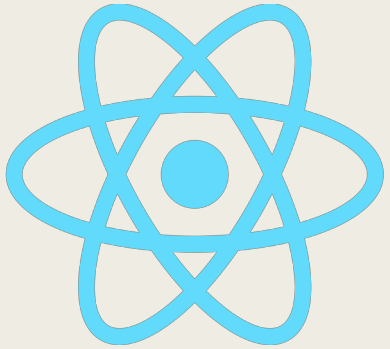


REACT

Single Page Application (SPA) Framework

Prashanth Puranik
Web Developer and Trainer

Basic software setup

Basic software setup

- Make sure to use an IDE with good support for JavaScript
  - VSCode is recommended
  - Also install an extension like **Reactjs code snippets** for VSCode by **Charalampos Karypidis**
- Install Git that shall be used to fetch code from GitHub.
- Code shall be shared using a private GitHub repo. Make sure you have a **GitHub ID** and it is added to the repo.

Git and GitHub

- What are Git and GitHub?
- Setting up Git global configs – `user.name` and `user-email`
- Cloning a repo
- Staging changes and committing them
- Pull changes
- Share your GitHub username? Don't have a personal GitHub account? Then create one and share it with me ☺
  - *The training material is shared via GitHub*

References:

1. Git site which links to its documentation
2. GitHub site

Pre-requisites

Pre-requisites

JavaScript

- **Basic JS** including functions and objects. Also
  - Function bind()
  - Object.assign()
  - Array forEach(), map()

- **ES6 features** used heavily in React
  - let, const
  - Default values for function arguments
  - Array and object destructuring
  - Rest and spread, including object rest and spread
  - Arrow functions
  - Classes and inheritance
  - Promises
  - Generators
  - Modules

JavaScript Toolchain Setup

JavaScript Toolchain Setup

- Front-end apps are popularly created as Node.js projects. Install Node.js.
  - Node.js is a JS runtime usually used to create backend of a web app
  - The npm tool that comes along, is used to manage (install, update etc.) app dependencies
  - The NPM registry has all popular JS libraries including React
- Install (using npm) and use Babel to convert ES6 code to ES5 code
- Install (using npm) and use Webpack to setup a development workflow

Pre-requisites

Node.js and npm

- What it is
- Creating a Node.js project
- Package.json
- npm, Node.js modules and npm registry
- Global and local installation
- Running a Node.js application and scripts in package.json

References:

1. https://nodejs.org/en/
2. Node package registry (npm registry)

Pre-requisites
Using Babel

- Install babel-cli (command line tool to run compiler) and @babel-core (core compiler) in the project folder

```
$> npm install --save-dev @babel/cli @babel/core
```

- Understand plugins and presets. Install (locally)
  - `@babel/preset-env` for ES6 feature support

- Configure Babel using .babelrc file and use script in package.json to build using Babel

Reference:

1. https://babeljs.io/setup#installation

Pre-requisites
Using Webpack

- Install webpack-cli (command line tool to run webpack), webpack (core compiler – a module bundler) and webpack-dev-server in the project folder

```
$> npm install --save-dev webpack-cli webpack webpack-dev-server
```

- Understand loaders. Install (locally)
  - `babel-loader` for transpilation via Babel
  - `style-loader, css-loader` for including and bundling CSS
- Understand plugins and Hot Module Replacement (HMR)
- Configure Webpack using webpack.config.js file and use a script in package.json to build using Webpack, and one more to serve using webpack-dev-server

Reference:

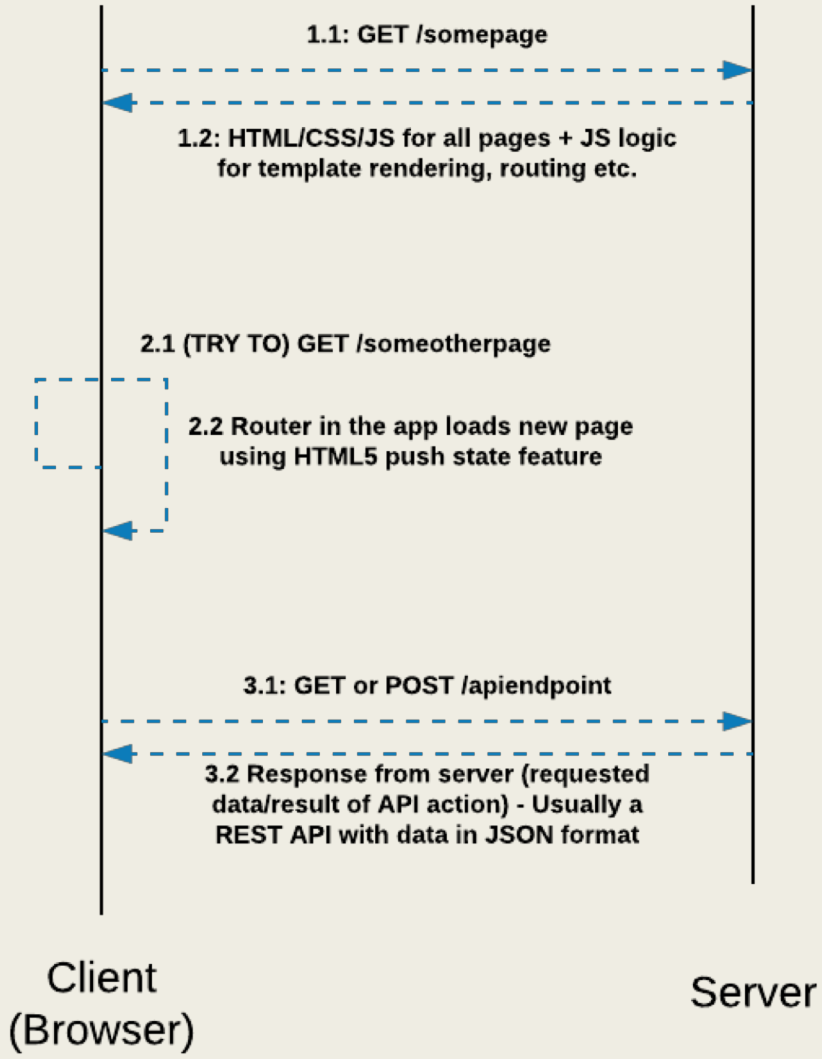
1. https://webpack.js.org/

Getting Started with React

Single Page Application (SPA)

- **Traditional web apps**
  - Multiple HTML pages for multiple views
  - Page refreshes when new view loads
  - Communication with server for HTML, data etc.
  - Page rendered with data on server-side
  - **Example**: Wikipedia

- **SPA**
  - App has only a single page!
  - Views load by manipulating DOM
  - Communication is usually for data only
  - Page rendered by JS running in browser
  - **Example**: Gmail, LinkedIn



What is React?

