayer.js` Component toggle the `model`

ViewPlayer.js

UpdatePlayer.js

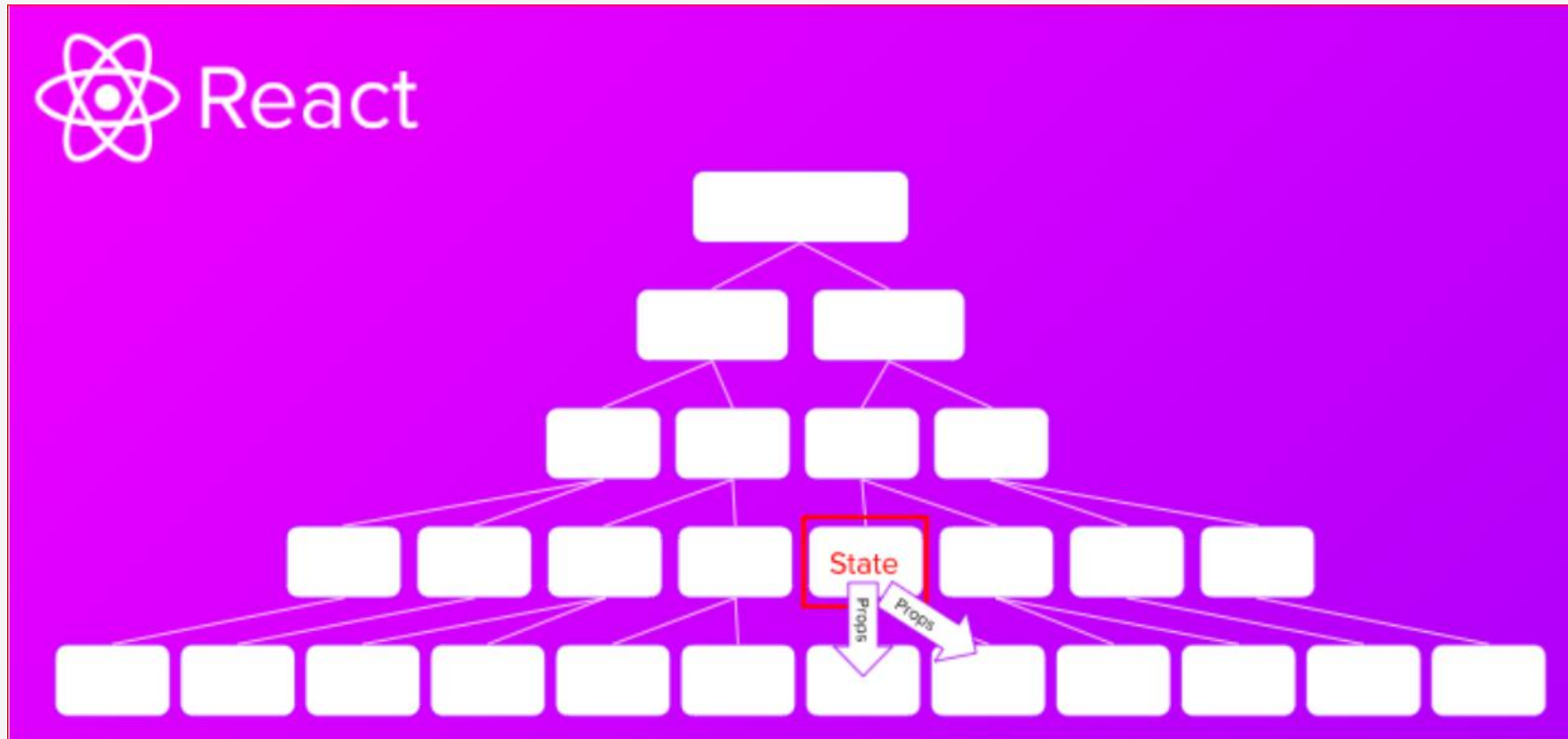
- 1) In constructor:
 - a) `this.state = {id: props.player.id,}`
 - b) `handleChange(event)`
`this.setState({[event.target.name]: event.target.value})`
- 2) `savePlayer(event)`
`let player = {id: this.state.id, name: this.state.name,}`
`updatePlayer(player).then(res=> {`
- 3) `render():` populate the form with existing values:

service/index.js

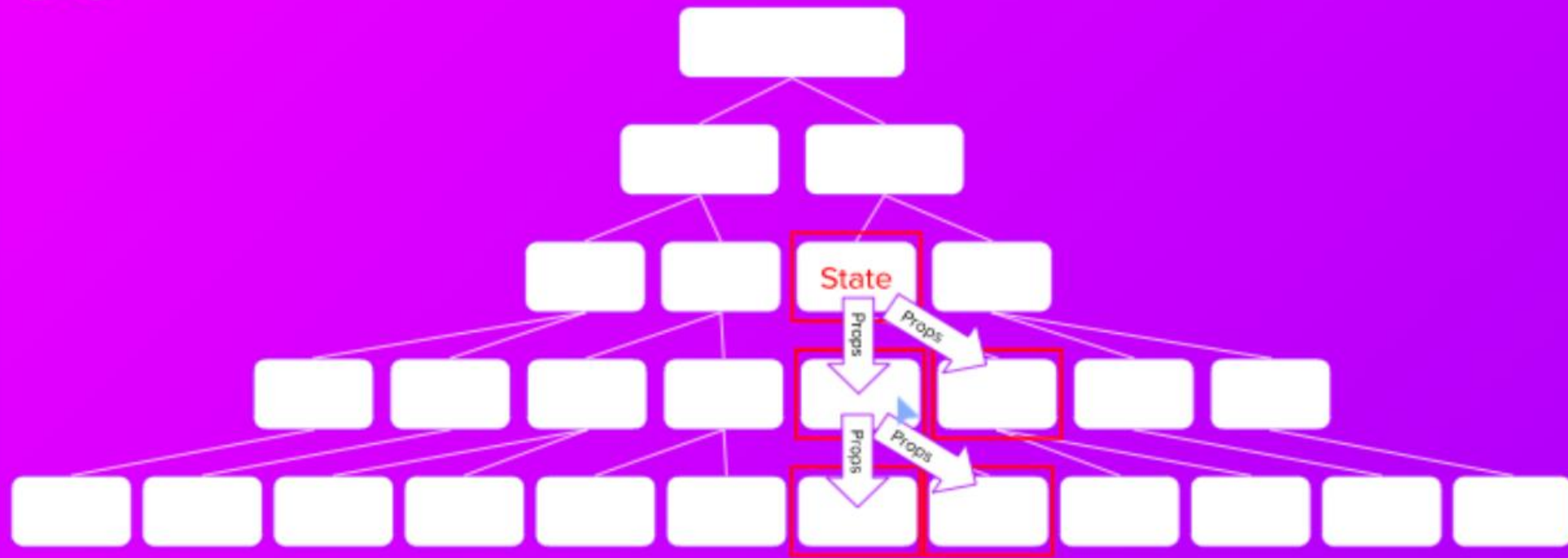
React Recap

- A React codebase is made up of components.
- Components are written using JSX.
- Data flows from parent to children, except when it comes to **state**, which originates inside a component
- Components possess a small set of lifecycle and utility methods
- Components can also be written as pure functions
- You should keep data logic and UI logic in separate components
- Everything in React can always be boiled down to functions and props
- Believe it or not, we've just covered 80% of the knowledge used by a React developer on a daily basis.

React passing data through props



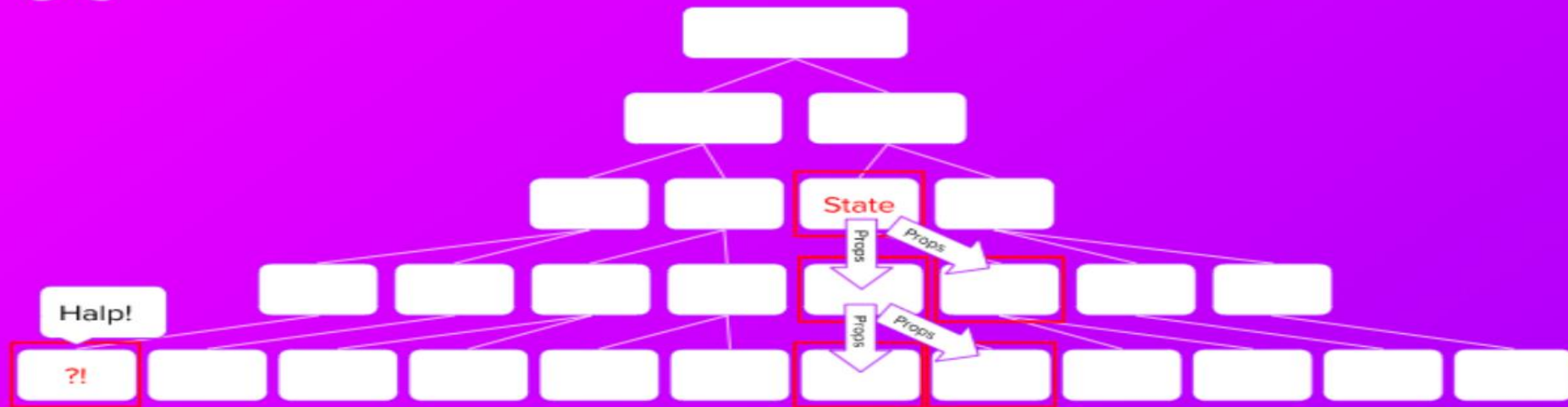
If we later find out that the sibling of the component with state also needs the data, we have to lift state up again, and pass it back down:



problems begin if a component on a different branch needs the data

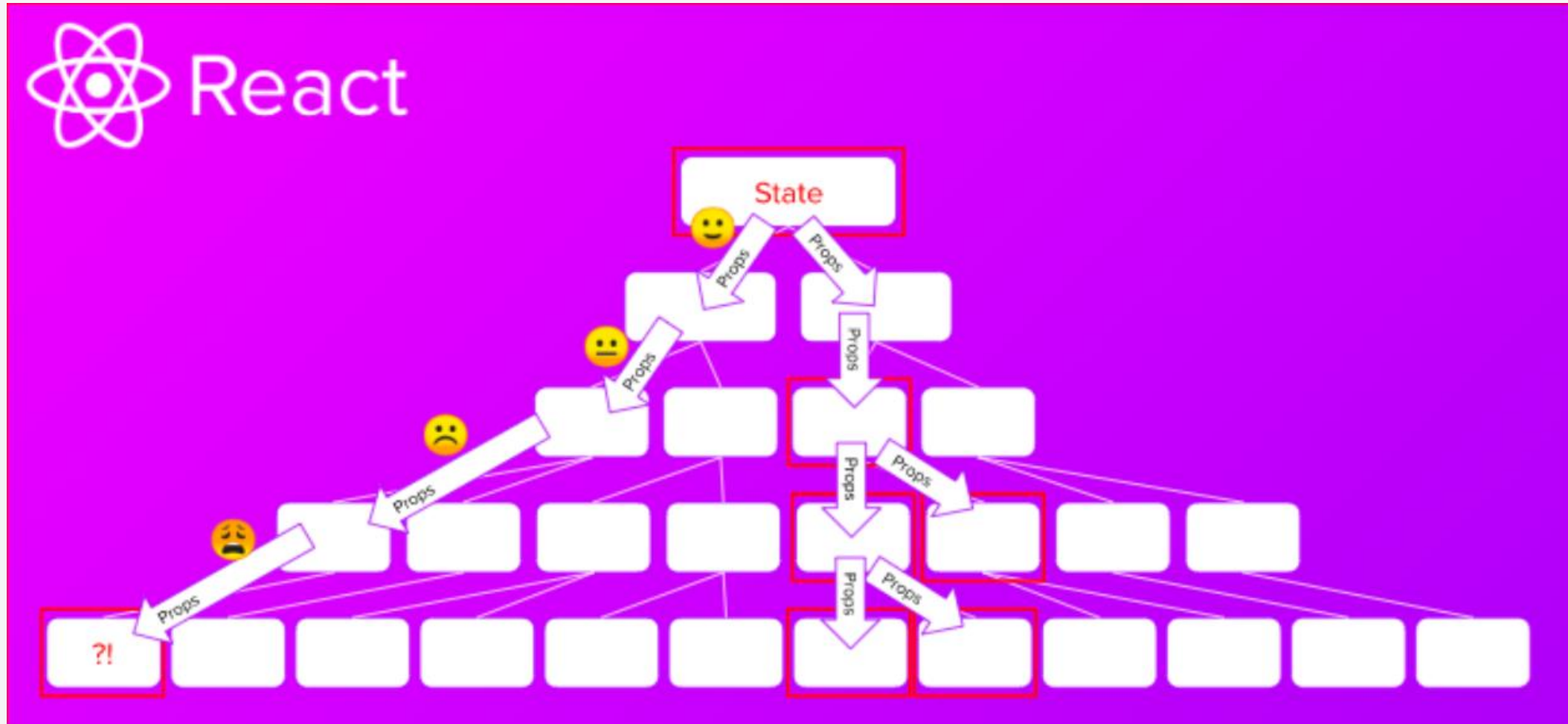


React



we need to pass state from the top level of the application through all the intermediary components to the one which needs the data at the bottom, even though the intermediary levels don't need it. This tedious and time-consuming process is known as *prop drilling*.

