



Assignment 01 - Inception



Owner



Pankaj Kumar

1: What is Emmet?

- **Emmet** is the essential toolkit for web-developers. It allows you to **type shortcuts** that are then expanded into full pieces of code for writing **HTML, XML, CSS and Other languages** based on an abbreviation structure most developers already use that expands into full-fledged HTML markup and CSS rules.
- In Visual Studio Code (VS Code), Emmet is built-in and can be accessed by typing abbreviations and then pressing the Tab key to expand them. For example, typing "div.container" and pressing Tab will expand to:

```
<div class="container"></div>
```

2: Difference between a Library and Framework?

- **library** is a collection of packages that perform specific operations whereas a **framework** contains the basic flow and architecture of an application. The major difference between them is the complexity. Libraries contain a number of methods that a developer can just call whenever they write code. React js is library and Angular is Framework.
- One key difference between libraries and frameworks is that libraries are called by the code, while frameworks call the code. In other words, when you use a library, you are in control of how and when it is used. With a framework, the framework is in control and dictates how the code should be organized and used.

3: What is CDN? Why do we use it?

(OR)

- CDN(Content Delivery Network) are used by a wide range of websites and applications, including e-commerce sites, media and entertainment platforms, and software as a service (SaaS) providers.
- **Improving the speed of web pages:** By serving content from a server that is physically closer to the user, CDNs can significantly improve the loading speed of web pages.
- **Reducing bandwidth costs:** CDNs can reduce the amount of bandwidth required by a website or application by serving cached copies of content from servers located closer to the user.
- **Enhancing security:** CDNs can provide additional security measures such as DDoS (Distributed Denial of Service) protection and SSL(Secure Socket Layer) encryption, helping to protect websites and applications from attacks.
- **Improving availability:** CDNs can help to improve the availability of a website or application by providing multiple copies of content that can be served to users if one server goes down.

4: Why is React known as React?

- React is a JavaScript-based UI development library. Facebook and an open-source developer community run it. It is designed to be declarative, meaning that developers specify what the UI should look like based on the current state of the application, and React takes care of updating the UI as needed. React is designed to be efficient and flexible, and is widely used in the development of web and mobile applications.

5: What is crossorigin in script tag?

- The crossorigin attribute sets the mode of the request to an HTTP CORS Request.
- The purpose of crossorigin attribute is used to share the resources from one domain to another domain. Basically, it is used to handle the CORS request. It is

used to handle the CORS request that checks whether it is safe to allow for sharing the resources from other domains.

- The crossorigin attribute on a `<script>` tag specifies that CORS is supported when loading an external script file from a third party server or domain. CORS is a standard mechanism used to retrieve files from other domains.

Syntax

```
<script crossorigin="anonymous|use-credentials">
```

6: What is difference between React and ReactDOM?

- React and ReactDOM are two separate libraries that are often used together in the development of web applications with React.
- `React` is a JavaScript library for building User Interfaces whereas `ReactDOM` is also JavaScript library that allows `React to interact with the DOM`.
- ReactDOM, on the other hand, is a library that provides an interface between React and the DOM (Document Object Model). The DOM is a tree-like structure that represents the HTML of a web page, and ReactDOM provides a set of functions that allow React components to be rendered to the DOM and updated efficiently.
- In short, React is a library for building user interfaces, while ReactDOM is a library for interacting with the DOM and rendering React components to the web page. While they are often used together, they serve different purposes and can be used independently of each other.
- The react package contains `React.createElement()`, `React.Component`, `React.Children`, and other helpers related to elements and component classes. You can think of these as the isomorphic or universal helpers that you need to build components. The react-dom package contains `ReactDOM.render()`, and in react-dom/server we have server-side rendering support with `ReactDOMServer.renderToString()` and `ReactDOMServer.renderToStaticMarkup()`.

```
const listElement = React.createElement( 'li', {
  className: 'list'
}, 'React.js' );

ReactDOM.render( listElement, document.querySelector( '#root' ) );
```

7: What is difference between react.development.js and react.production.js files via CDN?

- Development is the stage of an application before it's made public while production is the term used for the same application when it is made public.
- Development JS for development reasons. You have Source Maps, debugging and often times hot reloading ability in those builds.
- The production build, on the other hand, runs in production mode which means this is the code running on your client's machine.
- Development build is several times (maybe 3-5x) slower than the production build.

8: What is async and defer?

Async - The async attribute is a **boolean attribute**. The script is downloaded in **parallel(in the background)** to parsing the page, and **executed as soon** as it is available (do not block HTML DOM construction during downloading process) and don't wait for anything.

Syntax

```
<script async src="script.js"></script>
```

Defer - The defer attribute is a **boolean attribute**. The script is downloaded in **parallel(in the background)** to parsing the page, and **executed after the page** has finished parsing(when browser finished DOM construction). The **defer attribute** tells the browser **not to wait for the script**. Instead, the browser will continue to process the HTML, build DOM.

Syntax

```
<script defer src="script.js"></script>
```



Assignment 02 - Igniting Our App

wner



Panka umar

1: What is NPM?

- It is a tool used for package management and the default package manager for Node projects. NPM is installed when NodeJS is installed on a machine. It comes with a command line interface to interact with the online database of NPM. This database is called the NPM Registry and it hosts public and private packages. To add or update packages we use the NPM CLI to interact with this database.
- We use NPM because we want a lot of packages in our ProjectsReact app.

Note : npm does not stand for node package manager but everything else.

- npm alternative is yarn

How to initialize npm ?

```
npm init
```

`npm init -y` can be used to skip the setup step. npm takes care of it and creates the `package.json` file automatically but without configurations.

2: What is Parcel/Webpack ? Why do we need it?

- Parcel/Webpack is type of a web application bundler used for development and production purposes or power our application with different type functionalities and features.

- Parcel and webpack are the bundlers used mostly for JavaScript or Typescript code that helps you to minify, clean, and make your code compact so that it becomes easier to send a request or receive the response from the server when it usually takes you to transfer multiple files without using any bundler for loading the page of your application.
- Both of these bundlers substantially reduce the time it takes for the transfer of data and files to the server from the application.
- Along with that both bundlers parcel and webpack remove the unnecessary comments, new lines, any kind of block delimiters, and white spaces while the functionality of the code remains unchanged.
- It offers blazing fast performance utilizing multicore processing, and requires zero configuration. Parcel can take any type of file as an entry point, but an HTML or JavaScript file is a good place to start.

Parcel Features:

- HMR (Hot Module Replacement) —> parcel keeps track of file changes via file watcher algorithm and renders the changes in the files
- File watcher algorithm (written in C++)
- Minification
- Cleaning our code
- DEV and production Build
- Super fast building algorithm
- Image optimization
- Caching while development
- Compresses
- Compatible with older version of browser
- HTTPS in dev
- Port Number
- Consistent hashing algorithm

- Zero Configuration
- Automatic code splitting
- Tree Shaking —> Removed unwanted/unused code

installation commands:

```
npm install -D parcel
```

- `-D` is used for development and as a development dependency.
- Parcel Commands :
 - For development build:

```
npx parcel <entry_point>
```

- For production build :

```
npx parcel build <entry_point>
```

3: What is `.parcel-cache`

- `.parcel-cache` is used by parcel(bundler) to reduce the building time.
- It stores information about your project when parcel builds it, so that when it rebuilds, it doesn't have to re-parse and re-analyze everything from scratch. It's a key reason why parcel can be so fast in development mode.
- The dist folder contains the output of Parcel and the content of that folder is served by the web server.

4: What is `npx` ?

- The `npx` stands for Node Package Execute. It is a tool that is used to execute the packages. It comes with the npm, when you installed npm above 5.2.0 version then automatically npx will be installed. It is an npm package runner that can execute any

package that you want from the npm registry without even installing that package. npx is pre-bundled with npm.

- If the package is only to be used once or twice, rather than every time the project runs, it is preferable to utilize NPX, which will execute the package without installing it.
- NPM is used to install packages, which we should do if our project requires dependencies or packages.

5: What is difference between **dependencies** vs **devDependencies** ?

- Dependencies should contain library and framework in which your app is built on, needs to function effectively. such as Vue, React, Angular, Express, JQuery and etc.
- DevDependencies should contain modules/packages a developer needs during development.
such as, parcel, webpack, vite, mocha etc. These packages are necessary only while you are developing your project, not necessary on production.
- To save a dependency as a devDependency on installation we need to do,

```
npm install --save-dev
```

instead of just,

```
npm install --save
```

6: What is Tree Shaking?

