# React.js

## TABLE OF CONTENTS

## PREFACE

Welcome to the dynamic world of React.js, a cutting-edge JavaScript library that has revolutionized the landscape of front-end development. In this cheatsheet, we embark on a journey into the heart of React.js, exploring its core principles, capabilities, and the transformative impact it has had on building interactive and efficient user interfaces.

## INTRODUCTION

This cheatsheet is designed to be a quick reference, providing concise and essential information about the fundamental aspects of React.js, a JavaScript library for building user interfaces. Whether you're a beginner getting started with React or an experienced developer looking for a quick reference, this cheatsheet is designed to help you navigate and understand React's key concepts and features.

## REACT.JS OVERVIEW

React.js is a popular JavaScript library for building user interfaces, especially for single-page applications where the user interacts with the page without having to reload it. Developed and maintained by Facebook, React simplifies the process of building UI components by using a declarative syntax and a component-based architecture.

Here are some key concepts and features of React:

- **Declarative Syntax:** React allows you to describe how your UI should look at any given point in time, and it automatically updates and renders the components when the data changes. This is in contrast to imperative programming, where you would specify exactly how to achieve a task step by step.
- **Component-Based Architecture:** React applications are built using components, which are reusable, self-contained pieces of code that manage their state and can be composed to build complex UIs. Components make it easier to maintain and reason about your code.
- **Virtual DOM:** React uses a virtual DOM to improve performance. Instead of updating the actual DOM every time the state changes, React creates a virtual representation of the DOM in memory and updates only the parts that have changed in batches. This minimizes the number of direct manipulations to the actual DOM, resulting in faster rendering.
- **Unidirectional Data Flow:** React follows a unidirectional data flow, which means that data flows in a single direction through the components. This makes it easier to understand and debug the application, as data changes are predictable.
- **JSX (JavaScript XML):** React uses JSX, a syntax extension for JavaScript that looks similar to XML or HTML. JSX allows you to write HTML-like code in your JavaScript files, making it more readable and expressive.
- **React Router:** For building single-page applications with multiple views, React Router is commonly used. It enables navigation among views of different components, managing the URL, and updating the UI accordingly.
- **State and Props:** React components can have state, which represents the data that can change over time. Components can also receive data from a parent component through props (properties), making it easy to pass data down the component hierarchy.
- **Lifecycle Methods:** React components have lifecycle methods that allow you to execute code at various points in a component's life, such as when it is created, updated, or destroyed. This provides hooks for performing actions at specific times in the component's lifecycle.

When working with React, you'll need to install Node.js and you will typically use a React-powered tool like Create React App or framework like, Next.js, Remix, Gatsby or Expo to set up your project and then build your components to create a dynamic and interactive user interface. Frameworks provide features that most apps and sites eventually need, including routing, data fetching, and generating HTML. The combination of these features makes React a powerful and efficient library for building modern web applications.

## A "HELLO WORD" COMPONENT

Below is an example of a component that renders a message customizable based on input data.

```
import React from 'react';

// Functional component
const HelloWorld = (props) => {
  return (
    <div>
      <p>{props.message}</p>
    </div>
  );
};

// Example usage
const App = () => {
  return (
    <div>
      <h1>Greetings</h1>
      <HelloWorld message="Hello,
World!" />
    </div>
  );
};

export default App;
```

In this example:

- The `HelloWorld` component is a functional component that takes a `message` prop. It returns JSX, which represents the structure of the UI. It extracts the `message` prop using destructuring and displays it inside an `<p>` element, wrapped in a `<div>`.
- The `App` component renders the `HelloWorld` component and passes the prop with the message "Hello, World!".

