**POL Back Office Project:**

**Theme Configuration**

The project uses Material-UI (MUI) for its theme configuration. The muiTheme is defined in src/utils/muiTheme.ts and exports a theme object created using the createTheme function from @mui/material/styles.

The theme configuration includes settings for:

Color palette (primary, text, error)

Typography (font family, font size, line height)

Component styles (e.g., tabs, accordion, table)

Applying the Theme

The theme is applied to the application using the ThemeProvider component from @mui/material. This component wraps the entire application and provides the theme context to all components.

In src/components/v2/components/Inputs/Input/Input.stories.tsx, the ThemeProvider is used to wrap the Input component, applying the theme to the component:

**import { ThemeProvider } from '@mui/material';**

**import { muiTheme } from '@utils';**

**// ...**

**<ThemeProvider theme={muiTheme}>**

**<Input {...props} />**

**</ThemeProvider>**

**Common Styles**

Common styles are defined in various CSS files, such as Button.module.scss, CommoditiesView.module.scss, and header.module.scss. These styles are imported and used in the corresponding components.

For example, in src/components/Button/Button.tsx, the Button component imports its styles from Button.module.scss:

**Classnames Utility**

The project uses the classnames utility library to conditionally join CSS class names. This library is used to apply multiple classes to a component based on certain conditions.

For example, in src/components/Button/Button.tsx, the classnames utility is used to apply different classes based on the variant prop:

**Global Styles**

There are also global styles defined in src/utils/muiTheme.ts, which are applied to the entire application. These styles include settings for typography, spacing, and other global styles.

Overall, the project uses a combination of theme configuration, common styles, and utility libraries to manage its styling and layout.

MUI theme:

To apply an MUI theme to a React application, you'll define your theme, wrap your app with a ThemeProvider, and then use the theme within your components.

1. Define Your Theme:

* Create a theme file (e.g., theme.js) and import createTheme from @mui/material/styles.
* Use createTheme to define your desired theme, customizing aspects like palette (colors), typography, and components (component styles).

2. Create a Theme Provider:

* Import ThemeProvider from @mui/material/styles.
* Wrap your main application component (e.g., App or Root) with ThemeProvider, passing your theme object as the theme prop.

3. Use the Theme in Components:

* Once wrapped, any component within your application can access the theme via the useTheme hook or the theme prop passed to MUI components.
* You can then use the theme's colors, typography, and component styles directly in your component's JSX.

**Starting the server:**

The selected context appears to be a Node.js script, likely used as a development startup script for a Next.js project. The script is designed to be run from the command line, and it expects a single argument specifying the target environment (e.g., "dev").

The script uses the child\_process module to spawn(spawnSync) new processes, running TypeScript scripts using ts-node. The scripts are located in the ./scripts directory, relative to the current working directory.

The script's purpose is to perform some setup tasks, such as generating Git information and setting environment variables, before starting the Next.js development server using the next dev command.

Some possible assumptions about the context include:

The script is part of a larger Next.js project, possibly using a monorepo or a similar setup.

The git-info.ts and set-env.ts scripts are custom scripts written for this project, and they perform specific tasks related to Git and environment setup.

The script is intended to be run using a package manager like Yarn or npm, as hinted by the yarn start dev example in the error message.

**Notes:**

Based on the terminal input, the file corresponding to environment entered is copied to “env.local”, which is the file used for running the appln. (ts – typescript code)

The git log is updated with the latest commit msg, tag, current branch as well as hash for tracking (typescript). The content is appended to file

**ENV settings:**

In React, a .env file is used to store environment variables, which are configurations that can differ between development, testing, and production environments. These variables can include API keys, URLs, and other settings.

**How it works:**

**Creation:**

A .env file is created in the root directory of your React project.

**Variable Definition:**

Inside the .env file, variables are defined using the KEY=VALUE syntax.

Variable names must be prefixed with REACT\_APP\_ to be accessible in React. For example: REACT\_APP\_API\_KEY=your\_api\_key.

**Accessing Variables:**

In your React code, environment variables can be accessed using process.env.VARIABLE\_NAME. For example: const apiKey = process.env.REACT\_APP\_API\_KEY;.

**Security:**

The .env file should be added to your .gitignore file to prevent sensitive information from being committed to version control.