React props drilling





Redux

Redux three principles :

- **Single source of truth**
- **State is read-only**
- **Changes are made with pure functions**

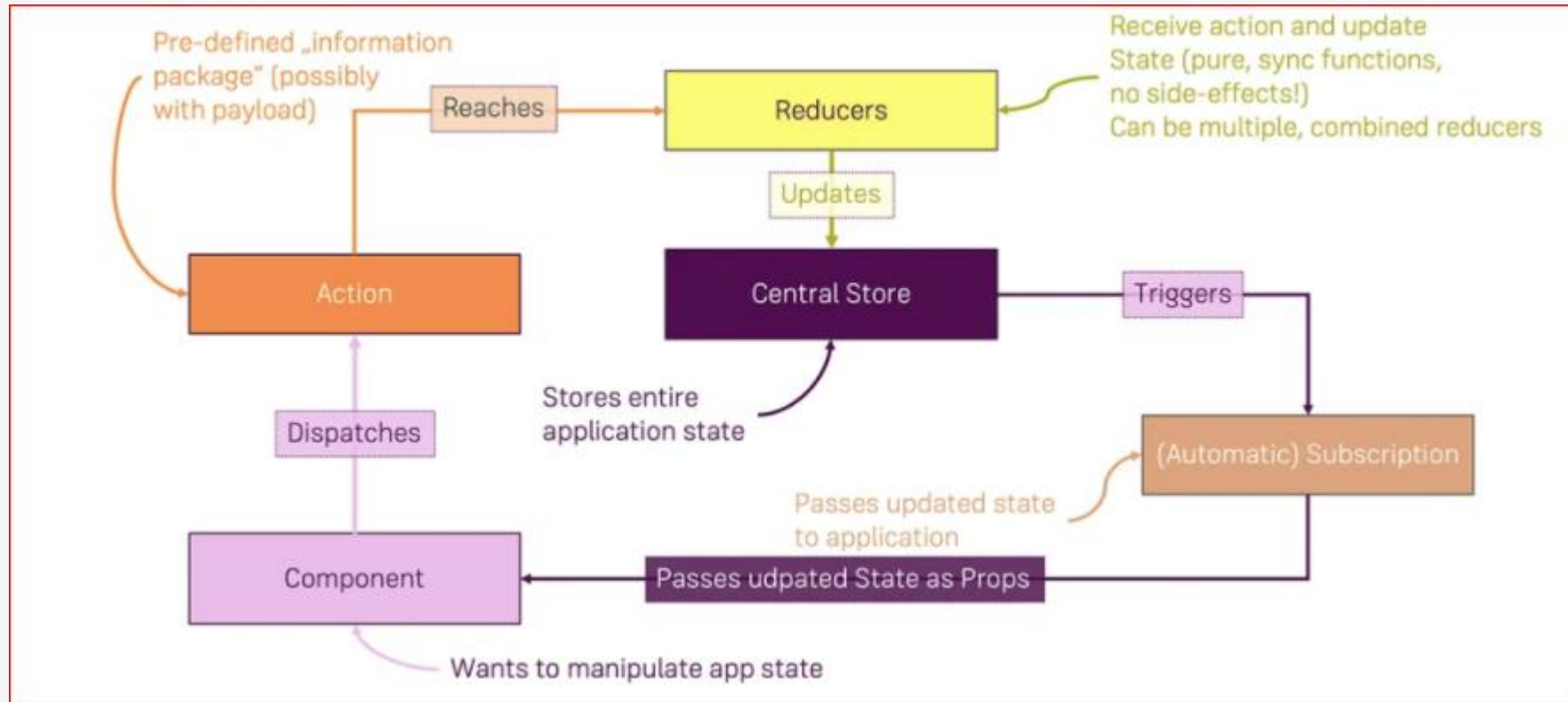
Redux

- **Redux** is an [open-source JavaScript library](#) for managing application [state](#) **centrally**.
- It operates on Higher Order Components,, a [functional programming](#) concepts & Observer Pattern, Composite pattern.
- Redux is a small standalone JS library
- Redux is a pattern and library for managing and updating application state, using events called "**actions**".
- Redux serves as a centralized **store** for state
- manage "global" state