- Tree shaking is a term to describe the removal of dead/unused code.
- “You can imagine your application as a tree. The source code and libraries you actually use represent the green, living leaves of the tree. Dead code represents the brown, dead leaves of the tree that are consumed by autumn. In order to get rid of the dead leaves, you have to shake the tree, causing them to fall.”

- In JavaScript land, tree-shaking has been possible since the ECMAScript module (ESM) specification in ES2015, previously known as ES6. Since then, tree-shaking has been enabled by default in most bundlers because they reduce output size without changing the program's behaviour.

7: What is Hot Module Replacement?

- Hot Module Replacement (HMR) exchanges, adds, or removes modules while an application is running, without a full reload.
- This can significantly speed up development in a few ways: Retain application state which is lost during a full reload.
- HMR is enabled for the dev server. When you change your files and save them, the HMR will automatically change your contents without recompile and reload whole project.
- HMR is the same as Live Reload with the difference that it only replaces the modules that have been modified, hence the word Replacement.
- The advantage of this is that it doesn't lose your app state e.g. your inputs on your form fields, your currently selected tab etc.

8: Superpowers of Parcel and describe any 3 of them in your own words.

- Zero config: Parcel supports many languages and file types out of the box, from web technologies like HTML, CSS, and JavaScript, to assets like images, fonts, videos, and more. It has a built-in dev server with hot reloading, beautiful error diagnostics, and much more. No configuration needed!
- HMR (Hot Module Replacement): Adds/removes/Updates modules while an application is running, without a full reload.
- File watcher algorithm: File Watchers monitor directories on the file system and perform specific actions when desired files appear.
- Automatic production optimization: Parcel optimizes your whole app for production automatically. This includes tree-shaking and minifying your JavaScript, CSS, and

HTML, resizing and optimizing images, content hashing, automatic code splitting, and much more.

- Ship for any target: Parcel automatically transforms your code for your target environments. From modern and legacy browser support, to zero config JSX and TypeScript compilation, Parcel makes it easy to build for any target or many!
- Scalable: Parcel requires zero configuration to get started. But as your application grows and your build requirements become more complex, it's possible to extend Parcel in just about every way. A simple configuration format and powerful plugin system that's designed from the ground up for performance means Parcel can support projects of any size.

9: What is `.gitignore` ? What should we add and not add into it?

- A gitignore is a text file where each line contains a pattern for files or directories to ignore. It is usually placed at the root of the project folder.
- A gitignore file specifies intentionally untracked files that Git should ignore. Files already tracked by Git are not affected. The `.gitignore` file is a text file that tells Git which files or folders to ignore in a project during commit to the repository.
- The types of files you should consider adding to a `.gitignore` file are any files that do not need to get committed. for example, For security, the security key files and API keys should get added to the gitignore.

Note: `package-lock.json` should not add into your `.gitignore` file.

- You should not commit these four types of files into your Git repository.
 - Files that don't belong to the project
 - Files that are automatically generated
 - Libraries (depends on the situation)
 - Credentials
- The entries in this file can also follow a matching pattern.

* is used as a wildcard match

/ is used to ignore pathnames relative to the `.gitignore` file

```
# is used to add comments to a .gitignore file
```

This is an example of what the .gitignore file could look like:

```
# Ignore Mac system files
.DS_store

# Ignore node_modules folder
node_modules

# Ignore all text files
*.txt

# Ignore files related to API keys
.env

# Ignore SASS config files
.sass-cache
```

10: What is the difference between `package.json` and `package-lock.json`

- `package.json`:
 - The `package.json` file is the heart of any Node project.
 - It records important metadata about a project which is required before publishing to NPM, and also defines functional attributes of a project that npm uses to install dependencies, run scripts, and identify the entry point to our package.
 - The file resides in the root directory of every Node.js package and appears after running the `npm init` command.
 - The `package.json` file contains descriptive and functional metadata about a project, such as a name, version, and dependencies.
- `package-lock.json`:
 - This file is automatically generated for those operations where npm modifies either the `node_module` tree or `package.json`.
 - The “`package.json`” file defines the rules required to run your application and install dependencies. On the other hand, the “`package-lock.json`” file holds

detailed information on all the dependencies installed based on the package.

- It is generated after an npm install and not designed to be manually edited and we should not delete it either.
- As name suggests, lock. json is created for locking the dependency with the installed version. It will install the exact latest version of that package in your application and save it in package.
- It allows future devs & automated systems to download the same dependencies as the project.
- it also allows to go back to the past version of the dependencies without actual committing the node_modules folder.
- It records the same version of the installed packages which allows to reinstall them. Future installs will be capable of building identical description tree.

~ or ^ in `package.json` file :

These are used with the versions of the package installed.

For example in `package.json` file:

```
"dependencies": {  
  "react": "^18.2.0",  
  "react-dom": "^18.2.0"  
}
```

- ~ : “*Approximately equivalent to version*”, will update you to all future patch versions, without incrementing the minor version.
eg. [major, minor, patch] is a tuple of versions where ~1.2.3 will use releases from 1.2.3 to <1.3.0.
- ^ : “*Compatible with version*”, will update you to all future minor/patch versions, without incrementing the major version.
eg. ^2.3.4 will use releases from 2.3.4 to <3.0.0.

If none of them is present , that means only the version specified in package.json file is used in the development.

11: Why should I not modify `package-lock.json` ?

`package-lock.json` file contains the information about the dependencies and their versions installed in the project. Deleting them would cause dependency issues in the production environment. So you should not modify it as it's being handled automatically by NPM.

12: What is `node_modules` ? Is it a good idea to push that on git?

- `node_modules` folder like a cache for the external modules that your project depends upon.
- When you npm install them, they are downloaded from the web and copied into the `node_modules` folder and Nodejs is trained to look for them there when you import them (without a specific path).
- we should not push `node_modules` in github because it contains lots of files (more than 100 MB), it will cost you memory space.

13: What is the `dist` folder?

- The `/dist` stands for distributable.
- The `/dist` folder contains the minimized version of the source code.
- The code present in the `/dist` folder is actually the code which is used on production web applications.
- Along with the minified code, the `/dist` folder also comprises of all the compiled modules that may or may not be used with other systems.
- It is easier to add files to the `/dist` folder as it is an automatic process. All the files are automatically copied to the `dist` folder on save.
- The `/dist` folder also contains all those files which are required to run/build a module for use with other platforms- either directly in the browser, or in an AMD system (eg. `require.js`).
- Ideally, it is considered a good practice to clean the `/dist` folder before each build.

14: What is **browserslist** ?

- There is a package called 'browserlist' & parcel automatically gives to us.
- Browserslist makes our code compactible for a lot of browsers.
- Browserslist is a tool that allows specifying which browsers should be supported in your frontend app by specifying "queries" in a config file.
- In package.json file, do:

```
"browserslist": [  
  "last 3 versions"  
]
```

This means my parcel will make sure that my app works in last 3 versions of all the browsers available.

- Browserslist helps you keep the right balance between browser compatibility and bundle size. With Browserslist, you will cover wider audience and have smaller bundle size.



Assignment 03 - Laying The Foundation



Owner



Pankaj Kumar

1: What is **JSX** ?

- JSX is a syntax extension to JavaScript. It is a HTML-like syntax (but not HTML) in javascript.
- Facebook developers build JSX.
- JSX sanitizes your code.
- JSX converts HTML tags into react elements, it easier to write and add HTML in React.
- JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.
- After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

Example 1 - WITH JSX:

```
const element = (  
  <h1 className="greeting">  
    Hello, world - With JSX!  
  </h1>  
);  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(element);
```

Example 2 - WITHOUT JSX:

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world - Without JSX!'
);
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(element);
```

2: Superpowers of **JSX** .

- JSX allows us to write HTML-Like(not HTML) syntax in JavaScript.
- JSX uses **React.createElement** behind the scenes.
- JSX \Rightarrow React.createElement \Rightarrow javascript-Object \Rightarrow HTML(DOM).
- Using JSX, you can write markup inside Javascript, providing you with a superpower to write logic and markup of a component inside a single .jsx file. JSX is easy to maintain and debug.
- JSX is very secure & makes sure that the app is safe.
- JSX Prevents Injection Attacks : This is safe. `{}` \rightarrow acts like a sanitiser.

Example

```
const title = response.potentiallyMaliciousInput;
// JSX Prevents Injection Attacks : This is safe
const element = <h1>{title}</h1>;
```

Note: In this example, by default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

3: Role of **type** attribute in script tag? What options can I use there?

The **type** attribute specifies the type of the script. The type attribute identifies the content between the `<script>` and `</script>` tags. The type attribute gives the language

of the script or format of the data. It has a Default value which is “text/javascript”.

type attribute can be of the following types:

- `text/javascript` : It is the basic standard of writing javascript code inside the `<script>` tag.

Syntax