## COMPONENT SPECIFICATION

Below is a table outlining the React functional component specification.

| Read/write Fields | Description |
|---|---|
| `props` | Object containing the properties passed to the component. |
| `state` | Object representing the internal state of the component. |
| `setState` | Function used to update the component's state. |

| Component Api | Description |
|---|---|
| `useState` | Hook that allows functional components to have local state. |
| `useEffect` | Hook for handling side effects in functional components. It is also used to implement component lifecycle related functionality. |
| `useContext` | Hook that allows functional components to subscribe to context changes. |
| `useReducer` | Hook for managing complex state logic in functional components. |
| `useCallback` | Memoizes a callback function to prevent unnecessary re-renders. |
| `useMemo` | Memoizes the result of a computation to optimize performance. |
| `useRef` | Creates a mutable object that persists across renders. |

These are fundamental aspects of a functional component in React. Keep in mind that the component API and available hooks may evolve as React is updated, so always refer to the official React documentation for the latest information.

## USING STATE AND PROPERTIES

Understanding when to use state and props is crucial for designing React components effectively. State is more suitable for managing internal component data that can change, while props are used for passing data between components in a controlled and unidirectional manner. Below is a table describing the differences between state and props in React, along with guidance on when to use each.

| Aspect | State | Props |
|---|---|---|
| **Definition** | Internal data managed by a component. | External data passed to a component. |
| **Mutability** | Can be changed using `setState` method. | Immutable; cannot be changed by the component. |
| **Initialization** | Defined within the component using `useState`. | Received from a parent component. |
| **Scope** | Local to the component where it is defined. | Received from a parent component; can be accessed by child components. |
| **Change Trigger** | Changes are triggered by events or async operations within the component. | Changes are triggered by a parent component updating the prop. |
| **Purpose** | Represents data that can change over time within the component. | Provides a way for a parent component to pass data down to its children. |
| **Immutability Principle** | Follows the principle of immutability; should not be modified directly. | Immutable; should not be modified by the receiving component. |
| **Ownership** | Owned and managed by the component itself. | Owned by the parent component and passed down. |

**Use State When:**

- Managing and representing internal state within a component.
- Needing to trigger re-renders based on events or asynchronous operations within the component.
- Handling data that is expected to change during the component's lifecycle e.g., tracking user interactions form input.

**Use Props When:**

- Passing data from a parent component to a child component.
- Configuring child components with data received from a parent.
- Establishing communication between components in a React application.

Let's update the "Hello World" example to include both props (external data) and state (internal data), as well as functions to alter the state. In the following example, we'll use the `useState` hook to manage the component's internal state.

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

const HelloWorld = (props) => {
  // State for the internal message
  const [internalMessage,
setInternalMessage] = useState(
'Default Internal Message');

  // Function to update the internal
message
  const updateInternalMessage = ()
=> {
    setInternalMessage('New Internal
Message');
  };

  return (
    <div>
      <p>External Message (from
props): {props.message}</p>
      <p>Internal Message (from
state): {internalMessage}</p>
      <button onClick=
{updateInternalMessage}>Update
Internal Message</button>
    </div>
  );
};

const App = () => {
  return (
    <div>
      <h1>Greetings</h1>
      <HelloWorld message="Hello,
World!" />
    </div>
  );
};

export default App;
```

In this updated example:

- The `HelloWorld` component now has an internal state called `internalMessage` managed by the `useState` hook.
- The `updateInternalMessage` function modifies the internal state using the `setInternalMessage` function, and it is triggered by a button click.
- The external message is still passed as a prop to the `HelloWorld` component.

## USING CONTEXT

Context in React is a feature that allows you to share data such as themes, user authentication status, language preferences or any other global state across components in a tree without explicitly passing props at every level. Context is often used to avoid "prop drilling," where you pass props down multiple levels of nested components.

**How to Use Context:**

- **Create a Context:** Use `createContext()` to create a context object.
- **Provide a Context Value:** Use a `Provider` component to specify the value you want to share.
- **Consume the Context:** Use the `useContext` hook or the `Consumer` component to access the context value in consuming components.

Let's modify the above example to include a context that provides a theme, in addition to using props and state.

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

```
// Create a context with a default
theme
const ThemeContext = createContext(
'light');

const HelloWorld = (props) => {
  const [internalMessage,
setInternalMessage] = useState(
'Default Internal Message');

  const updateInternalMessage = ()
=> {
    setInternalMessage('New Internal
Message');
  };

