Syncx LWC Documentation agencyScheduler

Description

This Salesforce Lightning Web Component (LWC) named `AgencyScheduler` is designed to manage and display the scheduling of shifts and availability for providers in a calendar format. It includes a range of functionalities such as adding, claiming, releasing, confirming, and canceling shifts, along with specifying the availability of providers. It also includes a variety of filters to narrow down the events shown on the calendar, such as shift type, location, candidate type, and specialty filters. It uses several custom components (`c-nu-page-layout`, `c-nu-calendar`, `c-nu-searchable-multi-picklist`, etc.) for layout and functionality purposes.

Methods

1. **connectedCallback**: Used to fetch initial data sets when the component is inserted into the DOM. It makes Apex calls to retrieve contact information, account by contact ID, placements by agency ID, and sets initial data and styles based on fetched data.
2. **renderedCallback**: Called after every render of the component. This is used to ensure that the client's CSS is loaded only once.
3. **handleSaveChangesClick**: Manages the saving of changes made to the shifts in the calendar. This includes actions like adding shifts, removing shifts, confirming shifts, claiming shifts, releasing claims, and canceling shifts.
4. **handleDiscardChangesClick**: Discards all the unsaved changes made to shifts, resetting any claims, adds, confirms, etc., and clears the filters.
5. **handleCalendarSelectedEventsUpdated**: Reacts to calendar events selection changes to manage state across the multiple calendar instances.
6. **processPlacementResults**: Processes placements data to extract and set information for shift options, provider options, and account locations.
7. **handleActionClick**: Determines actions based on selected options in the action panel and opens the respective modals for adding availability, confirming shifts, etc.

Properties

- **hasUnsavedChanges**: Indicates if there are any unsaved changes in the calendar.

- **shiftTypeOptions**: Options for shift type selection.
- **actionTypeOptions**: Options for actions to be performed on shifts.
- **providerOptions**: Options for provider selection.
- **specialtyOptions**: Options for specialty selection.
- **providerTypeOptions**: Options for provider type selection.
- **locationOptions**: Options for location selection.

Use Case

This component is suitable for use in scenarios where an agency needs to manage the scheduling of shifts and the availability of providers. It can be used to display scheduled shifts on a calendar, filter shifts based on various criteria, and perform actions such as adding, claiming, releasing, confirming, and canceling shifts.

Important Notes

- This component relies on several custom components and Apex methods for its functionality, so ensure those dependencies are properly set up in the Salesforce org.
- The component uses Lightning Data Service (LDS) and Apex to fetch data, emphasizing the need for correct Apex class permissions for the target users.
- Ensure that the CSS file defined in CLIENT_CSS is available in your Salesforce org and has the expected styles defined for the intended visual appearance.
- The component uses `localStorage` for storing some temporary state data; be mindful of the browser's restrictions and security considerations around its use.
- The component demonstrates a sophisticated use of LWC's reactivity and event handling to manage complex interactions within the calendar UI, making extensive use of template conditionals and iteration, event dispatching and handling, and dynamic CSS.

availableOrdersList

Description

This Salesforce Lightning Web Component named `AvailableOrdersList` is designed to display a list of available job orders fetched from an Apex class. It showcases job orders with various details such as Job Order ID, Client, Description, Specialty, Job Order Status, Start Date, and End Date. The component provides functionality for sorting and infinite loading of job order data.

Methods

- **connectedCallback**: Lifecycle hook that runs when the component is inserted into the DOM. It initializes component properties and fetches initial data.
- **renderedCallback**: Lifecycle hook that runs after the component has finished the rendering phase. It is used here for loading data into the datatable.
- **doSorting**: Function to handle sorting events emitted by the `lightning-datatable`.
- **sortData**: Helper function used to sort the datatable based on the column selected for sorting.
- **navigateToHomePage**: Utilizes the NavigationMixin to navigate to the home page of the Salesforce org.
- **loadMoreData**: Function triggered to load more job orders when the user scrolls down the data table, implementing infinite loading.

Properties

- **testAgencyId**: Public property (`@api` decorated) used for testing by allowing the agency ID to be set externally.
- **agencyId**: Stores the ID of the agency fetched based on the contact related to the current user.
- **contactId**: Stores the ID of the contact fetched based on the current user.
- **error**: Stores any error messages encountered during data retrieval.
- **columns**: Holds the column definitions for the `lightning-datatable`.
- **availableJobOrders**: Array that stores the job orders fetched.
- **defaultSortDirection, sortDirection**: Determine the current and default direction of sorting in the datatable.
- **sortedBy**: Specifies the field name that the data is currently sorted by.
- **recordCount**: The number of records currently displayed in the datatable.
- **totalNumberOfRows**: The total number of job order rows available.
- **renderConfig**: Holds rendering configurations for the datatable, particularly related to virtual scrolling.
- **loadMoreStatus**: Text status displayed based on whether more data is loading or all data has been loaded.

Use Case

This component is useful for applications within a Salesforce Org that need to present a list of available job orders to the user in a tabular format. It supports sorting by columns and infinite scrolling to load more data. It could be particularly useful for staffing agencies or companies looking to manage and display job orders accessed by their staff or users from a Salesforce org.

Important Notes

- **Dynamic Styling**: This component adapts its styling based on certain criteria such as the `Use_Portal_Theme__c` flag of the related account. If true, it dynamically adjusts border and background color styles based on account properties.
- **Infinite Loading**: Through the `lightning-datatable` attribute `enable-infinite-loading` and the `loadMoreData` JS method, the component supports loading data as the user scrolls.
- **Sortable Columns**: Columns are defined with sortable properties for enhanced user interaction, allowing users to sort the job order list based on column headers.
- **Page Lifecycle Hooks**: Utilizes `connectedCallback` & `renderedCallback` lifecycle hooks for fetching initial data and managing rendering logic.
- **NavigationMixin**: This component uses `NavigationMixin` for navigating to the home page, demonstrating how to embed navigation functionality within a Lightning Web Component.

```
navigateToHomePage() {
    this[NavigationMixin.Navigate]({
        type: 'standard__namedPage',
        attributes: {
            pageName: 'home'
        },
    });
}
```

Make sure that all Apex methods referenced in the JS controller (`fetchContactId`, `fetchAccountByContactId`, `fetchAvailableOrders`, `fetchAvailableOrderCount`) are properly implemented and accessible by the component.

candidateLookup

Description

The provided Salesforce Lightning Web Component (LWC), named `CandidateLookup`, is a custom lookup component designed to search and select records from any given Salesforce object. It allows users to type in search queries, view matching results, and then select a record from those results. Once a record is selected, it is displayed as a "pill" within the component, with an option to remove the selection. This component is a versatile tool for scenarios where a custom record search and selection functionality is needed beyond the standard Salesforce lookup fields.

Methods

1. **toggleResult(event):** Toggles the display of the search result section based on user interaction (such as clicking on the search input or clicking outside the component).
2. **handleKeyChange(event):** Handles input changes in the search field. It features debouncing to limit the frequency of search operations triggered by user input.
3. **handleRemove():** Clears the currently selected record, making the search input available for a new search.
4. **handelSelectedRecord(event):** Handles the selection of a record from the search results. It updates the component's state to reflect the selected record and adjusts the UI to display the selection as a "pill".
5. **handelSelectRecordHelper():** A helper function that updates the UI based on the current selection state of the component.
6. **lookupUpdatehandler(value):** Dispatches a custom event named `lookupupdate` to notify parent components of a selection change. The selected record (or null if a selection is cleared) is passed as the event's detail.

Properties

1. **label (String):** (API) The label for the lookup component.
2. **placeholder (String):** (API) The placeholder text for the search input field.
3. **iconName (String):** (API) The name of the icon to be used in the lookup component.
4. **sObjectApiName (String):** (API) API name of the Salesforce object this lookup component is searching against.
5. **recordType (String):** (API) Specified record type for the search query.
6. **jobOrderId (String):** (API) The Job Order Id used to fetch specialty and credentials for the job order.
7. **agency (String):** (API) Specifies the agency ID to be used in the search query.
8. **isSelected (Boolean):** Tracks whether a record is currently selected.

9. **isSearchLoading (Boolean):** Indicates whether a search is currently in progress.
10. **searchKey (String):** The current value of the search input field.
11. **lstResult (Array):** Stores the list of records fetched by the search.
12. **selectedRecord (Object):** Stores the currently selected record.

Use Case

This component is ideal for implementing custom lookup functionality in Salesforce Lightning applications where users need to search and select records from specific objects with custom filtering logic. It's particularly useful in scenarios requiring searches across different objects or customized search criteria beyond the capabilities of standard lookup fields.

Important Notes

- The component uses `@wire` adapters to fetch search results based on user input and to extract specific fields (`Specialty__c`, `Credential_Accepted__c`) from a given Job Order record.
- Debouncing is implemented to improve performance and reduce the number of executed queries when searching.
- The component illustrates a practical application of handling both input and selection events while providing visual feedback using SLDS styling.
- Ensure that the Apex controller (`CustomLookupController.fetchCandidate`) used for fetching records is properly implemented with necessary access controls and optimized query logic.
- This component assumes the use of Salesforce Lightning Design System (SLDS) styles for consistent UI rendering within the Salesforce ecosystem.

clientScheduler

Documentation for Salesforce Lightning Component: ClientScheduler

Description

The `ClientScheduler` component allows Salesforce users to schedule, view, edit, and manage shifts within a calendar view. It supports various actions, including adding, removing,

selecting claims, and canceling shifts based on specified criteria. It interacts with multiple Apex controllers to fetch and manipulate shift data, as well as user and account information.

Methods

fetchProviders

Fetches a list of providers based on specified locations, types, and specialties. It updates the component's provider-related options for filtering and creating shifts.

```
fetchProviders(locationOptionsIds);
```

computeTotalHours

Computes the total hours between the selected start and end times for a shift. It updates the `totalHours` property accordingly.

```
computeTotalHours();
```

setEventsMap & getEventsMap

Stores selected events from the calendar in a map for easy retrieval and synchronization across different calendar views.

```
setEventsMap(key, value);
getEventsMap(key);
```

Properties

- `selectedRateType`: Holds the currently selected rate type for filtering or creating shifts.
- `selectedShiftType`: Holds the selected shift type from a predefined list of options.
- `selectedLocation`: Stores the ID of the currently selected location.
- `selectedProviderType`: Stores the provider type selected by the user.
- `selectedSpecialty`: Holds the selected medical specialty.
- `selectedProvider`: Stores the selected provider's information.
- `weekDaySelectedOptions`: An array of selected weekdays for filtering shifts.

Use Case

A typical use case for the `ClientScheduler` component is within a Salesforce org where scheduling and managing shifts for healthcare providers or similar roles is required. This component caters to admins or managers who need to visualize shifts in a calendar view, perform actions like adding or removing shifts, and filter shifts based on various criteria such as location, provider type, or specialty.

Important Notes

- Ensure that all required Apex controllers and methods are available and have the necessary permissions to be called from the component.
- The user's timezone is considered for displaying dates and times correctly. Make sure to handle timezone conversions accordingly.
- Custom styling and theming are supported. Modify CSS according to the organization's branding requirements.

This component provides a comprehensive interface for managing shifts, integrating with Salesforce data, and offering a user-friendly calendar view for scheduling. It leverages Lightning Web Components (LWC) and Apex to interact with Salesforce backend data, providing a powerful tool for shift management within Salesforce.

cometdlwc

Description

The Lightning Web Component (LWC) named `Cometdlwc` is designed to connect to Salesforce Streaming API using the CometD library. It allows for real-time data streaming from a specified channel, making it useful for applications that require live data updates, such as chat applications or live dashboards.

Methods

- `initializecometd`: This method initializes the CometD library with necessary configurations such as the connection URL, authorization headers, and sets the WebSocket to false. It performs the handshake with the server. Upon a successful handshake, it subscribes to the specified channel and listens for incoming messages. When a message is received, it dispatches a `message` event with the message details.

Properties

- `channel` (`@api`): This public property allows the component to subscribe to a specific channel for listening to events. It should be set by the parent component or application using this component.
- `libInitialized` (`track`): A private property used to ensure the CometD library is initialized only once.
- `sessionId` (`track`): Stores the session ID needed for authentication with the CometD endpoint. It is obtained via a wire service to the `getSessionId` Apex method.
- `error` (`track`): Used to store any error messages, especially during the session ID retrieval or when there are issues initializing the CometD connection.

Use Case

The component is ideal for situations where you need real-time updates from Salesforce, such as:

- Display live updates for record changes or specific events within a Salesforce org.
- Real-time notification systems displaying alerts or messages as they happen.
- Chat applications where near-instant message delivery is crucial.

Important Notes

1. **Platform Resource Loader**: The component uses `lightning/platformResourceLoader` to dynamically load the CometD JavaScript library stored as a static resource within Salesforce. Ensure that the CometD library (`cometd.js`) is correctly uploaded as a static resource named `cometd`.
2. **Session ID Retrieval**: The component automatically retrieves the session ID using an Apex method (`getSessionId`) which needs to be implemented in the Salesforce org. This session ID is critical for authenticating the CometD connection.
3. **Event Dispatching**: When a message is received from the subscribed channel, the component creates and dispatches a `message` custom event. Any parent component using `Cometdlwc` needs to listen for this event to process the received messages.
4. **Subscription Channel**: The component subscribes to a channel based on the `channel` API property. It's important to set this property correctly to ensure the component listens to the intended events.

5. **Security and Access**: Make sure to handle the session ID securely and configure proper access rights for the Apex class to prevent unauthorized access.

Here's a simple example of how you can use this component in a parent component:

```
<c-cometdlwc channel="YourChannelName" onmessage={handleMessage}></c-cometdlwc>
```

And in the parent component's JavaScript:

```
handleMessage(event) {
    console.log('Message received: ', event.detail);
}
```

This setup ensures that your application can react to real-time events broadcasted to the specified channel.

customLookup

Description

The `CustomLookup` component is a Salesforce Lightning Web Component (LWC) designed to provide a customizable lookup field. It enables users to search for and select a record from a specified Salesforce object. This component includes features like search as you type, displaying search results, selecting a record, and showing the selected record with an option to clear the selection.

Methods

- **connectedCallback:** Initializes the component by fetching a default record if a `defaultRecordId` is provided.
- **searchResult:** A wire adapter that fetches search results based on user input (`searchKey`).
- **handleKeyChange:** Handles input field changes, implementing a debounce mechanism to limit server calls.
- **toggleResult:** Toggles the visibility of the lookup result section based on user actions (e.g., clicking outside the component).
- **handleRemove:** Clears the currently selected record and updates the UI to show the search input again.

- **handelSelectedRecord:** Handles a user's selection of a record from the search results, updating the UI and selected record state.
- **handelSelectRecordHelper:** A helper method used by `handleRemove` and `handelSelectedRecord` to update the UI when a record is selected or removed.
- **lookupUpdatehandler:** Dispatches a custom event to notify parent components of the selection change.

Properties

- **label (api):** The label for the lookup field.
- **placeholder (api):** The placeholder text for the search input.
- **iconName (api):** The name of the icon to display in the lookup field and search results.
- **sObjectApiName (api):** API Name of the Salesforce object to search.
- **defaultRecordId (api):** The Id of a default record to display upon initialization.
- **recordType (api):** Specifies the record type for filtering search results.
- **contactFieldName (api):** Field name to identify selections in custom events.
- **hideClearButton (api):** Boolean to control the visibility of the clear button.
- **lstResult:** Private property to store the list of search results.
- **hasRecords:** Flag to indicate if there are search results.
- **searchKey:** Stores the current value of the search input.
- **isSearchLoading:** Controls the visibility of the loading spinner.
- **selectedRecord:** Stores the currently selected record.

Use Case

This component is useful in situations where a user needs to associate a record from one Salesforce object with another record. For example, linking a `Contact` to an `Account`. It can be customized to search within any Salesforce object by setting the `sObjectApiName` and can be further configured with a default value, custom icons, and placeholder text.

Important Notes

1. **Debounce Mechanism:** The `handleKeyChange` method implements debouncing to reduce the number of server calls while the user is typing in the search input.
2. **Custom Events:** The component uses custom events (`lookupupdate`) to communicate the selection changes to parent components.

3. **CSS Classes:** The component makes extensive use of SLDS classes to conform to Salesforce's design system, ensuring that its look and feel are consistent with the Salesforce experience.
4. **Accessibility:** The component is developed with accessibility in mind, using ARIA attributes and roles to make it accessible to people using screen readers and other assistive technologies.

fullCalendar

Description

This Salesforce Lightning component, named `FullCalendarComponent`, integrates a robust calendar view into Salesforce Lightning Experience, enabling users to schedule, view, and manage events dynamically. Leveraging the popular FullCalendar library, the component offers various views such as day, week, month, and list views, and supports operations like event creation, modification, and deletion.

Methods

connectedCallback

Invoked when the component is inserted into the DOM. It initializes event listeners.

renderedCallback

Ensures FullCalendar and its dependencies are loaded and initialized. This is where the calendar is configured and rendered.

createEvent

Creates a new event on the calendar with specified details such as start time, end time, color, and more.

init

Initializes the FullCalendar instance with predefined options and plugins. It binds various FullCalendar events like `eventClick`, `dateClick`, and `select`.

nextHandler

Navigates the calendar to the next period based on the current view.

```
@api nextHandler() {
    this.calendar.next();
    this.calendarLabel = this.calendar.view.title;
    this.removeEmptyEvents(this.calendar.getEvents());
}
```

previousHandler

Navigates the calendar to the previous period based on the current view.

dailyViewHandler, weeklyViewHandler, monthlyViewHandler, listViewHandler

Change the current calendar view to day, week, month, or list view respectively.

today

Navigates the calendar to today's date.

refresh

Refetches events and refreshes the calendar view.

removeEmptyEvents

Removes events without an ID from the calendar.

Properties

The component exposes numerous `@api` properties such as `titleField`, `objectName`, `startField`, `endField`, etc., allowing you to customize the event source dynamically.

Use Case

This component can be utilized in any Salesforce Lightning application that requires scheduling or event management functionality. For example, it could be used to manage conference rooms, track project milestones, or schedule staff shifts. It offers extensive customization options to tailor the calendar view and event handling as per specific business requirements.

Important Notes

- The component depends on external libraries (`FullCalendar`, `FontAwesome`) loaded dynamically via `loadScript` and `loadStyle` methods. Ensure these resources are correctly uploaded to Salesforce static resources.
- The `eventSourceHandler` method integrates with custom Apex controllers to fetch and manipulate events. Make sure the Apex controllers are available and accessible by the Lightning component.
- Specific global variables (e.g., `objectName`, `startField`, `endField`, etc.) are declared outside the component class. These are utilized within `eventSourceHandler` for dynamic event fetching.
- The CSS classes (e.g., `.open-shift`, `.confirmed-shift`) mentioned define styles for different types of events and must be included in the component's stylesheet for optimal visual distinction.
- The component employs the Salesforce Lightning Design System (SLDS) for styling, ensuring a consistent look and feel with the Salesforce environment.
- It's important to handle all potential exceptions or errors during the lifecycle hooks (especially in `renderedCallback`) to avoid breaking the user interface.

hiringAuditList

Description

The `HiringAuditList` component is designed to fetch and display a list of Hiring Audit records dynamically from Salesforce. It provides a powerful UI to filter these records based on various criteria such as Status, Candidate Name, Specialty Name, Location, and more. The component uses Lightning Web Components (LWC) and Salesforce Apex to retrieve data and manipulates it within a Lightning Data Table for a seamless user experience. Additionally, it offers conditional rendering of filters and dynamically adjusts its styling based on theme settings from related accounts.

Properties

- `testAccountId`: API property used to pass the Account ID for testing purposes.
- `stage`: API property to define the current stage of the hiring audit process.
- `title`: API property to set the title of the component/page.

Methods

`connectedCallback()`

Executes when the component is inserted into the DOM. It initiates the `fetchHiringAuditsData` function to retrieve audit data.

`fetchHiringAuditsData()`

Fetches hiring audits data from Salesforce using Apex methods based on the Hired Account's ID. This method also sets up various filters' options used for filtering the displayed data.

`renderedCallback()`

Executes after every render of the component. It is used for any post-render actions but is currently not used in this component.

`loadMoreData(event)`

(Optional) Can be utilized to fetch more data when the end of a datatable is reached. Currently commented out and not implemented in this example.

`doSorting(event)`

Handles sorting of the datatable columns. It captures the columnName and direction (asc/desc) and sorts the data accordingly.

`sortData(fieldname, direction)`

Utility method for sorting the datatable rows based on the column selected and the direction of sorting (ascending or descending).

`handle[FilterName]Change(event)`

Each `handle[FilterName]Change` method is responsible for capturing filter value changes (e.g., `handleStatusFilterChange`, `handleCandidateFilterChange`) and calls `doFilter` to update the displayed data based on the selected filters.

`doFilter()`

Applies all the active filters to the hiring audits data and updates the datatable to only show the filtered results.

Use Case

The `HiringAuditList` component is ideal for Salesforce Community Pages or internal Salesforce org pages where there's a need to display and interact with a list of hiring audit records. This can be particularly useful for HR departments, recruitment agencies, or any organization managing a large number of hiring processes and looking to streamline their audit review process.

Important Notes

1. The component relies heavily on Apex backend calls, which requires the respective Apex classes and methods (`nuService_nuJobOrder`, `nuService_Contact`, etc.) to be correctly implemented and accessible.
2. It dynamically adjusts some of its styling based on the `RecordType` and `Use_Portal_Theme__c` fields of the account associated with the user viewing the page. This requires the Account object to have these fields populated correctly.
3. The component includes extensive use of conditional rendering in its template to either show or hide certain UI elements based on the state of the component (e.g., showing different filters based on the value of the `assignment` computed property).
4. There's commented-out functionality related to PDF generation and download, as well as loading more data into the datatable. If needed, these features could be implemented by uncommenting and completing the relevant code parts.
5. The component uses a mix of base Lightning components (like `lightning-datatable`, `lightning-combobox`) and custom components (like `c-nu-searchable-multi-picklist`), indicating a sophisticated UI design that might require interactions between multiple custom components.

jobOrderReleasesList

Description

This Lightning Web Component (LWC) named `OrdersList` is designed to display a list of Job Order Releases related to an account within a Salesforce Community. It dynamically fetches and displays data using Apex controller methods. Key features include a loading spinner while data

loads, a dynamic table for displaying releases, infinite loading of data, and custom styling based on account settings or global styles.

Methods

- `connectedCallback()`: Lifecycle hook that runs when the component is inserted into the DOM. It performs initial data fetching including user's contact ID, account details, and the first chunk of job order releases. It also sets custom styling based on account-specific theming or global styles.
- `renderedCallback()`: Lifecycle hook that runs after every render of the component. It's used to invoke `loadMoreData()` method on the first render to initiate the display and loading of job order releases data.
- `loadMoreData(event)`: This method is triggered when the user scrolls down the data table and more data needs to be loaded. It fetches the next set of job order releases and appends them to the current list, updating the table view.

Properties

- `@api testAccountId`: (Public property) This is an externally assignable property used primarily for testing, allowing a specific account ID to be set.
- `@api title`: (Public property) Allows the setting of a title for the page or the table from the component's parent.
- `@track accountId`: (Private reactive property) Used to hold the ID of the account fetched based on the current user's contact ID.
- `@track wiresLoaded`: (Private reactive property) A boolean used to determine if initial data loading is complete to control the display of the loading spinner.
- `columns`: Defines the columns of the `lightning-datatable`. This is a static, pre-defined array that includes configurations for each column in the table.
- `releases`: Holds the array of job order release records fetched from Apex to be displayed in the table.
- `recordCount`: Tracks the number of job order release records currently displayed in the table.
- `loadMoreStatus`: Indicates the status of data loading, especially useful when implementing infinite loading to provide user feedback.

Use Case

This component is particularly useful in a Salesforce Community where end-users need to view a list of Job Order Releases related to their account. Features like infinite loading and dynamic styling improve user experience by providing a seamless way to browse large datasets and a visually appealing interface that aligns with personalized or global design themes.

Important Notes

- **Apex Controllers**: The component relies on multiple Apex controllers (`nuService_Contact` and `nuService_nuJobOrder`) for backend data retrieval. These controllers must be properly set up and accessible by the component.
- **Permission and Sharing Settings**: Ensure that the running user has the appropriate permissions to access the data being queried by the Apex methods.
- **Styling**: Custom styling is applied based on account-specific themes (if available) or global styles through the `getPageStyles` method. The CSS leverages an external stylesheet (`nuCSS`) imported at the beginning of the CSS file.
- **Data Table Configuration**: The `lightning-datatable` is configured to support infinite loading, with a fixed height that triggers loading more data as the user scrolls. Columns and data properties are dynamically set within the JavaScript file based on fetched data.
- **Error Handling**: While some basic structures for error handling and loading status updates are provided, consider implementing more comprehensive user feedback mechanisms for error states and exceptions.
- **Testing**: The component's dependency on external Apex controllers and potential custom styling based on account information makes thorough testing essential across different user profiles and data scenarios to ensure consistent behavior and appearance.

listbox

Description

The given Salesforce Lightning Web Component (LWC) is designed to represent a custom listbox that supports both single and multiple selections. This component makes use of Salesforce Lightning Design System (SLDS) classes for styling and is dynamically populated with items passed through `options` property. It features a scrollable container where list items are rendered. Each item can be interactively selected, with the selection logic accommodating both single and multi-selection modes through mouse click interactions. An event named `optionchange` is dispatched whenever the selection changes, carrying the latest selection data.

Properties

- `label` (*@api*): The `aria-label` for the listbox used for accessibility.
- `isMultiSelect` (*@api*): A Boolean property to define if multiple items can be selected. Defaults to `false`.
- `options` (*@api*): An array of objects where each object represents an option in the listbox. Each object should have `label` and `value` properties.
- `selectedOptions` (*@api*): An array to track the selected options. It's updated based on user interactions.

Methods

- `get listOptions`: A getter method that processes the `options` and `selectedOptions` to generate a new array (`mutatedOptions`) that is used to render the listbox. It marks options as selected based on `selectedOptions` and applies a CSS class for styling selected options.
- `handleOptionClick`: An event handler for click events on listbox options. It updates the selection based on user interactions and `isMultiSelect` property. It also dispatches an `optionchange` event with the updated selection.

Use Case

The component can be used in Salesforce Lightning applications where a custom listbox is needed. For example, in a settings panel where users can select one or more options from a list, or in forms where users need to make a selection. The component's support for single and multiple selections makes it versatile for various use cases.

Important Notes

- The component is styled using SLDS classes and custom CSS for specific styling.
- Accessibility considerations such as `aria-label`, `aria-multiselectable`, and `aria-selected` are implemented.
- The component relies on the parent component to pass `options` and listens for selection changes through the `optionchange` event.
- To use this component, ensure the `options` and `selectedOptions` properties are correctly initialized and updated based on the application's logic.

- Proper handling of the `optionchange` event is necessary to update the parent component's state or to perform other actions based on the selection.

Example Implementation

Here's a quick example of how you might initialize this component in a parent component's HTML file:

```
<c-listbox label="Select Options"
           is-multi-select="true"
           options={myOptions}
           selected-options={mySelectedOptions}
           onoptionchange={handleOptionChange}>
</c-listbox>
```

And in the parent component's JavaScript file:

```
import { LightningElement, track } from 'lwc';

export default class ParentComponent extends LightningElement {
    @track myOptions = [
        { label: 'Option 1', value: '1' },
        { label: 'Option 2', value: '2' }
        // Add more options as needed
    ];

    @track mySelectedOptions = [];

    handleOptionChange(event) {
        this.mySelectedOptions = event.detail;
    }
}
```

This component is an excellent example of leveraging the LWC framework to create reusable UI elements that augment the Salesforce Lightning Experience with custom functionality.

nMyTask

Description

The Lightning Web Component (LWC) `NAgencyDashboard` is designed to display a dashboard for agencies within Salesforce. It utilizes Apex methods to fetch counts of open job orders, job orders synchronized and approved for the current month, and job orders closed. These counts

are then displayed within tiles using a custom Lightning component, `c-n-tile-box`, each with specific labels, icons, and colors.

Methods

1. **connectedCallback:** Lifecycle hook that fires when the component is inserted into the DOM. It initializes the component by fetching the agency ID (either passed as an attribute or retrieved from the logged-in user) and then calls methods to fetch data for open job orders, job orders synchronized/approved this month, and job orders closed.

```
connectedCallback() {
    if(this.testAgencyId) {
        this.agencyId = this.testAgencyId;
    }
    // fetch Agency ID logic for logged-in user is supposed to be
here
    this.fetchOpenJobOrdersCount(this.agencyId);
    this.fetchJobOrdersSynchronizedApprovedThisMonth(this.agencyId);
    this.fetchJobOrdersClosedCount();
}
```

2. **fetchOpenJobOrdersCount:** Calls the `countOpenJobOrders` Apex method, updating the tile array with the result for open job orders.

```
fetchOpenJobOrdersCount(agencyId) { /* Apex call and logic */ }
```

3. **fetchJobOrdersSynchronizedApprovedThisMonth:** Calls the `countJobOrdersSynchronizedApprovedThisMonth` Apex method, updating the tile array with the result for job orders synchronized and approved within the current month.

```
fetchJobOrdersSynchronizedApprovedThisMonth(agencyId) { /* Apex call
and logic */ }
```

4. **fetchJobOrdersClosedCount:** Calls the `countJobOrdersClosed` Apex method, updating the tile array with the result for job orders closed.

```
fetchJobOrdersClosedCount() { /* Apex call and logic */ }
```

Properties

1. **testAgencyId @api:** An API property that allows an external source to set the agency ID. This is likely for test or demonstration purposes.
2. **agencyId:** The agency ID used to fetch job order counts. It may come from `testAgencyId` or another source (e.g., the logged-in user's associated agency).
3. **tileArray:** An array designed to hold the data for each of the tiles to be displayed. It starts off with false values and is populated with objects containing the count, icon, label, and other metadata for each tile as data is fetched.
4. **boxLabel:** A simple string label for the box containing the dashboard's tiles. It's set to 'My Tasks'.
5. **isDataComplete:** A boolean flag indicating whether data for all tiles has been successfully fetched and populated.

Use Case

The component is useful in creating a dashboard for agencies within Salesforce, providing them with quick insights into the number of open job orders, orders approved in the current month, and orders closed. It's designed to be dynamic, fetching and updating data in real-time.

Important Notes

- This component relies on specific Apex methods (`countOpenJobOrders`, `countJobOrdersSynchronizedApprovedThisMonth`, `countJobOrdersClosed`) existing and being correctly configured in your Salesforce org.
- The `agencyId` can be set externally via the `testAgencyId` attribute. If not set externally, logic should be implemented to fetch it from the current user's context (not fully illustrated in the provided code).
- The component uses another custom component, `c-n-tile-box`, for displaying the data but does not include its implementation.
- The `tileArray` is critical for the rendering logic, and the component waits until all its elements are populated with data before rendering the dashboard.
- Error handling for the Apex calls is indicated but not fully implemented in the given code snippets.

nNavbar

Description

This Salesforce Lightning Web Component (LWC) named `NNavbar` is designed to dynamically display a navigation bar tailored for different portal types, such as Agency, Candidate, or Company (Client). It showcases various navigational links with submenu capabilities and adapts its layout responsively across devices. The component's behavior is controlled by JavaScript and its appearance is defined in CSS, both highly reliant on the component's state and properties fetched from the Salesforce database, including customization like dynamic styling and logos.