```
<script type="text/javascript"></script>
```

- `text/ecmascript` : this value indicates that the script is following the `EcmaScript` standards.
- `module` : This value tells the browser that the script is a module that can import or export other files or modules inside it.
- `text/babel` : This value indicates that the script is a babel type and required bable to transpile it.
- `text/typescript` : As the name suggest the script is written in `TypeScript` .

4: {TitleComponent} VS {<TitleComponent/>} VS {<TitleComponent></TitleComponent>} in JSX .

A: The Difference is stated below:

- `{TitleComponent}` : This value describes the `TitleComponent` as a javascript expression or a variable.
The `{}` can embed a javascript expression or a variable inside it.
- `<TitleComponent/>` : This value represents a Component that is basically returning Some JSX value. In simple terms `TitleComponent` a function that is returning a JSX value.
A component is written inside the `{< />}` expression.
- `<TitleComponent></TitleComponent>` : `<TitleComponent />` and `<TitleComponent></TitleComponent>` are equivalent only when `< TitleComponent />` has no child components. The opening and closing tags are created to include the child components.

Example

```
<TitleComponent>
  <FirstChildComponent />
  <SecondChildComponent />
  <ThirdChildComponent />
</TitleComponent>
```

5: What is tree shaking?

- Tree shaking is a term commonly used within a JavaScript context to describe the removal of dead code or unused code.
- Tree-shaking is an important way to reduce the size of your bundle and improve performance.
- It depends on the static syntax of import and export modules in ES6 (ES2015). By taking tree-shaking concepts into consideration when writing code, we can significantly scale down the bundle size by getting rid of unused JavaScript, thereby optimizing the application and increasing its performance.

6: What is React Element?

- React Element is finally an object.
- Elements are the smallest building blocks of React apps.
- An element describes what you want to see on the screen:

```
const element = <h1>Hello, world</h1>;
```

- Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.



Assignment 04 - Talk is Cheap, show me the code



Owner



Pankaj Kumar

1: Is **JSX** mandatory for React?

- JSX is not a requirement for using React.
- Using React without JSX is especially convenient when you don't want to set up compilation in your build environment.
- Each JSX element is just syntactic sugar for calling `React.createElement(component, props, ...children)`.
- So, anything you can do with JSX can also be done with just plain JavaScript.

Example of **JSX**

```
const header = <h1>Hello World!</h1>;
```

2: Is **ES6** mandatory for React?

- ES6 is not mandatory for **React** but is highly recommendable.
- ES6 is the standardization of javascript for making code more readable and more accessible.
- The latest projects created on React rely a lot on ES6. React uses ES6, and you should be familiar with some of the new features like: Classes, Arrow Functions, Variables(`let`, `const`).

- ES6 stands for ECMAScript 6. ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015.
- ES6 is the standardization of javascript for making code more readable and more accessible.

3: `{TitleComponent}` VS `{<TitleComponent/>}` VS `<TitleComponent></TitleComponent>` in JSX .

- `{TitleComponent}` : This value describes the `TitleComponent` as a javascript expression or a variable.
The `{}` can embed a javascript expression or a variable inside it.
- `<TitleComponent/>` : This value represents a Component that is basically returning Some JSX value. In simple terms `TitleComponent` a function that is returning a JSX value.
A component is written inside the `{< />}` expression.
- `<TitleComponent></TitleComponent>` : `<TitleComponent />` and `<TitleComponent></TitleComponent>` are equivalent only when `< TitleComponent />` has no child components. The opening and closing tags are created to include the child components.

Example

```
<TitleComponent>
  <FirstChildComponent />
  <SecondChildComponent />
  <ThirdChildComponent />
</TitleComponent>
```

4: How can I write `comments` in JSX?

JSX comments are written as follows:

- `{/* */}` - for single or multiline comments

Example

```
{/* A JSX comment */}  
{/*  
  Multi  
  line  
  JSX  
  comment  
*/}
```

5: What is `<React.Fragment> </React.Fragment>` and `<></>` ?

- `<React.Fragment> </React.Fragment>` is a A common pattern in React is for a component to return multiple elements.
- `<React.Fragment>` is a component exported by React.
- Fragments let you group a list of children without adding extra nodes to the DOM.
- `<></>` is the shorthand tag for `React.Fragment` .
- The only difference between them is that the shorthand version does not support the key attribute.

Example

```
return (  
  <React.Fragment>  
    <Header />  
    <Body />  
    <Footer />  
  </React.Fragment>  
);  
  
----- OR -----  
  
return (  
  <>  
    <Header />  
    <Main />  
    <Footer />  
  </>  
);
```

6: What is `Reconciliation` in React?

- Reconciliation is the process by which React updates the UI to reflect changes in the component state. The reconciliation algorithm is the set of rules that React uses to determine how to update the UI in the most efficient way possible.
- React uses a virtual DOM (Document Object Model) to update the UI. The virtual DOM is a lightweight in-memory representation of the real DOM, which allows React to make changes to the UI without manipulating the actual DOM. This makes updates faster, as changing the virtual DOM is less expensive than changing the real DOM.
- The reconciliation algorithm works by comparing the current virtual DOM tree to the updated virtual DOM tree, and making the minimum number of changes necessary to bring the virtual DOM in line with the updated state.
- The algorithm uses two main techniques to optimize updates:
 - *Tree diffing*: React compares the current virtual DOM tree with the updated virtual DOM tree, and identifies the minimum number of changes necessary to bring the virtual DOM in line with the updated state.
 - *Batching*: React batches multiple changes into a single update, reducing the number of updates to the virtual DOM and, in turn, the real DOM.
- The reconciliation algorithm is a critical part of React's performance and helps make React one of the fastest and most efficient JavaScript libraries for building user interfaces.
- After the reconciler compares the current and updated virtual DOM, it identifies the differences and makes the necessary changes to the virtual DOM to bring it in line with the updated state.

7: What is **React Fiber** ?

- React Fiber is a concept of ReactJS that is used to render a system faster, smoother and smarter.
- React Fiber is a backwards compatible, complete rewrite of the React core. In other words, it is a reimplement of older versions of the React reconciler.
- Introduced from React 16, Fiber Reconciler is the new reconciliation algorithm in React.

- Fiber brings in different levels of priority for updates in React. It breaks the computation of the component tree into nodes, or 'units' of work that it can commit at any time. This allows React to pause, resume or restart computation for various components.
- Fiber allows the reconciliation and rendering to the DOM to be split into two separate phases: Phase 1: Reconciliation and Phase 2: Commit.
- Since Fiber is asynchronous, React can:
 - Pause, resume, and restart rendering work on components whenever required.
 - Increases the suitability of the React library to create animations, layouts, and gestures.
 - Reuse previously completed work and even abort it if not needed.
 - Split work into chunks and prioritize tasks based on importance.

8: Why do we need **keys** in React? When do we need keys in React ?

- **key** is a special attribute you need to include when creating lists of elements in React.
- Keys helps React identify which items in the list have changed, are added, or are removed.
- In other words, we can say that keys are unique Identifier used to give an identity to the elements in the lists.
- The best way to pick a key is to use a string that uniquely identifies a list item among its siblings.
- When you don't have stable IDs for rendered items, you may use the item index as a key as a last resort.

Example

Recurring On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will match the two `first` trees, match the two `second` trees, and then insert the `third` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React will mutate every child instead of realizing it can keep the `Duke` and `Villanova` subtrees intact. This inefficiency can be a problem.

In order to solve this issue, React supports a key attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a key to our inefficient example above can make the tree conversion efficient:

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```



```
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Now React knows that the element with key '2014' is the new one, and the elements with the keys '2015' and '2016' have just moved.

9: Can we use **index as keys** in React?

Yes, we can use the **index as keys**, but React Developers don't recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state.

Keys are taken from each object which is being rendered. There might be a possibility that if we modify the incoming data react may render them in unusual order however when you don't have stable IDs for rendered items, you may use the item index as a key as a last option.

```
`NO key << INDEX as key <<<<<< Unique id as key from data`
```

10: What is **props in React** ? Ways to use props ?

- Props stand for "Properties." They are read-only components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.
- Props are immutable so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as props and can be used to render dynamic data in our render method.
- Props are used to store data that can be accessed by the children of a React component. They are part of the concept of reusability. Props take the place of class attributes and allow you to create consistent interfaces across the component hierarchy.

- Props act as a channel for component communication. Props are passed from parent to child and help your child access properties that made it into the parent's tree.

Every parent component can pass some information to its child components by giving them props. Props are similar to HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions.

Types of Props :

- Familiar Props —> HTML attributes like className, src, width, height passed in HTML tag
- Passing Props to Component —> props are the only argument to your component. React component functions accept a single argument, a props object.

