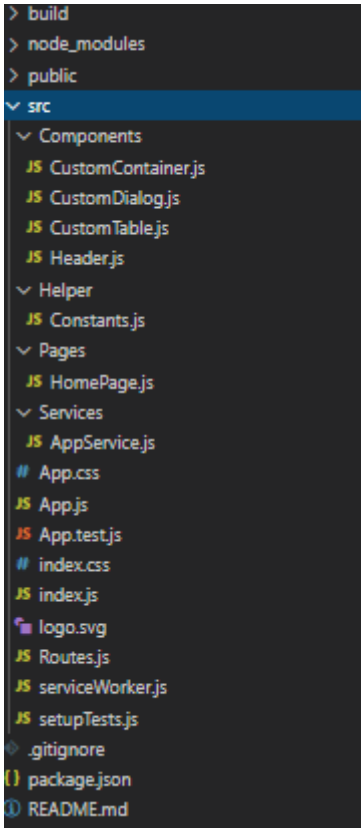
Table of Content

1. Best Practice #1: Project Structure

While creating a react project, the first step is to define a project structure that is scalable. You can create a new base react application structure by using the npm command ‘create-react-app’. The below screenshot displays the basic react app folder structure.



React folder structure may differ based on project specification and complexity. There are various react js best practices that can be taken into account while defining project architecture:

1. Folder Layout

The architecture focuses on reusable components of the react developer architecture so **that the design pattern** can be shared among multiple internal projects. Hence the idea of component-centric file structure should be used which implies that all the files related to a different component (like test, CSS, JavaScript, assets, etc.) should be kept under a single folder.

```
Components
        |
        --Login
                |
                --tests--
                --Login.test.js
                --Login.jsx
                --Login.scss
                --LoginAPI.js
```

folder_layout_1.txt hosted with ❤ by GitHub view raw

This is another approach used in grouping the file types. In this, the same type of files is kept under one folder. For example

```
APIs
  |
  --LoginAPI
  --ProfileAPI
  --UserAPI

Components
  |
  --Login.jsx
  --Login.test.js
  --Profile.jsx
  --Profile.test.js
  --User.jsx
```

folder_layout_2.txt hosted with ❤ by GitHub view raw

The above structure is the basic example. The folders can be further nested based on requirements.

2. CSS in JS

In a large project, styling and theming can be a challenging task like maintaining those big scss files. So, the concept of CSS-in-JS solutions (i.e. put CSS in JavaScript) came into the picture. Following libraries are based on this concept.

- EmotionJS
- Styled Components
- **Glamorous**

Among these libraries, you can use based on the requirement like for complicated themes, you can choose styled-components or Glamorous.

3. Children Props

Sometimes it is required to render method the content of one component inside another component. So we can pass functions as children props which get called components render function.

4. Higher-Order Components (HOC)

It's an advanced technique in React which allows reusing component logic inside the render method. An advanced level of the component can be used to transform a component into a higher order of the component. For example, we might need to show some components when the user is logged in. To check this, you need to add the same code with each component. Here comes the use of the Higher-Order Component where the logic to check the user is logged in and keep your code under one app component. While the other components are wrapped inside this.

2. Best Practice #2: React Component

Its components are the building blocks of a react project. Here are some of the React best practices that can be considered while coding with React in the component state and component hierarchy.

1. Decompose into Small Components

Try to decompose large components into small components such that component performs one function as much as possible. It becomes easier to manage, test, reuse and create a new small components. This makes sense right?

2. Use Functional or Class Components based on Requirement

If you need to show User Interface without performing any logic or state change, use functional components in place of class components as functional

components are more efficient in this case.

For instance:

```
// class component
class Cat extends React.Component {
  render () {
        let { badOrGood, type, color } = this.props;
        return <div classname="{type}">My {color} cat is { badOrGood } </div>;
  }
}

//function component
let Cat = (badOrGood, type, color) => <div classname="{type}">My {color} cat is { badOrGood }<
```

class_and_function_components.js hosted with ❤ by GitHub view raw

1. Try to minimize logic in React lifecycle methods like componentDidMount(), componentDidUpdate() etc. cannot be used with functional components, but can be used with Class components.
2. While using functional components, you lose control over the render process. It means with a small change in component, the functional component always re-renders.