- Front-end framework
- Used to build an app as a Single Page Application (SPA)
  - Alternatives are Angular (also popular), Vue, Ember and Knockout
- Supports most browsers including IE9+
  - Some polyfills are required for IE9, IE10
- Unlike Angular, React is concerned with building only the “view+controller” in an MV* application

Reference:

1. React: https://reactjs.org/

Supporting libraries

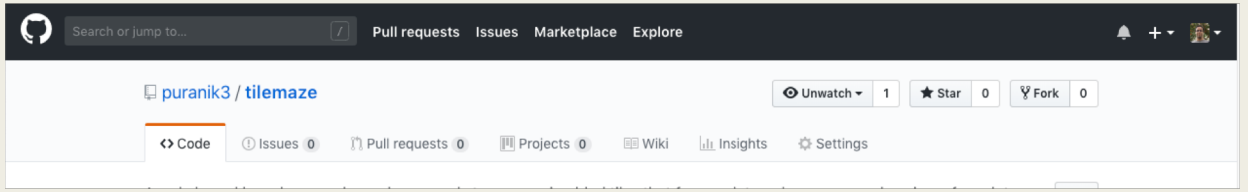
- Since React has a limited concern, it is used in conjunction with other libraries to build a large application
  - React router
  - Redux
  - Connected React Router
  - Axios
  - Redux Thunk, or Redux Saga

  etc.

- We shall use create-react-app to scaffold a React project
  - Helps set up workflow automatically - **build**, **serve**, **test** etc.
  - Uses Webpack to build and bundle modules under the hood

Thinking in terms of components

- React is a **component-based framework** (like Angular, Vue etc.)
- **Components are like custom HTML tags**, except that they encapsulate
  - Structure and Content of the custom HTML element (HTML)
  - Styling for the element (CSS)
  - Behavior for the element (JS)
- Help think in terms of **application-specific HTML elements**, instead of plain HTML elements.
- **Props** are React's equivalent of HTML element attributes.




```
<App>
    <NavBar>
        <BrandImage type="octocat" />
        <Search />
        <NavLinks>
        </Navlinks>
    </Navbar>
    …
</App>
```

A prop called **type** whose value is the string "octocat"

The GitHub page for a repo could be represented like so

Reference: [Thinking in React]

Script Includes for React

- React library consists of 3 JS scripts
  - React core library (creates **React** global variable)
  - React DOM (**ReactDOM** global)
  - Prop Types (**PropTypes** global)

Notes:

1. Use the **UMD (universal Module Loader) versions** that work in all JS environments (browser, Node.js etc.)
2. Use minified versions (.min.js) especially for production code
3. Use same versions for React and React DOM
4. React and React DOM are absolute essentials for any app
5. We shall learn about Prop Types later

Standalone React page

- Build a page that displays an h1 with text Hello World.
- Now add two child nodes to the h1
- Exercise
  - Log the return value of `React.createElement()` call. What does it look like? Note some of the object's properties.
  - What does `ReactDOM.render()` return?
  - `ReactDOM.render()` accepts a React element, or an array of React elements. Try both.

Notes:

1. Use the **UMD (universal Module Loader) versions** that work in all JS environments (browser, Node.js etc.)
2. Use same versions for React and React DOM
3. React and React DOM are absolute essentials for any app
4. Use minified versions (.min.js) especially for production code

Introduction to JSX

- JSX is React's extension to JS
  - It is not ES6, but Babel understands it – include Babel script
- It looks like HTML, but is really JS
- JSX code is transpiled to `React.createElement()` calls - these creates a **React element**

- Rewrite the *Hello World* page with JSX.

Exercise

- Repeat the previous exercise, this time using JSX
- Setup Babel locally using npm, and do away with the Babel script include

React Toolchain Setup

React Toolchain Setup

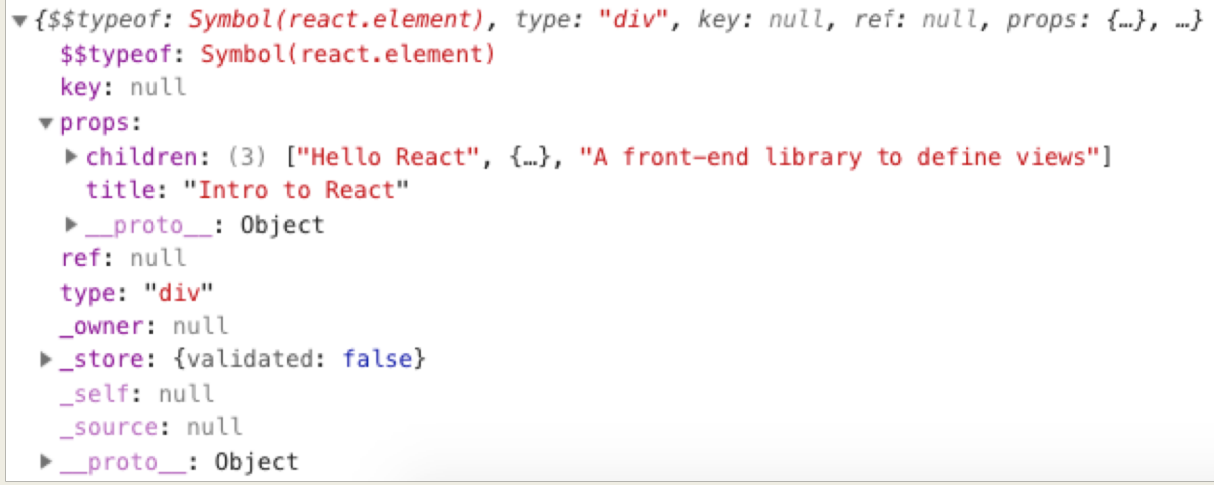
- You need to add more modules to the JS toolchain setup
  - Install `@babel/preset-react` for JSX support
  - Install `@babel/plugin-proposal-class-properties` for object (transforms properties declared with the property initializer syntax)
- Add Hot Module Replacement (HMR) plugin and configure for React app for HMR using react-hot-loader
- We shall hereafter use the create-react-app starter kit
  - Comes with great Webpack-based workflow
  - Many loaders pre-configured, testing using Jest is integrated, production optimized builds etc.