Ways to pass props to component	Ways to receive the props in another component
1. Add props to the JSX, just like you would with HTML attributes	All props are sent into a single props object
<pre><Profile name = { "Hello" } age={28} /></pre>	<pre>const Profile = (props) => { let name = props.name; let age = props.age; }</pre>
2. Similar to the way mentioned in 1.	Props object can be destructured using {} to receive only the required props
<pre><Profile name = { "Hello" } age={28} /></pre>	<pre>const Profile = ({name, age}) => { }</pre>
3. Using spread syntax	And props objects destructured using {}
<pre><Profile {...props} /></pre>	<pre>const Profile = ({name, age}) => { }</pre>

However, props are immutable which means unchangeable. When a component needs to change its props (for example, in response to a user interaction or new data), it will have to “ask” its parent component to pass it different props—a new object! Its old props will then be cast aside, and eventually the JavaScript engine will reclaim the memory taken by them.

11: What is Config Driven UI ?

- Config-driven UI is one of the UI design pattern in which the UI is rendered based on the configuration parameter sent by the server (backend). This is one of the popular pattern used in the industry now.
- **Config Driven UI** are based on the configurations of the data application receives. It is rather a good practice to use config driven UIs to make application for dynamic.
- It is a very common & basic approach to interact with the User. It provides a generic interface to develop things which help your project scale well. **It saves a lot of development time and effort.**
- Config driven UI not only make your code cleaner but also easier to maintain.
- You can any time change/add the validations in the config (Like maxValuue , or add a regex pattern, etc..) and change the renderer function to accept those keys.

12: Difference between **Virtual DOM** and **Real DOM** ?

DOM stands for **Document Object Model** , which represents your application UI and whenever the changes are made in the application, this DOM gets updated and the user is able to visualize the changes. DOM is an interface that allows scripts to update the content, style, and structure of the document.

- **Virtual DOM**
 - The Virtual DOM is a light-weight abstraction of the DOM. You can think of it as a copy of the DOM, that can be updated without affecting the actual DOM. It has all the same properties as the real DOM object, but doesn't have the ability to write to the screen like the real DOM.
 - Virtual DOM is just like a blueprint of a machine, can do the changes in the blueprint but those changes will not directly apply to the machine.
 - Reconciliation is a process to compare and keep in sync the two files (Real and Virtual DOM). Diff algorithm is a technique of reconciliation which is used by React.
- **Real DOM**
 - The DOM represents the web page often called a document with a logical tree and each branch of the tree ends in a node and each node contains object programmers can modify the content of the document using a scripting

language like javascript and the changes and updates to the dom are fast because of its tree-like structure but after changes, the updated element and its children have to be re-rendered to update the application UI so the re-rendering of the UI which make the dom slow all the UI components you need to be rendered for every dom update so real dom would render the entire list and not only those item that receives the update .

Real DOM	Virtual DOM
DOM manipulation is very expensive	DOM manipulation is very easy
There is too much memory wastage	No memory wastage
It updates Slow	It updates quickly
It can directly update HTML	It can't update HTML directly
Creates a new DOM if the element updates	Update the JSX if the element update
It allows us to directly target any specific node (HTML element)	It can produce about 200,000 Virtual DOM Nodes / Second
It represents the UI of your application	It is only a virtual representation of the DOM
It is not light-weight.	It is light-weight abstraction of DOM.



Assignment 05 - Let's get Hooked



Owner



Pankaj Kumar


1: What is the difference between Named export , Default export and as export?

- **Named export** - In named export, the function is exported as:

```
export <function>;
```

and imported as:

```
import { <function> } from '..';
```

. The function is exported inside  to other modules.

- **Default export**: In default export the function is exported as:

```
export default <function>
```

and imported as:

```
import <function> from '...'
```

- **as export**: * as export is used to import the whole module as a component and access the components inside the module.

for example: We have a module ABC.js and some components inside of it and a file named XYZ.js where we want to import the components.

ABC.js:

```
export const Comp1 = () => {...}
export const Comp2 = () => {...}
export const Comp3 = () => {...}
```

in **XYZ.js** we'll import then as

```
import * as Module from 'ABC.js'
```

Now we can use them JSX as:

```
<Module.Comp1 />
<Module.Comp2 />
<Module.Comp3 />
```

2: What is the importance of config.js file?

- `config.js` file is used for the hard coded values used in our application. We can use it import a configuration inside any component without having to copy it over and over again.

3: What are React Hooks?

- `React Hooks` are basically JavaScript functions which are provided by React. Hooks have some special capabilities that are useful for the development. like managing state, memory etc.

▼ **React provides a bunch of standard in-built hooks:**

- `useState` is a React Hook that lets you add a state variable to your component.
`const [state, setState] = useState(initialState);`
- `useEffect` is a React Hook that lets you synchronize a component with an external system.

(OR) To manage side-effects like API calls, subscriptions, timers, mutations, and more.

```
useEffect(setup, dependencies?)
```

- **useContext** is a React Hook that lets you read and subscribe to context from your component.

```
const value = useContext(SomeContext)
```

- **useReducer** is a React Hook that lets you add a reducer to your component.

(OR) A useState alternative to help with complex state management.

```
const [state, dispatch] = useReducer(reducer, initialArg, init?)
```

- **useMemo** is a React Hook that lets you cache the result of a calculation between re-renders.

(OR) It returns a memoized value that helps in performance optimizations.

```
const cachedValue = useCallback(calculateValue, dependencies)
```

- **useCallback** is a React Hook that lets you cache a function definition between re-renders.

```
const cachedFn = useCallback(fn, dependencies)
```

(OR) It returns a memorized version of a callback to help a child component not re-render unnecessarily.

- **useRef** is a React Hook that lets you reference a value that's not needed for rendering.

```
const ref = useRef(initialValue)
```

(OR) It returns a ref object with a current property. The ref object is mutable. It is mainly used to access a child component imperatively.

- **useLayoutEffect** is a version of **useEffect** that fires before the browser repaints the screen.

```
useLayoutEffect(setup, dependencies?)
```

(OR) It fires at the end of all DOM mutations. It's best to use **useEffect** as much as possible over this one as the **useLayoutEffect** fires synchronously.

- **useDebugValue** is a React Hook that lets you add a label to a custom Hook in React DevTools.

```
useDebugValue(value, format?)
```

- `useId` is a React Hook for generating unique IDs that can be passed to accessibility attributes.

```
const id = useId()
```

- `useTransition` is a React Hook that lets you update the state without blocking the UI.

```
const [isPending, startTransition] = useTransition()
```

4: Why do we need useState Hook?

- `useState` hook is used to maintain the state in our React application. It keeps track of the state changes. When a component is re-rendered it changes the state of our component.



Assignment 06 - Exploring the world



Owner



Pankaj Kumar

1: What is **Microservice** ?

Microservice - also known as the microservice architecture - is an architectural method that relies on a series of **Independently deployable services**. These services have their own business logic and database with a specific goal. **Updating, testing, deployment, and scaling** occur within each service. Microservices decouple major business, domain-specific concerns into separate, independent code bases.

- Benefits of Microservices:
 - Easier to Test
 - Flexible Scaling
 - Easy Deployment
 - Technological Freedom
 - Reusable Code
 - Resilience

2: What is **Monolith architecture** ?

A **Monolith architecture** is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications. A monolithic architecture is **a singular, large computing network with one code base that couples all of the business concerns together**. To make a change or a small change in it (We need to deploy the whole project init. For instance we changed a button we use deploy our Whole project.) to this sort of application requires **updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface**. This makes updates restrictive and time-consuming.

Means we are not dividing software into small, well-defined modules, we use every services like, database, server or a UI of the application, in one Application file.

▼ **Benefits of “Monolith architecture”:**

1. **Easy deployment:** One executable file or directory makes deployment easier.
2. **Development:** When an application is built with one code base, it is easier to develop.
3. **Performance** In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
4. **Simplified testing:** Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application. ****Easy debugging**** – With all code located in one place, it's easier to follow a request and find an issue.

3: What is the **difference** between `Monolith and Microservice?

Monolith	Microservice
Every service is inside the application	Services are scattered
Single code base	Code base is divided into separated applications
Hard to maintain	Easy to maintain
Deployment takes more time	Deployment is easy

Parameters	Monolith Architecture	Microservices Architecture
Development	When an application is built with one code base, it is easier to develop. This is true for small applications, but when the application takes larger, development becomes slower and complex.	Micro services add more complexity compared to monolith arch. If development sprawl isn't properly managed, it results in slower development speed and poor operational performance.