3. Use Functional Components with Hooks

After the release of React v16.08, it's possible to develop **function components with the state** with the new feature **'React Hooks'**. It reduces the complexity of managing states in Class components. So always prefer to use functional components with React Hooks like useEffect(), useState() etc. This will allow you to repeatedly use facts and logic without much modification in the hierarchical cycle.

4. Appropriate Naming and Destructuring Props

To keep readable and clean code, use meaningful and short names for props of the component. Also, use props destructuring feature of function which discards the need to write props with each property name and can be used as it is.

```
const funcDestruct = ({name, title}) => {
return (
<div>
<p>{name} - {title}</p>
</div>
)
}
```

destructuring_props.js hosted with ❤ by GitHub view raw

Herewith props destructuring, we can directly use name and title without using props.name or props.title.

5. Use propTypes for Type Checking and Preventing Errors

It is a good practice to do type checking for props passed to a component which can help in preventing bugs. Please refer below code for how to use

React.PropTypes:

```
import React, { Component } from “react”;
import PropTypes from “prop-types”;
class PropTypeExample extends Component {
 render() {
 const { username } = this.props;
 return
Welcome, { username }
 }
}
PropTypeExample.PropTypes = {
 name: PropTypes.string.isRequired
};
```

react_prototype.js hosted with ❤ by GitHub view raw

3. Best Practice #3: Code Style

1. Naming Conventions

A component name should always be in a Pascal case like 'SelectButton', 'Dashboard' etc. Using Pascal case for components differentiate it from default JSX element tags.

Methods/functions defined inside components should be in Camel case like 'getApplicationData()', 'showText()' etc.

For globally used Constant fields in the application, try to use capital letters only. Like const PI = "3.14";

2. Avoid the Use of the State as much as Possible

Whenever using state in the component, keep it centralized to that component and pass it down in the component tree as props.

3. Write DRY Code

Try to avoid duplicate code and create a common component to perform the repetitive task to maintain the DRY (Don't Repeat Yourself) code structure.

For instance: When you need to show multiple buttons on a screen then you can create a common button component and use it rather than writing markup for each button.

4. Try to Avoid Unnecessary Div

When there is a single component to be returned, there is no need to use .

```
return (
<div>
<Button>Close</Button>
```

```
</div>
);
```

unnecessary_div_1.js hosted with ❤ by GitHub view raw

When there are multiple components to be returned, use or in shorthand form <> as shown below:

```
return (
 <Button>Close</Button>
);
```

unnecessary_div_2.js hosted with ❤ by GitHub view raw

5. Remove Unnecessary Comments from the Code

Add comments only where it's required so that you do not get confused while changing code at a later time.

Also don't forget to remove statements like Console.log, debugger, unused commented code.

6. Use Destructuring to Get Props

Destructuring was introduced in ES6. This type of feature in the javascript function allows you to easily extract the form data and assign your variables from the object or array. Also, destructuring props make code cleaner and easier to read.

For example:

Example 1:

There is an objecting employee.

```
const employee= {
    firstName: "Linda",
    lastName: "Cris",
    city: "NY"
 }
```

destructuring_ex1_1.js hosted with ❤ by GitHub view raw

To access properties of object, you need to write:

```
const firstName = employee.firstName
const lastName = employee.lastName
const city = employee.city
```

destructuring_ex1_2.js hosted with ❤ by GitHub view raw

Which can be written as following with destructuring:

```
const { firstName, lastName, city } = employee;
```

destructuring_ex1_3.js hosted with ❤ by GitHub view raw

Example 2:

Let's take another example. Take an example of a cat that we want to display as a div by naming a class and its type. In between the div, we can see a statement which will tell the cat's color, its nature-good or bad, etc.

```
class Cat extends Component {
  render () {
        let { type, color, badOrGood } = this.props;
        return <div className={type}>My {color} cat is { badOrGood }</div>;
        }
}
```

destructuring_ex2_1.js hosted with ❤ by GitHub view raw

To maintain clarity with the codes, we can put all the ternary operators in its own variable and see the change.

```
class Cat extends Component {
  render () {
  let { type, color, isGoodCat } = this.props;
  let identifier = isGoodCat? "good" : "bad";
  return <div className={type}>My {color} cat is {identifier}</div>;
  }
}
```

destructuring_ex2_2.js hosted with ❤ by GitHub view raw

7. Apply ES6 Spread Function

It would be a more easy and productive way to use ES6 functions to pass an object property. Using {...props} between the open and close tag will automatically insert all the props of the object.

```
let propertiesList = {
  className: "my-favorite-props ",
  id: "myFav",
  content: "Hello my favourite!"
};
let SmallDiv = props => <div {... props} />;
let mainDiv = < SmallDiv props={propertiesList} />;
```

es6_spread_function.js hosted with ❤ by GitHub view raw

You can use the spread function:

There are no ternary operators required

There is no need to pass only HTML tag attributes and content

In case of repetitive use of functions, Don't use the spread function when:

There are dynamic properties

There is a need for array or object properties

In the case of render where nested tags are required

8. The Rule of 3

When there are three or fewer properties, then you should keep those properties in their line inside both the component and the render function.

For example: It would be fine in the code below to write one line to get properties.

```
class Gallery extends Component {
  render () {
  let { image, title } = this.props;
  return (
    <figure>
      <img src={image} alt={title} />
      <figcaption>
        <p>Title: {title}</p>
      </figcaption>
    </figure>
  );
  }
}
```

ruleof3_1.js hosted with ❤ by GitHub view raw

But, find the below code where more than 3 props are written in single line:

```
class Gallery extends Component {
  render () {
  let { image, title, artist, clas, thumbnail, breakpoint } = this.props;
  return (
    <figure className={clas}>
      <picture>
        <source media={`(min-width: ${breakpoint})`} srcset={image} />
        <img src={thumbnail} alt={title} />
      </picture>
      <figcaption>
        <p>Title: {title}</p>
        <p>Artist: {artist}</p>
      </figcaption>
    </figure>
  );
  }
}
```

ruleof3_2.js hosted with ❤ by GitHub view raw

And in render

```
<Gallery image="./src/img/image2.jpg" title="Scary Night" artist="Vani Garg" class="portrait" t
```

ruleof3_3.js hosted with ❤ by GitHub view raw

The above code becomes unreadable and clumsy. So when there are more than 3 props, write each one in a new line as below :

```
let { image,
  title,
  artist,
  clas,
  thumbnail,
  breakpoint } = this.props;
```

ruleof3_4.js hosted with ❤ by GitHub view raw

And in render

```
<Gallery
image="./src/img/image2.jpg"
  title="Scary Night"
  artist="Vani Garg"
  clas="landscape"
  thumbnail="./src/img/thumb/night.gif"
  breakpoint={320} />
```

ruleof3_5.js hosted with ❤ by GitHub view raw

9. Manage too many Props with Parent/Child Component

It's a tricky task to manage properties at any level in components, but with the help of React's state and ES6 destructuring feature, props can be written in a better way as shown below.

For example:
Let's create an application having a list of saved addresses and GPS coordinates of the current location.

The current user's location should be added in the favorite address and can be kept in parent component App section as shown below:

```
class App extends Component {
```

```
2    constructor (props) {
3    super(props);
4    this.state = {
5      currentUserLat: 0,
6      currentUserLon: 0,
7      isCloseToFavoriteAddress: false
8    };
9    }
10 }
```

parent_child_1.js hosted with ❤ by GitHub view raw

Now, to get data on how close current users are to the favorite address, we will pass at least two props from the App

In render() method of App:

```
1 <FavAddress
2   ... // Information about the address
3   addCurrentLat={this.state.currentUserLat}
4   addCurrentLong={this.state.currentUserLon} />
```

parent_child_2.js hosted with ❤ by GitHub view raw

In the render() for FavAddress Component:

```
1  render () {
2  let { addHouseNumber,
3      addStreetName,
4      addStreetDirection,
5      addCity,
6      addState,
7      addZip,
8      addLat,
9      addLon,
10     addCurrentLat,
11     addCurrentLon } = this.props;
12 return ( ... );
13 }
```

parent_child_3.js hosted with ❤ by GitHub view raw

As you can see in the above infographic, it's getting unwieldy. It is more feasible to keep multiple sets of options and separate them within their own internal

objects.