  // Use context to get the current
theme
  const theme = useContext
(ThemeContext);

  return (
    <div style={{ background: theme
=== 'dark' ? '#333' : '#fff', color:
theme === 'dark' ? '#fff' : '#333'
}}>
      <p>External Message (from
props): {props.message}</p>
      <p>Internal Message (from
state): {internalMessage}</p>
      <p>Theme (from context):
{theme}</p>
      <button onClick=
{updateInternalMessage}>Update
Internal Message</button>
    </div>
  );
};

const App = () => {
  // Use ThemeContext.Provider to
set the theme for the entire app
  return (
    <ThemeContext.Provider value=
"dark">
      <div>
        <h1>Greetings</h1>
        <HelloWorld message="Hello,
World!" />
      </div>
    </ThemeContext.Provider>
  );
};

export default App;
```

In this example:

- We've created a `ThemeContext` using `createContext('light')`, providing a default theme of 'light'.
- The `ThemeContext.Provider` in the `App` component sets the theme to 'dark' for the entire app.
- The `useContext(ThemeContext)` hook in the `HelloWorld` component allows us to access the current theme from the context.
- The component's styling is adjusted based on the theme.

## USING AJAX

In React, you can make AJAX (Asynchronous JavaScript and XML) requests using various techniques. One common approach is to use the `fetch` function or third-party libraries like Axios, jQuery or Zepto. Let's see an example.

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

const MyComponent = () => {
  const [data, setData] = useState
(null);
  const [error, setError] =
useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await
fetch('https://api.example.com/data'
);
        const result = await
response.json();
        setData(result);
      } catch (error) {
        setError(error);
      }
    };

    fetchData();
  }, []); // Empty dependency array
means the effect runs once after the
initial render

  return (
    <div>
      {data && (
        <div>
          <h1>Data Loaded:</h1>
          <pre>{JSON.stringify(data,
null, 2)}</pre>
        </div>
      )}
      {error && <p>Error: {error
.message}</p>}
    </div>
  );
};

export default MyComponent;
```

In this example:

- The `fetchData` function is an asynchronous function that makes the API request using the `fetch` function.
- The `useEffect` hook is used to trigger the data fetching when the component mounts.
- The fetched data is stored in the component's state using the `setData` function.
- Error handling is done using the `try/catch` block, and errors are stored in the component's state using the `setError` function.

## COMPONENT STYLING

In React, you can apply styles to components using a variety of methods. The two most popular ones are using inline styles or using CSS modules. Lets see an example of using both methods at the same time.

```
// styles.module.css
.container {
  color: red;
  font-size: 12px;
  // ... other CSS properties
}

import React from 'react';
import styles from
'./styles.module.css';

const MyComponent = () => {
  const mystyle = {
    color: 'blue',
    border: '1px solid black',
    // ... other CSS properties
  };

  return (
    <div style={mystyle}>
      <p className={styles.
container}>Hello, Styling!</p>
    </div>
  );
};

export default MyComponent;
```

## DOM REFERENCES

In React, accessing DOM elements directly is generally avoided, and the preferred approach is to use React's virtual DOM and state to manage component rendering. However, there are situations where you may need to interact with the actual DOM, such as focusing an input field, measuring an element, or integrating with third-party libraries that require direct DOM access.

React provides a feature called refs that can be used to get a reference to a DOM element. Refs are created using the useRef hook method.

```
import React, { useRef, useEffect }
from 'react';

const MyComponent = () => {
  const myInputRef = useRef(null);
  useEffect(() => {
    // Access the DOM element using
current property of the ref
    myInputRef.current.focus();
  }, []); // Empty dependency array
means the effect runs once after the
initial render

  return <input ref={myInputRef} />;
};
```

In the example above, we set focus on the input field right after its initial render.