Parameters	Monolith Architecture	Microservices Architecture
Testing	Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.	Teams can experiment with new features and roll back if something doesn't work. This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services.
Performance	In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.	Though performance could be an issue in microservices, it could be over come by various performance optimisation techniques.
Debugging	With all code located in one place, it's easier to follow a request and find an issue.	Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.
Scalability	You can't scale individual components.	If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure.
Reliability	If there's an error in any module, it could affect the entire application's availability.	You can deploy changes for a specific service, without the threat of bringing down the entire application.
Tech Adoption	Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.	Any new tech changes can easily be adopted as an independent service.
Deployment	One executable file or directory makes deployment easier. But, a small change to a monolithic application requires the redeployment of the entire monolith.	Microservices make it easier for teams to update code and accelerate release cycles with continuous integration and continuous delivery (CI/CD).
Agility	There is no agility in monolith.	Promote agile ways of working with small teams that deploy frequently.

4: Why do we need `useEffect` Hook ?

`useEffect` Hook is javascript function provided by `react`. The `useEffect` Hook allows you to `eliminate side effects` in your components. Some examples of side effects are: `fetching API data`, `directly updating the DOM`, and `setting up subscriptions or timers`, etc can be lead to unwarranted side-effects.

`useEffect` accepts `two arguments`, first is a `callback function` and a `dependency array`. The second argument is optional.

Syntax

```
`useEffect(() => {}, [])`
```

The `() => {}` is callback function and `[]` is called a empty dependency array. If anything that we pass (suppose `currentState`) inside the `[]` it trigger the callback function and changes the state of the application.

```
useEffect(() => {  
  setCurrentState("true");  
}, [currentState])
```

If we do not pass empty dependency array then the `useEffect` runs everytime when the UI is rendered.

```
useEffect(() => {})
```

5: What is `Optional Chaining` ?

`Optional Chaining` (`?.`) operator accesses an object's property or calls a function. If the object accessed or function called is `undefined or null`, it returns `undefined` instead of throwing an error.

`Optional Chaining` (`?.`) is good way of accessing the object keys, it prevents the application from being crashed if the key that we are trying to access is not present. If the key is not present then instead of a throwing key error, it returns `undefined`.

6: What is `Shimmer UI` ?

A shimmer screen is a version of the UI that doesn't contain actual content. Instead, it mimics the page's layout by showing its elements in a shape similar to the actual

content as it is loading and becoming available (i.e. when network latency allows).

A shimmer screen is essentially a wireframe of the page, with placeholder boxes for text and images.

7: What is the **difference** between **JS expression** and **JS statement** ?

JS Expression - A js expression returns a value. A number, string, ternary conditions (return true or false), math operations and array map method (returns new array) are all examples of js expression.

1. "Pankaj" ----> String is a valid expression
2. 1234 ----> Number is a valid expression
3. (isLoggedIn) ? "Logout" : "Login" ----> Ternary operator returning value is a valid expression
4. [1,2,3,4].map(num => num*2)----> Array map function is a valid expression which returns a new array after transformation
5. (1+2+3) ---> Math operation is a valid expression

JS Statement - A js statement just executes/performs an action but does not return/produce a value. A variable assignment, if condition (with no return) and for loops are examples of js statement.

1. console.log("This is a js statement") ----> This does not return any value, just prints the content on screen.
2. let name = "Pankaj"; ----> Variable assignment is a statement
3.

```
if(true){
  console.log("true");
} else {
  console.log("true");
}
```

 ----> This does not return any value
4.

```
for(let i=0; i< 5; i++) {
  arr.push(i);
}
```

Having said that, we can't put any js code inside {} in jsx of React. Only **js expressions** can be enclosed within {} of jsx.

If we want to use `JS expression` in JSX, we have to wrap in `{/* expression slot */}` and if we want to use `JS statement` in JSX, we have to wrap in `{(/* statement slot */)}`.

8: What is **Conditional Rendering** ? explain with a code example.

A conditional rendering is a way of rendering components based on the a state. If the condition is true for a component, then it gets rendered; otherwise, the other component is rendered.

For example : We load a shimmer UI before our component is loaded completely. We can create a state variable that will keep the value of our current application state. i.e.

```
const [isLoading, setIsLoaded] = useState(false)
```

In the above example, we are creating a state variable that is initially set to false, since our data has not been loaded in our application yet.

Untill our data is loaded completely we can show a shimmer UI to the user and when our data gets loaded we can render the data on the page. The conditional rendering is done via a ternary operator `?:` for example :

```
isLoading ? <Body /> : <Shimmer />
```

In the above expression, if `isLoading` is set to `false` then, `Shimmer` component will be loaded , when the data loading is completed, the `Body` component will be rendered.

- `&&` operator : if the condition is true, display the right-side code else display nothing.

```
{ errorMsg && {errorMsg} }
```

9: What is **CORS** ?

CORS stands for *Cross Origin Resource Sharing* , It is a HTTP-header based machanism that allows a server to indicate any origin other that it's own. We can create requests to other domains or ports to get the data from our browser.

(OR)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "pre-flight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that pre-flight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

10: What is **async and await** ?

Async/await are keywords to make a normal function behave like a asynchronous function.

async function always returns a promise, any values are automatically wrapped inside a resolved promise.

await: Await function is used to wait for the promise to be settled. It could be used within the 'async' block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

For example : Let's try to write a function `getRestaurants()` to fetch restaurant data from a public API.

First, let's try to write it with **Promise chaining** : `fetch(url)` returns a promise (resolve or reject), which can be consumed by the **then** (success) handler or **catch** (error) handler

```
function getRestaurants() {
  fetch(url).then((data)=>{data.json()})
    .then((json)=>{
      console.log(json);
    }).catch((err)=>{
      console.log(err);
    })
}
```

Using **async** and **await** : `await` waits until `fetch(url)` returns a promise with the data and headers which again needs to be resolved using `.json()` method to get the data. If any of promise inside try block is rejected, the control jumps to catch block.

```
async function getRestaurants() {
  try {
    const data = await fetch(url);
    const json = await data.json();
    console.log(json);
  } catch(err) {
    console.log(err);
  }
}
```

11: What is the use of `const json = await data.json();` in `getRestaurants()` ?

The `data` object, returned by the `await fetch()`, is a generic placeholder for multiple data formats.

so we can extract the `JSON object` from a `fetch` response by using `await data.json()`. `data.json()` is a method on the data object that lets you extract a `JSON object` from the data or response. The method returns a promise because we have used `await` keyword.

so `data.json()` returns a promise resolved to a `JSON object`.



Assignment 07 - Finding the Path

Owner  Pankaj Kumar

1: What are various ways to add images into our App? Explain with code examples

- Using the full URL of the image for the web (CDN) or any public images.

Example : ``

- Adding the image into the project Drag your image into your project and import it into the desired component

```
import reactLogo from "../reactLogo.png";

export default function App() {

  return <img src={reactLogo} alt="react logo" />
}
```

- The correct way to structure images in your project is to add them in an `images` folder. If you are using other `assets` than just images, you might want to add all the `assets` folders.

```
import reactLogo from "../../assets/images/reactLogo.png";

export default function App() {

  return <img src={reactLogo} alt="react logo" />
}
```

2: What would happen if we do `console.log(useState())`?

If we do `console.log(useState())`, we get an array `[undefined, function]` where first item in an array is `state` is `undefined` and the second item in an array

is `setState` function is bound `dispatchSetState`.

3: How will `useEffect` behave if we don't add a dependency array ?

Syntax : `useEffect(setup,[dependencies]?)`

`useEffect(setup)` When the dependency array is not included in the arguments of `useEffect()` hook, the setup function will be executed `every time` the component is rendered and re-rendered.

4: What is SPA?

`Single Page Application (SPA)` is a web application that dynamically updates the webpage with data from web server without reloading/refreshing the entire page. All the HTML, CSS, JS are retrieved in the initial load and other data/resources can be loaded dynamically whenever required. An SPA is sometimes referred to as a `single-page interface (SPI)`.

Example : Facebook is a Single Page Application which loads lot of components and refreshes only the required component.

5: What is difference between Client Side Routing and Server Side Routing?

- In `Server-side routing or rendering (SSR)`
 1. All our Pages load from server everytime.
 2. Make a network call, get HTML, CSS, JS and loads the whole page.
- In `Client-side routing or rendering (CSR)`, during the first load, the webapp is loaded from server to client, after which whenever there is a change in URL, the router library navigates the user to the new page without sending any request to backend(don't do full page reload). All `Single Page Applications uses client-side routing`.

6: What is an Image CDN?