**Environment-Specific Files:**

You can create environment-specific files (e.g., .env.development, .env.production) for different configurations.

**Key Considerations:**

Prefix: Always prefix your variables with REACT\_APP\_.

Restart: After modifying the .env file, you must restart your React application for changes to take effect.

Security: Never store secrets directly in your React app as they are embedded into the build and can be viewed by inspecting your app's files.

**// .env file**

**REACT\_APP\_API\_URL=https://api.example.com**

**REACT\_APP\_API\_KEY=your\_api\_key**

**// React component**

**const apiUrl = process.env.REACT\_APP\_API\_URL;**

**const apiKey = process.env.REACT\_APP\_API\_KEY;**

**Eslint Plugin:**

* We have the Eslint plugin configured to handle the following:
* Unused imports / exports
* Unused variables, functions
* Functions must have a return
* Spell check (custom list is generated to exclude few words we included in the project)
* Import orders (inbuilt , external, internal, parent/child, hooks , services etc)
* Functions with more than one line should have curly braces and return type is must

**Prettier Plugin:**

Prettier is a code formatter that ensures consistent code styling, while ESLint is a code linter that checks for code quality issues and enforces coding standards. They work well together to improve code quality.

**Typescript:**

In typescript we have turned on “strict = true”, so everything should be inferred with type function

**react-i18next:**

react-i18next is a powerful internationalization framework for React. All the UI texts are pulled from the transaction. So easily we can customize for the local language.

**SonarQube:**

SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs and code smells on 29 programming languages.

**PR checks:**

* SonarQube
* Pre-commit
* Eslint
* Gate Keeper
* Unit Testcase coverage as well as failure
* PR title should have the JIRA ticket number

**Orval**

Orval is a code generator that can automatically generate React hooks from OpenAPI or Swagger specifications. It simplifies the process of integrating APIs into React applications by creating custom hooks for each API endpoint. These hooks handle loading states, error handling, and caching, reducing boilerplate code.

Orval supports various clients, including React Query, SWR, and Vue Query. It generates TypeScript code with appropriate type signatures, ensuring type safety in your application. The tool can also generate models, requests, and mocks.

Key features of Orval include:

* **Automatic code generation:** Creates custom hooks for each API endpoint.
* **Type safety:** Generates TypeScript code with type signatures.
* **Support for multiple clients:** Works with React Query, SWR, Vue Query, and more.
* **Reduced boilerplate:** Handles loading states, error handling, and caching.
* **Configuration options:** Allows customization of generated code.

To use Orval, you need an OpenAPI or Swagger specification file. Orval then generates the necessary code based on this specification. This results in a more efficient and maintainable way to interact with APIs in React applications.

**When designing a React application, here are some key features to consider:**

**1. Component Hierarchy**

Break down the application into smaller, reusable components

Establish a clear hierarchy of components, with each component having a single responsibility

Use container components to manage state and props, and presentational components for rendering UI

**2. State Management**

Decide on a state management approach: props, state, context, Redux, MobX, etc.

Use a single source of truth for state, and avoid duplicating state across components

Consider using a library like Redux or MobX for complex state management

**3. Routing**

Choose a routing library: React Router, Reach Router, etc.

Define routes and navigation patterns for the application

Consider client-side rendering and server-side rendering for improved performance

**4. Data Fetching**

Decide on a data fetching approach: REST APIs, GraphQL, WebSockets, etc.

Use a library like Axios or fetch for making API requests

Consider caching and optimistic updates for improved performance

**5. Error Handling**

Implement error handling for API requests and component errors

Use try-catch blocks and error boundaries to catch and handle errors

Display user-friendly error messages and provide feedback

**6. Security**

Implement authentication and authorization for protected routes and data

Use HTTPS and secure protocols for data transmission

Validate user input and sanitize data to prevent XSS attacks

**7. Accessibility**

Follow accessibility guidelines: WCAG 2.1, Section 508, etc.