So, in App constructor:

```
this.state = {
 currentUserPos: {
   lat:0,
   lon:0,
},
 isCloseToFavoriteAddress: false,
}
```

parent_child_4.js hosted with ❤ by GitHub view raw

At a point before App render():

```
let addressList = [];
addressList.push({
 addHouseNumber: "12344",
 addStreetName: "Street Road",
 addStreetDirection: "N",
 addCity: "My City",
 addState: "ST",
 addZip: "12346",
 addLat: "019782356834",
 addLon: "02384575757"
});
```

parent_child_5.js hosted with ❤ by GitHub view raw

In App render():

```
<FavAddress
addressInfo={addressList[0]}
curretUserPos={this.state.currentUserPos}
/>
```

parent_child_6.js hosted with ❤ by GitHub view raw

For the FavAddress Component, inside the render function we can see:

```
render () {
let { addressInfo, currentUserPos } = this.props;
```

```
let { addHouseNumber,
    addStreetName,
    addStreetDirection,
    addCity,
    addState,
    addZip,
    addLat,
    addLon } = addressInfo;
return ( ... );
}
```

parent_child_7.js hosted with ❤ by GitHub view raw

10. Use Map Function for Dynamic Rendering of Arrays

In react, it is possible to create an object with props that return a dynamic HTML block without writing repeated code. For this, react provides a map() function to display arrays in order. While using an array with a map(), one parameter from the array can be used as a key.

```
render () {
  let cartoons = [ "Pika", "Squi", "Bulb", "Char" ];
  return (
  <ul>
    {cartoons.map(name => <li key={name}>{name}</li>)}
  </ul>
  );
}
```

map_function_1.js hosted with ❤ by GitHub view raw

Apart from this, ES6 spread functions can be used to send a whole list of parameters in an object by using Object.keys().

```
render () {
  let cartoons = {
  "Pika": {
    type: "Electric",
    level: 10
  },
  "Squi": {
    type: "Water",
    level: 10
```

```
  },
  "Bulb": {
    type: "Grass",
    level: 10
  },
  "Char": {
    type: "Fire",
    level: 10
  }
  };
  return (
  <ul>
    {Object.keys(cartoons).map(name => <Cartoons key={name} {... cartoon[name]} />)}
  </ul>
  );
}
```

map_function_2.js hosted with ❤ by GitHub view raw

Another example of mapping array is as follow:

```
import React, { Component } from "react";
class Item extends Component {
  state = {
  listitems: [
    {
      id: 0,
      context: "Primary",
      modifier: "list-group-item list-group-item-primary"
    },
    {
      id: 1,
      context: "Secondary",
      modifier: "list-group-item list-group-item-secondary"
    },
    {
      id: 2,
      context: "Success",
      modifier: "list-group-item list-group-item-success"
    },
    {
      id: 3,
      context: "Danger",
      modifier: "list-group-item list-group-item-danger"
    },
```

```
    {
      id: 4,
      context: "Warning",
      modifier: "list-group-item list-group-item-warning"
    }
  ]
  };

  render() {
  return (
    <React.Fragment>
      <ul className="list-group">
        {this.state.listitems.map(listitem => (
          <li key={listitem.id} className={listitem.modifier}>
            {listitem.context}
          </li>
        ))}
      </ul>
    </React.Fragment>
  );
  }
}
export default Item;
```

map_function_3.js hosted with ❤ by GitHub view raw

11. Dynamic Rendering with && and the Ternary Operator

In React, it is possible to perform conditional renderings the same as a variable declaration. For small code with conditions, it's easy to use ternary operators but with large code blocks, it becomes difficult to find those ternary operators. So the code can be written as below too:

```
class FilterResult extends Component {
  render () {
  let { filterResults } = this.props;
  return (
    <section className="search-results">
      { filterResults.length > 0 &&
        filterResults.map(index => <Result key={index} {... results[index] />)
      }
      { filterResults.length === 0 &&
```

```
        <div className="no-results">No results</div>
      }
    </section>
  );
  }
}
```

dynamic_rendering_1.js hosted with ❤ by GitHub view raw

The above way of conditional rendering will be useful when there are more than 2 conditions or we need to render some code on a specific condition and there is no else part. In those cases, you can use && operators with the condition.