Redux scenarios

- You have large amounts of application state that are needed in many places in the app
- · The app state is updated frequently over time
- · The logic to update that state may be complex
- · The app has a medium or large-sized codebase, and might be worked on by many people
- · Action: code that causes an update to the state when something happens.
- · The actions, the events that occur in the app based on user input, and trigger updates in the state
- · State describes the condition of the app at a specific point in time
- · The UI is rendered based on that state
- · When something happens (such as a user clicking a button), the state is updated based on what occurred

Redux Artifacts



Artifacts

- **Action:** An action is a JavaScript object describing an event in your application. They're typically generated by either a user interaction.
- **Stores:** Store manages the state of one domain within an application. state should always either return a new object state or should return the same object state. (Immutable)
- **Reducer:** A reducer is a function that determines changes to an application's state. It uses the action it receives to determine this change. ... Redux relies heavily on reducer functions that take the previous state and an action in order to execute the next state.
- **Provider**—A React component that you'll render at the top of the React app. Any components rendered as children of Provider can be granted access to the Redux store.

Redux prerequisites concepts

- Higher Order Components(HOC)
- React Memo
- How to Set Dynamic Property Keys
- Object.keys()
- reduce() – JavaScript app
- Functional programming

Redux Cricket Application setup

1) Create react app:

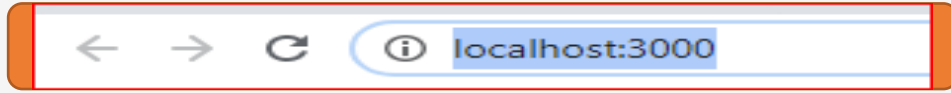
> npx create-react-app cricket-redux --template typescript

2) Add styling libraries sass is required for bootstrap:

> yarn add node-sass reactstrap @types/reactstrap bootstrap

3) Add redux libraries:

> yarn add redux react-redux @types/react-redux

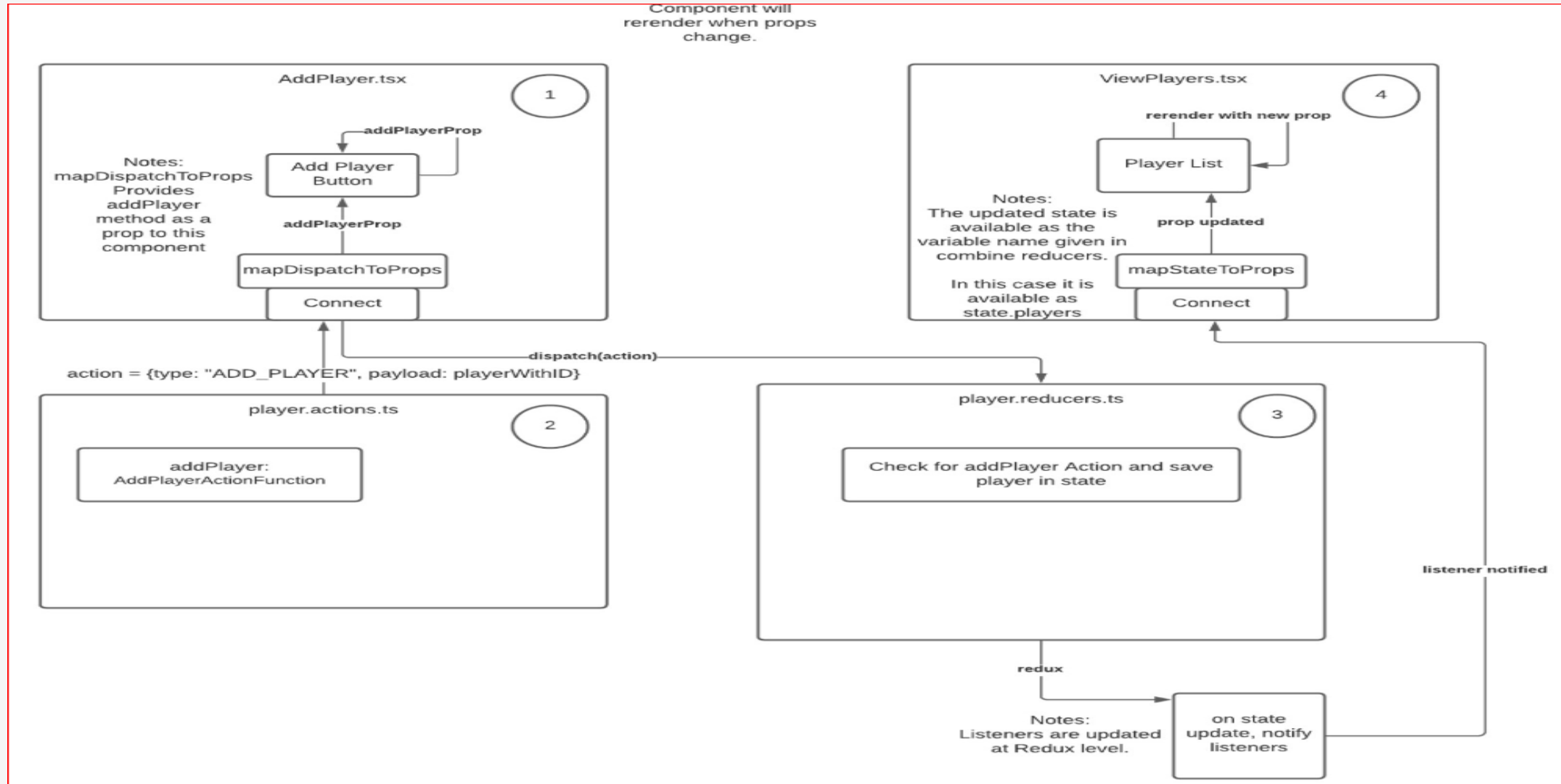


Add New Player	View Players
<div>Name</div> <input type="text"/>	
<div>Type</div> <input type="text"/>	
<div>Image</div> <input type="text"/>	
<div>Country</div> <div></div>	
<div>Reset</div>	
<div>Add Player</div>	

Directory structure

```
src
├── components
│   ├── add-player
│   │   └── index.tsx
│   └── view-players
│       └── index.tsx
├── pages
│   └── App.tsx
├── styles
│   └── index.scss
├── store
│   ├── reducers
│   │   ├── index.ts
│   │   └── player.reducer.ts
│   ├── actions
│   │   ├── index.ts
│   │   └── player.actions.ts
│   ├── action-types.ts
│   └── index.ts
└── types.ts
```

Redux Cricket Application Flow



connect() function

- **connect()**: is a Higher Order Component.
- It provides its connected component with the pieces of the data it needs from the store, and the functions it can use to dispatch actions to the store.
- It does not modify the component class passed to it; instead, it returns a new, connected component class that wraps the component you passed in.
- **function connect(mapStateToProps?, mapDispatchToProps?, mergeProps?, options?)**

connect() function parameter types

- `mapStateToProps?: Function`
- `mapDispatchToProps?: Function | Object`
- `mergeProps?: Function`
- `options?: Object`

mapStateToProps()

- If a **mapStateToProps** function is specified, the new wrapper component will subscribe to Redux store updates.
- Any time the store is updated, mapStateToProps will be called
- The results of **mapStateToProps** must be a plain object, which will be merged into the wrapped component's props
- If you don't want to subscribe to store updates, pass null or undefined in place of mapStateToProps.

mapDispatchToProps()