Reference:

1. Creating a toolchain for React from scratch using npm, Babel and Webpack
2. https://github.com/facebook/create-react-app#create-react-app

Diving Deep into React

React elements



- React's simplified representation for HTML elements (native and custom elements)
- These are used by `ReactDOM.render()` to render equivalent DOM nodes
- A react element is a plain JS object
- Props go intro props property
- Child nodes go into `props.children`

Exercise: Try changing the value of a prop in a React element.

Learning: *Props are immutable*

Inputs for a react element: **props** & **state**

- There are **2 things that affect the look and behavior of a React element**
  - **props**
    - These are like HTML attributes
    - HTML attributes are strings. **Props can be of ANY data type.**
    - Props act as input to the element from the outside world – they can't be changed inside
    - **Example**: The logo to be used in a custom `<IconButton />` element
      - `<IconButton type="octocat" />`
  - **state**
    - State is data maintained internally by a React element
    - It is usually time-varying data that is internal to the element
    - **Example**: The current time in a `<Clock />` element that displays current time

Creating Components

- Components are like custom HTML elements – they help reuse markup with functionality
- Their names MUST begin with a capital letter
- They can be
  - **Stateless** – defined using a **function**
  - **Stateful** – Defined using a **class** that extends React.Component

Exercise:

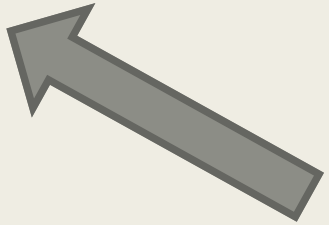
1. Define a Clock component as a stateless component. Take a display message as prop. Render a new instance of Clock every second to have time updated on the UI.
   - *What does `ReactDOM.render()` return?*
2. Define the same as a class component. Maintain the current time as state, and have it updated within the component every second.
   - *What does `ReactDOM.render()` return?*

Which approach would you prefer in this case? Why?

In summary...

Component definition

```
function Invoice( props ) { … }
(or)
class Invoice extends React.Component {
    constructor( props ) { super( props ); … }
    render() { … }
    …
}
```



React element creation

```
<Invoice title="Purchase Receipt" items={[
    { name: 'Soap', qty: 2 },
    { name: 'Chips', qty: 1 }
]}>
          <h2>Terms and Conditions</h2>
          <p>…</p>
</Invoice>
```

props object passed to Invoice function/constructor

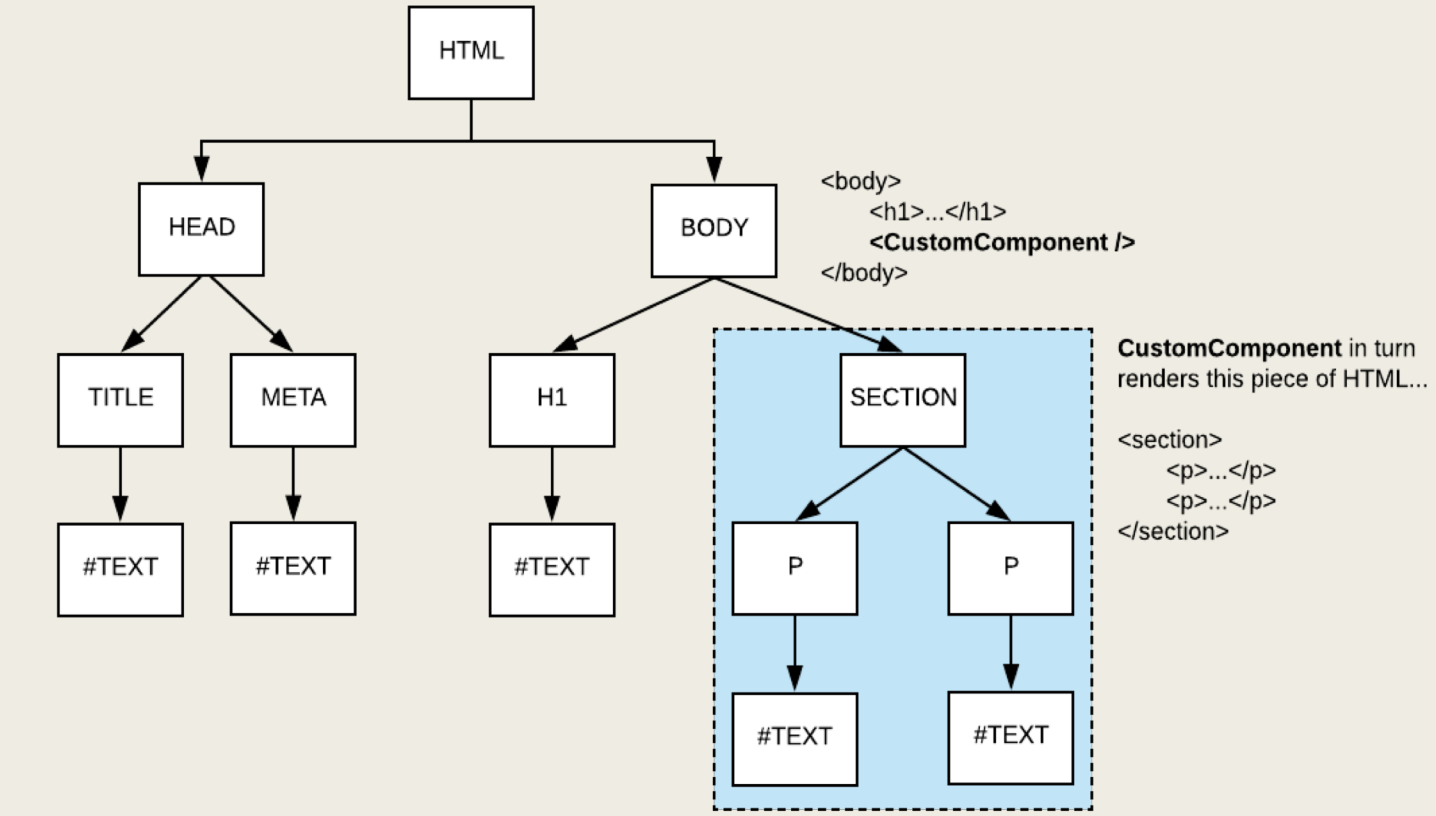
```
{
    title: "Purchase Receipt",
    items: [
        { name: 'Soap', qty: 2 }, { name: 'Chips', qty: 1 }
    ],
    children: [ { type: 'h2', props: { … } … }, { type: 'p', props: { … } … } ]
}
```