So the above code can be written in true ternary fashion:

```
class FilterResult extends Component {
  render () {
  let { filterResults } = this.props;
  return (
    <section className="search-results">
      { filterResults.length > 0 &&
        filterResults.map(index => <Result key={index} {... results[index] />)
      }
      { filterResults.length === 0 &&
        <div className="no-results">No results</div>
      }
    </section>
  );
  }
}
```

dynamic_rendering_2.js hosted with ❤ by GitHub view raw

Even though the above code is well organized, it could have become messy and unreadable if the render function had more than just one line as there would be more nested brackets.

```
class FilterResult extends Component {
  render () {
  let { filterResults } = this.props;
  return (
    <section className="search-result">
      { filterResults.length > 0
```

```
        ? filterResults.map(index => <Result key={index} {... filterResults[index] />)
        : <div className="no-result">No results</div>
      }
    </section>
  );
  }
}
```

dynamic_rendering_3.js hosted with ❤ by GitHub view raw

As you can see, in both cases code length is the same but there is one main difference, in the first example, there is rapid switching between two different syntaxes making visual parsing difficult as compared to the second, which is simple JavaScript code with variable assignments and one line return function. It can be inferred from the above code that if the JavaScript is kept inside a JSX object is more than two words (e.g. object. property), then keep that code before the return call.

12. Use es-lint or Prettier for Formatting

Follow the es-lint rules while writing code and use line breaks wherever required for a clean and formatted code. You can also use prettier for formatting the code.

13. Write Tests for Each Component

It is a good practice to write test cases for each component developed as it reduces the chances of getting errors when code is deployed. With the unit testing, you can check all the possible scenarios. Jest or enzymes are the most commonly used react test frameworks.

4. Best Practice #4: React JS Security

1. Add Security to HTTP Authentication

There are multiple applications where authentication is done on user login or account creation and this process should be secure as the client-side authentication and authorization can be exposed to many security defects that may destroy these protocols in the application.

The commonly used technique for adding authenticity can be validated using.

1. JSON Web Token (JWT)
2. OAuth
3. AuthO
4. React Router
5. PassportJs

Security with JWT

There are some points that to be taken into account while using JWT:

1. Please avoid keeping JWT tokens based on Local Storage. As it would be very easy for someone to get a token using the browser's Dev tools console and write.
   console.log(localStorage.getItem('token'))
2. Store your tokens to an HTTP cookie rather than localStorage.
3. Or, you can keep your tokens to your React app's state.
4. Tokens should be kept in the backend. It would be easy to sign and verify these keys at the backend side.
5. You should use long and unpredictable secrets similar to the passwords field while creating an account asking for a strong and long password.
6. Always make sure you use HTTPS in place of HTTP. This will give assurance for your web-based app to provide a valid certificate that can be sent over a secure SSL network.

2. Secure Against Broken Authentication

Sometimes when you enter authentication details, and the application crashes which might lead to exploitation of user credentials. So to remove this kind of vulnerability, make sure you follow the measures mentioned below.

1. Do use multi-factor and 2-step authorization.
2. You can use cloud-based authentication (for instance Cognito) for secure access.

3. Broken Access Control

With the improper management of restrictions and limitations on authenticated users can cause exploitation of unauthorized data and functionality of a React native app. Sometimes unauthorized users can also change the primary key of data and manipulate the functionality of the application. To ensure security from unauthorized access, follow these practices:

1. Add a role-based authentication mechanism to your react code
2. To secure your application, deny functionality access

4. Cross-Site Scripting (XSS)

1. You can create automated overseeing features that can sanitize the user input
2. Discard malicious and invalid user input from being rendered into the browser.

5. Secure Against DDoS Attacks

The vulnerable security concerns take place when the whole application state management has loopholes and it masks the IPs. This will restrict the

communication caused due to the termination of services. Here are some methods to stop this:

1. Limitation of rate on APIs- This will add limitations to the number of requests for a given IP from a specific source with a complete set of libraries using the Axios-rate limit.
2. Add app-level restrictions to the API.

6. SQL Injection

This attack is related to data manipulation. Due to this vulnerability, attackers can modify any data with or without the user's permission or can extract any confidential data by executing arbitrary SQL code.

Solution

1. To eliminate SQL injection attacks, first, validate API call functions against the respective API schemas. In order to handle the issue of time-based SQL injection attacks, you can use timely validation of the schema to avoid any suspicious code injections
2. Another effective way to secure against the SQL vulnerability is by using an SSL Certificate.