Image CDNs specializes in the transformation, optimization, and delivery of images. You can also think of them as APIs for accessing and manipulating the images used on your site. For images loaded from an image CDN, an image URL indicates not

only which image to load, but also parameters like size, format, and quality. This makes it easy to create variations of an image for different use cases.



Assignment 08 - Let's get Classy



Owner



Pankaj Kumar

1: How do you create **Nested Routes react-router-dom** configuration?

We can create a `Nested Routes` inside a react router configuration as follows:

first call `createBrowserRouter` for routing different pages

```
const router = createBrowserRouter([
  {
    path: "/", // show path for routing
    element: <Parent />, // show component for particular path
    errorElement: <Error />, // show error component for path is different
    children: [ // show children component for routing
      {
        path: "/path",
        element: <Child />
      }
    ],
  }
])
```

Now we can create a nested routing for `/path` using `children` again as follows:

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Parent />,
    errorElement: <Error />,
    children: [
      {
        path: "/about",
        element: <About />,
      }
    ]
  }
])
```

```

    children: [
      {
        path: "profile", // // nested routing for subchild
        element: <Profile />,
      },
    ],
  },
]
}
]
})

```

2: Read about `createHashRouter` , `createMemoryRouter` from React Router docs.

`createHashRouter` is useful if you are unable to configure your web server to direct all traffic to your React Router application. Instead of using normal URLs, it will use the `hash (#)` portion of the URL to manage the "application URL". Other than that, it is functionally the same as `createBrowserRouter` .

`createMemoryRouter` Instead of using the browsers history a memory router manages it's own history stack in memory. It's primarily useful for testing and component development tools like Storybook, but can also be used for running React Router in any non-browser environment.

3: What is the `order of life cycle method calls` in `Class Based Components` ?

Following is the order of lifecycle methods calls in `Class Based Components` :

1. constructor()
2. render ()
3. componentDidMount()
4. componentDidUpdate()
5. componentWillUnmount()

For more reference: [React-Lifecycle-methods-Diagram](#)

▼ Explanation:-

Class based components are executed in two phases : Render phase & commit phase.

Render phase is pure and no side effects. It may be paused, restarted or aborted by React (when child component is created for eg). The constructor(), render() and componentDidMount() happens in this phase.

In constructor, the props are passed to its parents.

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

Mounting :

1. constructor - The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.
 - Initializing local state by assigning an object to this.state
 - Binding event handler methods to an instance.

Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead.

1. componentDidMount() - componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). You may call setState() immediately in componentDidMount() so that it triggers re-render before the browser updates the screen.

Updating : 3. componentDidUpdate() - componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Unmounting : 4. componentWillUnmount() - componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

4: Why do we use `componentDidMount` ?

`componentDidMount` - In CBC it is the best place to make an API call. Like we make an API call inside `useEffect` in functional component. Bcz initially React first finishes the render() Phase and it updates the DOM, Then it makes an API call (it needs to load some data). So it takes some time to load and also we use `componentDidMount` as an `async function` so it delays the component to print. It is called after initial or every render. Example:

```
async componentDidMount() {
  //This is the best place we make an API call
  const data = await fetch("https://api.github.com/users/sam-0905");
  const json = await data.json();
  this.setState({
    userInfo: json,
  });
  console.log("userInfo", json);
  console.log("componentDidMount");
}
```

5: Why do we use `componentWillUnmount` ? Show with `example` .

`componentWillUnmount` is useful for the cleanup of the application when we switch routes from one place to another. Since we are working with a SPA the component process always runs in the background even if we switch to another route. So it is required to stop those processes before leaving the page. If we revisit the same page, a new process starts that affects the browser performance.

For example, in Repo class, during `componentDidMount()` a timer is set with an interval of every one second to print in console. When the component is unmounted (users moves to a different page), the timer will be running in the background, which we might not even realise and causing huge performance issue. To avoid such situations the cleanup function can be done in `componentWillUnmount`, in this example

`clearInterval` (timer) to clear the timer interval before unmounting Repo component.

6: (Research) Why do we use `super(props)` in `constructor` ?

`super(props)` is used to inherit the properties and access of variables of the React parent class when we initialize our component.

`super()` is used inside constructor of a class to derive the parent's all properties inside the class that extended it. If `super()` is not used, then `Reference Error : Must call super constructor in derived classes before accessing 'this' or returning from derived constructor` is thrown in the console.

A component that extends `React.Component` must call the `super()` constructor in the derived class since it's required to access this context inside the derived class constructor.

When you try to use props passed on parent to child component in child component using `this.props.name`, it will still work without `super(props)`. Only `super()` is also enough for accessing props in render method.

The main difference between `super()` and `super(props)` is the `this.props` is undefined in child's constructor in `super()` but `this.props` contains the passed props if `super(props)` is used.

7: (Research) Why `can't we have the callback function of useEffect async` ?

`useEffect` expects it's callback function to return nothing or return a function (cleanup function that is called when the component is unmounted). If we make the callback function as `async`, it will return a `promise` and the promise will affect the clean-up function from being called.

Solution to this is not making the callback function `async` but created another `async` function inside callback function of `useEffect()`.



Assignment 09 - Optimizing our App



Owner



Pankaj Kumar

1: When and why do we need lazy()?

- The `lazy()` function is a feature introduced in React 16.6 that allows for the lazy loading of components.
- The `lazy()` function is typically used in scenarios where you have large or less frequently used components that you want to load `asynchronously`.

▼ A few situations when you might need to use `lazy()` are:

1. Large component bundles: If your application has large components or dependencies, loading them synchronously during the initial render can cause significant delays. By using `lazy()` along with code splitting, you can split these components into separate chunks and load them only when necessary, improving the overall performance of your application.

2. Infrequently accessed routes: If you have certain routes or pages in your application that are rarely accessed, it might be inefficient to load their associated components upfront. Using `lazy()` allows you to lazily load these components when the specific route is visited, reducing the initial bundle size and improving the application's initial load time.

3. Enhancing performance: By employing `lazy()` and code splitting, you can ensure that only the necessary components are loaded upfront, while the rest are loaded on-demand. This approach helps reduce the initial bundle size and improves performance by minimizing the amount of JavaScript that needs to be downloaded and executed during the initial page load.

2: What is suspense?

- Suspense is a component that helps manage the loading state of dynamic imports, such as lazily loaded components, and provides a fallback UI to display while the requested content is being loaded. It enables a better user experience by showing a loading indicator or placeholder content until the desired component or data is ready to be rendered..
- Suspense component allows us to show some fallback content (such as a loading indicator/ Shimmer component) while we're waiting for the lazy component to load or the component is not yet rendered. It is similar to `catch` block. If a component suspends, the closest `Suspense` component above the suspending component `catches` it

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

const About = React.lazy(() => import('./About'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <About />
      </Suspense>
    </div>
  );
}
```

- The `fallback` prop accepts any `React elements` that you want to render while waiting for the component to load. You can place the Suspense component anywhere above the lazy component. You can even wrap `multiple lazy components` with a `single` Suspense component.

3: Why we got this error : A component suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix, updates that suspend should be wrapped with startTransition? How does suspense fix this error?

This error is thrown as Exception by React when the promise to dynamically import the lazy component is not yet resolved and the Component is expected to render in the meantime. If only the dynamic import is done and there is no `<Suspense />` component then this error is shown. React expects a Suspense boundary to be in place for showing a fallback prop until the promise is getting resolved. If showing the shimmer (loading indicator) is not desirable in some situations, then `startTransition` API can be used to show the old UI while new UI is being prepared. React does this without having to delete or remove the Suspense component or its props from your code.

4: Advantages and disadvantages of using this code splitting pattern?

Advantages	Disadvantages
1. Improved performance: Code splitting allows you to load only the necessary parts of your application when they are needed, reducing the initial bundle size. This results in faster load times and improved overall performance, particularly for larger applications.	1. Increased complexity: Code splitting adds complexity to your application's architecture, requiring you to manage and coordinate the loading of different code chunks. This can make the development process more challenging and may require additional tooling and configuration.
2. Faster initial load: By deferring the loading of non-essential code until it is actually required, code splitting can significantly reduce the time it takes for the initial page to load. This can greatly enhance the user experience, especially for users with slower internet connections or on mobile devices.	2. Potential for suboptimal user experience: If code splitting is not implemented carefully, it can lead to suboptimal user experiences. Poorly optimized code splitting may result in visible loading delays or multiple small network requests, which can negatively impact the perceived performance of your application.
3. Better resource utilization: With code splitting, you can optimize resource utilization by loading code chunks only when they are needed. This can reduce memory consumption and improve the efficiency of your application.	3. Additional network requests: Code splitting can lead to an increase in the number of network requests made by your application. While these requests may be smaller in size, the total number of requests may still impact the overall load time, especially in scenarios with slower network connections.