The Virtual DOM

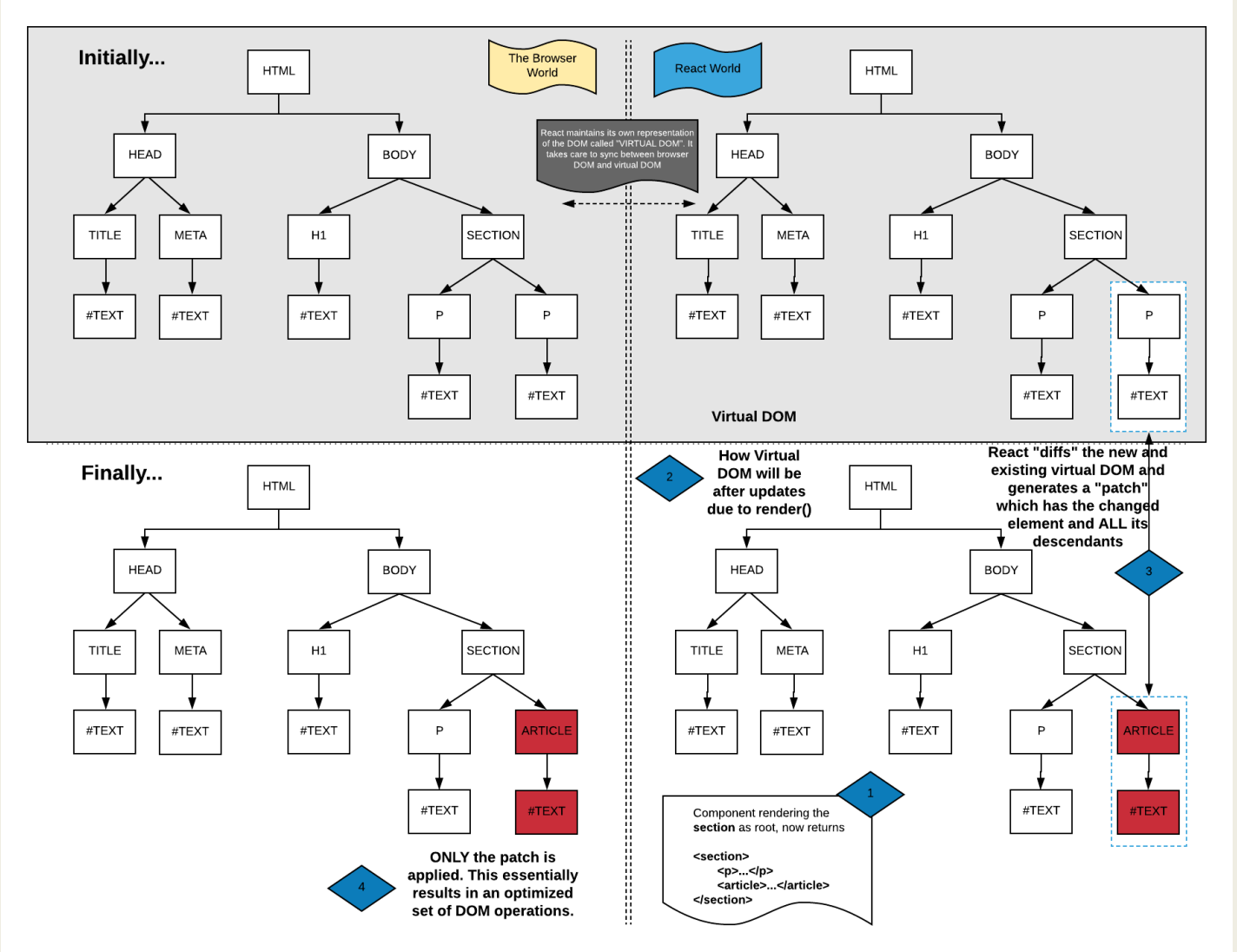
- **Virtual DOM**
  - The hierarchy of React elements with the `ReactDOM.render()` root element, that corresponds to the HTML DOM.
- React makes **intelligent updates to the HTML DOM** based on changes to the Virtual DOM (i.e. React elements within)
- There is a process of DOM **diffing** whenever changes occur
- Only the HTML DOM nodes corresponding to React elements that changed in a re-render are updated (**reconciliation**)
- Due to algorithmic complexity considerations, the subtree of any changed React element is also updated
  - *This works well in practice*

The Virtual DOM makes DOM updates in React very efficient

The Virtual DOM



The Virtual DOM



Props

- **Props can be of ANY data type**
  - *number*
  - *boolean*
  - *string*
  - *object*
  - *array*
  - *function*
  - *class (**including another component!**)*
- If the **prop is a string** you can set as *<Component customprop="stringvalue" />*
- **In general** you set as *<Component customprop={anyvalue} />*
- Pass **props available as key-value pairs in an object**, as *<Component {...obj} />*
- Props default to true `<Component prop1 />` ⇔ `<Component prop1={true} />`

Exercise: Create a component and pass it props of different data types and use those within the component.

Children

- Just like DOM nodes, React elements can have children.
- The children are available on **props.children** as
  - **null** : no child
  - **string** : single text child
  - **React element**: single non-text child
  - **Array of React elements** : multiple children

```
<Component propX1={valueX1} propX2={valueX2}>
          <ChildComponentA propY1={valueY1} propY2={valueY2}>
                              …
      </ChildComponentA>
      <ChildComponentB propZ1={valueZ1} propZ2={valueZ2}>
                              …
      </ChildComponentB>
</Component>
```

key

- When rendering an array of React elements, it helps to give a special attribute called key
- A key is to be unique among the array rendered
- It helps the React renderer make more sense of the virtual DOM and helps make DOM manipulations more efficient.