7. Using dangerouslySetInnerHTML

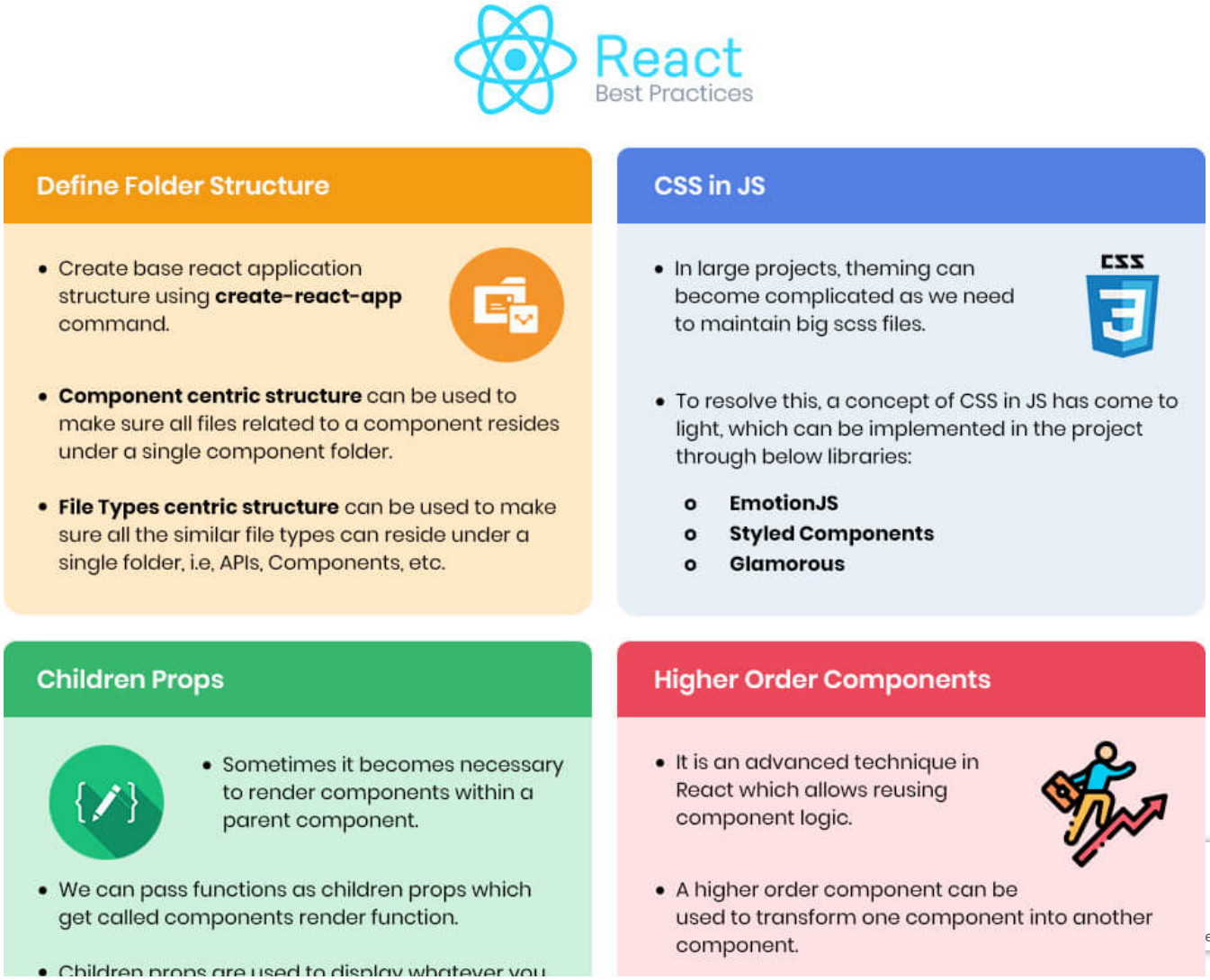
In React, you can use 'innerHTML' for an element inside DOM which is a risky practice as it's a wide-open gate for XSS attack. So to remove this issue, React has provided a "dangerouslySetInnerHTML" prop to safeguard against this type of attack.

Also, you can use libraries such as DOMPurify in order to sanitize user input and remove any malicious inputs. React already has an inbuilt function called dependency injection for properly managing user interfaces.

5. Conclusion

With this blog, we gave you a deeper insight into both reactjs developer and frontend developer on how ReactJS works, how to add security, how to build components and applications. The React best practices will offer you fewer typing options and more explicit codes. Once you will start using this, you will start liking its clear crisp features with code reusability, advanced react components, adding a separate state variable, and other smaller ready-made React.js features for simplified use. This list of best practices from React will help you place your ventures on the right path, and later down the line, eliminate any future development complications.