Advantages	Disadvantages
<p>4. Enhanced caching: Code splitting enables better caching and reusability of code. Once a code chunk is loaded, it can be cached by the browser, allowing subsequent visits to your application to benefit from faster load times.</p>	<p>4. Compatibility concerns: Code splitting relies on newer web standards and features, such as dynamic imports and ES modules. While these features are widely supported in modern browsers, older browsers may not fully support them, potentially leading to compatibility issues.</p>
<p>5. Smoother user interaction: By loading code asynchronously and showing loading indicators or fallback content during the loading process, code splitting provides a more seamless user experience. It prevents the entire application from freezing or becoming unresponsive while waiting for large chunks of code to load.</p>	<p>5. Build and deployment complexity: Implementing code splitting may require additional build and deployment configurations and tools. This can introduce complexity to your development workflow and may require learning and integrating new tools into your existing build process.</p>

5: When do we and why do we need suspense?

- **Lazy loading components:** When you want to load components lazily or on-demand, Suspense allows you to specify fallback content that will be displayed while the component is being loaded. This helps improve the user experience by showing a loading indicator or placeholder content until the component is ready to render.
- **Data fetching:** If your application needs to fetch data from an API or perform asynchronous operations, Suspense can be used to handle the loading state and display fallback content until the data is available. This simplifies the management of loading states and ensures a smooth transition between loading and displaying the data.
- **Code splitting:** Code splitting involves breaking your application's code into smaller chunks that can be loaded separately. When using dynamic imports and code splitting techniques, Suspense can be used to wrap the components that are being lazily loaded. It allows you to provide fallback content while the code chunks are being fetched and loaded, improving the perceived performance of your application.
- **Concurrent mode (experimental):** React's concurrent mode, introduces new features like rendering fallback content for slow components and prioritizing

updates. Suspense plays a crucial role in managing concurrent rendering and helps coordinate the rendering and fallback states of components.



Assignment 10 - Jo dikhta hai, vo bikta hai



Owner



Pankaj Kumar

1: Explore all the ways of writing css.

There are different ways of using CSS in our application.

1. Writing normal Native CSS : By creating `.css` files and adding them to our `html` files
2. By using CSS preprocessors like Sass or SCSS.
3. By writing inline CSS : writing CSS inside the components itself.
4. Using Libraries like Chakra UI, Tailwind CSS etc.

2: How do we configure Tailwind?

For configuring the Tailwind CSS with Parcel.

1. we need to install Tailwind library.

```
npm install -D tailwindcss postcss
```

1. Then we need to setup the config file for Tailwind.

```
npx tailwindcss init
```

This will create a `tailwind.config.js` file.

3: In tailwind.config.js, what does all the keys mean (content, theme, extend, plugins)?

The `tailwind.config.js` file contains the configurations for the our application. Following are the configurations we require to setup tailwind.

1. `content` : This configuration of the file formats, on which the styles are applied.

e.g.

```
content: [  
  "./src/**/*.html,js,ts,jsx,tsx",  
],
```

This configuration means that in files extensions of `html , js, ts ,tsx, jsx` files will use the stylings of Tailwind.

2. `theme` : Theme is where we design our own custom configurations for our project like colors and font-families for our application.

```
module.exports = {  
  theme: {  
    screens: {  
      sm: '480px',  
      md: '768px',  
      lg: '976px',  
      xl: '1440px',  
    },  
    colors: {  
      'blue': '#1fb6ff',  
      'purple': '#7e5bef',  
      'pink': '#ff49db',  
    },  
  },  
}
```

3. `extend` : Here we can extend more properties for Tailwind like adding values that does not exist for tailwind or overriding the existing the values for tailwind.

```
extend: {  
  spacing: {  
    '128': '32rem',  
  },  
}
```

```
    '144': '36rem',
  },
  borderRadius: {
    '4xl': '2rem',
  }
}
```

4. **plugins** : We can use to inject new style in our project using JavaScript instead of CSS.

```
const plugin = require('tailwindcss/plugin')
```

```
module.exports = {
  plugins: [
    plugin(function({ addUtilities, addComponents, e, config }) {
      // Add your custom styles here
    }),
  ]
}
```

4: Why do we have **.postcssrc** file?

It is just a transpiler that turns a special PostCSS plugin syntax into a Vanilla CSS. You can think of it as the Babel tool for CSS. Which will contain the PostCSS settings.



Assignment 11 - Data is the new oil



Owner



Pankaj Kumar

1: What is prop drilling?

- Prop drilling is a concept in React where data is passed down from a parent component to nested child components through props. It refers to the process of passing data through multiple layers of components in order to reach a deeply nested child component that needs access to that data.
- In a React application, components are organized in a hierarchical tree structure. Data typically flows from parent components to their child components through props. When a piece of data is required by a component deep down the component tree, it needs to be passed through all the intermediate parent components that do not directly use the data. This process of passing data down through several levels of components is known as prop drilling.

```
...  
import React from 'react';  
import ChildComponentA from './ChildComponentA';  
const ParentComponent = () => {  
  const data = "Hello, Prop Drilling!";  
  return (  
    <div>  
      <ChildComponentA data={data} />  
    </div>  
  );  
};  
...  
  
...  
// ChildComponentA.js
```

```

import React from 'react';
import ChildComponentB from './ChildComponentB';
const ChildComponentA = ({ data }) => {
  return (
    <div>
      <ChildComponentB data={data} />
    </div>
  );
};
export default ChildComponentA;
...

...

// ChildComponentB.js
import React from 'react';
import ChildComponentC from './ChildComponentC';
const ChildComponentB = ({ data }) => {
  return (
    <div>
      <ChildComponentC data={data} />
    </div>
  );
};
export default ChildComponentB;
...

...

// ChildComponentC.js
import React from 'react';
const ChildComponentC = ({ data }) => {
  return <div>{data}</div>;
};
export default ChildComponentC;
...

```

In this example, the data prop is passed from the ParentComponent down to ChildComponentA, then from ChildComponentA down to ChildComponentB, and finally from ChildComponentB to ChildComponentC. This pattern continues if we need to pass the data further down to more child components.

2: What is lifting the state up?

- Lifting the state up is a concept in React where the state data is moved from a lower-level component to a higher-level component in the component tree. This is done to share and manage the state data at a common ancestor, making it accessible to multiple child components that need to interact with the same data.

- In React, each component manages its own state data. However, when two or more child components need to share the same state or need to synchronize their data, it becomes beneficial to lift the state up to a common parent component. By doing so, the parent component becomes the "single source of truth" for the shared state, and any changes to the state will propagate down to all child components that use the state.
- Here's an example to illustrate lifting the state up:

```

...
// ParentComponent.js
import React, { useState } from 'react';
import ChildComponentA from './ChildComponentA';
import ChildComponentB from './ChildComponentB';
const ParentComponent = () => {
  const [count, setCount] = useState(0);
  const incrementCount = () => {
    setCount(count + 1);
  };
  return (
    <div>
      <ChildComponentA count={count} incrementCount={incrementCount} />
      <ChildComponentB count={count} />
    </div>
  );
};
export default ParentComponent;
...

...
// ChildComponentA.js
import React from 'react';
const ChildComponentA = ({ count, incrementCount }) => {
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
};
export default ChildComponentA;
...

...
// ChildComponentB.js
import React from 'react';
const ChildComponentB = ({ count }) => {
  return (
    <div>

```

```

    <p>Count: {count}</p>
  </div>
);
};
export default ChildComponentB;
```

```

- In this example, the state variable `count` and the function `incrementCount` are declared in the `ParentComponent`, which serves as the common parent for `ChildComponentA` and `ChildComponentB`. The count state is lifted up to `ParentComponent` and passed down as a prop to both `ChildComponentA` and `ChildComponentB`. When `ChildComponentA` increments the count using the `incrementCount` function, the updated count will be reflected in both `ChildComponentA` and `ChildComponentB`, as they share the same state.
- Lifting the state up promotes better data flow, simplifies state management, and helps to avoid inconsistencies in the application state. It is particularly useful when components need to interact with each other or when multiple components depend on the same data source.

### 3: What is Context Provider and Context Consumer?

1. **Context Provider** is used to provide access to a context between multiple components of the application. We can provide the access to the context or the data layer to the whole application to the and its subcomponents.

For example:

```

return (
 <UserContext.Provider value={{ user: user, setUser: setUser }}>
 <Head />
 <Outlet />
 <Footer />
 </UserContext.Provider>
);

```

In the code above , we are providing the access of `UserContext` to `<Head />` `<Outlet />` and `<Footer />` component

2. **Context Consumer** is used to consume the context data , provided by react context. We can do this using `useContext` hook for functional components

and `Context.Consumer` in class based components.

For example:

- In Class based components, we can use the Context and use the consumer.