Tip: Always set a key when rendering an array of similar React elements. Usually some unique identifying property will exist on each element

JSX

Interpolation – { }

- Use { } within JSX to interpolate variables (have their value show up). { } accepts
  - literals
  - Simple JS expressions (use operators)
  - Function calls
  - Conditional operator – `expression ? expr_if_true : expr_if_false;`
  - React element
  - null, undefined etc. results in nothing – **use this with ?: to conditionally show/hide**
  - An array of React elements – use Array map() to generate and display such an array

Exercise: Create an Invoice (say the bill at a supermarket) component. Have it accept

1. items
2. couponCodeOrDiscount

JSX
Styling

- Many prefer using inline styles in React
  - It groups related styles in the component code itself
  - No problems with global styles affecting the component
- It is upto you to decide
- The style prop gets a JS object as value
  - Hyphenated CSS properties are camel-cased
    - font-size becomes fontSize
  - CSS overrides can be achieved with `Object.assign( {}, styleObj1, styleObj2, …)`

Event handling

- React wraps native DOM events (it calls them **synthetic events**)
- Synthetic events behave uniformly across browsers
- Event handlers can be set only on React equivalents of native HTML elements
- Event handlers are set as props
  - `onClick={clickHandler} onMouseOver={mouseOverHandler}`
  - The props are camelCased versions of native events
  - **Note:** Unline in HTML, the functions are NOT called – only references are set

References

1. Guide on event handling in React
2. Reference for synthetic events

The event object

- The event object holds details of the event (just like in HTML/JS)
  - Event-specific details (eg. X-Y coordinates of click)
  - `preventDefault()` – prevents default action of browser
    - Prevent form submission on submit event
    - Prevent navigation on click of link

Context (“this”) of event handlers

- Just like in JS, event handlers are always called with global context (“this” will be window object). So we fix the context where required using either
  - Using function bind() when setting the handler, or centrally (like in constructor)
  - property initializer syntax for class methods `( classMethod = () => {} )`

- Sometimes we need to pass other arguments to an event handler
  - Example: The “item id” of the item being clicked
  - This may be done via Function bind()
    - `this.handler.bind( this, item.itemId );`
    - The handler now gets the event object as the 2nd parameter

Passing variables downstream as props

- A Component can pass variables to its child components via props
- Things commonly passed downstream from Component to a ChildComponent
  - *variable defined in Component*
  - *prop*
  - *property of state object*
  - *function (usually a method of Component class)*
  - *`props.children`*

```
<IconButton type=“octocat” value=“Home” />


function IconButton( props ) {
    return (
        <span className=“icon-button”>
            {this.props.value}
                <Logo
type={this.props.type} />
        </span>
    );
}
```

Example of a prop being passed downstream

Passing children downstream

- Sometimes it is required that a Component pass its children downstream
- **Example:**
  - A Dialog component passes children within to a child DialogBody that it uses internally

```
<Dialog>
    <h2>Terms and Conditions</h2>
    <p>…</p>
</Dialog>
```

```
function Dialog( props ) {
    return (
        <div className=“dialog”>
            <DialogBody>{props.children}</DialogBody>
        </div>
    );
}
```

Component state

- Create a stateful component when component is associated with an internal, (and usually) time-varying data.
  - The time in a Clock component
  - The products displayed by a ProductList component (pagination can change the list of products rendered)
  - The search suggestions in a search box (changes as user types)
- State is initialized by constructor on the component instance as the **state** property
- `setState()` changes state and results in re-render as state changes

setState()

- **Never modify `this.state` directly** – React won't come to know and component isn't re-rendered
- state and prop updates are (possibly) **asynchronous** operations
  - `setState()` is more like a request to update state (which React will honor)
- React may batch `setState()` calls that are called in quick succession, and run in order
- **`setState()` has 2 signatures.** The arguments for the 2 are
  1. `newState_delta_update` (**object**), `callback_on_state_change` (**function**)
  2. `callback_before_update` (**function**), `callback_on_state_change` (**function**)
- **In 1**, the `newState_delta_update` object is "merged" with current state (like in `Object.assign()`)
- **In 2**, `callback_before_update` **is passed the current state and props** just before React is about to actually perform state update**. Use these objects within the function.**

Use 2 if new state depends on current state

Exercise

- Implement a Counter app
  - It displays a counter value – initially 0
  - It has 2 buttons
    - + : Clicking this increments the value of counter
    - - : Clicking this decrements the value of counter

Which signature of `setState()` will you use in this case? Why?

Changing upstream state

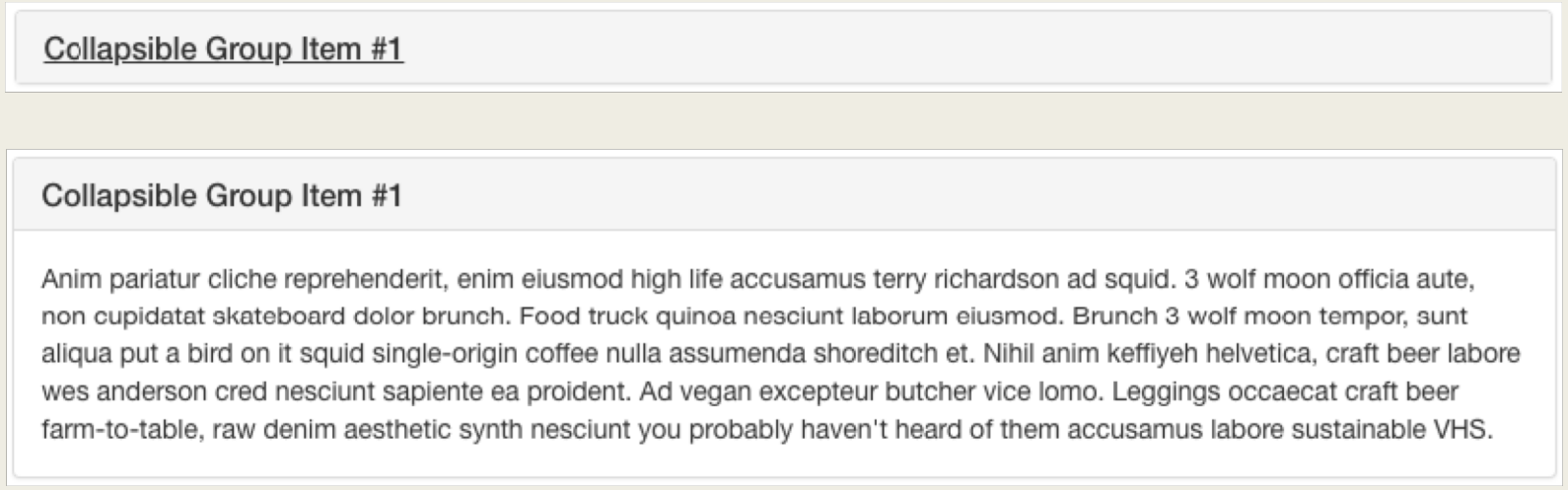
- Sometimes it is necessary for Child Component to changes something in the Parent Component – for example, the parent's state
- This can be done by parent passing a method that does the desired state change as prop downstream. When the child needs to update parent state, it calls the method.

```
Class Parent extends React.Component {
    …
    update() {
        this.setState( … );
    }
    render() {
        return <Child update={update} … />;
    }
}
```

```
function Child( props ) {
    return (
        <div onClick={update}>
            Click to update parent state
        </div>
    );
}
```