Methods

- **navigateToHome**: Navigates the user to the home page when invoked.

```
navigateToHome() {
    this[NavigationMixin.Navigate]({
        type: 'standard__namedPage',
        attributes: {
            name: 'Home'
        },
    });
}
```

- **navigateToPage**: Navigates to a specific page based on the data attribute 'data-navinfo' of the clicked element.

```
navigateToPage(e) {
    // Implementation; uses `this.navigationInformation` map and
`this[NavigationMixin.Navigate]`
}
```

- **handleMobileNav**: Toggles the visibility of the mobile navigation view.

```
handleMobileNav() {
    this.appClicked = !this.appClicked;
}
```

Properties

- **isAgency, isCandidate, isClient**: Computed properties determining the type of portal user.

```
get isAgency() {
    return (this.portalType && this.portalType === 'Agency');
}
```

- **linkContainerClass, closeClass**: Computed properties generating CSS class names based on the state of navigation.

```
get linkContainerClass() {
    if(this.appClicked) return 'link-container mobile-view';
    else return 'link-container';
}
```

- **portal, testContactId**: Public properties (`@api`) that can be set externally, influencing the component's behavior like the type of user or data to be tested.

Use Case

In a Salesforce Community Portal, `NNavbar` could be used to provide a consistent navigation experience across different portal types (Agency, Candidate, Company). It adapts the available navigation links and the appearance based on the user's role and other settings (like branding options specified in Salesforce records). For instance, an Agency user would see links to manage Candidates and Job Orders, whereas a Candidate user would have options related to Time Management and their Schedule.

Important Notes

- **Dynamic Styling and Branding**: The component supports dynamic theming based on the user's associated account settings (e.g., Primary Color, Logo URL), making it adaptable to various corporate identities.
- **Responsiveness**: It features responsive design through CSS media queries, changing the layout for mobile devices to offer a hamburger menu for compact navigation.
- **Navigation**: Uses Salesforce's `NavigationMixin` for navigating between different pages or views within the Salesforce platform, using a predefined map (`navigationInformation`) that pairs actions to specific page attributes.
- **Lightning Web Component (LWC) Lifecycle**: While there's a placeholder for the `connectedCallback` lifecycle hook with commented-out code, it hints at potential uses like fetch operations upon component initialization to retrieve user or account-specific data.

- **Commented Code**: The component contains commented-out sections that might serve as placeholders for future features (like notifications). It's important to review and clean these if they remain unused to maintain code clarity.

nNavigationBar

Description

The Lightning Web Component (LWC) named `NNavigationBar` provides a dynamic, role-based navigation bar for a Salesforce app. It adapts its appearance and available links depending on whether the user is identified as an "Agency", "Client", or "Candidate". This component uses a mix of standard Salesforce Lightning, custom CSS for styling, and JavaScript for behavior. It demonstrates usage of Salesforce's `NavigationMixin` for navigating to different pages within the Salesforce app and the `@wire` service to interact with the Salesforce data layer. The component is designed to be responsive, changing layout for mobile devices.

Methods

`navigateToHome()`

Navigates the user to the home page of the application.

```
navigateToHome(){
    this[NavigationMixin.Navigate]({
        type: 'standard__namedPage',
        attributes: {
            name: 'Home'
        },
    });
}
```

`navigateToPage(e)`

Navigates the user to a specific page based on the `data-navinfo` attribute of the clicked element. The method uses a switch case to determine the navigation target based on the attribute value.

```
navigateToPage(e){ /* Implementation */ }
```

`handleMobileNav()`

Toggles the mobile navigation view, showing or hiding the mobile navigation menu.

```
handleMobileNav(){
    this.appClicked = !this.appClicked;
}
```

Lifecycle Hooks

- `connectedCallback()`: Used to perform work in the lifecycle hook but the implementation is empty for this component.

Properties

`portal`

An API property (`@api`) that holds the role information such as "Agency", "Client", or "Candidate" to tailor the navigation bar based on the user's role.

Computed Properties

- `isAgency`, `isCandidate`, `isClient`: Computed properties that return a boolean value based on the `portal` property to conditionally render parts of the template.
- `linkContainerClass`, `closeClass`: Computed properties to determine the CSS class of the link container and close button, toggling mobile view styles based on the `appClicked` state.

Use Case

This component can be used in any Salesforce application that requires a dynamic and responsive navigation bar, capable of adjusting its content based on user roles. It's suitable for applications with different types of users, each needing access to different areas of the application. By detecting the user's role, the navigation bar ensures users can easily find and access the features and information relevant to them.

Important Notes

- The component utilizes Salesforce's `NavigationMixin` to handle navigation. Ensure that your Salesforce environment supports this feature.
- The `@wire(CurrentPageReference)` is used to log the current page reference but can be extended for more advanced routing and state management use cases.
- The CSS uses custom properties (e.g., `var(--primary-color)`) which should be defined in the imported `c/nSharedCss` file or another global CSS file.
- Media queries in the CSS ensure the navigation bar is responsive and adapts to different screen sizes, switching to a hamburger menu on smaller screens.
- The JavaScript `switch` statement in `navigateToPage` method outlines how navigation is handled based on the `data-navinfo` attribute. It's customizable based on the specific pages and navigation paths relevant to your application.
- This component assumes the presence of certain Salesforce object pages (e.g., `Job_Order__c`, `Timesheets__c`) that might need to be adjusted based on the actual Salesforce configuration and object schema in use.

nNotificationBtn

Description

The `NNotificationBtn` is a Salesforce Lightning Web Component (LWC) designed to serve as an interactive notification button, displaying notifications to the user when clicked. It has several key features:

- Shows an icon to indicate the notification status (new notifications available, no new notifications, etc.).
- Reveals a dropdown list that contains notification items when the notification icon is clicked.
- Supports navigation to a specific page when the notification icon is engaged with, particularly if no new notifications are present.
- Can dynamically display notifications count and handle the visibility of the notifications dropdown.

Methods

`handleNotification(e)`

The `handleNotification` method is an event handler for clicks on the notification icon. It toggles the visibility of the notification dropdown, updates the `showNotifications` property, and may navigate the user to another page based on specified conditions.

Example usage:

```
onclick={handleNotification}
```

`handleMouseOver(e)`

Currently commented out in the template. It is designed to upscale the notification icon on mouse over.

```
// onmouseover={handleMouseOver}
```

`handleMouseOut(e)`

Currently commented out in the template. It would reset the icon's scale back to normal when the mouse is moved away.

```
// onmouseout={handleMouseOut}
```

Properties

`notificationArr (@api)`

An array of notification objects. Each object in the array represents a notification item with properties that can be displayed in the notification dropdown.

`notificationDropdownStyle (@api)`

A string that represents custom styling for the notification dropdown. This allows for external customization of the dropdown's appearance.

`tileId (@api)`

A string representing the identifier of the specific tile associated with this notification button.

`tilePageName (@api)`

A string that specifies the name of the page to navigation to if certain conditions are met (no new notifications and the button is clicked).

`numberNotReadAccount (@api)`

An integer indicating the number of unread notifications relevant to the account.

Use Case

The `NNotificationBtn` component is useful in user interfaces where notifications play a critical role in user interaction and information delivery. It can be used:

- On dashboards, to alert users about new activities or updates that require their attention.
- In applications with event-driven updates or communications that need to be acknowledged by the user.

Important Notes

- The component's visibility and behavior can be customized using the exposed `@api` properties, such as `notificationArr` and `notificationDropdownStyle`.
- It is designed to be responsive, adapting its layout for desktop and mobile views using CSS media queries.
- The component relies on the parent component or application to provide notification data (`notificationArr`) and configuration (`tileId`, `tilePageName`).
- The CSS imports from 'c/nSharedCss' and 'c/nuCSS' indicate that the component is using shared styles, which need to be available in the Salesforce org for the component to render properly.
- Although commented out in the template, mouseover and mouseout handlers show potential for interaction effects that can enhance the user experience if enabled.

nNotificationItem

Description

This Salesforce Lightning Web Component (LWC) named `NNotificationItem` is designed to display individual notification items, which are part of a larger notification framework presumably. This component uses the Lightning Navigation Service to redirect users when they interact with a notification. It utilizes conditional styling to visually distinguish between read and unread notifications.

Methods

- **handleGotoUrl**

This method is responsible for navigating the user to a specific Salesforce record or named page based on the `notificationItem` property. Unread notifications are also presumably marked as read (code commented out), and the `notificationId` is stored in the localStorage for later use.

- **ntfItemTitleClass**

A getter that returns a CSS class based on whether the `notificationItem` has been read. It returns `'ntf-item-title-bold'` for unread notifications to make the title bold, and `'ntf-item-title'` for read notifications.

Properties

- **notificationItem** (exposed via `@api`)

This property is meant to hold the data of an individual notification item. It must be passed into the component from its parent.

Use Case

This component is used within a broader notification system in a Salesforce application. When included in a page, it visualizes notification items and allows users to interact with them. On click, it navigates the user to the related Salesforce record or a named page, making it an integral component for user engagement and action on received notifications.

HTML Structure

The component's template is straightforward, consisting of a `div` that binds the `handleGotoUrl` method to its `onclick` event. It displays the notification's title, reference text, and body. The title's CSS class changes based on whether the notification has been read or not, providing visual feedback to the user.

```
<template>
    <div onclick={handleGotoUrl} class="ntf-drop-item">
        <div class={ntfItemTitleClass}>{notificationItem.title}</div>
        <div>{notificationItem.referenceText}</div>
        <div>{notificationItem.body}</div>
    </div>
</template>
```

CSS

The component comes with styles that define basic font sizing, padding, margins, and hover effects to enhance user experience. Additionally, it imports shared styles with `@import 'c/nSharedCss';`.

```
.ntf-drop-item {
    font-size: 14px;
    padding: 5px;
    margin: 5px 5px 10px 5px;
    transition: all 100ms;
    cursor: pointer;
    border-bottom: 1px solid var(--grey-light-1);
}

.ntf-drop-item:hover {
    background-color: var(--primary-color-light-2);
}

.ntf-item-title {
    font-size: 16px;
}

.ntf-item-title-bold {
    font-size: 17px;
    font-weight: 700;
}
```

Important Notes

1. **NavigationMixin**: The component extends `NavigationMixin` to use the Navigation Service in Lightning Web Components efficiently, making it capable of navigating to Salesforce record pages or named pages dynamically.
2. **Read Marking Logic**: The logic for marking notifications as read (commented out in the JS code) indicates an intended feature to change the state of a notification upon the user's action. This functionality should be properly implemented and tested to ensure correct behavior.
3. **Local Storage Use**: Storing the `notificationId` in the localStorage is a notable method for passing values between components/pages. However, it's essential to consider security and potential limitations of localStorage in complex applications.

nSettingBtn

Description

The `NSettingBtn` component provides a customizable settings button for Salesforce Lightning Web Components. When clicked, the button displays a dropdown menu that includes options to select a view and profile, and input fields for Account ID and Contact ID. These settings allow users to configure their preferences or inputs directly within the UI, ensuring a flexible user experience.

Methods

- `handleSettings()`: Toggles the visibility of the settings dropdown.

```
this.showSettings = !this.showSettings;
```

- `handleViewInput(e)`: Updates the `viewValue` property based on the user's selection from the view combobox.

```
this.viewValue = e.detail.value;
```

- `handleProfileInput(e)`: Updates the `profileValue` property based on the user's selection from the profile combobox.

```
this.profileValue = e.detail.value;
```

- `handleAccountId(e)`: Updates the `accountIdValue` property based on the user input in the Account ID input field.

```
this.accountIdValue = e.detail.value;
```

- `handleContactId(e)`: Updates the `contactIdValue` property based on the user input in the Contact ID input field.

```
this.contactIdValue = e.detail.value;
```

Properties

- `showSettings`: A boolean indicating if the settings dropdown is visible.
- `viewValue`: Stores the selected view option.
- `profileValue`: Stores the selected profile option.
- `accountIdValue`: Stores the entered Account ID.
- `contactIdValue`: Stores the entered Contact ID.
- `viewOptions`: Returns an array of objects representing the view options.
- `profileOptions`: Returns an array of objects representing the profile options.

Use Case

This component is ideal for applications that require user customization or settings adjustments within the Salesforce environment. Users can easily configure their preferences, such as selecting a specific view or profile, and entering essential identifiers like Account ID and Contact ID directly from a neatly organized dropdown menu, enhancing the overall user experience.

Important Notes

- Make sure to import necessary Salesforce modules like `LightningElement`, `lightning-icon`, etc., for this component to function properly.
- The component utilizes a CSS file that includes both component-specific styles and an import statement for shared CSS, ensuring consistent styling across the application with the use of variables.
- Mobile responsiveness is considered in the CSS design, making the settings dropdown adapt to different screen sizes.
- The Lightning Web Components (LWC) framework handles reactivity efficiently, updating the UI in response to user inputs and selections without manual DOM manipulations.
- Ensure the property and method names used in the JavaScript class match those referenced in the template to maintain the component's functionality.

```
<template>
    <!-- Component HTML goes here -->
</template>
```

By adhering to these guidelines and implementing this component accordingly, developers can enhance Salesforce applications with a customizable, user-friendly settings interface.

nSharedCss

Description

This documentation explains the CSS (Cascading Style Sheets) code for a Salesforce Lightning component. The provided code focuses primarily on defining custom CSS variables that establish a color scheme for the component. These variables can be used throughout the component's styles to ensure consistency and simplification of theme adjustments.

Properties

The CSS code defines several custom properties (variables) that store color values. These variables are defined within the `:host` pseudo-class, making them available throughout the component's styles.

1. `--primary-color`: Represents the primary color used in the component (`#F5006B`).
2. `--primary-color-light-2`: A lighter variant of the primary color (`#d5f1f8`).
3. `--primary-color-dark`: A darker variant of the primary color (`#39b6dd`).
4. `--black`: Defines the color black (`#010101`).
5. `--grey`: Represents a standard grey color (`#5b5b5b`).
6. `--grey-light-1`: A lighter shade of grey (`#efefef`).
7. `--grey-light-2`: An even lighter shade of grey (`#f9f9f9`).
8. `--red`: Represents the color red.
9. `--green`: Represents the color green.
10. `--purple`: Represents the color purple.
11. `--orange`: Represents the color orange.
12. `--primary-background`: Defines the primary background color used in the component (`#f5f6f8`).

Use Case

These CSS variables can be employed throughout the component's CSS file(s) to maintain color consistency and facilitate theme customization. For example, to set the background color of a class named `.myComponent` to the primary color, you would use:

```
.myComponent {
    background-color: var(--primary-color);
```

```
}
```

Important Notes

- CSS variables defined in a Salesforce Lightning component using the `:host` pseudo-class are scoped to the component. This means they will not leak out and affect other components or the global style unless explicitly made global.
- Using CSS variables allows for easier maintenance and updates to themes or color schemes, as changes only need to be made in one place.
- Despite their benefits, ensure browser compatibility for your target audience since CSS variables have good but not universal support.
- Remember that CSS variables can be overwritten within the component to cater to specific needs, offering flexibility in styling different parts of the component differently while keeping the theme consistent.

This setup defines a clear, maintainable, and easily adjustable color scheme for Salesforce Lightning components, emphasizing the importance of using CSS variables for efficient style management.

nSquareTile

This Lightning Web Component (LWC) is designed to display a navigable tile representing either a Salesforce object or a specific page with rich visual cues including a customizable icon, optional notifications, and dynamic navigation based on the tile's configuration. It efficiently handles user interactions to navigate to desired targets within the Salesforce system. Below is the documentation for the component separated into sections.

Description

The `NSquareTile` component is a versatile, navigable tile that can represent different entities within Salesforce, such as a Salesforce object list view or a named page. It comes with customization options, including an icon with color variations, a label, and the ability to display notifications. The component makes use of conditional rendering to adjust its functionality and appearance based on the provided properties.

Methods

1. `navigateToObjectListView()`:
   Invokes the standard Salesforce object page navigation with specific filter criteria.

```
this[NavigationMixin.Navigate]({
    type: "standard__objectPage",
    attributes: {
        objectApiName: this.tileData.objectApiName,
        actionName: "list"
    },
    state: {
        filterName: this.tileData.filterName,
        [this.tileData.filterIdPropertyName]:
this.tileData.filterIdValue
    }
});
```

2. `navigateToPage()`:
   Navigates to a named Salesforce community page based on the `pageName` attribute.

```
this[NavigationMixin.Navigate]({
    type: 'comm__namedPage',
    attributes: {
        pageName: this.tileData.pageName
    }
}, false);
```

Properties

1. `tileData` (@api):
   An object containing configuration details for the tile, including `pageName`, `objectApiName`, `filterName`, `iconName`, `iconColor`, `label`, and `value`.
2. `notificationArr` (@api):
   An array of notifications to be displayed on the tile. If populated, it triggers the display of notification indicators.
3. `showNotifications`:
   A boolean indicating whether the notification indicator should be shown. Automatically set based on the presence and length of `notificationArr`.
4. `compClass` (getter):
   Dynamically determines the CSS class to be applied to the component's icon based on the `iconColor` attribute of `tileData`.

Use Case

The `NSquareTile` component is ideal for dashboard implementations within Salesforce Lightning Experience or Salesforce Communities where a visual, navigable interface is needed to represent and access objects or specific pages quickly. It enhances user experience by providing immediate visual cues and simplifying the navigation process.

Important Notes

- Ensure that the `tileData` property is correctly populated to define the tile's behavior and appearance. Misconfiguration may lead to unexpected behavior.
- The `notificationArr` property must be an array. Even if notifications are not required, this property should be defined, at least as an empty array, to avoid console errors.
- The component makes extensive use of Salesforce Lightning Design System (SLDS) classes for styling but introduces some custom styling that may need adjustments based on the context of usage.
- The component is dependent on the `lightning/navigation` module for navigating to the target pages or object lists, ensuring seamless integration within the Salesforce ecosystem.

The `NSquareTile` LWC provides a flexible, visually appealing way to represent and access various entities within Salesforce, enhancing navigation and user experience.

nTileBox

Description

The Salesforce Lightning component `NTileBox` presents a customizable, scrollable tile container for displaying a list of tiles. The tiles are represented by another custom component `c-n-square-tile`, and can be horizontally scrolled if enabled. The component is designed to encapsulate a responsive design approach, adapting its appearance and functionalities based on screen width and whether scrolling is enabled.

Methods

- **handleSideScroll(e):** This method implements the behavior for scrolling tiles to the left or right within the container, based on the user holding down the mouse button over the respective scroll icon (`utility:chevronleft` for left, `utility:chevronright` for right). It uses a set interval to continue scrolling as long as the mouse button is held down.

```
handleSideScroll(e){
    let direction = e.target.getAttribute('data-scrollbtn') === 'right'
? 20 : -20;
    let elem = this.template.querySelector('.tile-container');
    this.mouseInterval = setInterval(() => {
        elem.scrollLeft += direction;
    }, 25);
}
```

- **handleStopSideScroll():** This method stops the scrolling action initiated by `handleSideScroll` by clearing the interval set by the latter. It is called when the mouse button is released or the mouse leaves the scroll icon.

```
handleStopSideScroll(){
    clearInterval(this.mouseInterval);
}
```

Properties

- **tileArray (@api):** An array of objects where each object represents data for a tile to be displayed.
- **boxLabel (@api):** A string representing a label displayed at the top of the tile box.
- **scrollable (@api):** A Boolean indicating if the tile container should allow horizontal scrolling.
- **notificationArr (@api):** An optional array passed to tiles, presumably for displaying notifications.
- **tileBoxStyle (@api):** A string for inline CSS style to customize the tile box's appearance.
- **isUtility (@api):** A Boolean to indicate if the tile box is being used as a utility component, affecting tile visibility.

Use Case

The `NTileBox` component is ideal for scenarios requiring a visually appealing and interactive method to display a collection of related items or actions. For instance, representing a series of projects, contacts, or tasks in an application dashboard, where users can quickly navigate through the collection. Its responsive and scrollable features ensure a seamless user experience across devices.

Important Notes

- The component dynamically checks if there are any tiles to display through the `hasTiles` getter, making it versatile for dynamic data loading scenarios.
- The responsiveness is handled via CSS, particularly the `.tile-container` class, which changes its layout based on the `scrollable` property and screen size.
- Due to the component being highly customizable through its API properties, thorough testing is recommended to ensure its behavior aligns with the intended use case, especially for dynamic data loading and styling.

nuApproveTimesheetsPageLayout

Description

This Salesforce Lightning Web Component (LWC) is designed for the purpose of managing and approving timesheets and expenses. It dynamically renders tabs for timesheets and expenses depending on the availability of records. The component pulls data such as the user's contact ID, associated account information, and counts of records needing approval. It also adjusts its styling based on the associated account's preferences or falls back to a set of default styles. The component makes use of Apex controllers to fetch necessary data and utilizes conditional rendering within the template to enhance user experience.

Methods

connectedCallback()

Executes when the component is inserted into the DOM. This lifecycle hook initiates the process of fetching the current user's contact ID, their associated account information, and checks for the existence of timesheet and expense records. It conditionally sets the styling of the component based on account preferences.

- **Process Flow**:
  1. Checks if a `testContactId` is provided; if so, it uses this for testing purposes, otherwise, it fetches the actual contact ID.
  2. Fetches associated account information using the contact ID.
  3. Sets styling based on account preferences or defaults.
  4. Checks for the existence of timesheet and expense records and updates component state accordingly.
  5. Fetches counts of timesheets and expenses needing approval.

Properties

- **@track wiresLoaded**: Indicates if the necessary data has been loaded.
- **@api testContactId**: Optionally provided contact ID for testing purposes.
- **timesheetsExist**: Boolean indicating if timesheet records exist for approval.
- **expensesExist**: Boolean indicating if expense records exist for approval.
- **title**: Static title used in the page header.
- **countTimesheet**: String displaying the count of timesheets needing approval.
- **countExpense**: String displaying the count of expenses needing approval.

Use Case

This component can be used in an approvals dashboard where managers or approvers need to review and approve timesheets and expenses submitted by employees or team members. It provides a clear indicator of the pending approvals and organizes them into tabs for efficient processing.

Important Notes

1. **Styling**:
   The component's styling dynamically changes based on the associated account's preferences. If certain style-related fields (`Use_Portal_Theme__c`, `Tertiary_Color__c`, `Secondary_Color__c`, `Primary_Text_Color__c`) are set, the component uses these values. Otherwise, it uses default styles provided by `getPageStyles` from `nuUtils`.
2. **Data Fetching**:
   The component heavily relies on Apex controllers to fetch necessary data. Apex methods used include:

   - `fetchContactId`
   - `fetchAccountByContactId`
   - `fetchRecordsExist`
   - `fetchAllCounts`

   Proper error handling and loading indicators are implemented to enhance user experience.
3. **Conditional Rendering**:
   The use of `<template if:true={...}>` and `<template if:false={...}>` directives

facilitates conditional rendering based on the state of data (e.g., `wiresLoaded`, `timesheetsExist`, `expensesExist`).

4. **Testability**:
   Through the use of the `@api testContactId` property, the component allows for ease of testing by manually setting a contact ID, bypassing the need to fetch it dynamically.

nuAvailableAgencies

Description

This Salesforce Lightning Web Component (LWC) called `NuAvailableAgencies` is designed to select and save multiple agencies for a Job Order record. It utilizes a combination of Salesforce's UI components, Apex calls, and the Lightning Data Service to fetch, display, and store selections related to available agencies.

Methods

- **getJobOrder**: Utilizes the `@wire` service to fetch current Job Order details based on the `recordId` and sets the company and available agencies if present.
- **setAgencies**: Wired method that invokes the Apex method `fetchAvailableCompanyAgencyRelationships` to fetch available agencies related to the company. It also pre-processes the response to format it for the checkbox group.
- **handleAgencySelect**: Manages selection logic for agencies, including the special handling of a "Select All" option.
- **saveAndClose**: Saves the selected agencies back to the Job Order record and then closes the quick action modal.
- **closeAction**: Simply dispatches an event to close the quick action without saving.

Properties

- **recordId** (`@api`): The ID of the current Job Order record. This is an API property, making it accessible from the component's parent.
- **company** (`@track`): Tracks the company related to the Job Order.
- **agencyOptions** (`@track`): An array of objects storing agencies' labels and values to be used in the checkbox-group component.
- **isAllSelected** (`@track`): A boolean flag to determine if the 'Select All' option is selected.
- **agencyValues** (`@track`): An array that holds the values of selected agencies.

- **showAgency** (`@track`): This property does not appear to be used in the supplied code and could be a remnant from a previous version.

Use Case

This component could be used as a quick action on Job Order records to allow users to select multiple agencies that are available for that specific Job Order. The option to "Select All" makes it easier for users to quickly assign all agencies if necessary. The selection is persisted back to the Salesforce database when the "Save" button is clicked.

Important Notes

1. **Error Handling**: The code provides very basic error handling, mostly just console logging errors. In a production environment, it's recommended to implement user-friendly error messaging.
2. **Duplicate "All" Option**: The implementation pushes the "Select All" option into the `agencyOptions` array directly in the `setAgencies` method without checking if it already exists. This could lead to duplicates if the method is called multiple times.
3. **Security and Permissions**: Make sure that the Apex class `nuService_nuJobOrder` is properly secured for the intended user base, and users have the necessary permissions on the `Job_Order__c` object and related fields.
4. **Transitory @track**: The `@track` decorator is not necessary for properties in LWC unless you're working with arrays or objects and you want the UI to react to deep changes inside those objects. Simple fields like strings, booleans, and numbers are reactive by default.
5. **Lightning Data Service (LDS)**: Utilizes LDS (`getRecord`, `updateRecord`) for efficient data handling with minimal Apex reliance - fetches and updates record information declaratively. Ensure the current user has access to the fields being queried and updated by LDS.

```
// Example Method usage
saveAndClose() {
    let agencyString = this.agencyValues.filter(value => value !==
'All').join(';');
    updateRecord({ fields: { Id: this.recordId, Available_Agencies__c:
agencyString } })
        .then(() => {
            this.dispatchEvent(new CloseActionScreenEvent());
        })
        .catch(error => {
            console.error(error);
            // Handle error (show to user)
```

```
        });
}
```

nuCalendar

Description

The component, `NuCalendar`, is a Salesforce Lightning Web Component (LWC) designed to offer a rich interactive calendar interface. It is built using the FullCalendar library along with Salesforce Apex for data handling. The component allows users to view and manage shift-related events stored in Salesforce, including but not limited to shifting availability, confirmed shifts, and open shift claims. It supports various views such as month, week, and day views, along with a list view for a complete overview of scheduled events. Features include navigating through dates, adding new shifts, selecting and viewing event details, and interacting with the calendar using clicks and selections.

Methods

Public

1. `changeEventsHandler(info)`
   - **Description**: Handles updates to events when selections or modifications are made on the calendar.
   - **Parameters**:
     - `info` (Object): Object containing details about the action and the selected events.
2. `nextHandler()`
   - **Description**: Navigates the calendar to the next date range based on the current view.
3. `previousHandler()`
   - **Description**: Navigates the calendar to the previous date range based on the current view.
4. `dailyViewHandler()`
   - **Description**: Changes the calendar view to display daily events.
5. `weeklyViewHandler()`
   - **Description**: Changes the calendar view to display weekly events.
6. `monthlyViewHandler()`
   - **Description**: Changes the calendar view to a month grid.
7. `listViewHandler()`

- **Description**: Changes the calendar view to display events in a list format.

8. `today()`
   - **Description**: Sets the calendar view to the current date.
9. `refresh()`
   - **Description**: Refreshes the calendar, typically after changes in event data or filter criteria.
10. `handleScroll(event)`
    - **Description**: Prevents propagation of the scroll event to avoid unintended behaviors.
11. `handleEventClick(event)`
    - **Description**: Handles click actions on individual events in the calendar.
12. `handleSelect(event)`
    - **Description**: Allows for the selection of a range of dates or an individual date on the calendar.
13. `lookupRecord(event)`
    - **Description**: Fetches records based on the selection made in a lookup component (not explicitly covered within the provided code but implied by method signature).
14. `handleEditEventClick()`
    - **Description**: Opens a modal or navigates to a page for editing the selected event.
15. `handleBackOpenClick(event)`
    - **Description**: Handler for an action meant to reverse or cancel changes to an event, typically in the context of canceling a shift or similar action.
16. `handleCloseEventDetail()`
    - **Description**: Closes the modal displaying event details.

Private/Internal

1. `createTempEvent(startT, endT, title)`
   - **Description**: Creates a temporary event on the calendar, primarily for visual feedback during interaction.
   - **Parameters**:
     - `startT` (Date): Start time of the temporary event.
     - `endT` (Date): End time of the temporary event.
     - `title` (String): Title of the temporary event.
2. `hexToRgbA(hex)`
   - **Description**: Converts HEX color codes to RGBA format for CSS styling.
   - **Parameters**:

    - `hex` (String): The HEX color code.
3. `eventSourceHandler(info, successCallback, failureCallback)`
    - **Description**: Function to be used as an event source for the FullCalendar, fetching events dynamically from Salesforce.
    - **Parameters**:
        - `info` (Object): Contains information about the range of dates currently displayed on the calendar.
        - `successCallback` (Function): Callback function to be called on successful retrieval of events.
        - `failureCallback` (Function): Callback function to be called on failed retrieval of events.
4. `init()`
    - **Description**: Initializes the FullCalendar library with various options and event sources to render the calendar interface.

Properties

Public

- `aspectRatio`
- `height`
- `defaultDate`
- `calendarId`
- Various filtering and configuration properties such as `weekView`, `dayView`, `listView`, `title`, `startTime`, `endTime`, `whoId`.

Private

- `calendarLabel`
- `openEventDetailsModal`
- `selectedEvent`
- Various state tracking properties like `fullCalendarInitialized`, `selectedEventEditable`, `selectedEventDay`, `selectedEventStartTime`, `selectedEventEndTime`.

Use Case

An organization uses the `NuCalendar` component within their Salesforce Community to manage work shifts and availability for their staff and volunteers. The calendar provides functionalities like browsing through shifts by week, day, or month, adding new availability, and viewing detailed information about specific shifts including participant details and status.

Important Notes

- The component requires external libraries such as FullCalendar and Font Awesome to be loaded for full functionality.
- Salesforce Apex controllers are used for data operations, such as fetching events (`getEventsNearbyDynamic`) and saving event details (`saveEvent`, `saveEventClaim`).
- The component is designed to be versatile, supporting multiple views and interaction patterns tailored to scheduling and shift management use cases.
- Styling relies on SLDS classes, which should be consistently applied for a unified look and feel within Salesforce Lightning Experience.

nuCandidateCredentialsChecklist

Description

This is a Salesforce Lightning Web Component (LWC) designed for managing the compliance credentials of candidates through various stages in a placement process. The component allows users to view a list of placements, manage credentials for each placement, drag and drop credentials through different stages, and add new compliance credentials. It includes a modal form for specifying additional credential details. This component is built to be used within the Salesforce Lightning Experience and leverages the Salesforce Lightning Design System (SLDS) for styling.

Methods

1. **connectedCallback**
   - Executes when the component is inserted into the DOM.
2. **handleNeedToSpecify**
   - Handles the event for specifying additional details for a compliance credential. Triggers the modal to open.
3. **closeModal**
   - Closes the modal form.