```
<UserContext.Consumer>
 ({ { user } }) => <h1>{user.name}</h1>
</UserContext.Consumer>
```

- In Functional components we can use `useContext` hook to consume the context.

```
import UserContext from "../utils/UserContext";
import { useContext } from "react";

const { user } = useContext(UserContext);
```

## 4: If you don't pass a value to the provider does it take the default value?

If we do not override the values of context it takes the default values from the context, when we initialise the context.



# Assignment 12 - Let's build our Store



Owner



Pankaj Kumar

## 1: useContext vs Redux.

Both useContext and Redux are used to solve **props drilling**, a problem faced while passing props between components.

| Context API                                                                                                                                                             | Redux                                                                                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Context</b> provides a way to share values between components (throughout the application) without having to explicitly pass a prop through every level of the tree. | Redux is a central store for storing the data of the applications.                                                                                   |
| Context API is <b>built-in React tool</b> and does not have to be downloaded separately                                                                                 | Redux is an <b>third-party</b> open source library <b>not part of React</b> which provides a central <b>store</b> , and actions to modify the store. |
| Requires minimal Setup                                                                                                                                                  | Requires extensive setup to integrate it with a React Application                                                                                    |
| Specifically designed for static data, that is not often refreshed or updated                                                                                           | Useful for both static and dynamic data                                                                                                              |
| Difficult to debug                                                                                                                                                      | Easy to debug using Redux dev tool                                                                                                                   |
| Useful for small projects                                                                                                                                               | Useful for larger projects                                                                                                                           |

## 2: Advantage of using Redux Toolkit over Redux.

1. **Abstraction and Convenience:** Redux Toolkit provides a set of abstractions and conveniences on top of regular Redux, which make it easier to work with and manage the state of your application. This includes features such as

the `createSlice` function for creating slices of state and its associated `actions and reducer`, and the `createStore` function for creating a `Redux store` with pre-configured middleware and enhancers.

2. **Immutable updates:** Regular Redux requires you to create a `new state object` every time you make an update, which can become repetitive and error-prone. Redux Toolkit provides a way to `update the state immutably`, using its built-in `createSlice` function.
3. **Simplified Reducers:** In regular Redux, you write your `own reducers`, which can become complex and difficult to manage as your application grows. With Redux Toolkit, you can use the `createSlice` function to generate reducers for you, based on the state updates you define.
4. **Improved Performance:** Redux Toolkit uses advanced performance optimizations, such as memoization, lazy evaluation, and selective updates, to make your application faster and more efficient.
5. **Better Debugging:** Redux Toolkit provides better debugging tools, such as the ability to log and replay actions, inspect the current state of your application, and easily track the changes made to your state over time.

### 3: Explain Dispatcher.

A dispatcher is a `function` that dispatches actions to the store. In Redux, actions are used to describe changes to the state, and dispatching an action is the way to trigger those changes.

- How to create & use dispatcher function ?

```
const dispatch = useDispatch();
```

This hook returns a reference to the `dispatch function` from the `Redux` store. You may use it to dispatch actions as needed.

```
dispatch(actionCreator(data)); // returns an action payload object
```

When you dispatch an action creator, it returns an `action object` that the `reducer function` uses to update the `state`. The dispatcher function is used to dispatch the action creator and which in turns calls the reducer function to trigger the update.

## 4: Explain Reducer.

A reducer is a `pure function` in Redux that takes the `current state` of your application and an `action`, and returns a `new state` based on that `action`.

Example :

```
addItem: (state, action) => {
 const item = state.items[action.payload.id];
 const quantity = item && item.hasOwnProperty('quantity')
 ? state.items[action.payload.id]?.quantity + 1 : 1;
 state.items[action.payload.id] = { ...action.payload, quantity };
 state.totalItemsCount = state.totalItemsCount + 1;
},
```

Here based on the action object, the state is updated inside the reducer function.

## 5: Explain slice.

In Redux Toolkit, a `slice` is a piece of the state that is managed by a single set of actions and reducer.

## 6: Explain selector.

A `selector` is a pure `function` that takes the current `state` of your application and returns a derived value based on that state.

`useSelector` is a hook from the `react-redux` library that allows you to `subscribe` to the `state` of your Redux store from a React component. The `useSelector` hook takes a `selector function` as its argument, which is used to extract data from the state tree. The component will re-render whenever the state of your Redux store changes and the derived value returned by the selector function changes.

```
const totalItemsCount = useSelector(store => store.cart.totalItemsCount);
```



store => store.cart.totalItemsCount is the selector function which returns the totalItemsCount from the state. Now, useSelector() is used to subscribe to this totalItemsCount from the state.

## 7: Explain createSlice and the configuration it takes.

The `createSlice` function is used to create a store slice, a piece of the store.

The `createSlice` function takes an `object` as an argument, which contains the following properties:

- `name`: A string that represents the name of the slice.
- `initialState`: An `object` that represents the `initial state` of the slice. In our cartSlice example, the initial state is an object with two properties: `items` (an empty object) and `totalItemsCount` (which is 0).
- `reducers`: An `object` that contains the Redux reducers for the slice. Reducers are functions that take the current `state` and an `action`, and return a new state based on the action type and payload. In our example, there are three reducers: `addItem`, `removeItem`, and `clearCart`.

After creating the slice, the code `exports` the `actions` that can be dispatched on the store. In this example, there are three actions: `addItem`, `removeItem`, and `clearCart`.

Finally, the code `exports` the `reducer` for the slice using the `reducer` property of the slice. The reducer is responsible for managing the state of the slice and updating it in response to dispatched actions.



# Assignment 13 - Time for the Test



Owner



Pankaj Kumar

## 1: What are different types for testing?

- **Unit Testing:** focuses on individual units or components of the software, ensuring they work as intended.
- **Integration Testing:** combines different units and tests their interaction, ensuring they work together as a system.
- **Functional Testing:** tests the functionality of the software, verifying it meets the requirements and specifications.
- **End-to-end Testing:** tests the entire system, from start to finish, simulating real-world scenarios.
- **System Testing:** tests the system as a whole, verifying it meets the required performance, security, and reliability standards.
- **Acceptance Testing:** tests the software from the user's perspective, ensuring it meets the customer's expectations.
- **Performance Testing:** tests the performance of the software, such as response time, scalability, and stability under different load conditions.
- **Security Testing:** tests the security of the software, verifying it is protected against potential threats and vulnerabilities.
- **Regression Testing:** tests the software after changes have been made, ensuring the changes did not introduce new bugs or break existing functionality.
- **Smoke Testing:** a preliminary test to determine if the basic functions of the software work, before proceeding with more thorough testing.

## 2: What is Enzyme?

Enzyme is a JavaScript testing utility for React, developed and maintained by Airbnb. It can be used in both unit and integration testing.

## 3: Enzyme vs React Testing Library

| Features | Enzyme | React Testing Library | | :---- | :----- | : ----- | | API | Enzyme has a more comprehensive API with methods for `manipulating, traversing, and querying` the React component tree, which can be convenient for `unit testing`. | React Testing Library, on the other hand, has a simpler API that focuses on testing the `behavior` of the components from the user's perspective, making it more suitable for `integration` and `end-to-end` testing. | | Approach | more `implementation-focused` approach to testing, where you test the internal implementation details of the components, such as the `state or props` | React Testing Library, on the other hand, has a more `user-focused` approach, where you test the `behavior` of the components as a user would interact with them, such as `clicking buttons` or `filling out forms`. | | Maintenance | Enzyme requires `more maintenance` as the `internal` implementation of components changes, as the tests are `tightly coupled` to the implementation details. | React Testing Library, on the other hand, is `less` likely to break with changes to the implementation, as it tests the `behavior` of the components rather than the implementation details. |

## 4: What is Jest and why do we use it?

`Jest` is a `JavaScript testing framework` developed and maintained by `Facebook`. It is widely used for testing JavaScript applications, especially for `React` applications. Jest provides a complete and integrated testing solution, with features such as `automatic test discovery, mocking, code coverage, and assertion libraries`.

Jest is a popular and widely used testing framework for JavaScript applications due to the following reasons :

1. **Simplicity:** minimal configuration & low learning curve
2. **Speed:** fast test execution, automatic test caching, parallel test running
3. **Integration:** integrates well with popular JavaScript tools and frameworks, such as React, Babel, and Webpack.
4. **Feature:** mocking, spying, and code coverage reporting

Jest makes it easy for developers to write and run tests, ensuring the quality and reliability of their code.