Exercises

- Build a Bootstrap-like Collapsible Panel Component
  - It should have children Component instances – PanelHeading and PanelBody
  - You can toggle PanelBody display by clicking the PanelHeading
  - PanelBody's children are rendered as such in the final HTML
  - Try using both inline styling, as well as external stylesheet



Exercises

- Build a Stopwatch
  - You can start, stop and reset the stopwatch using buttons
  - It stores the seconds elapsed when it is in running state
  - Reset sets the seconds elapsed back to 0

Which signature of `setState()` will you use in this case?

Note: The answer may depend upon your implementation of Stopwatch

Note: `setInterval()` returns an integer id. Calling `clearInterval()` with the id, stops a function executing periodically via `setInterval()`

Detour – Making Ajax Calls using Axios

Axios

- You can use the **XMLHttpRequest** object of the browser to do Ajax calls
  - This is difficult to use and does not support promises

- You can use the **fetch API** in browsers
  - This supports promises, but is not supported in old browsers

- **Axios**
  - A promise-based API for Ajax calls that works in old browsers too!
  - Has methods like `get()`, `put()`, `post()`, `delete()` to do Ajax calls
  - Use axios.all() to parallelize Ajax requests

Reference: Axios project on GitHub

Making Ajax calls

- It is customary to define 3 states of application/component for every Ajax call
  - *REQUEST_SENT*
    - Set this as request state when Ajax call is sent
    - A component may show a spinner or a loading message in this state
  - *REQUEST_SUCCESS*
    - Set this when Ajax call returned result successfully
    - Additionally set the results as part of the state
    - A component may re-render to show results
  - *REQUEST_FAILURE*
    - Set this when Ajax call failed to return results
    - Additionally set the error details as part of the state
    - A component may re-render to an appropriate error message

Exercises

- Build a **QuestionList Component** that displays questions on Stack Overflow
  - It iterates through a supplied list of questions and displays individual questions in a **Question Component**
  - Use the `{...obj}` syntax to pass individual question details

Reference: https://api.stackexchange.com/2.0/questions?site=stackoverflow

Type-checking for props

- The data type of props can be specified using `PropTypes`
- Can be used for both stateless and stateless components
- The Prop Types script is to be included
  - prop type-checking is disabled in production
  - Use the development build (DO NOT use the .min.js file)
- The script allows you to define data type of props
- The library issues warning when there is mismatch

Reference: https://reactjs.org/docs/typechecking-with-proptypes.html

Default values for props

- Props can be assigned default values
- The default gets applied when the prop is not specified when the element is created
- Can be used for both stateless and stateful components

Exercise: Add propTypes & defaultProps

- Add type-checking for props in Invoice component
- Add default values for props in the Invoice component

Sharing data across component methods

- A React is only a plain class
- You can always create custom properties/methods on an object (apart from props, state etc.)
- You can also create static properties/methods
- Custom properties on the instance of a component are often used to share data across methods

ref

- A ref is a prop that is used to get a DOM node reference
- You can create a DOM node reference using
  - `React.createRef()` in constructor (new way)
  - callback (old way)
- The ref is usually setup as an instance property to be shared across methods
- `refObj.current` will be
  - DOM node for HTML element like
  - Component instance for custom component like <PanelBody />
- Refs are updated just before `componentDidMount()/componentDidUpdate()`

Note: Refs cannot be set on stateless (function) component elements

Working with forms

- Use **`onSubmit`** event for submit handling
- You can prevent form submission on invalid form (**`event.preventDefault()`**)
- **`defaultValue`** prop sets initial value of input element (same as **value in HTML**)
- A common practice is to create **controlled components**
  - Input element value is bound to a component state property (using **`value`** prop)
  - When inputs changes, the state property is updated in turn using an **`onChange`** listener
  - This is called 2-way data binding in other libraries/frameworks

Exercise

Write a contact form which uses controlled components to maintain form state in sync with input state

- The form should enable the submit button only if the form state is valid
- It should display error messages for invalid states of input elements
- On submit of the form, log the form state (which summarizes input element states) in the console.

Lifecycle methods
Phases

Three stages in the lifetime of a React component instance

- **Mounting phase**
  - Component instance is ***created*** and ***rendered into DOM first time***
- **Update phase**
  - ***state or prop changes*** and the instance's element tree is usually ***re-rendered***
- **Unmounting phase**
  - Element tree is ***removed from the DOM*** and the component instance is ***destroyed***

References

1. Guide on React site
2. Reference on React site

Lifecycle methods
Mounting phase

- `constructor( props )`
  - called when component is created
  - initializes component instance and sets the state up
- `componentWillMount()`
  - called before render method (before component's markup rendered to DOM)
- `render()`
  - returns markup to be rendered – rendering happens post this
- `componentDidMount()`
  - called after render method (after markup is rendered to DOM)

Lifecycle methods
Update phase

- `componentWillReceiveProps( nextProps )`
  - called when parent sends props or updates them
- `shouldComponentUpdate( nextProps, nextState )`
  - called before updation can start after state/props change
  - returns a Boolean. The updation can be cancelled by returning false
- `componentWillUpdate( nextProps, nextState )`
  - called when state/props change, but before render(). The state/props change after this.
- `render()`
  - called with new state/props. The new component tree is returned and rendered.
- `componentDidUpdate( prevProps, prevState )`
  - called after render in update phase

Lifecycle methods
Unmounting phase

- `componentWillUnmount()`
  - called when component is about to be removed from the DOM
  - since nothing is to be done after component unmounts, THERE IS NO componentDidUnmount()

How to use the lifecycle methods

- `constructor()`
  - Use to initialize state

- `render()`
  - Avoid `setState()` – results in recursive call!
  - Avoid DOM access
  - Avoid Ajax calls