4. **handleAddClick**
    - Handles the click event to add a new compliance credential. Opens the modal form in a specific mode for adding a credential.
5. **handleFormSuccess**
    - Handles the successful submission of the form within the modal. This triggers a refresh of data and closes the modal.
6. **handleManageButtonClick**
    - Handles the event when the "Manage" button is clicked. It selects the placement for which to manage credentials.
7. **handleBackClick**
    - Handles the event for returning to the list of placements from the credential management view.
8. **handleListItemDrag**
    - Captures the ID of the compliance credential being dragged in the drag and drop operation.
9. **handleItemDrop**
    - Handles the drop event of a compliance credential into a new stage. May trigger a prompt for a reason if moving to a previous stage.
10. **openReasonMovingBackStageModal**

- Opens a prompt for entering a reason when a credential is moved back to a previous stage.

11. **updateHandler**

- Handles the update operation for changing the stage of a compliance credential, including updating the reason for moving back, if applicable.

12. **showToast**

- Displays a toast message to the user for various actions, such as successful updates.

Properties

1. **recordId**
    - API: `@api`
    - ID of the current record.
2. **columns**
    - `@track`

- Columns configuration for the data table listing the placements.

3. **placementsResult, placements, selectedPlacement, selectedCompCreds**
   - `@track`
   - Various tracking properties for managing state related to placements and their corresponding compliance credentials.
4. **selectedStatus, selectedStage, pickVals, statusPickVals**
   - Various properties for managing the stages and statuses of compliance credentials.
5. **containerStyle, calcWidth, sectionHeadingStyle**
   - Styling properties computed based on component state or provided data.
6. **showModal, isCc, accessToNavigationMixin**
   - Boolean flags controlling UI behavior such as modal visibility and access control.

Use Case

This component is intended for use in scenarios where an organization needs to track and manage the compliance credentialing process of candidates through various stages. It allows users to interact with a visual representation of the credentialing process, move credentials through stages, and specify additional details when necessary. It's especially useful in healthcare staffing or similar fields where compliance to specific qualifications or credentials is critical.

Important Notes

- This component assumes that related Apex controllers (`nuService_Contact`, `nuController_CredentialsChecklist`, etc.) are implemented and available in the Salesforce org.
- Lightning Data Service (LDS) and Lightning UI API are heavily utilized for CRUD operations and record data retrieval.
- Usage of the Salesforce Lightning Design System (SLDS) ensures the component aligns with Salesforce's design guidelines.
- Proper error handling should be implemented (some console error logging is included, but UI feedback for errors may be expanded).
- Security features such as checking user permissions and field-level security considerations should be reviewed and implemented as necessary.

nuCandidateCredentialsZIP

Description

This Salesforce Lightning Web Component (LWC) named `NuCandidateCredentialsZIP` is designed to manage and interact with candidate credentials. It provides functionalities such as selecting files, viewing individual files, and downloading selected files either individually or as a ZIP. The component utilizes Apex controllers for backend data retrieval and supports navigation for file viewing and downloading.

Methods

- **selectAll**: Toggles the selection of all credentials in the list based on the user input. It updates the list of selected credentials accordingly.
- **handleFileSelect**: Manages the selection state of individual files. Adds or removes the selected file's identifier from the internal list depending on the checkbox's state.
- **viewFile**: Navigates the user to the URL of the selected file, allowing the user to view it.
- **downloadZIP**: Constructs a URL for downloading all selected files as a ZIP and navigates the user to that URL. It first checks if at least one file is selected.
- **downloadSingle**: Constructs a URL for downloading a single selected file and navigates the user to that URL.

Properties

- **recordId** (`@api`): The Id of the record the component is associated with. This is an API property allowing it to be set externally.
- **candidateCredentials** (`@track`): Tracked property holding the list of candidate credentials fetched from the backend.
- **selectedCredentials**: Internal property holding the list of credentials selected by the user for downloading.

Use Case

This component can be utilized in scenarios where managing candidate credentials is necessary. Typical use cases include:

- Allowing users to select multiple credentials from a list and download them as a ZIP file for easier handling.
- Providing functionality to view or download individual files for closer inspection or specific needs.

- Dynamically loading and displaying credentials based on the associated record, making it adaptable to various contexts like a hiring audit or placement compliance check.

Important Notes

- The component makes use of Salesforce Apex controllers named `nuController_RelatedLists` for backend operations like fetching the name of the object by record Id and retrieving candidate credentials. Ensure these controllers are properly defined and accessible.
- It leverages the `lightning/navigation` service for navigating to URLs for viewing or downloading files, hence requires proper URL constructions for these actions to work as expected.
- The visibility of certain elements, like the "Download ZIP" button or individual "Download" buttons, is controlled based on whether an object name is available or not, highlighting the component's adaptability to different object contexts within Salesforce.
- The `@salesforce/schema` import comment signifies a potential use case where you might want to import specific schema elements, such as fields from a `Hiring_Audit__c` object. This is commented out in the provided code but indicates possible extension points for future customization.

CSS

The component does not include specific CSS code, but it utilizes Salesforce Lightning Design System (SLDS) classes extensively to ensure a consistent look and feel with the Salesforce Lightning Experience.

XML

The component's configuration file (`*.xml`) is not provided, but it's essential to define tags like `<isExposed>`, `<targets>`, and possibly `<targetConfigs>` within this file to control where and how the component can be used within the Salesforce environment.

nuCandidateEnrollmentsChecklist Description

This Lightning Web Component (LWC) named `NuCandidateEnrollmentsChecklist` is designed to manage enrollment processes for assignments. It displays a list of placements in

the enrollment process which allows users to select a specific placement and manage its enrollments. Once a placement is selected, the component dynamically displays enrollment stages and related actions for each enrollment. It supports actions such as viewing enrollment details, adding a new enrollment, and modifying existing enrollments. The component makes use of Salesforce's Lightning Data Service (LDS) for CRUD operations and uses custom Apex methods to fetch data.

Methods

handleObjInfo

This method is used to fetch picklist values for the `Enrollment__c` object. It dynamically sets the `pickVals` and `statusPickVals` attributes based on the picklist values of 'Current Stage' and 'Status' fields, respectively.

```
@wire(getPicklistValuesByRecordType,{objectApiName: ENROLLMENT_OBJECT,
recordTypeId: '012000000000000AAA'})
```

handleResult

Used to fetch placements in the enrollment process via an Apex method `getPlacementsInEnrollmentProcess`. It processes and formats the fetched data for display in a datatable.

```
@wire(getPlacementsInEnrollmentProcess, {contactId : null})
```

handleManageButtonClick

Triggered by clicking the 'Manage' button. It sets the selected placement to the row selected in the datatable and transitions the UI to manage enrollments for the selected placement.

handleBackClick

Sets the component UI back to the state where the user can select a placement from the list.

handleAddClick

Opens a modal form for adding a new enrollment to the selected stage.

closeModal

Closes the modal form without saving changes.

handleFormSuccess

Refreshes the placements result to reflect the changes made in the modal form.

updateHandler

Updates the current stage of an enrollment record using the `updateRecord` method from LDS.

showToast

Displays a toast message indicating the success of an operation.

Properties

- `columns`: Configuration for the columns in the datatable.
- `title`: Title text displayed on the component.
- `headerText`: Header text giving instructions to the user.
- `placementsResult`, `placements`, `selectedPlacement`, `selectedEnrollments`: Variables holding data related to placements and enrollments.
- `contactId`, `accountId`: Store the contact ID and account ID related to the current user.
- `containerStyle`, `sectionHeadingStyle`: CSS styles applied dynamically based on account settings.
- `showModal`, `isEnrollment`: Boolean flags controlling UI state.
- `pickVals`, `statusPickVals`: Hold the picklist values for the enrollment stages and status fields.

Use Case

This component is useful in scenarios where there's a need to manage enrollments for various assignments within an organization. It streamlines the process of selecting placements, viewing/enrolling candidates, and updating enrollment stages, thereby facilitating efficient management of enrollment processes.

Important Notes

1. **Custom Apex Methods**: The component relies heavily on custom Apex methods for data retrieval and operations. Ensure these Apex methods are deployed and have the appropriate access permissions set.
2. **Datatable Selection**: The logic within `handleManageButtonClick` and `handleBackClick` assumes single-row selection from the datatable. Modifications may be required for multi-row selection support.
3. **Styling**: CSS variables and dynamic styles are used. Make sure to define these CSS variables (`--black`, `--background-color`) globally if they are not part of your Salesforce org's default theme.
4. **RecordTypeId Hardcoding**: The `recordTypeId` in `getPicklistValuesByRecordType` wire service is hardcoded. Consider dynamically fetching or parameterizing this value based on your org's requirement.
5. **Security**: Always ensure that user permissions and sharing settings are properly configured, especially when performing DML operations or accessing sensitive data.

nuChatWithAccountManager

Description

The Lightning component `NuChatWithAccountManager` provides a live chat interface within Salesforce, enabling users to send and receive messages in real time. Styled with Salesforce Lightning Design System (SLDS), it facilitates initiating a chat, displaying chat messages between the user and account manager, and sending new messages.

Methods

- **connectedCallback:** Initializes the component by subscribing to the chat channel.
- **disconnectedCallback:** Cleans up the component by unsubscribing from the chat channel before it's destroyed.
- **closeChat:** Closes the current chat session and updates the record to reflect that the chat is closed.
- **handleSubscribe:** Subscribes to a Platform Event to listen for incoming chat messages.
- **handleUnsubscribe:** Unsubscribes from the Platform Event to stop receiving chat messages.
- **handleInputChange:** Handles changes to the input field for new messages, updating the local `newMessageBody` property.

- **sendMessage:** Sends a new chat message by creating a `Chat_Message__c` record.
- **pubsishEvent:** Publishes a Platform Event to notify other subscribers about the new chat message.
- **messageReceived:** Callback for when a new chat message is received, triggering a refresh of chat messages.

Properties

- **isClosedChat (api):** Boolean indicating if the chat is closed.
- **recordId (api):** The record Id of the chat.
- **isPortalUser (api):** Boolean indicating if the current user is a portal user.
- **contactId (api):** The contact Id associated with the user.
- **chatMessages:** An array of chat messages fetched from the server.
- **chatCreatedBy:** The name of the user who created the chat.
- **chatCreatedDate:** The date the chat was created.
- **chatTitle:** The title of the chat.
- **newMessageBody:** Stores the body of the new message to be sent.

Use Case

This component can be used in a community or within internal Salesforce org to facilitate real-time communication between account managers and customers or internal users. It supports sending and receiving text messages, displaying them in a conversational UI, and provides functionalities like closing the chat and real-time notifications for new messages.

Important Notes

1. **Security:** Ensure that Platform Events and CRUD/FLS (Field-Level Security) permissions are properly set up for access to required fields and objects like `Chat__c` and `Chat_Message__c`.
2. **Subscription to Platform Events:** The component subscribes to a Platform Event (`New_Chat_Message__e`) to receive real-time updates. Ensure the named Platform Event exists in the org.
3. **Apex Controllers:** Several server-side behaviors such as fetching chat messages, publishing new chat message events, and sending emails are handled by Apex controllers (`nuService_Contact`). Ensure these Apex controllers are properly defined and accessible by the component.

4. **Styling with SLDS:** The component makes extensive use of SLDS for styling. Modification to these styles should adhere to SLDS guidelines to ensure consistent UX within Salesforce.
5. **Component Cleanup:** The component properly handles lifecycle hooks to subscribe and unsubscribe from Platform Events, crucial for avoiding memory leaks or unnecessary API calls when the component is not in use.

nuClassificationOfBookings

Description

The `NuClassificationOfBookings` component is designed for the Salesforce Lightning platform. It allows users to classify bookings based on employee types (1099, W2), order the booking records by a selected timeframe (weekly, monthly), and select specific days within that timeframe to view relevant booking data. Additionally, users can download this data as a CSV file. The component utilizes the `lightning-quick-action-panel` to present the UI, `lightning-combobox` for dropdown selections, `lightning-input` for date inputs, and `lightning-datatable` to display the fetched booking data as a table.

Properties

- `@api recordId`: Refers to the current record ID context of the component.
- `@track tableData`: Tracks the data to be displayed in the datatable.
- `columns`: Defines the structure of columns shown in the datatable.
- `defaultEmployeeType`, `selectedEmployeeType`: Manages the state of the selected employee type through combobox.
- `orberBy`: Determines the ordering of the booking records (weekly or monthly).
- `dateStart`, `dateEnd`: Stores the start and end dates for the selected timeframe.
- `isCsvDisabled`: Controls the disabled state of the CSV download button based on if there is data available.

Methods

- `handleAccResult(result)`: A wire method to fetch account name using the `getRecord` wire adapter.
- `handleEmployeeTypeSelect(event)`: Handles changes in the employee type selection.
- `handleDaySelect(event)`: Manages the day selection and calculates the start and end dates based on the order by selection (weekly/monthly).

- `handleOrderBySelect(event)`: Handles changes in the order by selection (weekly/monthly).
- `fetchRelatedData()`: Fetches related booking data based on selections of employee type, date start, and date end.
- `buildTableData(timesheets, expenses)`: Processes timesheets and expenses to prepare data for the datatable.
- `downloadCSV(event)`: Initiates the download of a CSV file containing the data displayed in the datatable.

Use Case

The component is ideally used to classify and analyze bookings within the context of a specific Salesforce record. This could be particularly useful for human resources or project management to review resource allocations or financial expenditures related to employees or contractors (1099, W2) over specified timeframes.

Important Notes

- Ensure that Apex controllers (`fetchTimesheetsForEmployees` and `fetchExpensesForEmployees`) are properly set up and accessible.
- The component makes use of `lightning/uiRelatedListApi` and `lightning/uiRecordApi` wire adapters which require `@salesforce/schema` imports specific to the Salesforce Org's schema.
- Custom styling is applied through CSS for layout adjustments.
- Important to check user permissions and field level access for fields used in the Apex controllers and component.

nuComplianceCredentialForm

Description

This Salesforce Lightning Web Component (LWC) named `NuComplianceCredentialForm` is designed for managing compliance documentation associated with placements in a Salesforce org. It operates within the Salesforce Lightning Design System (SLDS) framework, providing a user interface for adding, editing, and viewing compliance documents, credentials, and requirements. The component incorporates modals for interacting with different forms, such as starting a compliance document type form, adding a new document type, or creating a new compliance requirement.

Methods

The component includes several JavaScript methods to handle user interactions and backend communication, including:

- `renderedCallback`: Initializes the component by either setting default values or loading existing information based on the provided `recordFormId`.
- `handleSaveClick`: Handles the "Save" action for the compliance document being edited or created.
- `handleFormOnload`, `handleNewComplianceRequirementSuccess`, `handleNewCredentialSuccess`, `handleFormSuccess`: Methods to handle the successful submission of forms within the component.
- `openNewCredentialModal`, `closeNewCredentialModal`, `openNewComplianceRequirementModal`, `closeNewComplianceRequirementModal`: Methods to open or close modals for adding new credentials or compliance requirements.
- `previewHandler`: Navigates to a URL to preview documents related to the compliance credentials.
- `handleUploadCredentialFinished`, `handleReuploadCredentialFinished`, `handleUploadFinished`, `handleReuploadFinished`: Handle the completion of file upload or re-upload processes.
- `handleCredChange`: Manages changes to the credential input field, updating component state as needed.

Properties

Key properties (decorated with `@api` or `@track`) include:

- `recordFormId`: The ID of the record currently being viewed or edited.
- `instructions`, `specifyMode`, `selectedPlacement`, `selectedStatus`, `selectedStage`: Various metadata and configurations influencing the form's behavior and appearance.
- `filesResult`, `credentialValue`, `reqValue`, `ccValues`: Track state related to file uploads and field values within the forms.
- `credObject`, `reqObject`: Schemas for the `Credential__c` and `Compliance_Requirement__c` objects used in forms.
- `showNewCredentialForm`, `showStartlForm`, `showNewComplianceRequirementForm`: Control the visibility of different forms within the component.

Use Case

The component's primary use is within Salesforce orgs that manage compliance documentation related to placements. Use cases include:

- Adding new compliance credentials to a placement.
- Editing existing compliance documents or credentials.
- Viewing compliance requirements and uploading necessary documentation.

Important Notes

- This component relies on external Apex controllers (`nuController_CredentialsChecklist`) for backend operations such as fetching applications and handling document uploads.
- Custom utility methods, such as `fireToast`, are utilized for displaying messages to the user.
- Several commented code segments indicate areas for potential extension or modification, such as enabling additional form fields or handling different file types.
- The component's behavior is significantly dynamic, relying on the current state and user input to determine which forms to display and how to process submissions.

Code Samples

The following sample shows how a new credential form is toggled on and off within the component using JavaScript methods:

```
openNewCredentialModal(event) {
    this.showNewCredentialForm = true;
    this.showStartlForm = false;
}
```

```
closeNewCredentialModal(event) {
    this.showNewCredentialForm = false;
    this.showStartlForm = true;
}
```

These two methods control the display of the "New Credential" form by toggling the `showNewCredentialForm` and `showStartlForm` Boolean properties.

nuContactRelatedList

Description

This Salesforce Lightning Web Component (LWC) named `NuContactRelatedList` is designed to manage and display candidate records from a Salesforce Org. It utilizes a mixture of standard Salesforce UI elements such as `lightning-datatable`, `lightning-input-field`, `lightning-button`, etc., along with custom components to create a rich user interface for interacting with candidate data. The component's functionality includes searching, sorting, filtering candidates, and creating new candidate records through a modal form.

Properties

- `sortBy` (String, @track): Key by which the list of candidates is currently sorted.
- `sortDirection` (String, @track): The current direction of sorting (ascending or descending).
- `searchString` (String, @track): The current search query.
- `initialRecords` (Array, @track): The initial list of candidate records before any search or filter is applied.
- `enableContactPollsCreation` (Boolean, @track): Controls the visibility of a modal form for creating new candidate records.
- `contactId` (String): The contact Id related to the current user or context.
- `candidateRt` (String): Candidate record type Id.
- `accountId` (String): Account Id associated with the current user or context.
- `containerStyle` (String): Dynamic styling for the component container, based on account settings or theme.
- `sectionHeadingStyle` (String): Dynamic styling for the page header, based on account settings or theme.
- `columns` (Array): Defines the columns to display in the `lightning-datatable`.

Methods

- `handleProviderClassChange(event)`: Handles changes to the provider class selection.
- `handleEnableContactCreation()`: Toggles the visibility of the modal for candidate creation.
- `handleEnableContactCreationSuccess(event)`: Handles success after a new candidate is created, and updates the list.

- `createContact()`: Initiates the creation of a new contact based on input from the modal form.
- `validateFields()`: Validates the fields in the form before submission.
- `doSorting(event)`: Sorts the list of candidates based on selected column and direction.
- `sortData(fieldname, direction)`: Helper method for sorting data.
- `handleSearch(event)`: Filters the list of candidates based on a search query.
- `handleClassFilterChange(event)`: Filters the list of candidates based on selected classifications.
- `doFilter()`: Applies the selected filters to the list of candidates.

Use Case

This component is used for managing candidate records within a Salesforce Org. It allows users to view candidate details, search and filter candidates based on various criteria, and add new candidates through a modal form. It can be particularly useful in recruitment or HR applications where managing a pool of candidates is required.

Important Notes

1. **Security**: Ensure that Apex controllers and methods have the proper security annotations and access controls to protect sensitive data.
2. **Custom Styles**: The component dynamically applies styles based on account settings or themes. Ensure these settings are properly configured to reflect the desired UI.
3. **Component Communication**: The component demonstrates efficient use of LWC lifecycle hooks and wire service to fetch, display, and refresh data.
4. **Validation**: The component includes client-side validation before attempting to create a new record. Ensure server-side validations are also in place for robustness.
5. **Performance**: The component makes use of `@wire` to automatically cache records and reduce server calls, which helps in improving the performance.

nuCreditMemosRelatedList

Description

The Lightning Component `NuCreditMemosRelatedList` is designed to display a list of credit memos related to an invoice within a Salesforce environment using Lightning Web Components

(LWC). It utilizes the Salesforce Lightning Design System (SLDS) for styling and the `lightning-datatable` component for displaying the list of credit memos in a tabular format.

Methods

- `handlePageRef(result)`: Used to handle the current page reference and generate a base URL for navigating to individual credit memo records.
- `handleResult(result)`: Invoked by the `@wire` service to handle the result of fetching related credit memos. It processes data to add additional properties required for displaying in the datatable and handles potential errors.

Properties

- `invoiceId`: An `@api` decorated property that takes an external Invoice ID as input to fetch related credit memos.
- `columns`: Contains the configuration for the columns of the `lightning-datatable`.
- `creditMemos`: An `@track` decorated property that stores the processed list of credit memos to be displayed in the datatable.

Use Case

This component can be used in a Salesforce org to visually represent a list of credit memos associated with a particular invoice. It enhances the user experience by providing a clear, tabular view of critical information regarding each credit memo, including a clickable link that takes the user directly to the detailed view of a credit memo, the worklog associated with it, and its amount. This component is ideal for financial or sales applications where tracking financial transactions and associated details is crucial.

Important Notes

- The component expects the `invoiceId` to be provided externally, potentially as a page attribute or dynamically via another component, to retrieve and display the related credit memos.
- The backend Apex controller method `getInvoiceRelatedCreditMemos` must be implemented properly to return the necessary data fields (`creditMemoId`, `creditMemoName`, `worklogName`, and `amount`) for each credit memo related to the given invoice ID.

- It uses the Lightning Navigation service to generate URLs dynamically for navigating to the credit memo records, which means the format of the URL needs to be consistent with Salesforce's URL structure for records.
- The component is styled using SLDS and custom CSS to ensure consistency with Salesforce's design system, making it visually integrate well within the Salesforce UI.

```
@wire(getInvoiceRelatedCreditMemos, {invoiceId : '$invoiceId'})
handleResult(result) {
    const {error, data} = result;
    if (data) {
        this.creditMemos = data.map(cm => {
            return { ...cm, Id : cm.creditMemoId, nameUrl:
this.currentUrl + cm.creditMemoId}
        })
    }
    if (error) {
        console.error('nuCreditMemoRelatedList error ::: ',
JSON.stringify(error));
    }
}
```

The snippet above shows how the component dynamically handles the fetched credit memos data and potential errors, preparing the data for display in the `lightning-datatable`.

nuCSS

Description

The provided CSS styles are for a Salesforce Lightning component, defining a wide array of custom properties (CSS variables) primarily focused on defining color schemes across various UI elements within the Lightning component framework. These properties cover background colors, border colors, button styles, and more for different aspects of a UI such as forms, tiles, scheduling interfaces, notifications, and general component styling. The CSS leverages CSS custom properties to allow themes and colors to be changed dynamically or maintained more easily throughout the component's styling.

Methods

This piece of code does not define any methods as it is purely CSS for styling.

Properties

The CSS file declares several custom properties for different UI elements:

- `--background-color`: Main background color. Currently set to `#F5006B`.
- `--background-color-client`: Background color for client-specific elements, marked as `#160C28`.
- `--brand-color`: Primary branding color, noted as `#2b85a2`.
- `--form-buttons`: Color for form buttons, set as `#298ca8`.
- `--tile-color`: Background color for tiles, defined as `#F3F3F4`.
- `--notifications-not-read-color`: Color for unread notifications, `orange`.
- `--tile-text-color`: Text color on tiles, `#160C28`.
- Several others focusing on scheduling, spinner, button, accrual panel, and slot-specific styling.

Use Case

When implementing or customizing Salesforce Lightning components, these styles can be used to maintain a consistent look and feel across the application. For instance, by using these properties, a developer can easily change the theme or appearance of their app by modifying the values of these variables in one place rather than updating individual styles throughout the CSS file.

Important Notes

1. **Dynamic Theming**: CSS variables offer flexibility in theming; changing the value of the CSS variables at the root level can dynamically update the appearance without altering the component's logic.
2. **Commented Code**: Several color values are commented out, indicating possible previous options or placeholders for future themes. This is helpful for keeping track of alternative colors but remember to clean up any unnecessary code before production.
3. **Consistency Across Components**: Using these CSS variables across components ensures consistency in the appearance and feel, making the UI predictable and familiar to the users.
4. **Browser Support**: Ensure that the browsers targeted for the Salesforce application support CSS custom properties. Most modern browsers do, but this could be a concern for applications that need to support older browsers.
5. **Maintainability**: Maintaining a large list of CSS variables can become challenging, especially in large applications. It's good practice to comment on these variables or group them in a logical manner to enhance readability and maintainability.

nuDashboard

Description

This Lightning Web Component (LWC) named `NuDashboard` is designed to be a container or a dashboard-type component, which serves as a structure to incorporate other components (`<c-nu-new-dashboard>` as a placeholder for child components is shown in the template). While the component appears to be aimed at serving as a high-level dashboard, the provided code does not include much functionality within the component itself beyond setting up properties for potential use and initialization logging. External utility functions and pubsub modules suggest that this component is intended to work within a larger application where components communicate and data is fetched and manipulated outside the provided code snippet.

Methods

- `connectedCallback`:
  The only method provided in this component is the lifecycle hook `connectedCallback`, which is automatically invoked when the component is inserted into the DOM. The implementation here provides a simple `console.log` statement indicating the start of the `nuDashboard`'s connectivity process.

Properties

The component declares several public properties using the `@api` decorator, making them available for data binding and configuration by other components or external resources.

- `portal`: Intended use is not specified, could be an identifier or configuration for different portal types.
- `boxLabel`: Presumably a string to label or title part of the dashboard.
- `testContactId`: An ID for a contact, likely used for testing or demonstration purposes.
- `testAgencyId`, `testClientContactId`, `testClientAccountId`: These properties are similar to `testContactId` and suggest the component might be usable in contexts involving client and agency data handling or testing.

Use Case

A typical use case for this component would appear to be as a central dashboard within a larger Salesforce Lightning application, possibly for a portal that deals with contacts, agencies,

accounts, etc. The presence of test ID properties indicates this component may be useful in both development/testing environments and production, potentially displaying data related to these IDs or facilitating navigation and actions related to them.

Important Notes

- **Styling & Layout**: The commented CSS suggests a responsive design possibly intended for the dashboard but is currently commented out. The import of `'c/nSharedCss'` indicates that the component relies on externally defined shared CSS for its styling.
- **External Dependencies**: The component imports multiple modules and functions (`logIt`, `fetchContactAndRelatedData`, `registerListener`, `unregisterAllListeners`) from another custom component/module `c/nuUtils` and `c/pubsub`. This dependency on external libraries or components for functionality such as logging, data fetching, and event listening/unregistering is crucial for developers to note, as these components must be included in the application for `NuDashboard` to function correctly.
- **Asynchronous Operations**: The ESLint directive `/* eslint-disable @lwc/lwc/no-async-operation */` at the top of the JS file indicates that asynchronous operations may be used despite being discouraged in certain contexts within LWC, however, no such operations are explicitly included within the provided code snippet.

Code Snippet Examples

Sample `connectedCallback` Use in Console Log:

```
connectedCallback() {
    console.log('nuDashboard connected callback start');
}
```

Property Definition Example:

```
@api testContactId;
```

Understanding the structured design and potential usage scenarios can help further develop this component, integrate it with necessary data sources, and utilize it within a Salesforce Lightning application to serve as a dynamic and interactive dashboard.

nuDateTimeInput

Description

The `NgDateTimeInput` component is a Salesforce Lightning Web Component (LWC) designed to input dates and times. It provides a versatile and configurable input field for dates, times, or both, according to the provided properties. Users have the option to apply or strip out Salesforce Lightning Design System (SLDS) styling, making it adaptable for various UI requirements.

Methods

- **handleChange(event):** This method triggers when the input field loses focus (`onblur`). It prepares an object `realData` containing the field's name, value, a modified version of the value (`mil`), the type of input (time or the specified type), and the associated root date `dte`. It then updates the component's `value` property with the new value and fires a custom event named `datetimechange`, passing the `realData` object as event detail.

Properties

- **@api fieldName:** A unique name to identify the field.
- **@api value:** The current value of the input field. Default is an empty string.
- **@api type:** Specifies the type of input, such as "date", "time", etc. No default value.
- **@api label:** The label text associated with the input field. No default value.
- **@api rootDate:** An additional data point that can be passed for contextual use. No default value.
- **@api timeOnly:** A Boolean to indicate if the input is exclusively for times. Default is `false`.
- **@api stripStyling:** A Boolean to determine if SLDS styling should be applied. If `true`, styling is stripped. Default is `false`.

Use Case

In a Salesforce Lightning experience, where a user needs to input a date, time, or both, depending on the context, the `NgDateTimeInput` component can be utilized to capture this input efficiently. Its flexibility allows for usage in both styled and unstyled contexts, making it suitable for different UI designs. The component also supports additional data like a base date (`rootDate`) for more complex logic.

Important Notes

- The component distinguishes between time-only and other types of input through the `timeOnly` property. This affects how the `type` of the emitted `datetimechange` event's detail is determined.
- The `handleChange` method updates the component's `value` with every input change, ensuring the component's state is always up-to-date.
- Custom events in Lightning Web Components, such as `datetimechange` fired in this component, can be captured by parent components for further handling or processing of the input data.
- It's important to correctly bind the component's properties (`fieldName`, `value`, `type`, `label`, `rootDate`, `timeOnly`, `stripStyling`) when using it to ensure the component behaves as expected.
- The `stripStyling` property offers flexibility in utilizing the component within various design systems, not limited to Salesforce's Lightning Design System (SLDS).

nuDocumentUpload

Description

The Lightning Web Component (LWC) named `NuQuickActionPresentCandidate` is designed to facilitate the process of presenting a candidate in Salesforce. It features a modal dialog with dynamic content based on the candidate's information, particularly focusing on the candidate's CV. The component supports uploading a CV, previewing an existing CV, and submitting the candidate’s information for further processing. It also uses a custom child component `c-nu-provider-presentation-form` for a form presentation and integrates with the Salesforce Lightning Design System (SLDS) for styling.

Methods

- **handleHAResult**: Triggered when receiving the candidate ID from the `c-nu-provider-presentation-form`. This method fetches related files to check if the CV is available and updates the component's state accordingly.

```
handleHAResult(event) {
    ...
}
```

- **submitProviderPresentation**: Invoked when the “Next” button is clicked. This method submits the provider presentation form.

```
submitProviderPresentation() {
    ...
}
```

- **clearProviderPresentation**: Invoked when the “Clear Form” button is clicked. This method clears the provider presentation form.

```
clearProviderPresentation() {
    ...
}
```

- **handleUploadFinished**: Executed when the file upload is completed. It updates the state to reflect the availability of the CV.

```
handleUploadFinished() {
    ...
}
```

- **previewHandler**: Handles the event for previewing the candidate's CV by navigating to the CV's URL.

```
previewHandler(event) {
    ...
}
```

- **closeQuickAction**: Closes the quick action modal.

```
closeQuickAction() {
    ...
}
```

- **doPresent**: Handles the presentation process of the candidate by updating certain records and closing the quick action upon completion.

```
doPresent() {
    ...
}
```

Properties

- **hiringAuditId** (`@api`): The identifier for the hiring audit record.
- **candidateId**: The candidate's identifier, updated after fetching related files.
- **cvPreviewUrl**: The URL to preview the candidate's CV.
- **headerText**: The text displayed in the modal header, default is "Present Candidate".
- **isCvAvailable**: A boolean indicating if the candidate's CV is available.
- **showSpinner**: Controls the visibility of the spinner during asynchronous operations.
- **isPresentButtonDisabled**: A boolean to enable or disable the "Present" button based on certain conditions.
- **showPresentationForm**: Determines whether the presentation form or the CV upload interface should be displayed.