Use semantic HTML and ARIA attributes for screen readers

Test for accessibility using tools like Lighthouse and axe

**8. Performance Optimization**

Use React DevTools to identify performance bottlenecks

Optimize component rendering and reconciliation

Use techniques like memoization, shouldComponentUpdate, and React.lazy

**9. Testing**

Write unit tests and integration tests for components and functionality

Use Jest and Enzyme for testing React components

Consider using Cypress or Selenium for end-to-end testing

**10. Code Organization**

Follow a consistent naming convention and coding style

Organize code into logical folders and files

Use a linter and code formatter to enforce coding standards

**11. Internationalization**

Consider supporting multiple languages and locales

Use a library like i18next or react-intl for internationalization

Format dates, numbers, and currencies according to locale

**12. Deployment**

Choose a deployment strategy: create-react-app, Next.js, etc.

Set up continuous integration and continuous deployment (CI/CD)

Monitor application performance and errors using tools like New Relic or Sentry

By considering these features, you can design a robust, scalable, and maintainable React application that meets the needs of your users.

**Validating User Input and Sanitizing Data in ReactJS**

To prevent XSS (Cross-Site Scripting) attacks in ReactJS, it's essential to validate user input and sanitize data. Here are some best practices to follow:

**Validation**

Use a validation library: Libraries like react-validation, validator.js, or joi can help you validate user input.

**Command to install:**

npm i --save-dev @types/react-validation

Check input types: Ensure that user input matches the expected type (e.g., email, phone number, etc.).

Check input formats: Verify that user input conforms to expected formats (e.g., date, time, etc.).

Check input lengths: Validate the length of user input to prevent excessive data.

**Sanitization**

**Command:**

***npm install dompurify***

Import :

import DOMPurify from 'dompurify';

 console.log(DOMPurify.sanitize(event.target.value),event.target.value);

Use a sanitization library: Libraries like dompurify or sanitize-html can help you sanitize user input.

Escape special characters: Use functions like encodeURIComponent or escape to escape special characters in user input.

Remove HTML tags: Use a library or a custom function to remove HTML tags from user input.

Use a whitelist approach: Only allow specific, known-good input values to prevent unknown or malicious input.

**ReactJS Specific**

Use dangerouslySetInnerHTML with caution: This prop can be used to set HTML content, but it's vulnerable to XSS attacks. Use it only when necessary and with proper sanitization.

Use React.createElement instead of innerHTML: When creating elements dynamically, use React.createElement to prevent XSS attacks.

Use a library like react-dom-purify: This library provides a purify function to sanitize user input and prevent XSS attacks.

import React, { useState } from 'react';

import { validate } from 'react-validation';

import { sanitize } from 'dompurify';

const MyComponent = () => {

const [inputValue, setInputValue] = useState('');

const handleInputChange = (event) => {

const inputValue = event.target.value;

const sanitizedInput = sanitize(inputValue); // Sanitize user input

setInputValue(sanitizedInput);

};

const handleSubmit = (event) => {

event.preventDefault();

const isValid = validate(inputValue, { // Validate user input

type: 'email',

required: true,

});

if (isValid) {

// Process sanitized and validated input

}

};

return (

<form onSubmit={handleSubmit}>

<input type="text" value={inputValue} onChange={handleInputChange} />

<button type="submit">Submit</button>

</form>

);

};

**Token Handling:**

It appears that the authentication token is handled using the aws-amplify library. Specifically, the Auth class from aws-amplify is used to manage the authentication token.

In the src/app/AppWrapper.tsx file, there is an import statement for Auth from aws-amplify, and it is used in the authenticateUser function.

***import { Auth } from 'aws-amplify';***

Additionally, in the src/utils/authHelper.ts file, there is a function called getJwtToken that retrieves the JWT token from the Auth class.

**export async function getJwtToken() {**

**try {**

**// Calling Auth.currentSession automatically refreshes the id/access tokens using the refresh token**

**const session = await Auth.currentSession();**

**return session.getIdToken().getJwtToken();**

**} catch (err) {**

**console.error('Failed to get JWT token', err);**

**throw err;**

**}**

**}**

This suggests that the authentication token is stored in the Auth class and can be retrieved using the getJwtToken function. However, without more information, it is unclear how the token is used in the application or how it is passed to APIs or other services.

It's also worth noting that there is a requestTokenRefreshInterceptor function in the src/utils directory, which may be related to token refresh or authentication. However, without more context, it's difficult to say for certain how this function is used.

**import { requestTokenRefreshInterceptor } from '@utils';**