- `componentDidMount()`
  - can use `setState()`
  - can access DOM and set event listeners
  - can use timers (`setTimeout`, `setInterval`, `clearTimeout`, `clearInterval`)
  - Make Ajax calls

How to use the lifecycle methods

- `shouldComponentUpdate()`
  - Use this to improve performance by preventing unnecessary re-renders
  - If the UI is not going to change as a result of change in props/state, the return false to prevent re-rendering

- `componentWillUnmount()`
  - Remove event listeners
  - Cancel Ajax calls
  - Invalidate timers can be used (`clearTimeout`, `clearInterval`)

Routing – Creating an SPA

Routing

- Routing is logic that implements page transitions based on URL in address bar
- Example:
  - `www.mystore.com/` results in app's home page being loaded
  - `www.mystore.com/products` shows products listing page
  - `www.mystore.com/products/101` shows details of product with id = 101
  - `www.mystore.com/products/101/reviews` shows reviews of product with id = 101
  - `www.mystore.com/products/101/reviews/new` shows form to submit review for product with id = 101

What we shall build – Store app

- Store API available at https://awesome-store-server.herokuapp.com/
- The app shall have the following pages that appear on the given routes
  - `/` - the landing page that describes the app
  - `/products` - product catalog
  - `/products/:id` where `/:id` is a path fragment like `/1`, `/2` etc. - Product details for product with id 1, 2 etc.
  - `/products/:id/reviews` - reviews of product with given id
  - `/products/:id/reviews/new` – submit review form to add review for product with given id
  - A “Page not found page” is displayed for any other route

React router

- React router version 4 (react-router) is a popular solution for routing in React
- Custom components for routing
  - **<BrowserRouter />**, **<HashRouter/>** – Surround the top-level component with one of these
  - **<Link />**, **<NavLink />** – Use these in place of of an anchor tag (...). This prevents HTTP request and instead URL changes with React Router handling the change
  - **<Route />** - To configure component to load on change to specific route. The component renders as a child of <Route /> on a route match
  - **<Switch />** - To render only first component matching a route, from a group.
  - **<Redirect />** - Rendering this redirects app to the specified route

References

1. Overview
2. Quick start guide

React router

- History object to initialize router with. In a browser use `createBrowserHistory()` of history library to create this.

- Some useful props of React Router components are
  - `<BrowserRouter history={…}/>`, `<HashRouter history={…}/>`
  - `<Link exact to="some/path"/>`
  - `<NavLink exact activeClassName="name"/>`
  - `<Route exact path="some/path" component={ComponentX}/>`, `<Route path="some/path" render={() => {…}}`
  - `<Redirect to="some/path" />`

React router

- 3 props are passed to each descendant of top-level Router Component (essentially all component instances)
  - history
  - location
  - match
    - The dynamic parts of the URL (eg `id` in `/:id`) are available in the rendered element inside the element's `props.match.params` object

- Additionally, when using Redux, a method called `withRouter()` shall be used to wrap the top-level connected React Component

Exercises

- Build a 2 page app for a fictitious startup you own
  - my-startup.com/ - About page
  - my-startup.com/contact – Contact page. Display the following on the same page in a tabbed interface
    - my-startup.com/contact/address - address and map
    - my-startup.com/contact/form – contact form
- Style using Bootstrap

Exercises

- Build the store app using a Master-Detail view
- URL: /
  - A left pane displays a list of products (as a list with image thumbnails instead of table)
  - The right pane initially displays the About component
- URL: /products/:id
  - Clicking on a product list item in the left pane changes the URL to the one above, and displays the details of the product in the right pane
- Style using Bootstrap

Redux

- Redux is a predictable state management library for JavaScript apps
- Redux is independent of React and can be used with plain JS apps, Angular apps etc.
- It implements the Flux architecture recommended for React apps (however only one store).
- When app grows large it becomes difficult to pass state information across the component tree
  - *Example: One component updates its state and another component far away in the tree needs to re-render based on the state change*
- When Redux stores the app data, state is available centrally and any component can "subscribe" for notifications on state change
  - This way communication becomes simple

Reference: Redux site

Flux Pattern

- React apps do not follow the MVC pattern
- MVC vs Flux
  - Many models vs few stores (or one in Redux)
  - Bi-directional data flow between View and Controller, and Model and Controller in MVC
  - Uni-directional flow in Flux
    - Store to view allowed. View however cannot update store - it happens only indirectly via action and dispatcher