Use Case

The `NuQuickActionPresentCandidate` component is used within Salesforce to streamline the process of presenting candidates to clients. It provides a user-friendly interface for:

- Viewing if a candidate's CV is available and previewing it.
- Uploading a new CV if it's not available.
- Submitting candidate presentation information through a custom form.
- Dynamically updating the presentation status of a candidate.

Important Notes

- The component utilizes Salesforce's `lightning/uiRecordApi` for record operations like fetching and updating records.
- It makes use of custom Apex methods like `getRelatedFilesByRecordId` to fetch related documents (e.g., CVs).
- The component is integrated with SLDS for consistent Salesforce UI styling.
- NavigationMixin from `lightning/navigation` is used for navigating to external web pages, such as CV previews.
- This component should be placed in contexts where a `hiringAuditId` can be provided as it's essential for its operations.
- The component assumes that the `c-nu-provider-presentation-form` child component exposes methods for submitting and clearing the form, which is crucial for interaction.

nuDragAndDropCard

Description

This Salesforce Lightning Web Component (LWC) named `DragAndDropCard` displays draggable cards with data pulled from a Salesforce record, which is intended for use in stages such as compliance credentials, hiring audits, enrollments, and invoices. The component's appearance and functionality change based on the type of data it is meant to represent. It utilizes SLDS (Salesforce Lightning Design System) for styling to maintain consistency with Salesforce's UI.

Properties

1. `@api stage` - The current stage of the process, used to determine card display.
2. `@api record` - The Salesforce record to display information from.
3. `@api isHiringAudit, isComplianceCredential, isEnrollment, isInvoice` - Boolean flags indicating the type of card to display.
4. `@api isDisabled` - A Boolean property to indicate if the drag feature should be disabled.
5. `@api accessToNavigationMixin` - A Boolean to indicate if navigation functionality is available.

Methods

`connectedCallback()`

- Lifecycle hook that runs when the component is inserted into the DOM. Used here to log the record data for debugging purposes.

`get isSameStage()`

- A getter method to determine if the current record's stage matches the component's stage for conditional rendering.

`get verifyMode()`

- A boolean getter method to enable the "Verify" button based on the record's credential status and current stage.

`get itemStyle()`

- A getter method that dynamically generates styling for the item based on the record's current stage and the page path.

`navigateOppHandler(event), navigateAccHandler(event)`

- Event handlers for clicking links within the card. They call `navigateHandler` with the appropriate parameters based on the clicked element.

`navigateHandler(Id, apiName)`

- Utilizes the NavigationMixin to navigate to a specific Salesforce record page based on the given `Id` and `apiName`.

`throwNeedToSpecifyEvent()`

- Dispatches a custom event named `needtospecify` that bubbles up with the record's Id in the detail.

`itemDragStart(event)`

- Handles the drag start event for the card. If certain conditions are met, it dispatches a custom event named `itemdrag` or prevents the drag action.

Use Case

The `DragAndDropCard` component is suitable for implementing a draggable interface in Salesforce Lightning Experience where records from various stages (like compliance credentials, hiring audits, enrollments, and invoices) need to be displayed and interacted with using drag-and-drop functionality. It can be especially useful in kanban boards or any process requiring visual management of records through different stages.

Important Notes

- Ensure SLDS (Salesforce Lightning Design System) is properly loaded in your Salesforce org to maintain UI consistency.
- This component depends on record data being correctly formatted and passed to it. Make sure the `record` property is populated with relevant fields (e.g., `Name`, `Status__c`,

`currentStage`, `credential`).

- The component uses NavigationMixin for navigation purposes; hence, it works only within the Salesforce Lightning Experience and Salesforce Mobile App context.
- Custom events (`needtospecify` and `itemdrag`) must be handled by the parent component for the component to function as intended in broader application logic.
- The component specifically handles drag-and-drop functionality with conditions, such as preventing drag in certain stages (`Present`, `Interview`) or based on the `isDisabled` flag, ensuring flexibility in its usage across different application scenarios.

nuDragAndDropList

Description

This Lightning Web Component (LWC) named `nuDragAndDropList` is designed to work within the Salesforce Lightning Experience. It offers a drag-and-drop functionality primarily used to rearrange or categorize records visually. The component displays a list of records as draggable cards that can be dropped into designated areas. The functionality is beneficial for various business processes, such as hiring audits, compliance credential management, enrollment processes, and invoice management. Custom events are used to handle the specifics of dragging and dropping, ensuring a flexible component that can integrate with other parts of a Salesforce application.

Properties

The component has several public properties (`@api`) that can be set by the parent component:

- `records`: The list of records to be displayed.
- `stage`: A property indicating the stage or category of the dropped item.
- `isHiringAudit`: Flags whether the component is used for hiring audits.
- `isComplianceCredential`: Flags whether the component is used for compliance credentialing.
- `isEnrollment`: Flags whether the component is used for enrollments.
- `isInvoice`: Flags whether the component is used for invoice management.
- `isDisabled`: Determines if the drag and drop functionality should be disabled.
- `accessToNavigationMixin`: An optional property, likely intended for integration with Salesforce's NavigationMixin for navigating to Salesforce records or other destinations within the Salesforce Lightning Experience.

Methods

The component defines several JavaScript methods to handle user interactions and lifecycle events:

- `connectedCallback()`: Used to append additional styles if the `isDisabled` property is set to `true`.
- `handleNeedToSpecify(evt)`: Dispatches a custom event named `needtospecify` when a specific action is needed for a dropped item.
- `handleItemDrag(evt)`: Dispatches a custom event named `listitemdrag` when an item is dragged.
- `handleDragOver(evt)`: Prevents the default behavior when an item is dragged over a droppable area to allow for a drop.
- `handleDrop(evt)`: Dispatches a custom event named `itemdrop` when an item is dropped.

Use Case

This component is ideal for implementing drag-and-drop functionality within Salesforce Lightning applications where users need to visually manage a list of items or records. It can be used in scenarios such as sorting tasks, categorizing contacts based on criteria, managing stages in a process (e.g., hiring process, sales pipeline), or simply as a visual tool to improve user interaction in managing records.

Important Notes

- The component utilizes the standard `ul` HTML element as a container for the draggable items, ensuring semantic HTML structure.
- Custom events (`needtospecify`, `listitemdrag`, `itemdrop`) provide hooks for parent components to respond to actions within the drag-and-drop list, making the component flexible and integrable within larger applications.
- The component leverages Salesforce Lightning Design System (SLDS) classes for styling, ensuring consistent look and feel with the Salesforce Lightning Experience.
- It's important to note that the empty `.CSS` file suggests that custom styles specific to this component are not provided, relying instead on SLDS and inline styles for visual appearance.

Example

```
<c-nu-drag-and-drop-list
    records={yourRecordArray}
    stage="initialStage"
    is-hiring-audit={true}
    onneedtospecify={handleNeedToSpecify}
    onlistitemdrag={handleListItemDrag}
    onitemdrop={handleItemDrop}>
</c-nu-drag-and-drop-list>
```

This example shows how to use the `nuDragAndDropList` component within a parent component, passing an array of records (`yourRecordArray`) and configuring it for a hiring audit scenario. It also illustrates how to listen for the custom events emitted by the drag-and-drop component.

nuEnrollmentForm

Description

This Salesforce Lightning Web Component (LWC), `NuComplianceCredentialForm`, is designed to facilitate the process of payer enrollment by allowing users to fill out and submit enrollment records. It includes functionalities such as form submission, modal dialog interactions, and navigation to document review pages. The component leverages several Lightning base components like `lightning-record-edit-form`, `lightning-input-field`, and `lightning-button`, providing a user-friendly interface for record creation and modification.

Methods

- **renderedCallback**: Ensures the `getApplication` function is executed once when the component is rendered.
- **handleSaveClick**: Validates the enrollment document field before submission. It prevents form submission if the document field is empty.
- **previewHandler**: Utilizes the `NavigationMixin` to navigate to a URL, allowing users to review documents related to the enrollment process.
- **closeModal**: Dispatches a custom event `closemodal` to handle the closing of the modal.
- **handleFormSuccess**: Dispatches a custom event `formsuccess` when the form is successfully submitted.
- **handleUploadFinished**: Placeholder for implementation post-upload logic.

- **runGetApplication**: Function scaffold for fetching application-related URLs from an Apex controller. This is currently commented out but serves as a template for asynchronous data retrieval.

Properties

- **recordFormId** `[api]`: The ID of the record to be modified or created by the form.
- **instructions** `[api]`: Instructions or additional information provided to the user.
- **specifyMode** `[api]`: A boolean flag to control the rendering and behavior of certain form fields.
- **selectedPlacement** `[api]`: The chosen placement detail for the enrollment record.
- **selectedStatus** `[api]`: The current status of the enrollment record.
- **selectedStage** `[api]`: The current stage of the enrollment process.
- **reqApplicationFileUrl, compCredApplicationFileUrl, credentialFileUrl** `[track]`: URLs used in the rendering logic to display or hide sections and buttons based on the state of the application.

Use Case

This component is designed for implementation within Salesforce environments requiring a structured method to manage payer enrollments. It's particularly useful in scenarios where conditional rendering of content based on the record's current state or provided instructions is needed. The component facilitates displaying, editing, submitting enrollment records, and reviewing associated documents, improving the UX for end-users navigating the enrollment process.

Important Notes

- **Form Field Conditional Rendering**: The component uses the `specifyMode` flag to conditionally render input fields and manage their required status dynamically.
- **Modal Implementation**: The component encapsulates its content in a modal format using the SLDS modal classes for styling.
- **Custom Event Handling**: Events like `closemodal` and `formsuccess` are custom events that must be handled by the component's parent or enclosing context.
- **Apex Controller Dependency**: The commented-out `runGetApplication` method indicates an intended dependency on an Apex controller named

`nuController_CredentialsChecklist` for data fetching, which is crucial for full functionality but requires separate implementation.

- **Preview Functionality**: The `previewHandler` method demonstrates a pattern for implementing record-related document previews, highlighting the component's extensibility for document management within the enrollment process.

This component serves as a foundational piece for managing payer enrollments within Salesforce, capable of being extended or customized further to fit specific business processes or UI requirements.

nuExpenseApproval

Description

The Lightning Web Component (LWC) `NuExpenseApproval` is designed for approving, rejecting, or viewing expenses in a Salesforce environment. This component utilizes Salesforce Lightning Design System (SLDS) for styling and `@salesforce/apex` to communicate with the Apex classes for CRUD operations on expense records. The component is divided into tabs (`Submitted`, `Approved`, `Rejected`) each displaying relevant expenses in an accordion layout. The component also features modal dialogs for additional interactions such as rejecting an expense with a reason.

Methods

1. **handleRowAction**: Invoked when an action is performed on a row in the `lightning-datatable`. It handles different row actions like viewing the file associated with an expense.

```
handleRowAction(event) { ... }
```

2. **handleRejectClick & handleApproveClick**: These methods are triggered when the `Reject` or `Approve` button is clicked for an expense. They set up the UI and data for further operations.

```
handleRejectClick(event) { ... }
handleApproveClick(event) { ... }
```

3. **handleApproveAllClick**: Invoked when the `Approve All` button is clicked, iterating over all submitted expenses and approving them.

```
handleApproveAllClick() { ... }
```

4. **handleInputChange**: Handles changes in input fields, specifically for updating the rejection reason.

```
handleInputChange(event) { ... }
```

5. **rejectExpense & closeModal**: `rejectExpense` is called to proceed with the rejection of an expense, while `closeModal` is called to close the modal dialog.

```
rejectExpense() { ... }
closeModal() { ... }
```

6. **handleDateChange**: Handles changes in the date range filters.

```
handleDateChange(event) { ... }
```

Properties

1. **expenses**: Getter property to check if there are any expenses in the list.

```
get expenses() { ... }
```

2. **showApproveAll**: Determines if the `Approve All` button should be visible based on whether there are submitted expenses.

```
get showApproveAll() { ... }
```

Use Case

To utilize `NuExpenseApproval` in your Salesforce org, ensure that Apex classes for fetching and updating expenses (`fetchExpensesForApproverContact`, `approveExpense`,

`rejectExpense`) are correctly set up. This component is designed for approvers to manage expense records effectively, offering functionalities such as approval of individual or all expenses, rejection of expenses with a reason, and viewing attached files.

Important Notes

- The component relies on Apex controllers for backend operations. Make sure the Apex classes and methods are properly defined and have the necessary access rights.
- Custom styles are imported from `c/nuCSS`. Ensure this static resource or similar is available in your Salesforce org for styling purposes.
- The component uses SLDS classes extensively for its layout and styling. Familiarity with SLDS is recommended for customization.
- Ensure that the API names used in the component (e.g., field names in the `lightning-input` and `lightning-datatable`) match those in your Salesforce org's schema.

Conclusion

`NuExpenseApproval` offers a comprehensive solution for managing expense approvals within Salesforce. It leverages LWC capabilities, Apex backend operations, and SLDS for styling, ensuring a smooth experience for users dealing with expense records.

nuExpenseView

Description

This Salesforce Lightning Web Component (LWC) named `NuExpenseView` is designed to display and manage expense records in a Salesforce Community environment. It features a complex UI that includes tabs for different expense statuses (Unapproved, Approved, and Rejected), with each tab containing a dynamic list of expenses. The component allows users to filter expenses by candidates and specialties, view detailed expense information, approve, reject, and resubmit expenses.

Methods

1. **approveExpenseAsync:** Approves an expense based on the given expense ID, user name, and approver ID.
2. **rejectExpenseAsync:** Rejects an expense with the provided reason, based on the given expense ID, user name, and approver ID.

3. **filterExpense:** Filters the displayed expenses based on user inputs, such as candidate and specialty filters, and date range.
4. **handleViewFile:** Handles the event to view the attached file for an expense.
5. **handleAttachmentsClick:** Opens the page or modal to attach files to an expense.
6. **handleViewExpense:** Handles the event to view details of a rejected expense for potential resubmission.
7. **handleSectionToggle:** Toggles the visibility of sections in the accordion UI element when a section header is clicked.

Properties

- **expenses:** A getter that returns the list of expenses.
- **showApproveAll:** Determines whether the "Approve All" button should be shown based on the number of submitted expenses.
- **timesheetOptions:** Retrieves and formats options for selecting a timesheet.
- **showsOnWREXPENSE:** Determines if the work record expenses feature should be shown.
- **openModal:** Tracks the state of the modal for inputting a rejection reason.

Use Case

This component is used in Salesforce Communities to allow users (such as managers or approvers) to view, filter, approve, reject, or resubmit expense records. It's designed to handle different states of expenses, provide a user-friendly interface for managing them, and offer functionalities like file attachment viewing and expense filtering.

Important Notes

1. **Communication with APEX:** The component relies on methods (`approveExpense`, `rejectExpense`, `fetchExpensesForAgencyContact`, `fetchContactId`, `fetchSingleExpense`) from APEX classes to perform CRUD operations on expense records. Ensure these APEX methods are correctly defined and accessible.
2. **NavigationMixin:** The component uses `NavigationMixin` for navigation purposes, such as redirecting to a file attachment page or opening a resubmission form. Ensure import and usage follow Salesforce best practices.
3. **Accessibility:** Attention should be paid to making the UI accessible, including proper labeling, keyboard navigation, and ARIA attributes where applicable.

4. **Performance:** Given the potentially large set of expense records and dynamic data binding, consider implications on performance and user experience, such as loading times and responsiveness of filters.
5. **Community Compatibility:** Ensure the component and its features are fully compatible with Salesforce Community Cloud, considering permissions, visibility, and user roles.

nuExpiringDocuments

Description

The `NuExpiringDocuments` Lightning Web Component (LWC) is designed to provide users with a view of expiring documents related to specific candidates within Salesforce. This component displays a list of candidates with details such as their credentials, specialties, and the number of expiring documents. Users can expand a candidate's row to view detailed information about each expiring document, including the credential name, specialty, expiration date, document type, and status. Additionally, users can update a credential directly from within the component using a file upload functionality.

Methods

1. **handleContactResult**
   - Description: Wired method to fetch the contact ID from the user's record and subsequently load related account and expiring document information.
   - Parameters: contactResult (object) containing the contact data.
2. **doExpandCredentials**
   - Description: Toggles the visibility of the expanded view that shows details of expiring documents for a specific candidate.
   - Parameters: event (Event) used to retrieve the document ID (`docId`) from the clicked row.
3. **doViewCred**
   - Description: Handles the navigation to view a specific credential file by fetching the credential's URL and using the NavigationMixin to navigate.
   - Parameters: event (Event) used to retrieve the credential ID from the selected document.
4. **handleFileUploaded**
   - Description: Manages the file upload process for updating a credential, including size validation and invoking the APEX method to upload the credential.
   - Parameters: event (Event) containing the file selected by the user for upload.

Properties

1. **recordCount** (@track)
   - Description: Tracks the number of records (expiring documents) returned.
2. **expiringDocuments** (@track)
   - Description: Holds the list of expiring documents and their details.
3. **title**
   - Description: The title of the component, set to 'Expiring Documents'.
4. **showSpinner**
   - Description: Controls the visibility of a loading spinner to indicate when data is being fetched or an operation is being performed.

Use Case

This component can be useful in situations where a healthcare staffing agency uses Salesforce to track the credentialing of their staff. As credentials such as licenses and certifications have expiration dates that need to be monitored and updated, this component provides a user-friendly interface to view and manage these expiring documents. Users can quickly see which documents are expiring, expand to view more details about them, and even update the documents directly from the component.

Important Notes

- The component makes use of Apex methods to perform CRUD operations. Ensure that the Apex classes have the necessary permissions and are exposed to the LWC.
- Error handling for Apex methods is critical. While placeholder code for error logging exists, it's important to implement robust error handling to enhance user experience.
- The file size limit for credential uploads is hardcoded to 3MB. Consider validating this on the server side as well to ensure security and reliability.
- Styling is imported from a custom CSS module (`nuCSS`) and applied dynamically based on the account's theme preferences. Ensure that these styles are properly set up and tested.
- Since this component involves file handling (uploading), ensure that Salesforce file size limits and security considerations are followed.

nuFilesRelatedList

Description

This Lightning Web Component (LWC) named `NuFilesRelatedList` is designed to display a list of related files for a specific Salesforce record. It utilizes a `lightning-card` to encapsulate a file upload component and a list of files. Each file in the list has options to download or preview it. This component leverages Salesforce's Lightning Design System (SLDS) for styling and is built to be used within the Salesforce Lightning Experience or Salesforce App.

Methods

1. **connectedCallback**: This lifecycle hook is executed when the component is inserted into the DOM. It calls the Apex method `getRelatedFilesByRecordId` to fetch related files based on the `recordId`. It transforms the fetched data into a format suitable for rendering in the component.

```
connectedCallback() {
    getRelatedFilesByRecordId({recordId : this.recordId})
    .then(data => {
        // Handling data
    })
    .catch(error => {
        // Error handling
    })
}
```

2. **previewHandler**: This method is triggered when the "Preview" button next to a file is clicked. It uses Salesforce's `NavigationMixin` to navigate to the file's preview URL in a new tab or window.

```
previewHandler(event){
    this[NavigationMixin.Navigate]({
        type:'standard__webPage',
        attributes:{
            url: event.target.dataset.url
        }
    })
}
```

Properties

1. **recordId** *(api)*: The ID of the Salesforce record for which related files are to be displayed. It is decorated with `@api`, making it publicly accessible for the component to be configured with a specific record ID.

```
@api recordId;
```

2. **filesList** *(track)*: An array that stores the list of files related to the `recordId`. This property is reactive, meaning any change to its value re-renders the component UI where `filesList` is used.

```
@track filesList = [];
```

3. **acceptedFormats**: A getter method that returns an array of accepted file formats for upload, currently configured to only accept PDF files.

```
get acceptedFormats() {
    return ['.pdf'];
}
```

Use Case

This component can be utilized in Salesforce record pages where there's a requirement to show related files for that record. It's particularly useful for cases like Job Application records where CVs or resumes (in PDF format) are associated with records and need facilities for upload, download, and preview.

Important Notes:

- **Apex Class Dependency**: This component depends on an Apex class named `nuController_RelatedLists` and specifically a method `getRelatedFilesByRecordId`. You must ensure that this Apex class is properly implemented and accessible by the component to fetch related files.
- **NavigationMixin and File Preview**: For the file preview feature, the `NavigationMixin` is utilized which requires the Salesforce `lightning/navigation` module. This allows files to be previewed in a new browser tab, depending on how the URLs are handled by the browser.
- **RecordId Requirement**: The component requires a valid `recordId` to fetch and display related files. Ensure that the component is placed in contexts where the `recordId` is

available, such as a record page.

- **Accepted Formats for Upload**: Currently, the component is restricted to accept only PDF files for upload. This can be adjusted by modifying the `acceptedFormats` getter method to include other file types as needed.

Incorporating this component into your Salesforce org provides an elegant solution for handling file uploads, and displaying related files with options to download or preview directly from a record's page, enhancing the overall user experience.

nuHiringAuditsKanban

Description

This Salesforce Lightning Web Component (LWC) is designed to manage and display hiring audits in a Kanban-style interface. It allows users to filter hiring audits based on various criteria such as specialty, candidate summary ID, candidate, location, and status using multi-select picklists. The component supports dragging and dropping hiring audits between stages and presents modals for certain actions like presenting a candidate or setting up an interview. It dynamically generates and updates the UI based on the available hiring audit data and user interactions.

Methods

- **connectedCallback**: Lifecycle hook that runs when the component is inserted into the DOM. It fetches initial data such as contact ID, account ID, and hiring audits.
- **handleListItemDrag**: Captures the ID of the hiring audit being dragged.
- **handleItemDrop**: Handles the drop action of a hiring audit item, potentially opening confirmation modals or updating the hiring audit's stage.
- **handleConfirmModal**: Handles the user's response in a confirmation modal, potentially proceeding with updating a hiring audit's stage.
- **handleFinishQuickAction**: Refreshes the hiring audits data after a quick action modal (presenting a candidate or setting up an interview) is completed.
- **updateHandler**: Updates the stage of a hiring audit using the `updateRecord` API.
- **handleSpecFilterChange, handleCSIDFilterChange, handleCandidateFilterChange, handleLocFilterChange, handleStatusFilterChange**: These methods handle changes in their respective filter picklists and trigger the filtering of hiring audits.
- **doFilter**: Applies all set filters to the hiring audits data, updating the view to reflect the filtered results.

Properties

- **recordId**: Stores the ID of the currently selected hiring audit.
- **pickVals**: Tracks the picklist values for hiring audit stages.
- **hiringAudits**: Stores hiring audit data for display and manipulation.
- **containerStyle, sectionHeadingStyle**: Stores inline CSS styles for dynamic styling based on account settings.
- **isHa, showPresentsModal, showInterviewModal, accessToNavigationMixin**: Boolean flags for various component states and features.
- **locOptions, csIdOptions, specOptions, candidateOptions, statOptions**: Arrays storing options for each of the filterable criteria's picklists.
- **changeStageModal, confirmChange, changingStage**: Properties related to handling stage change confirmation modals.
- **wiresLoaded**: A boolean to check if wire service calls are completed.

Use Case

This component is ideal for a recruitment or HR application within Salesforce, where managing and tracking the progress of hiring audits is crucial. It caters to the need for a visual representation of hiring audits across different stages of a recruitment process. The component allows users to filter the visibility of hiring audits based on diverse criteria making it easier to handle a large number of candidates. Additionally, it supports actions such as presenting candidates or setting up interviews directly from the Kanban board.

Important Notes

1. **Dynamic Styling**: The component dynamically adjusts its styles based on account data, ensuring consistency with the portal's theme.
2. **Drag and Drop Functionality**: The implementation of drag-and-drop for moving hiring audits between stages may require additional libraries or platform support not shown in the provided code.
3. **Promise Handling in `handleItemDrop`**: This method uses Promises to handle asynchronous operations involved in updating a hiring audit's stage, including user confirmation.
4. **Refresh After Action**: The component refreshes its data after actions like updating a stage or completing a quick action to ensure the UI is up-to-date.

5. **Wire Service for Picklist Values**: The use of wire service to fetch picklist values for stages emphasizes the dynamic nature of the component, ensuring it adapts to changes in the underlying Salesforce metadata.

```
// Sample code snippet to illustrate handling filter change
handleSpecFilterChange(event){
    let selectedValues = event.detail.value;
    let filterCriteria = [];
    selectedValues.forEach(selectedValue => {
        filterCriteria.push(selectedValue.label);
    });
    this.specFiter = filterCriteria;
    this.doFilter();
}
```

This documentation provides an overview of the NuHiringAuditsKanban component's capabilities and its application in a Salesforce Lightning environment.

nuInvoicesList

Description

This Salesforce Lightning Web Component (LWC) provides functionality for displaying a list of invoices in a datatable, with options to view detail, generate a PDF version, and dispute invoice entries. It leverages Apex controllers for fetching data, performing updates, and handling the dispute logic. The component dynamically updates styles based on Account record type and configurations. It supports navigation between the list view and detail view of invoices and includes modal dialogs for disputing entries and generating PDFs.

Methods

connectedCallback(): Initialization method that fetches contact, account details, and sets up initial UI settings like styles based on the account's configuration. It enables the display of either agency or bill amounts in the invoice table based on the account type.

handleInvoiceSelect(event): Handles the selection of an invoice from the list, fetching detailed invoice data including work logs.

setRecordCount(event): Updates the component with the number of records in the table view.

onPdfClick(event): Triggers the PDF generation for the selected invoice.

onDisputeClick(event): Opens the modal to initiate the dispute process for selected time entries.

doDispute(event): performs the dispute operation by calling an Apex method with selected entries and user-provided reasons.

saveInvoice(event): Saves any edits made to the invoice in detail view.

closeModal(event): Closes any open modal dialogs.

goBack(event): Navigates back to the list view from the invoice detail view, resetting the view state.

getInvoice(): Fetches detailed information about a selected invoice, including work logs and timesheet entry slots.

Properties

- **selectedInvoiceId**: The ID of the currently selected invoice.
- **selectedInvoice**: Detailed information about the selected invoice.
- **workLogs**: Collection of work logs related to the selected invoice.
- **invoices**: Collection of invoice summaries for display in the list view.
- **recordCount**: The number of records displayed in the list view.
- **agencyId**: The ID of the agency, if applicable.
- **accountId**: The ID of the account associated with the invoices.
- **status**: The filtering status for displayed invoices.
- **readOnly**: Boolean indicating if the detail view is read-only.
- **isReadOnly**: Boolean indicating if the invoices table is read-only.
- **columns**: Configuration for the columns in the datatable.
- **containerStyle**: Dynamic styling for the container based on account.
- **sectionHeadingStyle**: Dynamic styling for section headings.
- **title**: The title displayed in the header, dynamic based on the context.
- **showSpinner**: Indicates whether the loading spinner should be shown.
- **openDisputeModal**: Controls the visibility of the dispute modal.
- **showTable**: Controls the visibility of the invoices table.

Use Case

This component is suitable for use in an application where managing invoices, viewing invoice details, and performing operations like disputing charges or generating PDFs of invoices are required. It can be customized to handle different types of invoices and associated workflows based on the user's account type, such as distinguishing between agency invoices and other invoices.

Important Notes

- This component relies on several Apex controllers not included in the snippet, which must be implemented in the Salesforce org.
- The component uses dynamic styling, which requires setting up corresponding fields and styles in the Account object.
- Given the complex UI interactions, thorough testing is recommended to ensure behavior aligns with user expectations, particularly around error handling and state management across calls to Salesforce Apex.
- To use custom events like `onrecordcount` and `oninvoiceselect`, ensure the corresponding event handlers are correctly set up in the child components.

nuInvoicesTable

Description

This Lightning Web Component (LWC) named `NuInvoicesTable` is designed to list invoices with various filtering and action capabilities such as dispute handling, PDF generation, and downloading attachments. It includes functionality to filter invoices by dates, numbers, and several picklists including status, candidates, and work locations. The component provides a rich user interface for viewing detailed invoice and worklog information. Users can also dispute invoice entries and expenses with reasons and comments.

Methods

`runGetInvoices`

Fetches invoices based on agency and work location IDs. It processes fetched invoices for display and initializes options for picklists.

`doFilter`

Applies various filters to invoice data, including agency, company, date range, status, classification, specialty, and candidate filters.

`handleDateChange`

Handles changes in date filters and applies the date filter.

`doSelectInvoice`

Handles the selection of an invoice to toggle the display of its worklog details.

`poNumberHandler`, `WRpoNumberHandler`, `EXPpoNumberHandler`

Handles enabling editing for PO numbers on invoices, worklogs, and expenses, respectively.

`updateInvPO`, `updateWR`, `updateExp`

Updates PO numbers for invoices, worklogs, and expenses, respectively.

`doSelectWorklog`

Fetches and displays slots for a selected worklog.

`doOpenDisputeModal`, `doDispute`, `doOpenExpenseDisputeModal`, `doDisputeExpense`

Controls the opening of dispute modals and processing of disputes for either worklog entries or expenses.

`doSelectAllSlots`, `doSelectSlot`

Handles the selection of all slots or individual slots for disputing within a worklog.

`doSelectAllExpenses`, `doSelectExpense`

Handles the selection of all expenses or individual expenses for disputing.

`doCloseModal`, `doCloseExpenseModal`

Handles the closing of the dispute modals.

`onPdfClick`, `onDownloadAllClick`

Initiate PDF generation and downloading of invoice attachments.

`updateStatus`

Updates the status of an invoice based on user actions.

`doExpandExpenses`, `navigateToPlacement`

Controls expanding and collapsing expense details and navigating to placement records.

Properties

- `agencyId`, `accountId`, `containerStyle`, `status`, `isReadOnly`
  Configuration properties to adapt component behavior and display.
- `invoices`, `filteredInvoices`, `slotCols`, `classificationOptions`, `specialtyOptions`
  Hold the data processed for display or selection within the component.
- `showDisputeModal`, `showExpenseDisputeModal`, `showSpinner`
  Control visibility of modals and loading spinner.
- `selectedDisputeReason`, `slotsForDispute`, `expsForDispute`
  Track selected reasons for disputes and IDs of entries or expenses selected for disputing.

Use Case

The `NuInvoicesTable` component is intended for use within a Salesforce Lightning environment where an organization needs to manage, display, and action on invoices. It is particularly useful for agencies or companies that handle numerous invoices with the need for detailed tracking and action capabilities like disputing charges.

Important Notes

1. **Apex Classes Dependency**: This component depends on Apex classes (`nuController_InvoicesList`, `nuController_TimeSheetPDF`, `nuController_Files`, `nuService_Contact`) for backend data operations. Ensure these classes are present and accessible in your org.

2. **Custom Sub-Components**: The component uses custom sub-components (ex: `c-nu-searchable-picklist`, `c-nu-searchable-multi-picklist`, `c-nuUtils`). Ensure these components are deployed in your org for this component to function correctly.
3. **Community and Standard Navigation**: The component uses `NavigationMixin` for navigation. Ensure URLs and navigation targets are properly configured, especially when deploying in Salesforce Communities vs. the standard Salesforce environment.
4. **Data Security**: Ensure proper security settings are applied to the data accessed by this component to avoid unauthorized data exposure.
5. **Input Field Validations**: The component implements input validations especially for dispute reasons and comments. Ensure input criteria meet your organizational requirements.