## VALIDATING PROPERTIES

In React, you can validate the props that a component receives using PropTypes. PropTypes is a built-in type-checking feature that helps you catch common bugs by ensuring that components receive the correct types of props. React will issue warnings to the console if the passed props do not match the specified types. Below is an example of how you can use PropTypes for prop validation.

```
import PropTypes from 'prop-types';

const MyComponent = ({ name, age,
isStudent }) => {
  // Component logic
};

MyComponent.propTypes = {
  name: PropTypes.string.isRequired,
// String is required
  age: PropTypes.number,
// Number is optional
  isStudent: PropTypes.bool
.isRequired, // Boolean is required
};

MyComponent.defaultProps = {
  age: 25,
};
```

Here are some common PropTypes:

- PropTypes.string: A string.
- PropTypes.number: A number.
- PropTypes.bool: A boolean.
- PropTypes.array: An array.
- PropTypes.object: An object.
- PropTypes.func: A function.
- PropTypes.node: A React node (e.g., string or ReactElement).
- PropTypes.element: A React element.
- PropTypes.instanceOf(MyClass): An instance of a particular class.
- PropTypes.oneOf(['value1', 'value2']): A value from a specified set.
- PropTypes.oneOfType([PropTypes.string, PropTypes.number]): One of a specified type.
- PropTypes.arrayOf(PropTypes.number): An array of a certain type.
- PropTypes.objectOf(PropTypes.string): An object with values of a certain type.
- PropTypes.shape({ name: PropTypes.string, age: PropTypes.number }): An object with specific properties.

Using .isRequired ensures that the prop is passed and is of the specified type.

PropTypes are a powerful tool for catching potential issues early in development and providing clear documentation for your components. They are particularly helpful when working on larger projects or collaborating with a team.

### CUSTOM VALIDATION

Custom prop validation in React allows you to define your own rules for prop validation beyond the standard data types provided by PropTypes. You can create custom validation functions and use them to check the values of your props.

```
import PropTypes from 'prop-types';

// Define a function that checks whether the prop meets your custom validation criteria.
// The function should return null if the prop is valid, and a Error object if the prop is invalid.
const isValidEmail = (props, propName, componentName) => {
  const value = props[propName];

  if (!value || typeof value !== 'string' || !value.includes('@')) {
    return new Error(`Invalid email prop in ${componentName}.`);
  }

  return null;
};


const MyComponent = ({ name, email }) => {
  // Component logic

};

// Combine your custom validation with standard PropTypes checks of your component.
MyComponent.propTypes = {
  name: PropTypes.string.isRequired,
// Standard PropTypes check
  email: isValidEmail,
// Custom validation check
};
```

## THE FLUX APPLICATION ARCHITECTURE

Flux is an application architecture developed by Facebook for building client-side web applications. It complements the React library but can be used with other libraries or frameworks as well. Flux is not a library or a framework but rather a set of design principles that guide how data flows through an application. The primary goal of Flux is to provide a unidirectional data flow, making it easier to reason about and manage the state of a complex application.

The key components of the Flux architecture include:

- **Actions:** Actions represent events or user interactions that trigger changes in the application state. They are simple objects containing a type property that describes the action, and optionally, a payload with additional data.
- **Dispatcher:** The Dispatcher is responsible for distributing actions to registered stores. It is a central hub that manages the flow of data within the application. When an action is dispatched, the Dispatcher notifies all registered stores about the action. In other words it is essentially an event system and there can be only one global dispatcher.
- **Stores:** Stores contain the application state and logic for handling actions. They respond to actions by updating their state and emitting change events. Each store is responsible for a specific domain or part of the application state. A Store is a singleton and its the only entity in the application that is aware of how to update data.
- **Views (React Components):** Views are React components that display the application's user interface. They subscribe to changes in the stores and update their presentation accordingly. Views can trigger actions based on user interactions.
- **Action Creators:** Action Creators are utility functions that encapsulate the logic for creating actions. They are responsible for defining the different types of actions that can occur in the application.