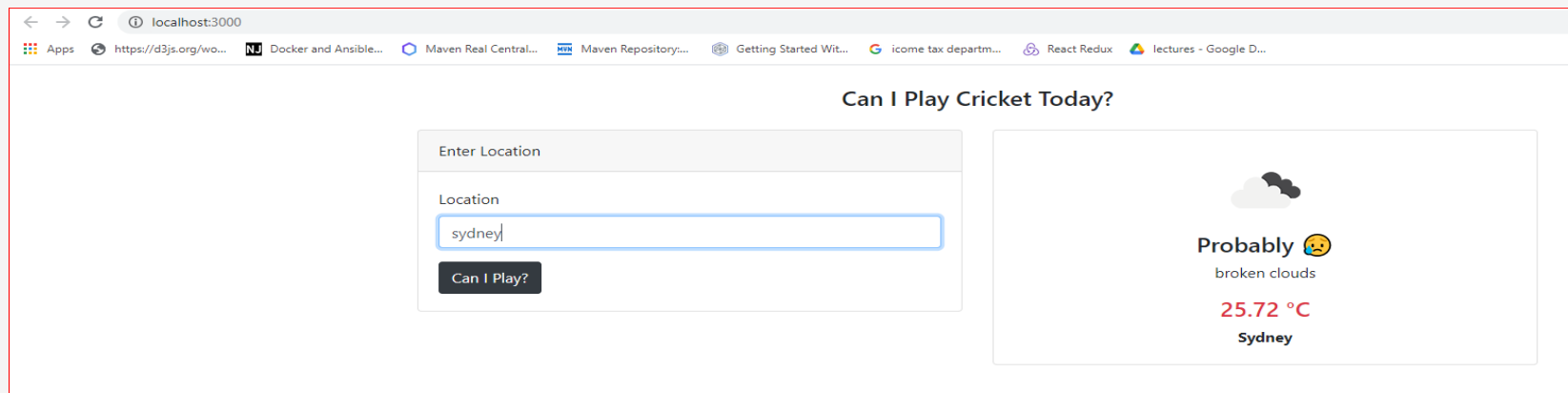
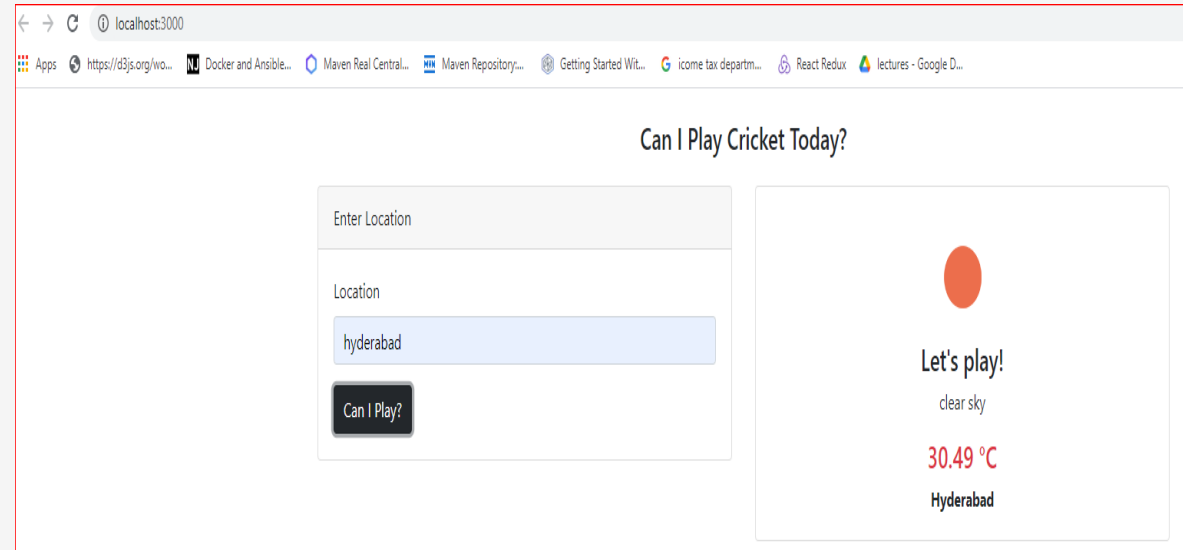
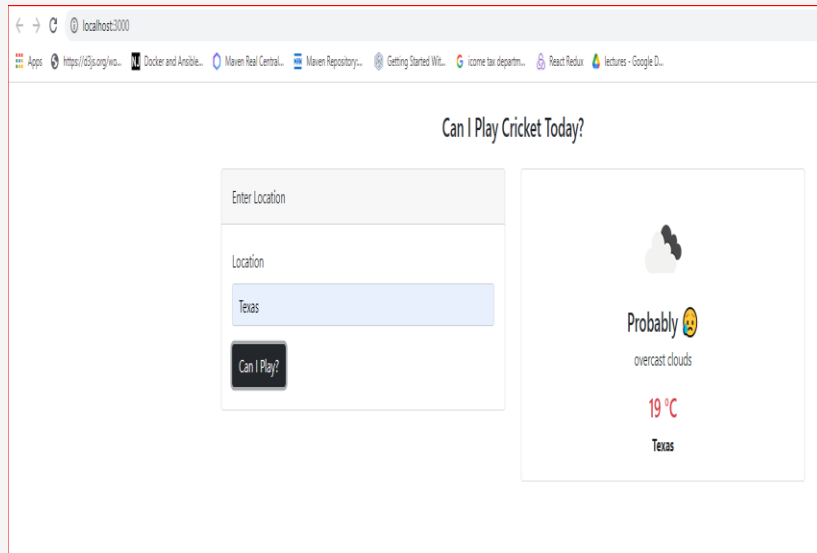
- **mapDispatchToProps**, this second parameter to connect()
- May either be an object, a function, or not supplied.
- Your component will receive dispatch by default, i.e., when you do not supply a second parameter to connect()
- **mapDispatchToProps** argument is responsible for **enabling** a component to **dispatch** actions