This component is a comprehensive solution for managing invoices within a Salesforce Lightning environment, providing robust filtering, viewing, and action capabilities.

nuInvoiceTableRecordPage

This documentation outlines the structure and functionality of a Salesforce Lightning Web Component (LWC) designed for managing, displaying, and interacting with invoice records, including detailed work logs and expenses.

Description

The `NuInvoiceTableRecordPage` component displays a table of invoices with detailed information such as invoice number, client, agency, amount, status, etc. It allows users to expand each invoice to view associated work logs and expenses. The table supports actions like invoice selection for further details, PO Number editing, work log selection, slot selection, dispute entry, and downloading attachments. It utilizes Salesforce's Lightning Design System (SLDS) for styling and layout.

Methods

- **runGetInvoices**: Fetches invoices by ID and processes them for display.
- **doSelectInvoice**: Handles the action when an invoice is selected to toggle the visibility of associated work logs.
- **poNumberHandler**: Toggles the read-only state of the PO Number input field, allowing editing.
- **poNumberChange**: Updates the PO Number field in the database when changed.
- **doSelectWorklog**: Fetches and displays entries associated with a selected work log.

- **doOpenDisputeModal**: Opens the modal to dispute entries for a work log.
- **doOpenExpenseDisputeModal**: Opens the modal to dispute expenses.
- **doDispute**: Handles disputing the selected entries.
- **doDisputeExpense**: Handles disputing the selected expenses.
- **doSelectAllSlots**: Selects all time slots for disputing within a worklog.
- **doSelectAllExpenses**: Selects all expenses for disputing within an invoice.
- **doSelectSlot**: Handles the selection of an individual time slot for disputing.
- **doSelectExpense**: Handles the selection of an individual expense for disputing.
- **doCloseModal**: Closes the dispute modal dialog.
- **doCloseExpenseModal**: Closes the dispute expenses modal dialog.
- **onPdfClick**: Initiates PDF generation for the invoice.
- **onDetailsClick**: Navigates to a detailed view of the invoice.
- **doExpandExpenses**: Toggles the display of detailed expenses associated with an invoice.
- **onDownloadAllClick**: Initiates downloading of all documents associated with the invoice.
- **navigateToPlacement**: Navigates to the placement record associated with a work log or expense.

Properties

- **agencyId**: Stores the agency ID if applicable.
- **containerStyle**: Style for the container housing the component.
- **recordId**: The ID of the currently viewed record.
- **isReadOnly**: Indicates if the component should be in read-only mode.
- **showDisputeModal**: Controls the visibility of the dispute modal.
- **showExpenseDisputeModal**: Controls the visibility of the expense dispute modal.
- **filteredInvoices**: Contains the processed invoices for display in the UI.

Use Case

Ideal for use in environments where invoice management is critical, such as staffing agencies or professional services firms. The component provides a detailed, interactive view of invoicing data, allowing for comprehensive management actions such as disputing entries or expenses, editing PO numbers, and downloading associated documents.

Important Notes

- The component relies heavily on Apex controllers for backend data processing and actions like disputing entries or generating PDFs, ensure controllers are properly configured and have necessary permissions.
- Before using in production, thoroughly test all interactive features such as disputing entries or expenses, and downloading PDFs to ensure they meet your organization's requirements.
- Consider customizing dispute reasons and options based on your specific business processes.
- Ensure that record type names and field API names used in the component match those in your Salesforce org to avoid runtime errors.

nuJobOrderForm

Description

The `NuJobOrderForm` Lightning Web Component (LWC) facilitates the creation of job orders within Salesforce by providing a detailed and comprehensive form. The form encompasses various sections, including Company Information, Hospital Profile Information, Contact Details, Job Basic Information, Provider Requirements, Job Details, and Practice Setup Information. The information captured is extensive, ensuring all necessary details for job orders are gathered. The form supports conditional rendering of content based on user inputs and fetches dynamic picklist values from the Salesforce org, enhancing the flexibility and relevance of the data collection process.

Methods

- **handleExpandCollapse**: Toggles the display of the Hospital Profile Information section.
- **handleLocationListboxChange**: Handles changes to the selected location and updates corresponding location-dependent data.
- **handleBlur**: Captures user input on form field blur events to keep the job order object updated.
- **handleSubmitClick**: Validates form inputs and triggers the job order saving process.
- **handleClearClick**: Resets the form to its initial default state, clearing any user input.
- **handleLookupRecord**: Updates contacts within the job order based on selection from custom lookup fields.
- **loadCostCentres**: Fetches and populates the Cost Centers options based on the selected location.

- **saveJobOrder**: Processes the save action for the job order, capturing the data inputted by the user into the form and storing it accordingly.

Properties

- **isSaving**: A boolean flag indicating if the job order is currently being saved.
- **containerStyle, sectionHeadingStyle**: Styles applied to the form container and heading sections.
- **showRelatedCompanies**: A boolean flag indicating if related company options should be displayed.
- **jobOrder**: The main object holding all job order related data captured in the form.
- **showStartDate, showEndDate**: Flags to conditionally display start and end date inputs.
- **priorityOptions, credentialAcceptedOptions, etc.**: Arrays holding picklist values for various form inputs to ensure data consistency and integrity.

Use Case

This component is intended for use within the Salesforce ecosystem for organizations needing an extensive and customizable job order creation process. It could be particularly useful in staffing, recruitment, or any industry where detailed job descriptions and requirements need to be captured and processed.

Important Notes

- The form heavily relies on `@wire` services to fetch picklist values, ensuring that the form inputs are always aligned with the org's customization and metadata.
- It employs custom events like `handleLookupRecord` for populating lookups, providing a smoother user experience by integrating custom lookup components.
- Handlers for blur events on inputs (`handleBlur`) ensure the captured data is always current and correctly reflects user inputs.
- The component features a sophisticated method of clearing the form (`handleClearClick`) and resetting it to its original state, improving usability by allowing users to start afresh easily.
- CSS from an external source is imported (`@import 'c/nuCSS';`), indicating that the component might use a shared styling resource within the Salesforce org.
- Various sections like "Practice Setup Information" and "Job Details" include detailed fields and structured data capture, suggesting that the component is designed for complex data

entry requirements.
- The component showcases advanced LWC features, including conditional rendering, dynamic class application, and comprehensive use of Salesforce Lightning Design System (SLDS) classes for styling.

nuLinkedChatTable

Description

This component, named `NuLinkedChatTable`, is a Salesforce Lightning Web Component (LWC) designed to display a table of linked chats related to a specific record, and provides the functionality to create new chat records by selecting a contact. The component makes use of `lightning-datatable` for showcasing the data, a custom modal for creating new chats, and interacts with an Apex controller for data retrieval and creation of chat records.

Methods

- **handleNewChat:** Opens the modal to create a new chat.
- **handleComboChange:** Handles the selection change in the combobox within the modal, updates the selected contact value.
- **handleSave:** Handles the save action within the modal. It creates a new chat record based on the selected contact and navigates to the newly created chat's record page.
- **handleCancel:** Closes the modal without performing any action.
- **LinkedChatTable:** Invokes an Apex method to retrieve linked chat records for displaying in the datatable.
- **renderedCallback:** Lifecycle hook used for fetching and displaying the linked chat table data once the component is rendered.

Properties

- **recordId:** Record ID of the parent record. It's an `@api` property meaning it can be set from outside the component.
- **objectApiName:** Object API name, another `@api` property for dynamic binding.
- **initialised:** Indicator to prevent re-fetching data on every component re-render.
- **data:** Holds the data for the datatable.
- **columns:** Column definitions for the datatable.
- **selectedOption:** Tracks the selected contact option from the modal's combobox.

- **allFields, filteredFields, contactFields:** Used for managing the selectable options based on the record's available contact fields and processing them accordingly.
- **isModalOpen:** Controls the visibility of the modal for creating a new chat.

Use Case

The component is ideal for use cases where there is a need to display related chat records in a table format within a Salesforce record page and provide users with an ability to create new chat records by picking a contact. Such functionality can be useful in CRM scenarios where managing communication history and initiating new conversations directly from the CRM platform are required.

Important Notes

- The component requires Apex controller methods (`getLinkedChatTable`, `createChatRecord`, `getCommunityUserIdByContactId`) to be defined and accessible.
- The `@wire` decorators are used for reactive data fetching based on the component's reactive properties (`recordId`, `objectApiName`).
- Utilizes the `NavigationMixin` for navigating to the newly created chat record page upon successful creation.
- The modal operation is handled purely on the client side, with its visibility toggled by the `isModalOpen` property.
- Custom styling and layout are managed through inline styles and SLDS (Salesforce Lightning Design System) classes.

Example:

Embedding the component within a Salesforce record page:

```
<c-nu-linked-chat-table record-id={recordId} object-api-
name="YourObjectName"></c-nu-linked-chat-table>
```

This enables users on the specified record page to view linked chats and initiate new chats as needed.

nuListTimesheetEntryView

This documentation outlines the structure and functionality of a Lightning Web Component used for displaying a list of timesheet entries in Salesforce. The component is called `NgListTimesheetEntryView` and is designed to render timesheet entries in either a vertical layout or a horizontal layout, with additional configuration options such as showing shifts, unpaid breaks, and notes.

Description

`NgListTimesheetEntryView` is a Salesforce Lightning Web Component (LWC) used to present timesheet entries. It supports various configurations like layout modes (vertical or horizontal), showing shifts, and handling time entries that spread across multiple days. This component is flexible and can be used in different contexts by adjusting its properties via the API.

Methods

The component primarily relies on its lifecycle hooks (`connectedCallback`, `renderedCallback`) to execute code on component connection and post-render. It does not explicitly define custom methods but makes use of getters to compute derived state.

Properties

The component exposes the following API properties:

- `debugVar`: Used for debugging purposes.
- `debugTarget`: Specifies the debug target.
- `mode`: Mode of the layout, e.g., 'Agency'.
- `layout`: Specifies the layout orientation ('vertical').
- `entries`: Holds the timesheet entries.
- `showSlotInput`: Boolean to show/hide slot input.
- `updatedDate`: Represents the date when entries were last updated.
- `editMode`: Boolean to toggle edit mode.
- `showShift`, `showUnpaidBreak`, `showNote`: Booleans to control the visibility of shifts, unpaid breaks, and notes, respectively.
- `autoBreakValue`: Numerical value for auto break.
- `shiftOptions`, `otShifts`, `multidayShifts`, `hoursOnlyShifts`, `breakShifts`, `unpaidBreakOptions`: Arrays that hold options for different configurations related to shifts and breaks.

- `clockInClockOut`: Boolean to toggle between clock in/out and simple hours mode.
- `multiDay`: Boolean to indicate if the entry spans multiple days.

Use Case

This component can be utilized in applications or pages within Salesforce that require displaying or managing timesheet entries in a flexible manner. It supports various configurations to fit different business rules or UI requirements, such as displaying detailed shifts, handling multi-day entries, and allowing users to input or edit timesheet data directly.

Important Notes

- This component makes extensive use of conditional rendering (`template if:true/if:false`) in the HTML template to dynamically adjust the display based on the component's state.
- It employs Salesforce Lightning Design System (SLDS) for styling, ensuring a consistent look and feel with the Salesforce platform.
- The component reacts to changes in its properties to fetch and refresh entries as needed (`get cachedEntries`).
- Debugging utilities are embedded for development and troubleshooting (`debugVar`, `debugTarget`, and logging functions).

Example Usage

```
<c-ng-list-timesheet-entry-view
    mode="Agency"
    layout="horizontal"
    entries={timesheetEntries}
    show-slot-input
    updated-date={lastUpdatedDate}
    show-shift
    show-note
    clock-in-clock-out
    multi-day>
</c-ng-list-timesheet-entry-view>
```

This example demonstrates how to use `NgListTimesheetEntryView` to display timesheet entries in a horizontal layout, showing the shift and note information for entries that have clock-in/clock-out times and span multiple days.

nuListTimesheetSlotView

Description

The component `NuListTimesheetSlotView` is a Salesforce Lightning Web Component (LWC) designed to render timesheet slot entries in both a view and edit mode. It manages a collection of timesheet slot data, displaying each slot's details such as shift name, hours worked, clock-in/clock-out times, and enabling actions like edit, save, and delete on individual slots. The component dynamically adjusts its UI to accommodate various slot types, including overtime, multiday shifts, hours-only shifts, and allows for inputs when in edit mode. It makes use of Salesforce's Lightning Design System (SLDS) for styling to ensure consistent UI across the Salesforce ecosystem.

Methods

1. **handleSlotSelected**
   - Triggered on slot selection; emits a custom event with the slotId.
2. **handleAddSlotClick**
   - Handles the addition of a new slot; emits an `addslotclick` event.
3. **handleRemoveSlotClick**
   - Triggered on clicking the remove slot button, emitting a `removeslotclick` event with the slot details.
4. **handleEditSlotClick**
   - Invoked when an edit action is initiated on a slot; sets the component into edit mode for the specified slot.
5. **handleDateTimeChange**
   - Processes changes to date and time inputs during slot editing.
6. **handleShiftValueChange**
   - Handles changes in shift selection during slot editing.
7. **handleSaveSlotClick**
   - Validates and saves the edited slot information, emitting a `saveslotclick` event if the validation passes.
8. **handleBlur**
   - Processes input field blur events, applying changes to the slot data.
9. **mutateTimeSlots**
   - Internally modifies slot data based on editing actions or initial property inputs.

10. **computeHours**
    - Calculates and sets hours based on in/out times and any breaks.
11. **validateSlot**
    - Validates slot data before saving, ensuring required fields are populated.

Properties

1. **timeSlots** (`@api`)
    - An array of slot objects to be rendered.
2. **showSlotInput** (`@api`)
    - A boolean value determining if slot input fields should be shown (edit mode).
3. **isReadonly** (`@api`)
    - Specifies if the slots are to be rendered in a readonly mode.
4. **mode**, **showShift**, **showUnpaidBreak**, **showNote**
    - Various `@api` properties controlling the visibility of certain UI elements based on the usage context.
5. **shiftOptions**, **otShifts**, **multidayShifts**, **hoursOnlyShifts**, **breakShifts**
    - Lists (`@api`) defining the types and categories of shifts available for selection or display.
6. **localShiftOptions** (`@track`)
    - A tracked copy of `shiftOptions` for internal reactivity.

Use Case

The `NuListTimesheetSlotView` component is ideal for applications requiring a visual display and management of timesheet entries, such as employee work hours tracking, shift planning, or volunteer activity logging. Its flexible design supports a variety of slot types and user interactions, making it suitable for both viewing and editing timesheet data within Salesforce or custom Lightning applications.

Important Notes

- The component relies on a set of utilities from `c/nuUtils` for logging, event handling, and deep cloning, suggesting these are custom implementations specific to the larger application.
- It makes extensive use of Salesforce's Base Components (`lightning-*`) and the SLDS for consistent experience and styling.

- All dynamic behavior (e.g., slot addition, modification) is communicated through custom events (`fireCustomEvent`) to enable interaction with parent or surrounding components.
- Date and time calculations consider slot types, unpaid break selections, and auto-break calculations, with visual feedback and validation to ensure data integrity.

This component demonstrates a comprehensive approach to handling time tracking requirements, exemplifying complex LWC development practices, including event handling, dynamic templating, and API property usage.

nuListTimesheetView

Description

This Salesforce Lightning Web Component (LWC) named `NgListTimesheetView` is designed to display timesheets in either a vertical or horizontal layout, based on the property `layout` provided to it. It allows for dynamic interaction with timesheets, which can be selected by users. The component supports numerous configurable properties to customize the appearance and behavior of the timesheets, such as whether to show entries, headers, submit buttons, and more. It employs CSS for styling, uses imported utility methods for functionality like deep copying and logging, and is designed to potentially interact with a messaging system for application-wide event handling, though those parts are commented out in the given code.

Methods

- **handleListClick**: Handles clicks on list items (timesheets), firing off a custom event named `timesheetselectionchange` with the selected timesheet's ID.
- **mutateTimesheets**: Adjusts the class of each timesheet based on its selected state to update its appearance in the UI.
- **connectedCallback**: Lifecycle hook that runs when the component is inserted into the DOM. Initially sets up debugging information.
- **renderedCallback**: Lifecycle hook that runs after the component renders. Used here for performance logging.
- **disconnectedCallback**: Lifecycle hook that runs when the component is removed from the DOM. Used here for cleanup, if necessary.

Properties

- **debugVar**, **debugTarget**: Properties for enabling debugging.

- **layout**: Determines the layout orientation of the timesheet display (`vertical` or not).
- **timesheets**: An array of timesheet objects to display.
- **updatedDate**, **showEntries**, **showHeader**, **showSubmitButtons**, **showFiles**: Configurable properties that control the visibility of various UI elements related to the timesheets.
- **otShifts**, **multidayShifts**, **hoursOnlyShifts**, **breakShifts**: Arrays or properties that describe specific types of shifts within the timesheets.
- **showSlotInput**, **positionLabel**, **weekStartLabel**, **weekEndLabel**, **totalHoursLabel**, **guaranteedHoursLabel**, **showPosition**, **showWeekStart**, **showWeekEnd**, **showTotalHours**, **showGuaranteedHours**: More configurable properties to tailor the timesheet display and info further.
- **clockInClockOut**, **multiDay**, **showUnpaidBreak**, **autoBreakValue**, **showNote**, **showShift**: Boolean flags and values affecting the display and behavior of each timesheet entry.

Use Case

A Salesforce application that requires displaying a list of timesheets, where each timesheet includes detailed entries and attributes that can be shown or hidden based on user preferences or application context. This component would be particularly useful in HR or project management applications within Salesforce where tracking time and activities is essential.

Important Notes

- **Layout Flexibility**: The component can dynamically switch between vertical and horizontal layouts, making it adaptable to different UI designs or screen sizes.
- **Event-Driven Interactions**: Though much of the message service-related code is commented out, the structure suggests an intended use within a larger system where components communicate through events, such as selection changes.
- **Debugging and Performance Logging**: The component contains an extensive setup for debugging and logging, demonstrating a pattern for tracking component behavior and performance in a development or testing environment.
- **Styling Approach**: The component relies on both standard Salesforce Lightning Design System (SLDS) classes and custom styling defined within the component, demonstrating a combination of global and local styling practices.

nuManageDashboardPreferences

Description

The `NuManageDashboardPreferences` component is designed to manage dashboard preferences within a Salesforce Lightning environment. It features a form for editing records related to dashboard preferences, offering fields for basic information, contact association, and template selection. This component utilizes a combination of Lightning Web Components (LWC) and Salesforce Apex to interact with Salesforce records and metadata dynamically. Functionality includes creating new dashboard preference records, updating existing ones, and applying templates to dashboard preferences.

Methods

handleContactResult

Triggered by a wire service to fetch and process the contact's related information like account record type, user type, and account id based on the `recordId` provided.

connectedCallback

Lifecycle hook that runs when the component is inserted into the DOM. It initializes the component by fetching object names and setting up the fieldsBySection based on the result.

getTemplatesAndContactPrefs

Fetches dashboard preference templates and contact-specific dashboard preferences, preparing the UI for interaction.

handleFormSubmit

Handles form submission, either creating a new dashboard preference template or updating an existing record based on the user's input.

handleFormSuccess

Executes actions upon successful form submission, including stopping the spinner and displaying a success toast message.

handleChange

Handles changes to the combobox for template selection, updating UI state based on the selected template.

handleCancel

Resets the form to its initial state or the last saved state.

handleClearForm

Clears the form to enable creating a new record by resetting relevant component properties.

Properties

- **recordId**: The Id of the currently selected record.
- **contactId, accountId, userType**: Trackable properties related to the contact's information.
- **fieldsBySection**: Structure to hold field information organized by sections for rendering.
- **templatesPickvalues**: Array holding picklist options for the templates combobox.
- **dpId, contactDpId, dpName, contactDpName**: Properties for holding dashboard preference record details.
- **selectedTemplateValue, defaultPicklistvalue**: Manage selected values in templates combobox.
- **showSpinner**: Boolean to control the display of the loading spinner.
- **displayContactValue, displayTemplatePicklist**: Booleans to manage UI elements display based on conditions.
- **isNewTemplate**: Indicates when a new template is being created.
- **dpObject**: References the `Dashboard_Preferences__c` object.

Use Case

This component can be used in Salesforce orgs where dashboard preferences need to be personalized per contact or user. It allows for the creation and application of templates to quickly apply predefined dashboard configurations. Its core functionalities enable users to:

- Create new dashboard preference records.
- Update existing dashboard preference records.
- Select and apply dashboard preference templates to specific contacts.

Ideal for organizations looking to provide a customized dashboard view for their users or for admins needing to manage dashboard preferences efficiently.

Important Notes

- The component is dependent on the `recordId` to perform operations. Without a valid `recordId`, the component cannot function as expected.

- The component expects specific Salesforce object and field schemas, namely `Dashboard_Preferences__c` and its fields. Customization or changes in these schemas may require corresponding adjustments in the component.
- Commented code (e.g., the button for creating new templates directly from the UI) suggests potential features or adjustments that are currently inactive but can be enabled with slight modifications.
- The error handling is primarily logged to the console. In production, consider implementing user-friendly error messages.

nuNewDashboard

Description

The `NuNewDashboard` component is a Salesforce Lightning Web Component (LWC) designed to display various sections like "Hot Jobs", "Activity & Alerts", "Insights", "Pending Actions", and "Quick Tasks". It dynamically loads content based on the availability of the data and user permissions. The data is retrieved via Apex controllers and static resources. It also supports navigation to different pages or records within the Salesforce environment based on user interactions.

Properties

- `contactId` & `accountId`: Track the ID of the contact and account, respectively.
- `pendingActionsTiles` & `quickTasksTiles`: Arrays of objects representing the tiles to be displayed in the "Pending Actions" and "Quick Tasks" sections.
- `hotjobs`: An array to contain information about "Hot Jobs".
- `notifications`: An array to contain notification details for the "Activity & Alerts" section.
- `notificationDropdownStyle`: Style information for the display of notifications.
- `keyInsightsId`: A unique identifier for key insights data.
- `isDataComplete`: A boolean indicating if the data loading is complete.
- These flags control the visibility of the respective sections:
  - `enableHotJobs`
  - `enableInsights`
  - `enableActivityAlerts`
  - `enableInsightsUtility`
- `showSpinner`: Controls the visibility of a spinner during data loading.
- Static resources for the "Insights" section:

- `spendWorkedYTD`
- `clinicianScheduled`
- `spendScheduled`

Methods

- `connectedCallback()`: Initializes the component by fetching data required for rendering.
- `navigateToPage(event)`: Handles navigation to a specific page as per user interaction.
- `navigateToHotjob(event)`: Navigates the user to a detailed view of a selected hot job.
- `navigateToReferenceUrl(event)`: Handles navigation based on the provided record ID or reference URL.

Use Case

This component is intended to serve as a dashboard within a Salesforce Community or within the Salesforce Lightning Experience. It provides quick access to various sections of interest such as "Hot Jobs", "Activity & Alerts", "Insights", "Pending Actions", and "Quick Tasks". The component is designed to enhance user engagement by dynamically displaying information relevant to the user and providing quick navigation routes to pertinent areas within Salesforce.

Important Notes

1. **Dynamic Data Loading**: Data for this component is dynamically loaded using Apex controllers (`fetchContactAndRelatedData`, `fetchHotJobOrders`) and static resources. The presence of data dictates the visibility and content of various sections.
2. **Navigation**: This component makes extensive use of the Salesforce NavigationMixin to handle navigation, demonstrating how components can facilitate user navigation within Salesforce.
3. **Styling**: It leverages SLDS (Salesforce Lightning Design System) for its styling needs, ensuring a consistent look and feel with the Salesforce ecosystem.
4. **Responsive Design**: Media queries are included to ensure the component is responsive and adapts to different screen sizes.
5. **Visibility Logic**: The visibility of various sections is controlled through boolean flags that are set based on the data received from the server.
6. **Event Handling**: Event handlers (`navigateToPage`, `navigateToHotjob`, `navigateToReferenceUrl`) are used to manage user interactions, such as clicked links for navigation.

Example of a method for navigating to a specific page:

```
navigateToPage(event) {
    const clickedPageName = event.target.dataset.pagename;
    if (clickedPageName) {
        let destination = {
            type: 'comm__namedPage',
            attributes: {
                pageName: clickedPageName
            }
        };
        this[NavigationMixin.Navigate](destination, false);
        event.stopPropagation();
    }
}
```

nuObjectChecklist

Description

This Salesforce Lightning component, named `NuObjectChecklist`, is designed to display a setup checklist for a particular record. The checklist items are fetched from a Salesforce Apex controller and displayed in a series of `lightning-card` components. Each item shows its completion status indicated by an icon, its name, and optionally a "Go To" button linked to a URL, along with a list of values beneath it if present.

Methods

`wiredChecklist`

This method is triggered by the `@wire` service, invoking the `returnChecklist` Apex method with the current `recordId`. It processes the returned data, displaying it within the component. In case of any error, it logs the error to the console.

`handleGoToClick`

Handles click events on the "Go To" buttons. It uses the `NavigationMixin.GenerateUrl` to generate a standard URL based on the `data-id` attribute of the clicked button, which it then opens in a new window.

Properties

- **@api recordId**: This is an externally accessible property used to pass the record ID into the component, determining which checklist items to fetch and display.
- **checklistList**: An array that stores the checklist items fetched from the Apex controller for display.

Use Case

This component is used to provide users with a visual checklist related to a specific Salesforce record. Each checklist item indicates whether the task is complete, provides a name for the task, and optionally includes a direct link to another page and a list of related values. This is useful in guiding users through setup processes or ensuring that all necessary tasks related to a record are completed.

Important Notes

1. **CSS Styling**: Custom CSS styles are applied to list items and headers. Ensure these styles meet your org's design guidelines.
2. **Deep Copy Utility**: The component uses a `deepCopy` method from a custom utility module `nuUtils`. This method must be provided within your org to ensure the component functions correctly.
3. **Error Logging**: The component logs errors to the console but does not provide user-facing error messages. Consider enhancing error handling for a better user experience.
4. **Data Fetching**: The checklist data is fetched using an Apex controller named `nuChecklistController` and its method `returnChecklist`. Ensure this controller and method exist and are properly configured in your org.
5. **Standard Web Page Navigation**: When opening URLs, the component assumes that they can be navigated to as standard web pages. Ensure the URLs provided in checklist items are accessible to users and consider modifying navigation handling for different types of URLs if needed.

Example Implementation

```
import { api, LightningElement, wire } from 'lwc';
import returnChecklist from
'@salesforce/apex/nuChecklistController.returnChecklist';

export default class ExampleComponent extends LightningElement {
    @api recordId;
```

```
    // Define additional methods and properties as needed
}
```

This component serves as a dynamic, user-friendly way to display task completion and link users to necessary resources or additional actions within the Salesforce UI.

nuOrderJobDetailsAndRelatedList

Description

The component `NuOrderJobDetailsAndRelatedList` is a comprehensive Salesforce Lightning Web Component (LWC) designed for rendering detailed views associated with various Salesforce objects like Job Orders, Hiring Audits, and Placements. It supports viewing and editing details, quick actions for record processing, related lists for associated files, work logs, credit memos, chat functionalities, and more. It dynamically adjusts its behavior and presentation based on the object type and context, handling both client and agency portals while offering capabilities like PDF generation and ZIP file downloading for job orders.

Properties

- `hiringAuditId`: For storing Hiring Audit ID.
- `recordId`: Universal property to store the current record ID being viewed or manipulated.
- `recordType`: Stores the type of the record.
- `isorderApprovalActionPanel`: Indicates if the approval action panel should be shown.

Methods

- `handleVfPageMessage(event)`: Handles messages from the embedded Visualforce page, particularly for adjusting iframe heights.
- `runInit()`: Initial method called to fetch object names, setup styles, and prepare the component based on the context (client or agency).
- `handleFieldChange(event)`: Handles any change made to input fields on the form, adjusting UI dynamically based on field-specific logic.
- `handleEditMode()`: Enables or disables edit mode for the fields in the component.
- `handleFormSubmit(event)`: Placeholder for custom logic on form submission.
- `handleQuickAction(event)`: Triggers when a quick action is initiated by the user.
- `finishQuickAction(event)`: Handles the completion of a quick action and potentially navigates to a different record page based on the action's result.

- `loadMoreData(event)`: Method for loading more data dynamically into the datatable present in the component.
- `handleAssignmentFilesRowAction(event)`: Handles row actions in the Assignment Files datatable.

Use Case

This LWC is well-suited for complex Salesforce objects related to job management or recruitment processes, where a detailed and interactive view of the object is required. This would include:

- Viewing detailed information of Job Orders, Hiring Audits, and Placements.
- Performing object-specific actions directly from the detail view (e.g., close job order, submit candidate).
- Managing related files, chat conversations, and other related records from the same interface.
- Editing records inline and handling dynamic UI changes based on the record type and field values.

Important Notes

- The component uses a mix of LWC framework capabilities along with Apex backend support to fetch records, handle actions, and manage related lists.
- It utilizes `NavigationMixin` for handling navigation to different record pages or external URLs (like generated PDFs).
- The CSS part involves advanced styling to ensure the component maintains a consistent look and feel with Salesforce Design System standards, as well as responsive design considerations.
- Integration with Visualforce for specific functionalities like PDF generation is handled through message passing techniques.
- The component demonstrates a complex use of LWC lifecycle hooks, Salesforce schema import for referencing object fields, and wire service for reactive data fetching.
- Extensive use of `@api`, `@track`, and `@wire` decorators indicates a highly dynamic and data-driven component.

nuPageHeader

Description

The Lightning Web Component (LWC) named `NuPageHeader` is designed to serve as a custom page header within Salesforce Lightning Experience. This component integrates with Salesforce's design system (SLDS) to display breadcrumb navigation and a page title. It also embodies navigation functionality, allowing users to return to the homepage with a click.

Methods

- **navigateToHomePage**:

  This method enables the component to navigate back to the Salesforce homepage when invoked. The navigation is achieved by using the `NavigationMixin.Navigate` method provided by the `lightning/navigation` module.

  ```
  navigateToHomePage() {
      this[NavigationMixin.Navigate]({
          type: 'standard__namedPage',
          attributes: {
              pageName: 'home'
          },
      });
  }
  ```

  The `navigateToHomePage` function is tied to the `Home` breadcrumb link within the component's template, and it's triggered via an `onclick` event.

Properties

- **@api heading**:

  This public property (`@api`) allows the parent component to pass a string value that serves as the heading or title of the webpage. This title is displayed in a `<h1>` tag with appropriate styling as per Salesforce Lightning Design System (SLDS).

  The usage in the template is demonstrated as follows:

  ```
  <h1 class="slds-text-heading--medium slds-truncate" title="Available
  Orders">{heading}</h1>
  ```

Use Case

The `NuPageHeader` component can be employed across various custom pages within a Salesforce Lightning application to provide a consistent user experience. It's particularly useful in scenarios where navigation to the home page and displaying page titles are requisite elements of the user interface.

For instance, it could be used on a custom orders page, where the header displays "Available Orders" and offers a breadcrumb navigation back to the home page for easy user orientation and navigation.