The data flow in Flux follows a unidirectional cycle:

- **Action Creation:** A user interacts with the application, triggering the creation of an action by an Action Creator.
- **Dispatch:** The Action is dispatched to the Dispatcher, which forwards it to all registered stores.
- **Store Update:** Stores receive the action and update their state based on the action type. They emit a change event to notify the views.
- **View Update:** Views (React components) receive the change event and update their presentation based on the new state from the stores.
- **User Interaction:** The cycle repeats when the user interacts with the updated views, creating new actions and continuing the unidirectional flow.

This architecture helps to maintain a clear and predictable flow of data in the application, making it easier to understand and debug. It's important to note that Flux is a set of design principles, and there are various implementations and variations of Flux in the wild, including popular libraries like Redux.

## TESTING COMPONENTS

Testing React components can be done using testing libraries such as Jest and React Testing Library. Let's create a simple example of testing the "Hello World" component using these libraries.

Firstly, you'll need to install the necessary packages by running the following command:

```
npm install --save-dev jest @testing-library/react @testing-library/jest-dom
```

Now, let's create a test file for our "Hello World" component.

```
import React from 'react';
import { render, screen, fireEvent }
from '@testing-library/react';
import '@testing-library/jest-
dom/extend-expect'; // for
additional matchers

import HelloWorld from '
./HelloWorld'; // Assuming your
component is in HelloWorld.js

describe('HelloWorld Component', ()
=> {
  test('renders the component with
the provided props', () => {
    const { container } = render(
<HelloWorld message="Testing Hello
World" />);

    // Check if the component
renders correctly
    expect(container.firstChild
).toMatchSnapshot();
```

```
    // Check if the rendered text matches the provided message
    expect(screen.getByText('Testing Hello World')).toBeInTheDocument();
  });

  test('updates internal message on button click', () => {
    render(<HelloWorld message="Initial Message" />);

    // Check if the initial message is rendered
    expect(screen.getByText('Initial Message')).toBeInTheDocument();

    // Trigger the button click to update the internal message
    fireEvent.click(screen.getByText('Update Internal Message'));

    // Check if the internal message is updated after the button click
    expect(screen.getByText('New Internal Message')).toBeInTheDocument();
  });
});
```

This test file includes two tests:

- **renders the component with the provided props:**
  - Checks if the component renders correctly with the provided prop.
  - Verifies if the rendered text matches the provided message.
- **updates internal message on button click:**
  - Checks if the initial message is rendered.
  - Simulates a button click to update the internal message.
  - Verifies if the internal message is updated after the button click.

To run the tests, add the following script to your `package.json`:

```
"scripts": {
  "test": "jest"
}
```

Then, run the tests by issuing the `npm test` command.

## REACT ADDONS

The table below provides an introduction to some popular React addons. These addons enhance and extend the capabilities of React applications, covering areas such as routing, state management, styling, animations, testing, and more. Keep in mind that the React ecosystem evolves, and new addons may emerge, so it's always a good idea to check the official documentation for the latest information.

| Addon | Description |
|---|---|
| React Router | A complete routing solution for React applications. Allows for navigation between views in a React app. |
| Redux | A state management library for managing the state of a React application in a predictable way. |
| React-Redux | Official bindings for using Redux with React. Provides the `connect` function to connect components to the Redux store. |
| Redux Thunk | Middleware for Redux that allows you to write asynchronous logic in your Redux actions. |
| Styled-components | A CSS-in-JS library for styling React components. Allows you to write actual CSS in your JavaScript files. |

| Addon | Description |
|---|---|
| React Helmet | A library for managing the document head in React applications. Useful for updating meta tags and titles dynamically. |
| React-Query | A library for managing, caching, and updating remote data in React applications. |
| Formik | A form management library for React. Makes it easy to handle form state, validation, and submission. |
| Yup | A JavaScript schema builder for value parsing and validation. Often used with Formik for form validation. |
| React Spring | A spring-physics-based animation library for React. Enables smooth and natural animations in React components. |
| Enzyme | A JavaScript testing utility for React that makes it easy to test React components' output and behavior. |
| React DevTools | A browser extension that allows you to inspect and debug React component hierarchies in your application. |