Redux Architecture

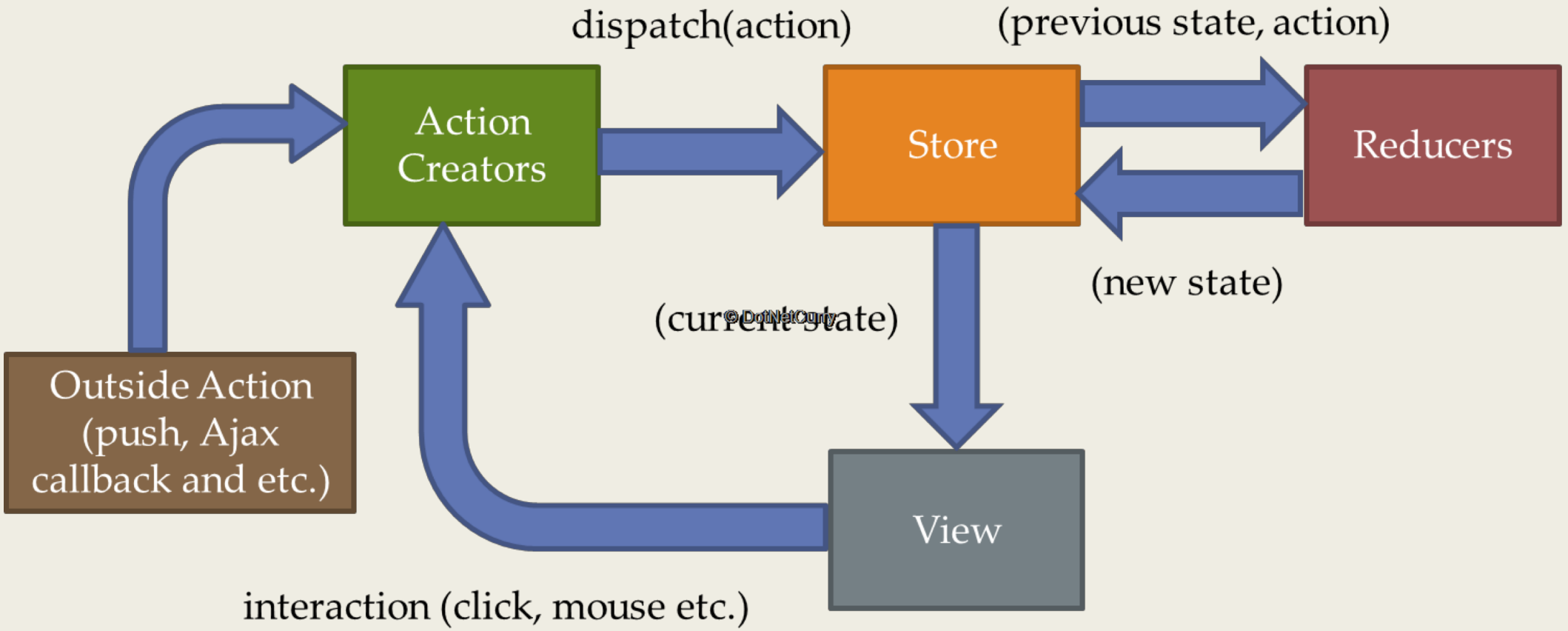


Image Credits: https://www.dotnetcurry.com/reactjs/1356/redux-pattern-tutorial

Redux app - Building blocks

- **Store**
  - stores data for entire app. Redux uses a single store.
- **Actions**
  - Objects representing user actions etc. in the app. Each MUST have type whose value is unique for the action
- **Action creators**
  - Function which simply return some action
- **Reducers**
  - Functions that take current state and action as input and return a new state
  - The store calls reducers when an action occurs
- **Middleware**
  - Hooks (functions) called between dispatch of action by the UI, and call to reducers
  - They allow transformation on the action (even cancelling them), side effects etc.

Reducers

- Reducers SHOULD be **pure functions**
  - Do not alter arguments
  - No side effects - no change in global state, no DB and network calls (no Ajax)
  - Do not use global state and non-deterministic methods (eg. Math.random())
  - Always give the same output for same inputs
  - Always return a value

- Reducers return
  - A new state object to change state (however it need not be a complete deep copy – only parts that changed need be copied)
  - The original state if no change in state is to be effected

Redux API

- `const store = createStore( reducer, initialState, enhancer );`

- `const combinedReducer = combineReducers( stateTree );`
  - Multiple reducers needed to modularize state changes in a large app
  - `combineReducers()` generates a single reducer using all of the reducers
  - Reducers are specified in a `stateTree` whose keys become the keys in the store's state object
  - `initialState` follows the same structure as `stateTree`
  - Every reducer is called on every action dispatched, but is passed only that part of the state it is concerned with

- `const enhancer = applyMiddleware( middleware_1, middleware_2, …)`
  - Middleware are executed in the specified sequence

Redux API

- `store.getState()`
  - returns the current state
- `store.dispatch( action )`
  - Used by the UI to dispatch actions
- `store.subscribe( listener )`
  - Used by the UI to listen for state changes

Exercise

- Write the Counter app using React and Redux

Note: You will need to use a React component's forceUpdate() method in order to force re-render after state changes in Redux

Writing custom middleware

- The function curry pattern is used to define the middleware function
- It is defined as a series of closures (one enclosing the other)

```
const middleware = store => next => action => {
        // write stuff that should be executed before the reducer executes, here
        …
        next( action );
        // write stuff that should be executed after the reducer executes, here
        …
};
```

Reference: Middleware in Redux

Async action using Redux Thunk

- It is better to have a module that makes Ajax calls (web service interactions)
  - This way code that accesses services is shared
- Every time an action involving an Ajax call is dispatched we need to also make the Ajax call in our component because
  - action creators are supposed to only return actions
  - reducers are pure functions and can have no side-effects (like making Ajax calls)
- Redux Thunk provides a simple solution – it is a middleware that allows your app to call dispatch actions that are functions (and also objects)
- **Redux Thunk intercepts function that are dispatched, executes them (passing dispatch as argument).**
  - These functions can dispatch normal action objects and also trigger Ajax calls etc.

Reference: Redux Thunk

Exercise

- Explore and use redux-logger middleware
- Implement your own Redux middleware that does the same thing as Thunk middleware
- Implement the QuestionList component using React and redux
- Implement the Store app using React and Redux

Redux DevTools Extension

- A Chrome extension. Two steps to use it…
  - Install the Chrome extension
  - Include necessary code in the app

- This tool helps you debug Redux apps
  - Dispatch actions from its UI
  - See state changes
  - Time-travel debugging – Go back and forth in action history, even skipping previously dispatched actions to see what the result would be

- **Usage**: `composeWithDevTools( applyMiddleware( … ) )`

Reference: http://extension.remotedev.io

React-Redux

- React-Redux is the official React binding for Redux (“ties” React to Redux)
- Defines a **container component that wraps a component** using state/changing state
  - Container subscribes to stores and dispatch actions
  - Wrapped component simply acts as a presentation layer (unaware of Redux)

- **Advantages**
  - **Optimizes performance** – re-renders components only when needed (only when state changes it is concerned with happen)
  - **Hides Redux from components**, making them easily testable and reusable

Reference: https://react-redux.js.org/

React-Redux API

- Wrap the top-level app component with `<Provider />` (encloses even the `<Router />` component if it exists)
- Supply `store` as a prop
- `connect( mapsStateToProps, mapDispatchToProps )`
  - wrap components making state changes, or requiring state using it
  - returns a connected component (one that binds Redux store and component)
- `mapsStateToProps( state, ownProps )` – returns stateProps
- `mapDispatchToProps( dispatch, ownProps )` – returns dispatchProps

The connected-react-router library

- Helps move router state also to Redux

Reference: https://github.com/supasate/connected-react-router

Exercise

- Implement the Store app using React-Redux along with connected-react-router

Advantages of using Redux

- Flux pattern helps structure interactive applications and avoid spaghetti code
- Predictable state management system
- Centralized state management a boon for large scale applications.

THANK YOU!

Questions are welcome

Copyright Notice

Prahara Consulting Private Limited

204, Sri Vari Enclave

Horamavu Agara Road, Horamavu, Bangalore - 560043.

puranik@digdeeper.in