Important Notes

- The component uses the Salesforce Lightning Design System (SLDS) for styling, ensuring that it adheres to the visual language of the Lightning Experience.
- Navigation functionality is leveraged through the `NavigationMixin` from `lightning/navigation`, which is a Salesforce standard for enabling navigation in Lightning Web Components.
- The breadcrumb navigation is designed to be accessible, with an assistive text "You are here:" that's aimed at screen readers, improving the component's usability for users with disabilities.
- The commented-out `<p>` tag within the template suggests a provision for displaying additional information (like record counts) which can be uncommented and utilized if needed.
- Remember to replace `javascript:void(0)` in the anchor tag's `href` attribute with a more appropriate, navigational link if extending the breadcrumb functionality.

nuPageLayout

Description

The provided code defines a Salesforce Lightning Web Component (LWC) named `NuPageLayout`. This component is crafted to serve as a customizable and complex layout for a Salesforce Community Page or any Salesforce Lightning Experience App page. It incorporates a navigation top bar with dynamic links to different functionalities such as Jobs, Candidates, Assignments, and Analytics, along with submenus with actionable items. Additionally, it supports a chat feature, notification system, and a unique ability to handle unsaved changes, enriching user interaction and functionality within the Salesforce ecosystem.

Methods

1. **navigateToHome:** Navigates the user to the Home page, checking for unsaved changes before leaving.

```
navigateToHome()
```

2. **navigateToPage:** Navigates the user to the specific page clicked on.

```
navigateToPage(event)
```

3. **handleMyATSClick:** Navigates the user to an external ATS (Applicant Tracking System) URL.

```
handleMyATSClick()
```

4. **handleGotoUrl:** Handles navigation to a specific URL based on the target ID and updates the read status of a notification.

```
handleGotoUrl(event)
```

5. **handleMobileNav:** Toggles mobile navigation view.

```
handleMobileNav()
```

6. **doShowInbox:** Toggles the inbox display, showing or hiding it.

```
doShowInbox(event)
```

7. **handleInboxItemClicked:** Handles the action when an inbox item is clicked.

```
handleInboxItemClicked(event)
```

8. **handleSaveChangesClick:** Manages the saving of data when the user intends to save changes.

```
handleSaveChangesClick(event)
```

9. **doShowChat:** Toggles the chat display, showing or hiding it.

```
doShowChat(event)
```

10. **showSelectedChat:** Shows the selected chat from the list of user chats.

```
showSelectedChat(event)
```

11. **messageReceived:** Handles the incoming messages from `c-cometdlwc`, acting upon new chat message events.

```
messageReceived(event)
```

12. **navigateToReportDashboard:** Opens a new browser tab with the Salesforce standard record page for the selected report or dashboard.

```
navigateToReportDashboard(event)
```

Properties

- **standalone**: Determines if the page layout is in standalone mode.
- **showUnsavedChanges**: Indicates whether there are unsaved changes.
- **hasUnsavedChanges**: Tracks if there are unsaved changes to alert the user before navigating away.
- **hideFooter**: Controls the visibility of the footer.
- **jobsMenuTiles, candidateTrackingMenuTiles, placementsMenuTiles, analyticsMenuTiles**: Arrays holding data for respective menu tiles.
- **inbox**: Array holding inbox notification data.
- **chatId**: Holds the identifier for the current chat.
- **reports, dashboards**: Arrays containing report and dashboard data.
- **userChats**: An array of user's chat data fetched and used for displaying the chat area.

Use Case

This component can be utilized in any Salesforce Lightning Experience application or Community Cloud portal that requires a navigable, feature-rich layout. Each of the navigation

items (Jobs, Candidates, Assignments, Analytics) leads to different functional areas of the application, potentially corresponding to different Salesforce objects or external pages. Additionally, it supports real-time chat functionalities and notifications, making it suitable for a dynamic, interactive user portal.

Important Notes

1. **Adaptive Design:** The component contains responsive design properties enabling it to adapt between desktop and mobile views smoothly.
2. **Requires Apex Controllers:** The JavaScript controller references multiple Apex controller methods, such as `fetchContactById` and `getUsersChats`, which are necessary for fetching backend data.
3. **External ATS Link:** Offers an integration point with an external Applicant Tracking System via `handleMyATSClick`, enhancing the component's flexibility for recruitment-based applications.
4. **Unsaved Changes Handling:** The component is sensitive to unsaved changes, prompting the user before navigating away from the current page, ensuring data integrity and user confirmation before leaving the page.

nuPageLayoutFooter

Description

This Lightning Web Component (LWC) named `NuPageLayout` is designed to provide a dynamic user interface for navigating through various pages within a Salesforce Lightning Experience or Community. The component includes a navigation bar, dropdown menus, mobile responsiveness, and the functionality to navigate to different pages based on the user's selection. It also handles the display of notifications and updates their read status upon user interaction.

Methods

- **navigateToHome**:
  Navigates the user to the home page.

  ```
  navigateToHome()
  ```

- **navigateToPage**:
  Navigates the user to a specific page based on the clicked element's `data-navinfo` attribute.

  ```
  navigateToPage(e)
  ```

- **handleGotoUrl**:
  Handles redirection to a specified URL when a notification is clicked and updates the read status of the notification.

  ```
  handleGotoUrl(event)
  ```

- **handleMobileNav**:
  Toggles the visibility of the mobile navigation menu.

  ```
  handleMobileNav()
  ```

Properties

- **testContactId** `@api`:
  External ID that can be set by the parent component.
- **jobOrderTiles**, **applicantTrackingtiles**, **providersTiles**, **agenciesTiles** `@api`:
  Data properties that are expected to be populated through an API call to render specific tiles or links dynamically.
- **isDataComplete** `@track`:
  A boolean flag indicating the completeness of data loading. It controls the rendering of the main content.
- **imageResource**, **headingStyle**, **linkContainerStyle**, **linkStyle**, **notificationDropdownStyle**, **submenuStyle**, **appClicked**:
  Internal component properties used for styling and managing state.

Use Case

This component is designed for use in Salesforce Lightning Experience or Communities where navigation between different pages or modules is required. It can be particularly useful in

custom-built Salesforce apps that require a dynamic, responsive navigation bar with support for notifications and custom navigation links.

Important Notes

1. **External Styling**:
   Make sure that the SLDS (Salesforce Lightning Design System) is included in your project, as this component may rely on SLDS classes and design tokens for styling.
2. **NavigationMixin**:
   This component uses the `NavigationMixin` for navigating to Salesforce pages. Ensure that the correct page references are provided in the `navigationInformation` property.
3. **Responsive Design**:
   The component is designed to be responsive and includes CSS for mobile views. Test thoroughly on different devices and screen sizes to ensure compatibility.
4. **Data Fetching**:
   Data is fetched asynchronously in the `connectedCallback` lifecycle hook. Ensure error handling is robust and consider loading states for a better user experience.
5. **Security**:
   Always consider security best practices when developing Lightning Web Components, especially when handling navigation and URL redirection.
6. **Community Usage**:
   When deploying this component within a Salesforce Community, make sure to correctly configure the `sitePath` and ensure that named pages (`comm__namedPage`) are correctly defined in the Community Builder.

nuPageLayoutHeader

Description

The Salesforce Lightning component `NuPageLayout` is a complex, navigation-oriented component designed to provide modern, responsive navigation menus for Salesforce Lightning Experience or Communities. It dynamically displays various navigation tiles such as Job Orders, Applicant Tracking, Providers, and Agencies, each with a potential submenu for notifications. The component utilizes various API properties to fetch and display relevant data and incorporates styling for both desktop and mobile views. It implements programmatic navigation using the `NavigationMixin` for navigating to Salesforce pages or external URLs based on user interactions.

Methods

1. `navigateToHome`

Navigates the user to the Salesforce Home page.

```
navigateToHome() {
    this[NavigationMixin.Navigate]({
        type: 'standard__namedPage',
        attributes: {
            name: 'Home'
        },
    });
}
```

2. `navigateToPage`

Handles navigation to different pages based on the target's `data-navinfo` attribute. Utilizes the `navigationInformation` object for mapping navigation data.

```
navigateToPage(e) {
    // Implementation of navigation based on 'data-navinfo' attribute
}
```

3. `handleGotoUrl`

Directs the user to a URL when a notification is clicked after marking it as read on the server-side.

```
handleGotoUrl(event) {
    // Handles redirection to specified URL after processing the
notification
}
```

4. `handleMobileNav`

Toggles the mobile navigation view on and off.

```
handleMobileNav() {
    this.appClicked = !this.appClicked;
}
```

Properties

1. `testContactId` (API)

An API property to receive the contact ID for testing purposes.

2. `jobOrderTiles`, `applicantTrackingtiles`, `providersTiles`, `agenciesTiles` (API)

API properties that hold arrays of objects representing different tiles for navigation such as Job Orders, Applicant Tracking, etc.

3. `isDataComplete` (Track)

A tracked property to control the rendering of the component's template when data fetching is complete.

4. `headingStyle`, `linkContainerStyle`, `linkStyle`, `notificationDropdownStyle`, `submenuStyle`

These properties contain styles applied to various parts of the component.

5. `brandingStyle`, `linkContainerClass`, `closeClass`

Computed properties used for dynamic styling and class assignment based on component state.

Use Case

This component is designed for use within Salesforce Lightning Experience or Communities where a dynamic, responsive navigation bar is required. It could be particularly useful for a custom home page or a navigation side panel where users need access to different Salesforce entities or external resources, with additional context provided by notifications.

Important Notes

1. **Dynamic Data Fetching**: Data for tiles like Job Orders and Applicant Tracking is intended to be dynamically fetched and passed via API properties. Ensure the data structure matches what is expected in the component.
2. **Navigation Data Setup**: The component includes a `navigationInformation` object to map user clicks to Salesforce navigation events. This object must be tailored to include valid Salesforce named pages, object pages, filters, etc.
3. **Mobile Responsiveness**: Mobile navigation is handled through a combination of CSS and JavaScript. Ensure the CSS media queries and JavaScript methods (`handleMobileNav`) align with your layout requirements.
4. **Action Handling**: The component includes handlers for both navigating to Salesforce pages and external URLs (`handleGotoUrl`). Ensure that actions such as `updateReadOn` correctly interface with backend logic for marking notifications as read.
5. **Style Customization**: Component styling is defined within the CSS file. Modify these styles to match the desired look and feel, paying attention to both desktop and mobile versions.

nuProviderPresentationForm

Description

This Salesforce Lightning Web Component (LWC) named `NuProviderPresentationForm` is designed to create and present candidate information in an organized manner. It incorporates various Salesforce and custom components to display and manage data related to agencies, companies, providers, certifications, education, experiences, and more. The form allows for the display of data fetched from Salesforce, manipulation of data, and submission of a comprehensive presentation.

Methods

- **handleSubmitClick**: Validates the form data to ensure all required fields are filled and then calls `savePresentation` to save the data.
- **handleClearClick**: Clears the form and resets it to its initial state.
- **handleInputChange**: Handles changes to input fields by updating the component's reactive properties accordingly.
- **checkLength**: Checks the length of text areas and warns the user if they are approaching the maximum character limit.
- **savePresentation**: Invokes the `savePresentation` Apex method to save the presentation data to the Salesforce org.

- **fireToast**: Fires a toast event to display notifications to the user, indicating success or error messages.

Properties

- **isSaving**: Tracks whether the data is currently being saved.
- **presentation**: Holds the presentation data bound to the form.
- **hasActiveLicenseOptions**: Stores options for the 'Active License' picklist.
- **boardStatusOptions**: Stores options for the 'Board Status' picklist.
- **certificationsOptions**: Stores options for the 'Certifications' picklist.
- **credentialTypeOptions**: Stores options for the 'Credential Type' picklist.
- **specialtyOptions**: Stores options for the 'Specialty' picklist.
- **travelRequirementOptions**: Stores options for the 'Travel Requirements' picklist.
- **isStartDateOrEstimateOptions**: Stores options for the 'Is Start Date Known or Estimate' picklist.
- **startEstimateOptions**: Stores options for the 'Start Estimate' picklist.

Use Case

This component is intended for use in situations where a detailed presentation of a candidate's information is necessary, such as in staffing or recruitment processes within healthcare agencies. The presentation form includes diverse sections such as provider information, board status, certifications, licensure, education & training, experience, availability, travel, and other additional information.

Important Notes

- The component uses several wire services to fetch picklist values and other initial data necessary for the form.
- Local storage is used to persist form data temporarily, ensuring data isn't lost if the page is refreshed.
- Custom events like `"passproviderid"` are dispatched to communicate with other components in the application.
- The component makes liberal use of the `lightning-*` components to create a standardized UI that follows Salesforce Lightning Design System (SLDS) guidelines.
- CSS from a custom static resource (`nuCSS`) and specific styles defined within the component are used for styling.

Example Usage

While the component code provides the entire structure of the `NuProviderPresentationForm` component, here is a brief outline on how it might be included within another component or application:

```
<c-nu-provider-presentation-form standalone={true} hiring-audit-id=
{hiringAuditId}></c-nu-provider-presentation-form>
```

This inclusion creates an instance of the provider presentation form, potentially using properties like `standalone` and `hiringAuditId` to control its behavior and data loading, respectively.

nuProviderPresentationRelatedList

Description

This Salesforce Lightning Web Component (LWC), named `NuProviderPresentationRelatedList`, is designed to display and manipulate records related to a specific object within Salesforce. The component allows users to dynamically view records organized by sections and fields as configured in Salesforce Layouts. It also includes functionality to trigger an Apex method to generate a PDF and subsequently download it as a ZIP file, integrating custom error and loading handling.

Methods

wiredRecordEdit()

- **Description**: This method is wired to the `getRecordUi` method from `lightning/uiRecordApi`. It organizes the record data into sections and fields based on the layout configuration in Salesforce, adjusting the visibility of fields based on the component's `startEstimatePicklist` property.
- **Parameters**: None directly, but uses `recordId` to fetch data.
- **Usage**: Automatically invoked whenever the `recordId` is updated due to the `@wire` decorator.

handleClick(event)

- **Description**: Handles the click event on the "Presentation PDF" button. It triggers the `generatePDF` Apex method, handles errors, displays a loading indicator during processing, and invokes the `downloadZIP` Apex method upon success. It uses NavigationMixin to navigate to the generated file URL.
- **Parameters**:
  - `event`: The click event object.
- **Usage**: Bound to the click event of the `lightning-button` in the component's template.

Properties

- **recordId** (`@api`): The ID of the Salesforce record the component operates on.
- **startEstimatePicklist** (`@api`): An API property that influences which fields are displayed based on its value.
- **objectName**: Stores the API name of the object the record belongs to, as determined by `wiredRecordEdit`.
- **isDisabled**: A boolean that controls the disabled state of input fields. Initially set to `true`.
- **isLoading**: A boolean to indicate whether an asynchronous operation (like generating a PDF) is in progress. It controls the visibility of a loading spinner.
- **fieldsBySection** (`@track`): A tracked property that stores the configuration of sections and fields derived from the Salesforce Layout, used to render the record edit form dynamically.

Use Case

This component can be used in Salesforce Lightning Experience or Communities to provide a detailed view and edit functionality for records, segmented according to Salesforce Layout configurations. Additionally, it gives users the ability to download related files as a PDF packaged in a ZIP file. An example use case could be for managing provider agreements or contracts where users need to review details, make updates, and generate a presentation PDF directly from the record detail page.

Important Notes

- **Dynamic Field Display**: The visibility of specific fields based on the `startEstimatePicklist` property shows an advanced level of dynamic UI manipulation based on the component's state.
- **Error Handling**: The component includes basic structures for error handling and user feedback through toast notifications but might require expansion based on specific business

logic and error types encountered in the Apex methods `generatePDF` and `downloadZIP`.

- **Apex Integration**: The component relies on Apex methods for PDF generation and file downloading, highlighting the need for server-side logic to support its functionalities.
- **Loading State Management**: The implementation of `isLoading` to manage the presentation of a loading spinner during asynchronous operations enhances the user experience by providing visual feedback during data processing.

nuProviderStartingNDays

Description

The `NuProviderStartingNDays` component is a Salesforce Lightning Web Component designed to display a table of candidates starting assignments within a certain number of days (either within 7 or 10 days, based on the `cmpVariant` attribute). It dynamically fetches this data using Apex controller methods to retrieve the relevant candidate and assignment information based on the current user's contact and account information. The component also supports themed styling based on account properties or default styles.

Properties

- `cmpVariant`: Used to specify the variant of the component. Determines whether the component is fetching data for candidates starting in 7 days or 10 days. It is an API property so it can be set externally.
- `title`: The title of the component, automatically set based on the `cmpVariant`.
- `recordCount`: The count of records (placements) retrieved and displayed in the datatable.
- `accountId`, `agencyId`, `contactId`: Track properties that store IDs necessary for data retrieval.
- `placements`: Stores the data to be displayed in the datatable.
- `sectionHeadingStyle`, `containerStyle`: Dynamic styling for the component elements based on account data or default styles.
- `columns`: Configurations for the columns in the `lightning-datatable`.

Methods

- **connectedCallback:** Lifecycle hook that runs when the component is inserted into the DOM. It sets the component's title based on the `cmpVariant`.

- **renderedCallback:** Lifecycle hook that runs after every render of the component. It is used to add an additional column to the datatable based on whether the account is recognized as an agency.
- **populateContactId, populateAccountData:** Wired methods to fetch contact ID and account data respectively. Account data is used to fetch placements and determine styles.
- **handleResult:** Processes the fetched placements and prepares them for display in the datatable.

Use Case

This component is ideal for use in a Salesforce Community or any Salesforce App where there's a need to display recently starting candidates within a specified timeframe (7 or 10 days) related to the logged-in user's account. It's particularly useful for agencies or organizations that need to track placements and candidate assignments efficiently.

Important Notes

- Custom Apex methods (`fetchContactId`, `fetchAccountByContactId`, `fetchPlacementsWhereProviderStartingSevenDays`, `fetchPlacementsWhereProviderStartingTenDays`) are required for the component to function correctly. Their implementations should be ensured in the Apex class `nuService_Contact`.
- The component makes use of dynamic styling based on account properties. The `Use_Portal_Theme__c`, `Tertiary_Color__c`, `Secondary_Color__c`, `Primary_Text_Color__c`, and `RecordType.DeveloperName` fields on the account are particularly significant.
- The component's appearance and columns may change dynamically based on whether the account is recognized as an "Agency" and based on the component variant chosen (7 days or 10 days).
- The `lightning-datatable` is used to display the data with a custom column configuration defined in the component's JavaScript.
- This component leans on the utility class `nuUtils` for default page styles, ensuring that even in the absence of specific account-based styling, the component remains visually consistent with the application.

nuQuickActionAcceptOffer

Description

The component `NuQuickActionAcceptOffer` is designed to handle the quick acceptance of an offer related to a hiring audit record in Salesforce through the Lightning Web Components (LWC) framework. It primarily interacts with Salesforce's data layer to update specific fields of a hiring audit record (`Hiring_Audit__c`) and uses an Apex method `createPlacement` for further processing. The component displays a spinner while processing and emits a custom event `finishquickaction` upon completion or throws an error if something goes wrong.

Methods

renderedCallback()

The `renderedCallback` lifecycle hook is used in this component to execute logic after the component has been inserted into the document but re-evaluates only when `isRecordUpdated` is `false` to prevent repeated executions. It performs the following actions:

- Marks `isRecordUpdated` as `true` to indicate processing has started and will not rerun the enclosed logic on subsequent renders unless the component is destroyed and recreated.
- Invokes the Apex method `createPlacement` with the current `hiringAuditId`.
- On promise resolution, it hides the spinner, and dispatches the `finishquickaction` event with the result from `createPlacement`.
- On promise rejection, logs an error to the console.

```
renderedCallback() {
    if (!this.isRecordUpdated) {
        this.isRecordUpdated = true;

        createPlacement({hiringAuditId: this.hiringAuditId}).then(result
=> {
            this.showSpinner = false;
            this.dispatchEvent(new CustomEvent(
                'finishquickaction',
                {detail: result}
            ));
        })
        .catch(error => {
            console.error('Error Updating HA to Placement',
JSON.stringify(error));
        });
```

```
    }
}
```

Properties

1. **hiringAuditId (@api)**: External (public) property to accept the `Hiring_Audit__c` record ID to be updated or processed.
2. **showSpinner**: A private boolean property to control the visibility of the spinner component based on asynchronous operations.
3. **isRecordUpdated**: A private boolean property used to ensure the logic within `renderedCallback` executes only once.

Use Case

This component is designed to be used as part of a Salesforce Lightning experience, embedded within pages where users interact with `Hiring_Audit__c` records. It's particularly useful in scenarios where a hiring audit process reaches a milestone that requires the status to be quickly and programmatically updated to reflect the acceptance of an offer, automatically handling the transition without manual user input beyond initiating the action. It visually indicates processing with a spinner and confirms the operation completion through a custom event.

Important Notes

- The component leverages a mix of standard Lightning Web Components (`lightning-spinner`) and Salesforce-specific modules (from `lightning/uiRecordApi` and `@salesforce/schema`).
- It embraces best practices for data manipulation by using `@salesforce/apex` to call server-side logic in a secure and efficient way.
- The renderedCallback method's logic is encapsulated in a condition to prevent unnecessary or repeated Apex calls, optimizing performance.
- Custom event `finishquickaction` should have a listener implemented in the component's consuming context to handle post-action workflows or navigation properly.
- Error handling is done through console logging, which might need enhancement in a production environment for user feedback or retry mechanisms.
- The commented-out section of `renderedCallback` suggests there was an initial approach to directly update the record using `updateRecord` from `uiRecordApi`, which has been

replaced with a more complex Apex interaction, presumably for additional server-side logic encapsulated in `createPlacement`.
- The component necessitates the `hiringAuditId` to be set for proper operation, making it dependent on a parent component or context to provide this data, thus not being entirely standalone.

nuQuickActionApproveNameClear

Description

This Salesforce Lightning web component (LWC) is designed to update a specific record's fields in Salesforce, specifically for a `Hiring_Audit__c` object. The component updates the `Current_Stage__c` and `Status__c` fields of a `Hiring_Audit__c` record to "Name Clear" and "Name Clear Approved", respectively. It is intended to be used as part of a quick action to streamline the process of changing the audit status. Upon successful update of the record, it emits a custom event named `finishquickaction` to signal the completion of the action.

Methods

- **renderedCallback()**: This lifecycle hook is used to perform actions after the component has been inserted into the DOM. In this scenario, it checks if the record has been updated (`isRecordUpdated`); if not, it proceeds to update the record's relevant fields (`Current_Stage__c` and `Status__c`) using the `updateRecord` method from `lightning/uiRecordApi`.

Properties

- **hiringAuditId (@api)**: This public property allows the component to receive the `Id` of the `Hiring_Audit__c` record which needs to be updated.
- **showSpinner**: A boolean used to control the visibility of a spinner (loading indicator), which could be used in the template to show the user that an action is in progress.
- **isRecordUpdated**: A boolean to prevent multiple updates by keeping track if the record has been updated during the component's lifecycle.

Use Case

This component can be utilized as a custom action in Salesforce for hiring auditors to approve and clear names quickly. When the action is triggered (e.g., through a button in the Salesforce

UI), it automatically updates the `Current_Stage__c` and `Status__c` fields of a specified `Hiring_Audit__c` record without further user input, improving efficiency and user experience in hiring audit processes.

Important Notes

- **Data Security**: Ensure that users who have access to this action have the necessary permissions to update the `Hiring_Audit__c` record.
- **Error Handling**: While there's a basic setup for logging errors in the `catch` block, it might be beneficial to implement more sophisticated error handling or user feedback mechanisms.
- **Event**: The component dispatches a `finishquickaction` event upon successful completion. Ensure that the enclosing application or component listens for this event to react accordingly (e.g., refreshing relevant data or showing a success message).
- **Lightning Data Service (LDS)**: This component leverages the `updateRecord` method from LDS for committing the record changes, which provides benefits such as reduced Apex code and automatic UI refresh if the updated record data is displayed elsewhere in the UI.

nuQuickActionCancelByAgency

Description

This Lightning Web Component (LWC) is designed to provide a modal popup that facilitates the cancellation of a hiring audit. When users invoke this action, they are presented with a form within a modal that captures the reason for the cancellation. It utilizes Salesforce's `lightning-record-edit-form` to interact with the Hiring Audit object's records directly. This form is initially populated with some default values but allows the user to input a cancellation reason. The component also updates related records upon successful form submission.

Methods

1. **handleResult:** A wired method to fetch data for the initial state of the component based on the provided `placementId`. It sets the `hiringAuditId` for further operations.
2. **doSubmit:** Invoked when the user clicks the submit button. It submits the form and shows a spinner if the cancellation reason is provided.
3. **successHandler:** Handles the successful submission of the form. It updates the related placement record to mark it as 'Cancelled' and then dispatches a `finishquickaction` event to close the modal.
4. **errorHandler:** Logs errors to the console if the form submission fails.

5. **closeQuickAction:** Hides the spinner and dispatches the `finishquickaction` event to close the modal. Triggered when the cancel button is clicked.

Properties

- **hiringAuditId**: stores the Id of the Hiring Audit record. It's marked with `@api`, making it publicly accessible to the parent component.
- **placementId**: stores the Id of the Placement record. This is also exposed to the parent component via `@api`.
- **hiringAuditObject, statusField, stageField, reasonField**: represent schema details of the Hiring Audit object and its fields.
- **showSpinner**: controls the visibility of a spinner during async operations.
- **statusValue, stageValue, reason**: default values and user-entered reason for the cancellation form.

Use Case

This component is ideal for situations where a user needs to cancel a hiring audit, requiring a reason for such action. The component is meant to be used as a Quick Action, integrated into record pages where it can be invoked to easily cancel hiring audits and automatically update related records with minimal user input.

Important Notes

1. **Schema Dependency**: This component depends on specific schema elements (`Hiring_Audit__c` and its fields) being present in the Salesforce org. Ensure these schema elements exist and are accessible to the component.
2. **Security**: Ensure that users who have access to invoke this action have the necessary permissions to read from and write to the related records.
3. **Lightning Data Service**: Leveraging `lightning/uiRecordApi` for record fetching and updates facilitates ease of development with built-in caching and reduced Apex reliance. However, ensure that the org's data limits are considered when scaling usage.
4. **Event Handling**: The component dispatches a `finishquickaction` event upon successful updation or cancellation. The parent component or application needs to handle this event appropriately to ensure a seamless user experience.
5. **UI**: The component uses SLDS classes for styling to adhere to Salesforce's Lightning Design System standards, ensuring a consistent user interface across the Salesforce platform.

nuQuickActionCancelByClient

Description

This Lightning Web Component (LWC) named `NuQuickActionCancelByClient` provides functionality to cancel a hiring audit by displaying a modal dialog where a user can submit reasons for the cancellation. It leverages the Salesforce Lightning Design System (SLDS) for styling a modal layout that includes a form for capturing cancellation details. The component uses Salesforce's `lightning-record-edit-form` to interact with the Hiring Audit record, displaying pre-filled statuses and requiring a reason for the cancellation. Additionally, it supports dynamic data fetching with `@wire` service to retrieve related record data and utilizes the Salesforce UI API for record updates.

Methods

- **doSubmit()**: Submits the form data captured in the `lightning-record-edit-form`. It also triggers a spinner to indicate the processing state until a response is received.
- **successHandler(event)**: Handles the successful submission of the form. It updates the status of a related Placement record and dispatches a custom event to signal the completion of the quick action.
- **errorHandler(event)**: Logs errors encountered during form submission to the console.
- **closeQuickAction()**: Dismisses the modal and stops the spinner. It also dispatches a custom event signaling the user has closed the action without completing it.

Properties

- **hiringAuditId**: (Decorated with `@api`) Represents the record ID of the Hiring Audit that is being cancelled.
- **placementId**: (Decorated with `@api`) The ID of the Placement related to the Hiring Audit.
- **hiringAuditObject, statusField, stageField, reasonField**: These properties hold the schema descriptors for the Hiring Audit object and its fields to be used in the component.
- **showSpinner**: Boolean flag used to control the visibility of the spinner indicating processing state.
- **statusValue, stageValue**: Pre-defined values to be displayed for the status and stage fields, indicating that the Hiring Audit has been canceled by the client.

Use Case

This component can be utilized in a Salesforce org where tracking and management of hiring audits are required, and there is a need for a quick and seamless process to cancel these audits. It's particularly useful in scenarios where an audit needs to be marked as cancelled and a reason for the cancellation needs to be captured, all while updating related records based on this action.

Important Notes

- The component is designed to be used within contexts where a `placementId` is available, from which it retrieves and sets the `hiringAuditId`.
- The `.css` code related to the styling of this component is not provided but is assumed to utilize standard SLDS classes for modals.
- The component's functionality relies on the Salesforce schema names being correctly provided and existing in the org where it's deployed.
- Error handling in this component is minimal and primarily logs the errors to the console. Depending on the use case, more sophisticated error handling and user feedback mechanisms should be considered.
- Custom events like `finishquickaction` are dispatched to signal completion states but require corresponding event handlers in the parent component or application for further action.

nuQuickActionCloneJobOrder

Description

This Salesforce Lightning Web Component (LWC) named `NuQuickActionCloneJobOrder` provides functionality to clone a Job Order record in Salesforce. It displays a modal window with a dynamic form, allowing users to submit new Job Order records with pre-populated fields based on the original record's data. The component also offers customization options for handling specific fields differently, such as conditionally displaying Start Date and End Date fields.

Methods

- **closeQuickAction:** Closes the quick action modal without submitting any data.
- **connectedCallback:** Lifecycle hook that runs when the component is inserted into the DOM. It fetches the Job Order's details and prepares the fields for cloning.

- **renderedCallback:** Lifecycle hook that runs after every render of the component. It customizes the options for the 'Is_End_Date_Known_or_Estimate__c' field.
- **doSubmit:** Programmatically clicks the hidden submit button to initiate form submission.
- **handleFormSubmit:** Handles the form submission event, creates a new Job Order record, and navigates to the new record's page upon success.
- **handleFieldChange:** Handles changes to any of the field inputs in the form, specifically adjusting the display of date fields based on certain conditions.

Properties

- **recordId:** The ID of the Job Order record being cloned (@api).
- **showEndDate:** Determines whether to show the end date based on a condition (@api).
- **showStartDate:** Determines whether to show the start date based on a condition (@api).
- **fieldsBySection:** Tracks the dynamically constructed fields and their organization for rendering in the form (@track).

Use Case

To use this component, you should have a Job Order (`Job_Order__c`) record which you intend to clone. This component is designed to be initiated as a quick action from a Job Order's detail page. It dynamically fetches the Job Order's fields, excluding certain fields defined in the `FIELDS_TO_EXCLUDE` constant, and presents them in an accordion-style input form within a modal. The user can submit this form to create a new Job Order record that clones data from the original, with options to manage the inclusion of Start and End Date fields.

Important Notes

- The component makes use of the `lightning-record-edit-form` to leverage Salesforce's built-in record editing functionality, making it efficient in handling field data types and validations without extra code.
- Dynamic handling of the Start Date and End Date visibility is provided based on the component's `showStartDate` and `showEndDate` properties.
- Custom logic is included to filter out certain fields from cloning (defined in `FIELDS_TO_EXCLUDE`) and to mark specific fields as required (using `requiredLabels`).
- Upon submission, a new Job Order record is created with the provided data, excluding specific system fields, and the user is navigated to the newly created record's detail page.

Example

Using this component as a quick action on the Job Order object page allows users to quickly clone Job Orders with much of the data pre-populated, streamlining the process of creating similar Job Orders.

nuQuickActionCloseCandidate

Description

The component, `NuQuickActionCloseCandidate`, is designed to facilitate the closing of Job Orders in Salesforce by providing a modal dialog that allows users to input a reason and comments for the closing action. The dialog appears within the Salesforce Lightning Experience and utilizes Salesforce Design System (SLDS) for its UI elements. The modal allows users to either save their input and close the job order or cancel the action. Internally, it uses the Salesforce Lightning Data Service (LDS) and Apex to update the Job Order record and potentially handle record locking scenarios gracefully.