We have presented an info-graphical representation of React Best Practices. Take a look:



include between the opening and closing tags when you invoke a component.

- Concretely, a higher-order component is a function that takes a component and returns a new component.

Decompose into Small Components

- Try to decompose large components to small components such that each component performs one function as much as possible.

- It becomes easier to manage, test and reuse small components.

Use of Functional & Class Components

- If you need to show UI without performing any logic or state change, use functional components in place of class components as functional components are more efficient in this case.

```
// Class Component
class Cat extends React.Component {
  render () {
    let { badOrGood, type, color } = this.props;
    return (
      <div className={type}>
          My {color} cat is { badOrGood }
      </div>);
  }
}

// Function Component
let Cat = (badOrGood, type, color) =>
  <div className={type}>
        My {color} cat is { badOrGood }
  </div>;
```

- React lifecycle methods like componentDidMount(), componentDidUpdate() etc. cannot be used with functional components, but can be used with Class components.
- While using functional components, you lose control over the render process. It means with the small change in component, the functional component always re-renders.

Functional Components with Hooks

- After release of React v16.08, it is possible to develop function components with state with the new feature 'React Hooks'.
- It reduces complexity of managing states in Class components. So always prefer to use functional components with React Hooks like useEffect(), useState() etc.

User PropTypes for Checking & Preventing Errors

- It is a good practice to do type checking for props passed to a component which can help in preventing bugs.
- Please refer the code for how to use React.PropTypes:

```
import React, { Component } from "react";
import PropTypes from "prop-types";
class PropTypeExample extends Component {
   render() {
     const { username } = this.props;
     return <h1>Welcome, { username }</h1>;
   }
}
PropTypeExample.PropTypes = {
   name: PropTypes.string.isRequired
};
```

Naming Conventions

- A component name should always be in Pascal case like 'SelectButton', 'Dashboard' etc.Using Pascal case for components differentiate it from default JSX element tags.
- Methods/functions defined inside components should be in Camelcase like 'getApplicationData()', 'showText()' etc.

Apply ES6 Spread function

- It would be a more easy and productive way to use ES6 functions to pass an object property. Using {...props} between the open and close tag, will automatically insert all the props of the object.

- For globally used Constant fields in application, try to use capital letters only. Like const PI = "3.14";

Use Destructuring to get props

- Destructuring was introduced in ES6.
- This javascript feature allows you to extract multiple parts of data from an object or array and assign them to their own variables.
- Destructuring props makes code cleaner and easier to read.

Use when :

- No ternary operators required
- Need to pass only HTML tag attributes and content
- For repetitive use of functions

Avoid when :

- There are dynamic properties.
- Need of array or object properties.
- In case of render where nested tags are required.

Use Linting Packages

ESLint

- ESLint is one the most popular linting package used to check possible errors in code and code styles to meet best practices standards.
- Follow the ESLint rules while writing code and use line breaks wherever required for a clean and formatted code. You can also use prettier for formatting the code.

Don't Repeat Yourself

DRY

- Try to avoid duplicate code and create a common component to perform the repetitive task to maintain **DRY (Don't Repeat Yourself)** code structure.
- For example: When you need to show multiple buttons on a screen then you can create a common button component and use it rather than writing markup for each button.

Dynamic Rendering

- In react, it is possible to create an object with props that return a dynamic HTML block without writing repeated code.
- For this, react provides **map()** function to display arrays in order. While using array with **map()**, one parameter from array can be used as a key.

```
render () {
 let cartoons =
    [ "Pika", "Squi", "Bulb", "Char" ];
 return (
  <ul>
    {cartoons.map
        (name => <li key={name}>{name}
            </li>)}
  </ul>
 );
}
```

React JS Application Security

Secure HTTP Auth

Use methods like **JWT, OAuth, AuthO,** React Router, PassportJS, etc. to ensure robust HTTP Authentication.

No Broken Authentication

Use **multi-factor and 2-factor authentication** as well as **cloud based authentication** to make sure all vulnerabilities are negated.

Access Control

Add User Role-based authorization methods to control access to the private data.

Cross Site Scripting (XSS)

Prevent DDoS Attacks

SQL Injection Prevention