THUNKS
IN REDUX

Redux Thunk

- **Redux Thunk is** middleware that allows you to return functions, rather than just actions, within **Redux**.
- Thunk allows for delayed actions, including working with promises.
- One of the main **use** cases for this middleware **is** for handling actions that might not be synchronous.
- **Thunks** are a functional programming technique used to delay computation.

Thunk Weather app

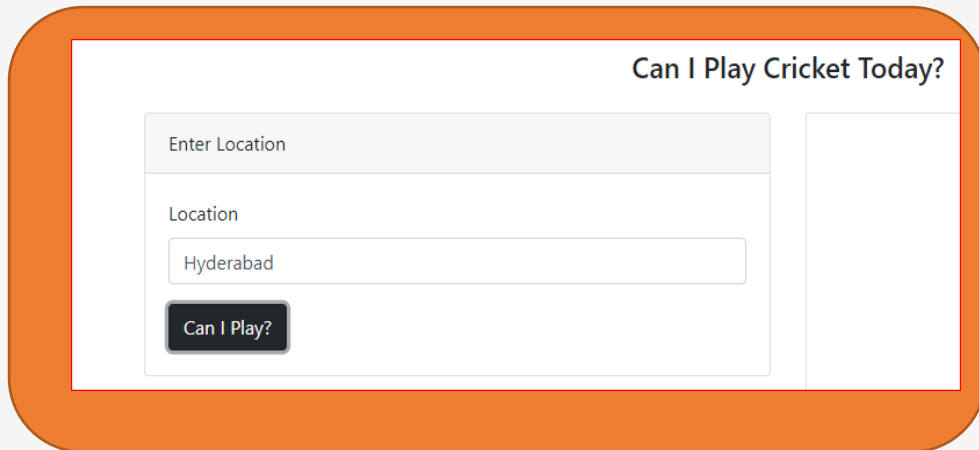


Thunk Weather App

➤ Create 2 components: WeatherInput, Verdict.

➤ <https://openweathermap.org/> : create account and sign in first

WeatherInput Component

The WeatherInput component is a mobile app interface with an orange border. It features a title "Can I Play Cricket Today?" at the top right. Below the title is a form with a header "Enter Location" and a sub-header "Location". A text input field contains the word "Hyderabad". At the bottom left of the form is a dark button labeled "Can I Play?".

Can I Play Cricket Today?

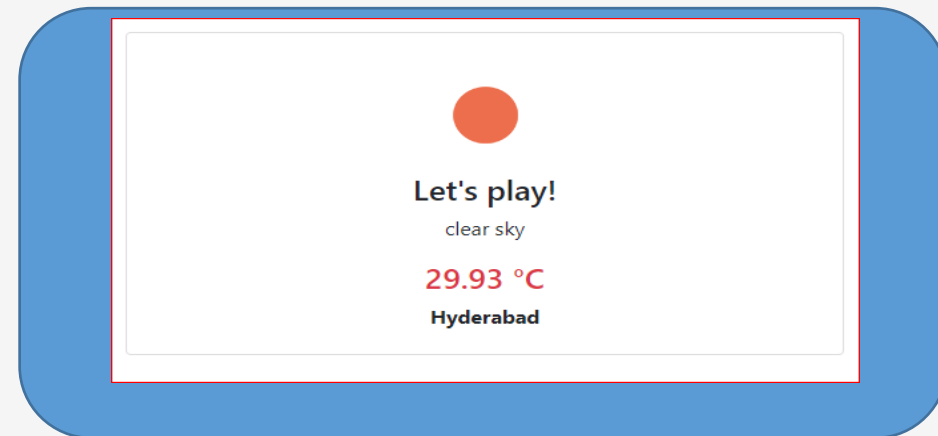
Enter Location

Location

Hyderabad

Can I Play?

Verdict Component

The Verdict component is a mobile app interface with a blue border. It displays a red circle at the top, followed by the text "Let's play!", "clear sky", "29.93 °C", and "Hyderabad".

Let's play!

clear sky

29.93 °C

Hyderabad