Methods

- **closePopupSuccess**: This method gathers the inputs from the reason and comments fields, validates them to ensure they are not empty, constructs a record input object with the necessary fields including status, closed on date, closed by user, and the text for reason and comments, then attempts to update the record. It handles errors and specific scenarios like locked records gracefully.
- **handleConfirmClose**: An async method that triggers a confirmation dialog when an attempt to close a locked Job Order is made. It proceeds to forcefully close the Job Order through Apex processing if confirmed by the user.
- **closeJobOrderProcess**: Invokes an Apex method to handle the closing of a Job Order, especially under conditions where the record might be locked or requires special processing.
- **closePopup**: Dispatches an event indicating that the modal should be closed, canceling the action.

Properties

- **@api joborderId**: The Id of the Job Order record being closed. It's a public property that should be set when the component is utilized.

- **objectApiName**: Internal property used to store the API name of the Job Order object. It is hard-coded to `'Job_Order__c'`.
- **reasonField**: Stores the API name of the field used for storing the close reason.
- **commentsField**: Stores the API name of the field used for storing closing comments.
- **showSpinner**: A boolean flag used internally to control the display of a loading spinner.
- **isRecordUpdated**: A boolean flag indicating whether the record has been successfully updated or not.
- **userId**: The Id of the current user derived from "@salesforce/user/Id".

Use Case

This component is designed to be used within a Salesforce Lightning application where it can be triggered from a list view or record page of Job Orders (`Job_Order__c`). Its primary function is to allow users to close Job Orders by providing a reason and additional comments before saving. It's particularly useful for streamlining processes that require formal closure of job orders, enforcing data integrity by ensuring necessary information is captured during the closure process, and handling records that may have approval processes attached.

Important Notes

- The component makes extensive use of Salesforce Lightning Web Component (LWC) features, Salesforce Lightning Design System (SLDS) for styling, and leverages Lightning Data Service (LDS) for record creation and updates which ensures built-in performance benefits and data consistency.
- Error handling and record locking logic is implemented to ensure a smooth user experience even when Salesforce record-level security and record locking come into play.
- The component relies on a custom Apex method `closeJobOrderProcess` which should be present and accessible in the Salesforce org for handling specific business logic tied to the closing process.
- Toast notifications are hinted at in the code but are commented out. Implementations should consider utilizing a toast utility function like `fireToast` for user notifications.
- The component dispatches a `finishquickaction` event upon user actions that should be handled by the parent component to control the visibility of the modal and possibly refresh the view or redirect the user after a successful operation.

nuQuickActionCreatePlacement

Description

The Lightning Web Component (LWC) named `NuQuickActionCreatePlacement` is designed to perform a backend action using Apex when rendered on the page and communicate the action's result to its parent component. The component uses a spinner to indicate that an asynchronous operation is in progress and utilizes an Apex method named `createPlacement` to perform a task related to a "Hiring Audit" object. After the backend operation is done, it communicates back to the encompassing component or application through a custom event named `finishquickaction`.

Methods

- **renderedCallback()**: This lifecycle hook gets called after the component has been rendered to the DOM. It checks if the record has not been updated already to prevent redundant operations. It then makes an Apex call to `createPlacement`, passing the `hiringAuditId` as a parameter. On successful execution, it stops displaying the spinner and dispatches the `finishquickaction` event with the result of the operation. If there's an error, it logs the error message in the console.

Properties

- **hiringAuditId** `[api]`: This public property is used to pass the ID of the Hiring Audit record to the component. It must be set by the parent component and is mandatory for the Apex call.
- **showSpinner**: Used to control the visibility of the `lightning-spinner` component based on the asynchronous operation's status. It is `true` by default, indicating that the spinner is visible initially.
- **isRecordUpdated**: Internal property to ensure that the Apex call is made only once after the component is rendered.

Use Case

This component is ideal for scenarios where a quick action is required to perform a backend operation related to a Hiring Audit record and to notify other parts of the Lightning application that the operation is complete. It can be particularly useful in custom Salesforce Lightning applications where workflows need to automate and process Hiring Audit data seamlessly and inform the user or other components of the completion without manual intervention.

For example, it can be leveraged in a recruitment application within Salesforce where, upon viewing details of a candidate's Hiring Audit, an automatic processing step is required to create a Placement record, and the UI needs to reflect this change or notify other components to update accordingly.

Important Notes

1. **@salesforce/apex import**: Ensure that the Apex method `createPlacement` is correctly annotated with `@AuraEnabled` in your Apex class `nuController_QuickActions` and is accessible by the current user's profile or permission set.
2. **Custom Event Dispatching**: The custom event named `finishquickaction` should be handled in the parent component to take action based on the result sent by `NuQuickActionCreatePlacement`.
3. **Error Handling**: This component logs errors to the console but does not display them to the user. Depending on the application's requirement, it’s advisable to implement user-friendly error handling to indicate when the operation fails.
4. **Spinner Visibility**: The spinner's visibility is managed through an internal property. Ensure that any operation that could impact the spinner (e.g., additional Apex calls or UI updates) respects this property to maintain a coherent user experience.
5. **Lifecycle Hooks**: The use of `renderedCallback` is pivotal here but ensure it does not lead to infinite loops or redundant operations by properly managing flags like `isRecordUpdated`.

nuQuickActionDeclineNameClear

Description

This Lightning Web Component (LWC) named `NuQuickActionDeclineNameClear` is designed to provide a user interface for declining a name clearing action in Salesforce. It utilizes a modal dialog that includes a combobox for status selection, options for inputting reasons for decline (including interaction with another Salesforce object through record editing form or text input), and functionality to specify an expiration date. The component communicates with Apex controllers to fetch account details and update specific records based on user input.

Methods

nameClearDeclineMethod

```
async nameClearDeclineMethod(agencyId, expirationDate, textForDecline,
option)
```

Performs the action of marking a record as declined by calling an Apex method with parameters such as `agencyId`, `expirationDate`, `textForDecline`, and a boolean `option`.

handleChange

```
handleChange(event)
```

Handles changes on the `Status` combobox, updating the UI based on the selection.

closePopupSuccess

```
async closePopupSuccess()
```

This method is triggered when the "Save" button is clicked. It validates inputs and performs the decline operation, updating related records and displaying toast notifications based on operation success or failure.

closePopup

```
closePopup()
```

Closes the popup without saving changes by dispatching a custom event `finishquickaction`.

getAgencyName

```
async getAgencyName(inputField)
```

Fetches the name of an agency based on an account ID by calling an Apex method, aimed at assisting with autocompleting or validating input data.

Properties

- `hiringAuditId`: Record ID for the Hiring Audit entry.

- `objectApiName`: API name of the Salesforce object associated with the record edit form.
- `secondPart`, `thirdPart`, `datePart`: Boolean properties controlling the visibility of certain parts of the UI.
- `textForDecline`: Text string aggregating reasons for the name clear decline.
- `accName`: Stores the name of the account after fetching it from Salesforce.
- `value`: Stores the value of a selected option from the combobox.
- `fieldToUpdate`: API name of the field to be updated based on user input.

Use Case

When a user needs to decline a name clear request within Salesforce, they can use this component to specify the reasons, select or input an agency, and set an expiration date for the decline. This information will be recorded in the associated Salesforce objects, updating the relevant record fields with user-provided details.

Important Notes

- It's essential to load and understand the dependencies from Salesforce's schema module accurately to ensure field API names are correctly referenced.
- The component relies on external Apex methods (`fetchAccountById` and `nameClearDecline`) and a utility method (`fireToast`) which are not provided within this code. These methods should exist and be properly designed in your Salesforce org to handle backend logic and user notifications, respectively.
- The usage of `@salesforce/schema/` in importing field references is a Salesforce standard way to safely reference schema elements across environments.
- Proper error handling and input validation are implemented to enhance user experience and data integrity.
- This component demonstrates an advanced use case of handling dynamic UI pieces based on user interaction, showcasing the handling of multiple conditions to dynamically render parts of the form and perform complex business logic based on user input.

nuQuickActionDeclineOffer

Description

The `NuQuickActionDeclineOffer` Lightning Web Component (LWC) provides a modal pop-up interface for users to input a reason for declining an offer. The component dynamically

adjusts which field to update based on whether the offer is declined by an agency or a client and supports the inclusion of a comment when the specified reason is "Other (require comments)."

Methods

- `connectedCallback`: Initializes the component by determining the correct field to update based on the context (agency or client). This method is invoked after the component is inserted into the DOM.
- `handleInputChange`: Captures and logs changes to the reject reason text area. This method handles input change events for the rejection reason comment.
- `handleFieldChange`: Monitors changes to the decline reason selection field and determines if the comment section should be shown based on the selected reason.
- `closePopupSuccess`: Validates the input field values and updates the related Hiring Audit record with the decline reason and optional comment. It also posts a toast message based on the operation's outcome before closing the modal.
- `closePopup`: Closes the modal without saving changes by dispatching a custom `finishquickaction` event.

Properties

- `@api hiringAuditId`: The ID of the Hiring Audit record being updated.
- `@api isDeclineOfferByAgency`: A boolean indicating if the decline operation is initiated by an agency.
- `@api isDeclineOfferByClient`: A boolean indicating if the decline operation is initiated by a client.
- `@track showSpinner`: Used to control the display of a spinner, indicating a process is taking place.
- `@track showCommentRequire`: Indicates whether the comment input should be displayed based on the selected decline reason.

Use Case

This component is intended to be used in scenarios where a user needs to document the reason for declining a hiring offer in a Salesforce community. It adapts to different contexts (agency or client-initiated decline) and enforces input validation, ensuring that a reason and, where necessary, an accompanying comment are provided before submission.

Important Notes

- Users should ensure that the `@salesforce/schema` fields utilized (`Status__c`, `Presentation_Decline_Reason__c`, and `Reason_for_Decline__c`) exist and their API names are correctly referenced.
- It leverages the `updateRecord` method from `lightning/uiRecordApi` for seamless integration with Salesforce's data layer.
- The component employs custom toast notifications (`fireToast` utility) for user feedback, which suggests that there's a dependency on another LWC (`c/nuUtils`) for displaying these messages. Ensure this utility is included in your Salesforce org.
- The modal utilizes SLDS classes and attributes to align with Salesforce Lightning Design System standards for consistent styling and accessibility.

Example Use

```
<c-nu-quick-action-decline-offer
  hiring-audit-id={someHiringAuditId}
  is-decline-offer-by-agency={trueOrFalse}
  is-decline-offer-by-client={!trueOrFalse}>
</c-nu-quick-action-decline-offer>
```

Note: Replace `{someHiringAuditId}` with the actual ID of the Hiring Audit record and `{trueOrFalse}` with appropriate boolean values based on the context.

nuQuickActionFillRateTable

Description

This Salesforce Lightning Web Component (LWC) named `NuQuickActionFillRateTable` is designed to provide functionalities for handling rates related to hiring audits, job orders, and assignments. It enables users to view and edit rates, manage draft values for rate amendments, validate data, and navigate through different tabs presenting rate information in various contexts.

Methods

- **connectedCallback**: Lifecycle hook which runs when the component is inserted into the document. It modifies the columns displayed based on the context (e.g., hiring audit vs.

assignment).

- **fetchRates**, **fetchRatesFromJobOrder**, **fetchRatesFromAssignment**: Wired methods to fetch rates data from Apex controllers based on different contexts, such as hiring audits, job orders, and assignments.
- **updateDataValues**: Updates data values with the changes captured in the editable rate table.
- **updateDraftValues**: Updates the draft values as the user edits rows in the rate table, performs inline validations like checking if values deviate from recommended ranges.
- **handleCellChange**: Handler function that captures changes in the cell values and updates draft values.
- **handleSubmit**: Handles the form submission, which involves persisting the edited rates. Displays success or error messages as needed.
- **handleCancel**: Handles cancellation of edits by reverting to the last saved data.
- **refresh**: Refreshes the rate data displayed by re-fetching from the server.
- **handleNextButtonClick**: Handles click event on the "Next" button, validating the current content before moving forward in a multi-step process.
- **performValidation**: Performs validation on the rates data, ensuring data integrity before allowing further operations.

Properties

- **columns**, **columnsNotEditable**, **columnsNotEditableJobOrder**, **columnsNotEditableHiringAudit**: Arrays defining the structure and behavior of columns in the data tables used for displaying rates information. These columns are adjusted based on the context, like whether the data is editable or not.
- **data**, **draftValues**, **errors**: Tracks the rates data, any draft (unsaved) changes, and any validation errors that occur.
- **presentedRateTab**, **rateTabJob**, **rateTabAssignment**: Boolean variables to control the visibility of different tabs or sections based on the context like presented rates, job order rates, etc.
- **showSpinner**: Boolean property used to control the display of a spinner UI component, indicating ongoing data operations.

Use Case

This component is suited for scenarios where it is necessary to manage and edit rates associated with different entities like hiring audits, job orders, and assignments within

Salesforce. It offers functionalities such as inline editing, validation against custom business logic, and dynamic adjustments of visible UI elements based on the context.

Important Notes

- The component relies on Apex controllers for data retrieval and operations (`nuService_Contact`, `nuService_RateTable`), implying that it is part of a larger application with server-side logic.
- Data validation plays a significant role in the component's operation, highlighting the importance of accurate and meaningful data entry.
- The component is designed with reusability in mind, making adjustments based on the context like the type of rates being managed. This dynamic behavior necessitates careful testing across different scenarios to ensure consistency and reliability.

This documentation gives an overview of a complex Lightning Web Component designed for managing and interacting with rates data in a Salesforce application, highlighting its structure, functionality, and contextual flexibility.

nuQuickActionMakeOffer

Description

This Salesforce Lightning Web Component (LWC), named `NuQuickActionMakeOffer`, is designed to update a specific record in Salesforce with new values for its `Current_Stage__c` and `Status__c` fields. The component visualizes a spinner to indicate ongoing processing and utilizes the Salesforce Lightning Data Service (`lightning/uiRecordApi`) to perform the record update operation without requiring Apex code. The action takes place in the `renderedCallback` lifecycle hook of the component, ensuring that the operation is attempted only once per component lifecycle due to the `isRecordUpdated` flag.

Methods

- **renderedCallback()**: This lifecycle method is overridden to perform actions after the component has been inserted into the DOM but before rendering control back to the browser. It updates the record identified by `hiringAuditId` once per component lifecycle.

Properties

- **hiringAuditId (@api)**: This is a public property that allows the component to receive the ID of the `Hiring_Audit__c` record to be updated. Being an `@api` property, `hiringAuditId` can be set externally, e.g., when using the component within another component or when the component is being instantiated dynamically.
- **showSpinner**: A private property used to control the visibility of the `lightning-spinner` component. It is initialized as `true` and is set to `false` once the record update operation is either completed successfully or fails.
- **isRecordUpdated**: Another private property to ensure that the record update operation occurs only once. It prevents the method within `renderedCallback` from being called multiple times during the component's lifecycle.

Use Case

The primary use case of `NuQuickActionMakeOffer` is in scenarios where a Salesforce record (specifically of the `Hiring_Audit__c` object) needs to have its `Current_Stage__c` and `Status__c` fields updated to "Confirm Assignment" and "Client Accepted", respectively. This can be particularly useful in hiring or recruitment applications within Salesforce where moving a candidate to a new stage in the pipeline necessitates a quick and straightforward way to update the record's status. The component can be placed on any Lightning page, utility bar, or used in a Lightning application where such an operation is needed.

Important Notes

- The component hinges on the `lightning/uiRecordApi` module's `updateRecord` function for record updates, which requires the object fields' API names. These are imported using the `@salesforce/schema` namespace.
- It incorporates a basic loading spinner via the `lightning-spinner` component, aiding in enhancing user experience by providing visual feedback during the update process.
- The component assumes that the `Hiring_Audit__c` object and its fields (`Current_Stage__c` and `Status__c`) are present in the Salesforce org where this component is deployed. Missing objects or fields can lead to deployment errors or runtime exceptions.
- Errors during the update process are logged to the console, but no user-friendly error handling is implemented. Depending on the use case, developers might want to extend this component to display error messages directly on the UI.
- This component triggers a custom event (`finishquickaction`) after a successful update operation. If used within a larger application, other components can listen to this event and

respond accordingly, though this event handling aspect is not fleshed out in the provided code.

```
this.dispatchEvent(new CustomEvent('finishquickaction'));
```

- The use of the `renderedCallback` lifecycle hook to perform data manipulation is efficient in this specific use case but might not be suitable for all scenarios, especially where multiple re-renders might occur, leading to unintended operations or API calls.

nuQuickActionPresentCandidate

Description

The `NuQuickActionPresentCandidate` component for Salesforce Lightning is designed to facilitate the candidate presentation process within a hiring audit workflow. This component allows users to present candidates by showcasing their CVs, filling out a presentation form, and completing rate tables. It integrates with Salesforce's Apex controllers for backend operations such as checking existing CVs and attaching files to the provider. The component uses Salesforce Lightning Design System (SLDS) classes to ensure a consistent and responsive UI.

Methods

1. **handleNext:**
   - Description: Handles the "Next" action, primarily for navigating through the component's steps.
   - Code:

     ```
     handleNext() {
         // Handle logic for the "Next" action
     }
     ```

2. **handleNextButtonClick:**
   - Description: Triggers a custom event when the "Next" button is clicked. It's designed for transitioning to the next step within the rate table component.
   - Code:

     ```
     handleNextButtonClick() {
         // Custom event or logic for the "Next" button
     ```

```
}
```

3. **doAcceptRates:**
   - Description: Changes the component state to show the CV upload section after accepting rates.
   - Code:

```
doAcceptRates(event) {
    // Logic for accepting rates and showing CV upload
}
```

4. **handleHAResult:**
   - Description: Processes the result from the hiring audit and updates the UI accordingly to proceed to the rate table or CV upload sections.
   - Code:

```
handleHAResult(event) {
    // Logic for handling the hiring audit result
}
```

5. **checkCurrentRecordCv:**
   - Description: Checks if the current record has an existing CV attached.
   - Code:

```
checkCurrentRecordCv() {
    // Logic for checking existing CV
}
```

6. **submitProviderPresentation, clearProviderPresentation, handleUploadFinished, handleUploadOtherDocsFinished, previewHandler, refreshView, closeQuickAction, fireToast, doPresent:**
   - Description: These methods handle various actions such as submitting the presentation form, uploading documents, previewing CV, refreshing the view, closing the quick action modal, showing toasts for feedback, and presenting the candidate.

Properties

- **hiringAuditId:** Referenced ID for the hiring audit.

- **candidateId:** ID of the candidate being presented.
- **cvPreviewUrl:** URL to preview the candidate's CV.
- **headerText, isCvAvailable, showSpinner, isPresentButtonDisabled:** Various states to control UI elements display and interactivity.
- **showPresentationForm, showRateTable, showCVUpload, disabledFilesUploader:** Boolean properties that control the visibility of specific sections within the component.

Use Case

This component is best suited for scenarios within recruitment processes where a hiring audit involves presenting candidates, submitting candidate details, and uploading necessary documents like CVs and additional files. It can be used as a quick action within Salesforce to streamline the candidate presentation process directly from Salesforce records related to hiring audits.

Important Notes

- **SLDS:** The Salesforce Lightning Design System classes are used to maintain the styling consistent with the Salesforce Lightning Experience.
- **Apex Integration:** The component integrates closely with Apex controllers for back-end operations. Ensure the referenced Apex methods (`checkExistingCV`, `attachFilesToProvider`) are correctly implemented and accessible.
- **Refresh UI:** The component makes use of explicit UI refresh calls to ensure that UI state is consistent with the backend state after operations like file upload or record update.
- **NavigationMixin:** Used for navigating to web pages, like the CV preview URL. Pay attention to URL validity and access rights.
- **Event Handling:** Custom events are used to communicate between parent and child components (e.g., handling next button clicks or submitting forms). Proper event handling is crucial for the component's workflow.
- **Accessibility:** Ensure that accessibility features such as labels and ARIA attributes are correctly implemented for a broader user acceptance.

nuQuickActionPresentToClient

Description

`NuQuickActionPresentToClient` is a Salesforce Lightning Web Component (LWC) intended for presentation purposes in the Hiring Audit process flow. The main functionality of this component centers around marking a specific Hiring Audit record as "Presented to Client" within Salesforce. It utilizes standard Salesforce UI elements like modals and spinners for feedback during the operation.

Properties

1. `hiringAuditId`:
   - Type: `@api`
   - Description: A public property to store the ID of the Hiring Audit record. The value is passed into the component, enabling it to perform operations related to that specific record.
2. `quickActionLabel`:
   - Type: `@api`
   - Description: A public property intended to store a label for the quick action. This property can be used to dynamically set the modal's title or button labels but is currently not utilized in the visible part of the provided code.
3. `showSpinner`:
   - Type: `Boolean`
   - Description: A private property used to control the visibility of the loading spinner, providing user feedback during asynchronous operations.

Methods

1. `connectedCallback()`:
   - Visibility: Private
   - Description: Lifecycle hook that's called when the component is inserted into the DOM. It starts an operation to update the Hiring Audit record's status to "Presented to Client" (`HA_STATUS_FIELD`).
2. `closeQuickAction(event)`:
   - Visibility: Private
   - Parameters: `event` - The event object that initiated the method call.
   - Description: Dispatches a `finishquickaction` event to signal the completion of the quick action, allowing parent components or handlers to respond accordingly.
3. `doSubmit(event)` (Commented out):

- Visibility: Private
- Parameters: `event` - The event object that initiated the method call.
- Description: *Commented Out* — Intended for submitting form data collected from the user through input fields in the component's template. Dispatches updates to the record and emits a `finishquickaction` event upon completion.

Use Case

In a Salesforce implementation related to hiring processes, this component can be used to quickly mark a Hiring Audit record as "Presented to Client" without the need for the user to manually edit the record. This can streamline workflows and improve user experience by reducing the number of steps required to update record statuses.

Important Notes

- The component contains sections of code that have been commented out, including a modal form intended for user input and a method (`doSubmit(event)`) for handling form submission. To fully utilize these parts of the component, the corresponding code must be uncommented and potentially modified to suit specific requirements.
- The `@wire` decorator and associated handler method for fetching the Hiring Audit record have also been commented out. To incorporate data retrieval into the component, this section should be uncommented and adjusted as needed.
- There are Salesforce schema imports (`HA_CANDIDATE_FIELD`, `HA_STAGE_FIELD`, `HA_STATUS_FIELD`) used in the JS code, indicating the component's reliance on specific custom fields that must exist in the Salesforce environment for proper function.
- Use of `updateRecord` from the `lightning/uiRecordApi` module is demonstrated, showcasing how to programmatically update Salesforce records within LWCs.

Example of emitting an event upon closing the quick action:

```
this.dispatchEvent(new CustomEvent('finishquickaction'));
```

The component's current configured functionality primarily involves automatic background processes (e.g., updating a record's field upon component initialization) rather than direct interaction via a UI form.

nuQuickActionResubmitJobOrderForApproval

Description

This component `NuQuickActionResubmitJobOrderForApproval` is designed to operate within the Salesforce Lightning Experience. It functions primarily as a UI modal dialog that allows users to resubmit a Job Order for approval through a Salesforce Lightning Flow. It displays a modal with a title "Resubmit Job Order for Approval". Upon record ID availability, it dynamically feeds the record ID to the flow named ‘Job_Order_resubmit_for_approval’ as an input variable, thus facilitating the approval process. The component also features functionality to close the modal dialog, handle the result status from the Lightning Flow execution, and present relevant messages to the user through toast notifications.

Methods

1. **recordIdpopulated**: A getter method that checks if the `recordId` property is populated. If true, it sets the input variables for the flow and returns true.

```
get recordIdpopulated(){
    if (this.recordId != null && this.recordId != undefined){
        this.inputVariables =  [{
            name: 'recordId',
            type: 'String',
            value: this.recordId
        }];
        return true;
    }else{
        return false;
    }
}
```

2. **closePopup**: Invokes the dispatch event to close the modal dialog. This is designed to be called when the close button within the modal header is clicked.

```
closePopup(){
    this.dispatchEvent(new CustomEvent('finishquickaction'));
}
```

3. **handleStatusChange**: Handles the status change of the Lightning Flow. Upon completion (`FINISHED` status), it closes the modal. In case of errors (`ERROR` status), it presents an error toast to the user and then closes the modal.

```
handleStatusChange(event) {
    if(event.detail.status === 'FINISHED'){
        this.dispatchEvent(new CustomEvent('finishquickaction'));
    }
    else if(event.detail.status === 'ERROR'){
        const evt = new ShowToastEvent({
            title: 'Error',
            message: 'Most likely record do not match any approval
process criteria',
            variant: 'error',
        });
        this.dispatchEvent(evt);
        this.dispatchEvent(new CustomEvent('finishquickaction'));
    }
}
```

Properties

1. **recordId** (`@api`): This property is intended to receive the Salesforce record ID on which the job order resubmission for approval takes place. It's marked with `@api` for it to be publicly exposed to be set by the component's parent.

```
@api recordId;
```

Use Case

This component is useful in scenarios where there is a need to resubmit Job Orders for approval directly from a UI without navigating away to different components or pages. For instance, this can be embedded in record pages for easy access to resubmit approvals.

Important Notes

- This component is dependent on the `recordId` being passed by the parent component or context in which it is placed.

- It dynamically interacts with the Salesforce Flow named `Job_Order_resubmit_for_approval`, and as such, the flow needs to exist and be properly configured to accept the input variables as set by this component.
- The `handleStatusChange` method is crucial for feedback to users, particularly to handle any exceptions or errors gracefully.
- The CSS classes used are part of the Salesforce Lightning Design System (SLDS) and ensure that the modal dialog's UI is consistent with the Salesforce Lightning Experience UI.

nuQuickActions

Description

The `NuQuickActions` component is a versatile Salesforce Lightning Web Component (LWC) designed to handle a wide array of quick actions related to recruitment processes, like submitting candidates, presenting candidates, setting up interviews, making offers, and much more. This component dynamically renders child components based on the type of action determined by the `quickActionApiName` property. Each child component represents a specific quick action and is capable of emitting a `finishQuickAction` event upon completion.

Methods

- `connectedCallback()`: Lifecycle hook that runs when the component is inserted into the DOM. It evaluates the `quickActionApiName` property and sets flags to conditionally render the appropriate child component.
- `finishQuickAction(event)`: Handler for the `finishQuickAction` event emitted by any of the child components. It further dispatches a `finishQuickAction` event, optionally with details from the event it received.

Properties

- `recordId` (`@api`): The Salesforce record ID related to the quick action.
- `agencyId` (`@api`): The Salesforce record ID for the agency, relevant for certain actions.
- `quickActionLabel` (`@api`): Label for the quick action, used for display purposes in child components.
- `quickActionApiName` (`@api`): API name of the quick action, used to determine which child component to display.

- `showEndDate`, `showStartDate` (`@api`): Boolean flags to control the visibility of start and end date fields in certain actions.
- Various boolean flags like `isSubmitCandidate`, `isMakeOffer`, etc., which are set based on the `quickActionApiName` and control the rendering logic for different actions.

Use Case

This component can be used in a Salesforce org where recruitment-related Quick Actions need to be performed directly from a Lightning page. For example, a recruiter could use this component on a Job Order record page to perform actions such as submitting candidates to the job order, presenting them to the hiring manager, or making an offer to the candidate, all without leaving the page.

Important Notes

- The `connectedCallback()` method plays a crucial role in component initialization by deciding which child component needs to be displayed based on the `quickActionApiName` property.
- The child components are expected to emit a `finishQuickAction` event upon completing their tasks. This event is then captured by the parent `NuQuickActions` component and can be handled or propagated further as needed.
- The component employs Salesforce Lightning Web Component (LWC) technology, making it suitable for embedding within Lightning Experience, Salesforce Mobile App, or any web technology that supports LWC.

Example

To use this component for a specific quick action, ensure it is placed on a page with its `quickActionApiName` property set to match the API name of the desired action. For example:

```
<c-nu-quick-actions quick-action-api-
name="Job_Order__c.Submit_Candidate" record-id="a1S3h0000074hELEAY"></c-
nu-quick-actions>
```

This instantiation of the component will render the UI for submitting a candidate because the `quickActionApiName` matches the case in the `connectedCallback()` method that sets `this.isSubmitCandidate = true`.

nuQuickActionSetupInterview

Description

This Salesforce Lightning Web Component (LWC) is designed to display a spinner while performing automated updates in the background, specifically for updating a `Hiring_Audit__c` record to indicate that an interview stage has been set up. The component operates primarily in the background, displaying a spinner to the user when activated, and then automatically updating specific fields of a provided `Hiring_Audit__c` record.

Methods

connectedCallback()

This lifecycle hook gets invoked when the component is inserted into the DOM. In this method, it checks if the record has not been previously updated during the component's lifecycle, then sets the interview stage and status on the `Hiring_Audit__c` object before dispatching a `finishquickaction` event to indicate completion.

```
connectedCallback() {
    this.showSpinner = true;
    if (!this.isRecordUpdated) {
        // Updates the record with predefined values
    }
}
```

closeQuickAction(event)

This method handles the closing of the quick action by dispatching a `finishquickaction` event, signaling to the parent or host component that the action has concluded.

```
closeQuickAction(event) {
    this.dispatchEvent(new CustomEvent('finishquickaction'));
}
```

Properties

1. `@api hiringAuditId`: An API property to pass the ID of the `Hiring_Audit__c` record to the component.

2. `@api quickActionLabel`: An API property intended to provide a label or title for the quick action, though it appears unused in the current version of the component.
3. `showSpinner`: A private property utilized to control the visibility of the `lightning-spinner` component based on whether an action is in progress.

Use Case

This component is designed to be used in scenarios where a user initiates an interview setup action from within the Salesforce UI. Upon activation, it displays a loading spinner and automatically updates the associated `Hiring_Audit__c` record to reflect the new status of "Interview Scheduled" and the stage as "Interview". The component then signals completion by dispatching an event.

Important Notes

- The component is currently focused on showing a spinner and performing a background update to the `Hiring_Audit__c` record without providing any UI for user input or interaction beyond the initial activation and completion stage.
- The majority of the component's potential functionality, such as form inputs for the interview details and the ability to manually submit these details, is commented out. This suggests that the component is either in an incomplete state of development or has been simplified for a specific use case.
- The component utilizes the `updateRecord` method from the `lightning/uiRecordApi` module for performing DML operations, indicating a declarative approach to interacting with Salesforce data.
- Error handling is present in the connectedCallback method to catch and log errors during the update process.

nuQuickActionSubmitCandidate

Description

This Salesforce Lightning Web Component (LWC), `NuQuickActionSubmitCandidate`, provides a modal interface for submitting a candidate selection related to a job order and agency within Salesforce. The component displays a modal with a candidate lookup field that allows the user to search and select a candidate record. Once a candidate is selected, their ID is captured for submission when the "Save" button is clicked. The component utilizes an Apex controller method `submitCandidate` to submit the selected candidate's information.

Methods

- **handleLookupRecord(event):** Reacts to the "lookupupdate" event emitted by the `c-candidate-lookup` component. It grabs the selected candidate's ID from the event's detail.
- **doSubmit(event):** Handles submission by invoking the Apex method `submitCandidate` with the selected candidate ID, job order ID, and agency ID. It utilizes promises to handle the response and potentially triggers a toast message if submission is duplicated.
- **closeQuickAction(event):** Dispatches a custom event, `finishquickaction`, signaling that the modal should be closed without performing any action.
- **@wire(getObjectInfo, { objectApiName: CONTACT_OBJECT }):** Wired method to get the record type ID for 'Candidate' from the `Contact` object's metadata.

Properties

1. **@api jobOrderId:** External ID for a job order passed into the component.
2. **@api agencyId:** External ID for an agency passed into the component.
3. **@api quickActionLabel:** External label for the quick action, displayed as the modal's heading.
4. **candidateRecordTypeId:** Stores the record type ID for "Candidate", derived from the Contact object.
5. **candidateId:** The ID of the selected candidate to be submitted.

Use Case

This component can be utilized in Salesforce as a quick action or a part of a bigger flow that requires selection and submission of candidate records linked to specific job orders and agencies. It's particularly useful in recruitment processes within Salesforce, allowing for streamlined candidate submissions with minimal user input.

Important Notes

- **Lookup Component Dependency:** This component uses a custom component, `c-candidate-lookup`, for looking up candidate records. Ensure this component is available and properly configured in your org.
- **Apex Controller Dependency:** The component relies on an Apex controller, `nuController_QuickActions`, specifically the `submitCandidate` method. This method

must exist and be accessible by the component.

- **Error Handling:** The component uses a utility, `fireToast`, to display error messages. Make sure this utility is implemented and handles custom toast messages properly.
- **Record Type ID Fetching:** The component fetches the Record Type ID for "Candidate" dynamically; ensure the Contact object has a "Candidate" record type set up.

CSS

Here is the CSS snippet used in the component:

```
.hidden-submit {
    display: none;
}
```

This class is utilized to initially hide the submit button and programmatically trigger the form submission if needed (though the related JS functionality is commented out in the provided code).

Example Usage

The component could be placed on a Lightning Page or within another component, and invoked with specific `jobOrderId` and `agencyId` values to tie the candidate submission to those records:

```
<c-nu-quick-action-submit-candidate job-order-id="a1S4P000000k9UJ"
agency-id="a2D4P000000k9UI" quick-action-label="Submit Candidate">
</c-nu-quick-action-submit-candidate>
```

This would render the modal for submitting a candidate for the specified job order and agency.

nuRatesTable

Description

The `NuRatesTable` component extends the Salesforce `LightningDatatable` component to add custom functionality specifically for managing picklist columns. This component allows users to use a custom data type `picklistColumn` for displaying and editing picklist values within a table cell. The implementation includes separate templates for display (`picklisstatic.html`) and edit modes (`picklistColumn.html`), along with the

capability to customize the picklist behavior and appearance through a set of defined attributes.

Methods

This component inherits all methods from the `LightningDatatable` component but does not define any additional methods of its own.

Properties

The `NuRatesTable` component introduces a custom data type `picklistColumn` with several properties defined under `typeAttributes`. These properties include:

- `label`: The label of the picklist field.
- `placeholder`: A placeholder text that appears when the picklist field is empty.
- `options`: A list of options for the picklist.
- `value`: The currently selected value of the picklist.
- `context`: Any additional context needed for the picklist field.
- `variant`: The visual variant of the picklist field.
- `name`: The unique name of the picklist field.

Use Case

The custom `NuRatesTable` component is particularly useful in scenarios where a Salesforce Lightning datatable needs to include one or more columns that allow users to select from predefined options (picklists). For instance, an application managing financial rates might use this component to allow users to specify rate types (e.g., Fixed, Variable) directly within a table row.

Important Notes

1. **Templates**: The custom component requires two separate HTML templates – one for the static display mode (`picklistst atic.html`) and another for the edit mode (`picklistColumn.html`). These templates need to be properly defined and stored in the same directory as the JavaScript class.
2. **Inheritance**: Since `NuRatesTable` extends `LightningDatatable`, it inherits all of its properties, methods, and behaviors. This means that apart from the custom functionality introduced, all standard datatable functionalities are available.

3. **Custom Types**: The usage of `static customTypes` defines a way to extend the datatable with custom data types. This is a powerful feature that allows developers to introduce complex field types beyond the standard ones (e.g., text, number, date) provided by Lightning Web Components (LWC).
4. `typeAttributes`: These attributes provide a flexible way to pass configurations and controls to the custom picklist column, enabling a wide range of use cases and custom behaviors.

Example usage within a Lightning Web Component could involve defining a datatable in the component's HTML template and specifying columns in the JavaScript class that utilize this custom `picklistColumn` type, setting up the necessary options, and handling value changes as desired by the application logic.

nuRatesTableComp

Description

This Lightning Web Component (LWC) named `NuRatesTableComp` is designed to manage and display rate-related information within a Salesforce environment, specifically designed for the Rate Management System. The component utilizes a lightning-card layout to display a table (`c-nu-rates-table`, a custom component) populated with records fetched from an Apex class, allows for inline editing of records, and includes functionalities to add, update, and delete rate records. It also features dynamic picklist fields based on schema information and showcases the use of various Lightning Data Service and Apex method invocations to manipulate and display data relevant to "Rates".

Methods

- **handleAddNewRow:** Adds a new empty rate row to the table, allowing users to input new rate information.
- **handleCellChange:** Triggers when cell data in any row is edited, updating the draft values state to reflect changes made.
- **handleSave:** Saves all changes made in the draft to the backend. This involves separating records that need to be inserted from those that need to be updated, making respective `createRecord` and `updateRecord` calls, and handling deletion requests for removed records.
- **handleCancel:** Reverts changes made by resetting the data in the table to the last saved state and clears any draft values.

- **updateDataValues:** Helper method to update the component's data with changes made by the user. It's used internally to synchronize the component's state with user actions.
- **updateDraftValues:** Updates draft values based on inline editing, handling special conditions for rate class and rates class selections.
- **showToast:** Utility method to display toast notifications for success or error messages.
- **refresh:** Refreshes the table data after a save operation, ensuring the displayed data is up-to-date.
- **pollFunction:** An internal function used by the `refresh` method to periodically check if data needs to be updated from the server.

Properties

- **data:** Stores the current rates to be displayed in the table.
- **draftValues:** Tracks changes made by the user to the data before saving.
- **pickListOptions, typeOptions:** Stores picklist options for 'Rate Class' and 'Rate Type' fields.
- **showSpinner:** A boolean to control the display of a loading spinner during data operations.

Use Case

This component could be utilized in a Salesforce org where managing rate information is crucial, for example, in a financial services application where rates are frequently updated and need to be reflected across various parts of the system in real-time. Users can view, add, edit, or delete rate records directly within a user-friendly interface without navigating away from the page.

Important Notes

1. **Custom Apex Calls:** The component relies on custom Apex methods like `fetchRates` to retrieve rate information. Proper setup and access to these methods are crucial for the component to function as expected.
2. **Dynamic Picklist Values:** The component dynamically fetches and assigns picklist values for 'Rate Type' and 'Rate Class' from the Rate object fields, which requires the setup of these fields in the org where the component is deployed.
3. **Inline Editing and Data Management:** The component handles inline editing and manages draft states for potentially complex data manipulation before saving. This ensures a smooth user experience but requires careful handling of data states to prevent unsaved data loss.

4. **LDS and Refresh Strategy:** Leveraging Lightning Data Service (LDS) functionalities like `updateRecord`, `createRecord`, and `refreshApex` for data operations, the component ensures data consistency across the Salesforce UI without manual refreshes, utilizing an asynchronous polling mechanism to refresh data post-save.
5. **CSS Styling:** The `.cardSpinner` CSS class applied to the lightning-card element positions the loading spinner. Ensure this styling does not conflict with other CSS in the Salesforce org where the component is deployed.

nuReviewCancelledEvents

Description

The provided Salesforce Lightning component, named `NuReviewCancelledEvents`, is designed for rendering within a Salesforce environment. This component integrates a custom layout, encapsulated within the `c-nu-page-layout`, alongside the Salesforce Lightning `lightning-flow` component for the purpose of executing a specific flow identified by the `Acknowledge_Cancellation_Requests` API name. Its primary function appears to be providing a user interface for reviewing and acknowledging cancellation requests through the specified Salesforce Flow.

Methods

This component does not define custom JavaScript methods in its `.JS` file. Its functionality is primarily driven by the Salesforce Lightning components it incorporates within its template.

Properties

There are no explicit properties defined within the `.JS` file for `NuReviewCancelledEvents`. However, the component utilizes the `lightning-flow` component's `flow-api-name` property to specify which Salesforce Flow to execute - in this case, `Acknowledge_Cancellation_Requests`.

Use Case

The typical use case for the `NuReviewCancelledEvents` component is to embed it within a Salesforce page where users need to manage and process cancellation requests. For example, in a customer service application within Salesforce, this component could be utilized to streamline the process of reviewing cancellation requests by directly embedding the necessary

Flow into a user-friendly interface. Users can interact with the Flow's steps to acknowledge or act upon these requests without needing to navigate away from the page.

Important Notes

- **Custom Layout**: The component is wrapped within a custom layout component `c-nu-page-layout`. Make sure that the custom layout is properly defined and available within your Salesforce org, as it's a custom component that's not provided out-of-the-box by Salesforce.
- **Flow Integration**: The component integrates a Salesforce Flow using the `lightning-flow` component. The specific Flow to be executed is `Acknowledge_Cancellation_Requests`. It is essential to ensure that this Flow exists and is correctly configured within your Salesforce org to function as expected.
- **Flow Customization**: Any customization or adjustments needed for the Flow's execution (i.e., input variables or behavior after completion) would need to be managed within Salesforce Flow Builder and is not directly handled within this component's code.
- **Permissions**: Users who are expected to use this component must have the necessary permissions to view and execute the specified Flow, `Acknowledge_Cancellation_Requests`.
- **Embedding the Component**: To utilize the `NuReviewCancelledEvents` component within your Salesforce org, you'll need to embed it in a Lightning page, Salesforce Experience Cloud site, or any other area that supports Lightning Web Components (LWC).

Example usage in a Lightning page setup:

```
<c:nu-review-cancelled-events></c:nu-review-cancelled-events>
```

Ensure the API version of your Salesforce org supports the features utilized in this component.

nuScheduler

Description

This documentation covers a Salesforce Lightning Web Component (LWC) named `nuScheduler`. `nuScheduler` is designed to provide a comprehensive scheduling interface within the Salesforce platform, allowing users to create, view, and manage shifts or appointments. It supports different modes (e.g., Client, Agency), making it versatile for various

business needs. The component integrates with custom Apex controllers to fetch relevant data, such as contacts, accounts, and placements, to populate its interface dynamically.

Methods

1. **setColors()**: Determines the color scheme for the scheduler based on the selection and status of shifts or appointments. It adjusts the colors for confirmed, unconfirmed, open, and available shifts.
2. **computeTotalHours()**: Calculates the total hours between the start and end times of a shift or appointment.
3. **formatTime(time)**: Formats a given time string to a more user-friendly format.
4. **handleSelectorChange(event)**: Handles changes in the shift or appointment type selector.
5. **handleLocationChange(event)**: Handles changes in the location selector.
6. **handleProviderTypeChange(event)**: Handles changes in the provider type selector.
7. **handleSpecialtyChange(event)**: Handles changes in the specialty selector.
8. **handleTimeChange(event)**: Handles changes in the start and end time inputs and recalculates the total hours.
9. **handleConfirmChange(event)**: Handles changes in the confirmation status of a shift or appointment.
10. **lookupRecord(event)**: Handles the selection of a record from the provider lookup component.
11. **previousHandler()**, **nextHandler()**, **refresh()**, **today()**: Navigation and refresh controls for the calendar views.

Properties

1. **selectedLocationOption**: Tracks the currently selected location option.
2. **selectedProviderTypeOption**: Tracks the currently selected provider type option.
3. **selectedSpecialtyOption**: Tracks the currently selected specialty.
4. **selectedOption**: Tracks the currently selected shift or appointment option.
5. **confirmed**: Tracks the confirmation status of the shift or appointment.
6. **colors**: Tracks the current color theme for shifts or appointments based on their status.
7. **colorState**: A boolean flag used to toggle between color states.
8. **mode**: Indicates the mode in which the scheduler is operating, e.g., "Client" or "Agency".
9. **totalHours**: Stores the computed total hours for a shift or appointment.

10. **providerTypeOptions**, **specialtyOptions**, **locationOptions**: Store dynamic options for the related selectors fetched from the backend.

Use Case

`nuScheduler` can be utilized in scenarios where scheduling and time management are crucial, such as in staffing agencies, healthcare facilities, or any organization that needs to manage shifts, appointments, or resource allocations. Its dynamic fetching of contacts, accounts, and customizable options makes it adaptable to specific business models and requirements.

Important Notes

- The component is tightly coupled with custom Apex controllers for fetching required data, meaning that those controllers must be correctly set up and configured in the Salesforce org where the component is deployed.
- The component uses sessionStorage and localStorage for persisting some states across page reloads. This behavior is essential for understanding how user selections are retained.
- CSS classes in the component leverage the Salesforce Lightning Design System (SLDS) for styling, ensuring a consistent look and feel with the Salesforce platform.

The component's dynamic nature, combined with its integration into Salesforce's ecosystem, makes it a powerful tool for managing scheduling needs within a Salesforce org.

nuSearchableMultiPicklist

Description

The `NuSearchableMultiPicklist` component enables users to search, select, and view multiple options from a dropdown list. Users can add options by clicking on them in the dropdown, which then appear as tags or pills above the search box. Users can also remove previously selected options by clicking an 'x' button on each tag.

Methods

- **fetchNewValue()**: Simulates fetching new search results asynchronously.
- **search(event)**: Handles the search input event, dynamically filtering dropdown options based on the user's input.

- **handleRemoveOption(event)**: Removes a selected option from the `selectedOptions` array when the user clicks the close icon on a tag.
- **selectSearchResult(event)**: Adds the clicked option from the dropdown to the `selectedOptions` array.
- **onblurClear()**: Clears the search results under certain conditions which are determined by the `dontClose` flag.
- **clearSearchResults(event)**: Clears the current search results, returning the state to no results shown.
- **showPicklistOptions()**: Displays all initial options when the search input is focused, before any search is performed.
- **handleDropdownClick(event)**: Prevents the dropdown from closing when clicking inside of it.
- **handleInputClick(event)**: Manages focus state to control dropdown visibility.

Properties

- **options**: An array of objects representing all available options. Each object in the array should have `label` and `value` properties.
- **optionsSearch**: An array of objects representing filtered options based on search input.
- **selectedSearchResult**: The currently selected search result.
- **defaultvalue**: A value to preselect an option on the component initialization.
- **selectedOptions**: An array of objects that hold the options selected by the user.

Use Case

This component could be used in scenarios where a user needs to select multiple options from a large list, such as assigning tags to a blog post or selecting skills on a job search platform. By allowing search and selection in a compact UI, it enhances user experience, especially when dealing with extensive options.

Important Notes

- **Accessibility**: Ensure that interactions with the component are fully accessible, including keyboard navigation and screen reader support.
- **Singleton Selection Prevention**: The component currently does not prevent the same option from being selected more than once, which could be implemented based on the specific use case requirements.

- **@track Decorator**: The use of the `@track` decorator can be optimized further based on the property's usage. For properties expected to change frequently, `@track` ensures the UI re-renders to reflect those changes.
- **Asynchronous Handling**: The simulated `fetchNewValue` method showcases the capability to handle asynchronous operations, which in real scenarios would likely involve fetching data from a server.

Example Implementation

The component requires the `options` property to be populated with available selections. Here is an example of how it could be used in a Salesforce Lightning Web Component:

```
<c-nu-searchable-multi-picklist options={picklistOptions}></c-nu-searchable-multi-picklist>
```

`picklistOptions` should be an array of objects structured as follows:

```
[
  { label: 'Option 1', value: 'opt1' },
  { label: 'Option 2', value: 'opt2' },
  // more options...
]
```

The component also emits custom events `valueselect` and `optionsselect` that can be listened to in order to perform additional actions based on the user's selection.

nuSearchablePicklist

Description

This Salesforce Lightning Web Component, named `NuSearchablePicklist`, creates a customizable, searchable picklist input. It leverages the Salesforce Lightning Design System (SLDS) to provide a styled and interactive UI. Users can search through the picklist options by typing, and the component will filter and display matching options. Users can select an option by clicking it, and the component can operate in both editable and read-only modes.

Methods

- **search(event):** Filters the `options` based on the user's input. It updates the `optionsSearch` with the filtered results, allowing users to see only the matching options.

```
search(event) {
    this.defaultvalue = '';
    const input = event.detail.value.toLowerCase();
    const result = this.options.filter((picklistOption) =>
      picklistOption.label.toLowerCase().includes(input)
    );
    this.optionsSearch = result;
    this.selectedSearchResult = null;
}
```

- **selectSearchResult(event):** Updates the `selectedSearchResult` with the user's selection from the filtered options.

```
selectSearchResult(event) {
    this.defaultvalue = '';
    const selectedValue = event.currentTarget.dataset.value;
    this.selectedSearchResult = this.options.find(
        (picklistOption) => picklistOption.value === selectedValue
    );

    this.clearSearchResults();
}
```

- **clearSearchResults:** Clears the search results and sets the focus state to `false`.

```
clearSearchResults(event) {
    this.optionsSearch = null;
    this.focussed = false;
}
```

- **showPicklistOptions:** Displays the options when the input field is focused, if no search has been performed yet.

```
showPicklistOptions() {
    if (!this.optionsSearch) {
        this.optionsSearch = this.options;
    }
}
```

- **handleDropdownClick, handleInputClick:** Prevents the dropdown from closing unexpectedly by managing focus and click events.

Properties

- **@api options:** The list of options that can be searched and selected from.
- **@api optionsSearch:** The filtered list of options based on the search term.
- **@api selectedSearchResult:** Currently selected option from the filtered list.
- **@api defaultvalue:** The predefined value if any is to be selected upon component initialization.
- **@api selectedOption:** (Optional) Explicitly selected option, useful in scenarios where an external action determines the selection.
- **@track readOnly:** Determines if the component is in read-only mode.
- **@track type:** Indicates the type of input, dynamically switching between 'search' and 'text' based on editable state.

Use Case

This component is ideal for situations where users need to choose from a large list of options but are likely to know what they're looking for. For instance, selecting a product code from a large catalog, or choosing a user from a massive user base, where typing to filter is much faster than scrolling through options.

Important Notes

- The component transitions the input field between editable and read-only states. This behavior is essential for preserving UI consistency in different application contexts.
- Remember, this component uses custom events like `fireCustomEvent` from an imported module `c/nuUtils`. Ensure this module is available in your Salesforce org.
- Proper handling of the `readonly` attribute is crucial for accurate data representation and interaction with the component.
- The component's styling and behavior heavily rely on the Salesforce Lightning Design System (SLDS), ensuring a consistent look and feel with the Salesforce experience.

nuSimpleFooter

Description

The given Lightning Web Component (LWC) is a simple, reusable footer component named `NuSimpleFooter`. It's designed to display a text message that includes a copyright notice for "Syncx ©2023". The text is centrally aligned, colored, and padded, making it aesthetically pleasing at the bottom of a Salesforce Lightning Experience page or any other LWC-based project.

Properties

This component does not have any properties defined within the JavaScript (`JS`) class. Therefore, it showcases static content without any reactive or dynamic data binding.

Methods

As with properties, no methods are defined within the component's `JS` class. It means this component does not perform any actions or respond to events; it simply displays static content.

Use Case

The `NuSimpleFooter` component can be used in Salesforce Lightning Experience, Salesforce App, or any LWC-based project where a footer is needed to display copyright, company name, or any generic information at the bottom of a page. Due to its simplicity and lack of dynamic content, it's best suited for static pages or areas of an application where the footer content does not need to change based on user interaction or other variables.

Important Notes

- **Styling**: The styling of the footer is defined within the component's CSS file, meaning it's encapsulated and won't affect other elements outside the component. If you need to adjust the styling (e.g., color, padding), modifications should be made directly within the component's `.CSS` file.
- **Customization**: Although this component does not support properties or methods for dynamic behavior, it can be easily extended to do so. For instance, copyright text could be passed as a `@api` property to make the component more versatile.
- **Usage**: To use this component in a Lightning page or another LWC, ensure you import it correctly using its file name (assuming `nuSimpleFooter`). For example:

```
<c-nu-simple-footer></c-nu-simple-footer>
```

- **Salesforce-Specific Considerations**: Within Salesforce, ensure that the component's XML configuration file (`metadata.xml`, not provided) is correctly set up for the intended type of usage (e.g., available for Salesforce App, Lightning Experience, etc.).

This straightforward yet elegant component encapsulates the essentials of a Lightning Web Component, demonstrating how HTML, CSS, and JS files come together to create functional elements within the Salesforce ecosystem or any LWC-compatible environment.

nuTimesheetApproval

Description

This Salesforce Lightning Web Component (LWC) named `NuTimesheetApproval` provides functionality for displaying and managing timesheets. It features several sections including Submitted, Approved, and Rejected timesheets, and offers capabilities such as filtering, approval, rejection, and viewing of detailed timesheet entries. This component is designed to be used by approvers to manage timesheets efficiently and supports interactions like approving or rejecting individual or multiple timesheets, filtering timesheets based on various criteria like date range, candidate, location, and viewing modal dialogs for actions like rejection justification.

Methods

1. **connectedCallback()**: Initializes component data upon insertion into the DOM.
2. **handleRowAction(event)**: Handles actions on rows within the datatable component.
3. **handleRejectClick(event)**: Opens a modal for rejecting a timesheet.
4. **handleApproveClick(event)**: Approves a timesheet.
5. **handleApproveAllClick()**: Approves all displayed timesheets.
6. **handleInputChange(event)**: Handles changes in input fields within the component.
7. **handlePickChange(event)**: Handles selection changes in picklist fields.
8. **rejectTimesheet()**: Confirms the rejection of a timesheet with specified reasons.
9. **closeModal()**: Closes an open modal dialog.
10. **handleDateChange()**: Handles changes in date fields for filtering.

Properties

- **timesheets**: Getter that returns the timesheetList data.
- **showApproveAll**: Getter that indicates whether 'Approve All' button should be shown.

- **otherComment**: Getter that indicates whether an other comment is needed.

Use Case

This component is particularly useful in scenarios where a manager or an approver needs to review timesheets submitted by employees. It allows for a detailed review process, enabling filtering by specific criteria (e.g., dates, candidate, location) and provides direct actions for approving or rejecting timesheets. The component enhances the efficiency of the timesheet approval process within organizations using Salesforce.

Important Notes

- Component relies on Apex controller methods `approveTimesheet`, `rejectTimesheet`, and `fetchTimesheetsForApprover` for backend operations.
- `@wire` and `@track` decorators are used extensively for reactive properties and to wire Apex methods to component properties.
- Component utility functions `fireToast`, `deepCopy`, and `logIt` from `c/nuUtils` are utilized for common operations.
- Component is designed with responsiveness in mind, although specific CSS class `slds-modal__container` is overridden to adjust modal width.
- Interaction with this component requires the user to have corresponding permissions to view and manage timesheets in the Salesforce org.

nuTimesheetEntryView

Description

This documentation covers the Salesforce Lightning Web Component (LWC) `NuTimesheetEntryView`. The component displays a timesheet entry in either a vertical or horizontal layout, depending on the configuration. It includes functionality for adding shift slots to a given entry, conditional rendering based on the component's state, and customization through various properties.

Methods

connectedCallback()

Executed when the component is inserted into the DOM. It initializes component state, specifically determining the `allowSlotAdd` property based on the presence of time slots in

the `entry` property.

renderedCallback()

Executed after the component is rendered in the DOM. It is used to query child components and perform debugging operations, along with initializing the `accessSlots` property with child component references.

handleAddSlotClick()

Handles the click event on the Add Shift button by firing a custom event `addslotclick` with the current entry data.

updateAllowAdd(event)

An event handler for internal events that updates the `allowSlotAdd` property based on incoming event details.

Properties

- `debugVar` (api): Used for debugging purposes.
- `debugTarget` (api): Target component for debugging.
- `mode` (api): Specifies the mode of the component.
- `layout` (api): Determines the layout (`vertical` or `horizontal`) of the component.
- `entry` (api): The timesheet entry data.
- `allowSlotAdd` (api): Controls the visibility of the Add Shift button.
- `showSlotInput` (api): Determines whether the slot input is visible.
- `updatedDate` (api): Date when the entry was last updated.
- `editMode` (api): Flag to toggle edit mode.
- `showShift`, `showUnpaidBreak`, `showNote` (api): Flags to control the visibility of various elements.
- `autoBreakValue` (api): Value for automatically calculated break.
- `shiftOptions`, `otShifts`, `multidayShifts`, `breakShifts`, `hoursOnlyShifts` (api): Arrays representing different configuration options for shifts.
- `unpaidBreakOptions` (api): Options related to unpaid break times.
- `verticalLayout`: Computed property based on `layout`, determines if the layout is vertical.

Use Case

The `NuTimesheetEntryView` component can be used in a Salesforce application where timesheet management is essential. Users can see an overview of their shifts or specific timesheet entries presented in a chosen layout. The component supports additions of new shift slots, offering flexibility in managing work times, especially for scenarios with complex or varying shift schedules.

Important Notes

- To use custom CSS and utility functions (`nuUtils`), ensure these are properly imported and available in your Salesforce org.
- The structure of the `entry` property should align with the component's expectations, especially for properties accessed directly within the template.
- Handling of events (`addslotclick` and internal component events) requires proper setup in the parent component to respond to user actions effectively.
- The component dynamically updates based on its properties, making it versatile for different use cases but requiring careful state management.

nuTimesheetFormattedTabs

Description

This Salesforce Lightning Web Component (LWC) named `NuTimesheetFormattedTabs` is designed to display timesheet data across different tabs categorizing them based on their status: Action Required, Submitted, Approved, and All. Each tab contains a nested custom component `<c-nu-timesheets-formatted>` which is responsible for rendering the timesheet data according to the given filters and settings.

Methods

- `connectedCallback()`: This lifecycle hook gets called when the component is inserted into the DOM. In this example, it is used to log the values of `showWorkLocation` and `workLocationLabel` to the console.

```
connectedCallback() {
    console.log(this.showWorkLocation);
```

```
    console.log(this.workLocationLabel);
}
```

Properties

The component has several public properties (decorated with `@api`) which allow customization and control from the parent component:

- `testContactId`: Allows the identification of a specific contact.
- `renderMode`: Determines how the component should be rendered.
- `positionLabel`: Label for the position field (default: "Position").
- `weekStartLabel`: Label for the week starting date field (default: "Week Starting").
- `weekEndLabel`: Label for the week ending date field (default: "Week Ending").
- `totalHoursLabel`: Label for the total hours field (default: "Total Hours").
- `guaranteedHoursLabel`: Label for the guaranteed hours field (default: "Guaranteed Hours").
- `fileAttachmentsPageName`: The name of the page for file attachments (default: 'timesheet-files').
- `showFiles`: Controls whether file attachments are shown or not.
- `title`: The title of the component (default: 'Timesheets').
- `clockInClockOut`, `showUnpaidBreak`, `autoBreakValue`, `showNote`, `showShift`, `showPosition`, `showWeekStart`, `showWeekEnd`, `mode`, `showTotalHours`, `showGuaranteedHours`: Various settings to control the visibility and behavior of timesheet entry fields.
- `statusFilter`: Set to 'All' by default, used to filter timesheet entries in the "All" tab.
- `showWorkLocation`: Boolean indicating if the work location should be shown (default: false).
- `workLocationLabel`: Label for the work location field.

```
@api showWorkLocation=false;
@api workLocationLabel;
```

Use Case

This component is ideal for use in Salesforce applications where managing timesheets and tracking employee work hours are required. Specifically, it allows for the organization of

timesheets based on their submission status, making it easier for users to navigate through and manage timesheets accordingly.

Important Notes

- Since `@api` properties are exposed, they can be set and modified by the parent component that includes `<c-nu-timesheet-formatted-tabs>`, allowing for flexible customization according to specific requirements.
- The `connectedCallback()` method is useful for initializing the component with specific logic but should be used cautiously as it fires every time the component is inserted into the DOM.

Here's an example of how the component can be utilized in a parent component:

```
<c-nu-timesheet-formatted-tabs
    test-contact-id="003xx000004TmiZAAS"
    render-mode="compact"
    show-files={true}>
</c-nu-timesheet-formatted-tabs>
```

This snippet demonstrates how a parent component can pass specific properties to customize the `NuTimesheetFormattedTabs` component, like setting a contact ID, rendering mode, and controlling the visibility of file attachments.

nuTimesheetHeaderView

Description

This Lightning Web Component (LWC) is designed to display details of a timesheet record in Salesforce, highlighting various attributes such as position, week starting, week ending, total hours, guaranteed hours, work location, expense status, and reasons for rejection if any. It also provides capability to navigate to further details or actions related to the timesheet, like uploading or viewing expenses and initiating a chat regarding the timesheet record.

Methods

`handleUrlClick(event)`

Handles the click event on the timesheet URL to fire a custom event named `timesheeturlclick` with the `timesheetId` as detail.

```
fireCustomEvent(this, 'timesheeturlclick', { timesheetId:
this.timesheet.timesheetId });
```

`handleAttachmentsClick(event)`

Invoked when the button to view or upload expenses is clicked. It checks whether to view or upload based on the button's label and fires a custom event `fileattachmentsclick` with `timesheetId` and `viewOnly` status.

```
const view = this.expensesLabelButton == 'View Expenses' ? true : false;
fireCustomEvent(this, 'fileattachmentsclick', { timesheetId:
this.timesheet.timesheetId, viewOnly: view });
```

`navigateToWorkRecord(event)`

Navigates to the work record related to the timesheet using Salesforce's standard record navigation functionality, focusing on the chat tab if applicable.

```
this[NavigationMixin.Navigate]({
    type:'standard__recordPage',
    attributes:{
        recordId: recordId,
        actionName: 'view',
    },
    state: {
        focusOnChatTab: true
    }
});
```

Properties

- `timesheet`: An object holding the details of a timesheet record.
- `timesheetLabel`, `positionLabel`, `weekStartLabel`, `weekEndLabel`, `totalHoursLabel`, `guaranteedHoursLabel`, `RejectedReasonLabel`, `workLocationLabel`, `expenseStatus`: String properties to hold labels for timesheet-related data.
- `showPosition`, `showWorkLocation`, `showWeekStart`, `showWeekEnd`, `showTotalHours`, `showGuaranteedHours`, `showFiles`: Boolean properties controlling the visibility of respective fields in the UI.

- Various getter methods like `TimesheetLabel`, `rejected`, `expenseButtonExceptRejected`, `showExpenseStatus`, provide computed values for use within the component.

Use Case

The component is ideal for use in a project management or HR system where tracking timesheet details, including hours worked, expenses incurred, and reviewing specific timesheet records are vital aspects. It allows users to get a quick overview of timesheet status and perform actions like uploading expense details directly from the summary view.

Important Notes

- Requires `import` statements for specific Salesforce modules like `LightningElement`, `NavigationMixin`, and Apex methods (`fetchTimesheet`, `fetchContact`), indicating that this component is tightly coupled with Salesforce's ecosystem.
- This component utilizes Salesforce's Lightning Design System (SLDS) for styling, ensuring consistency with the Salesforce UI.
- It's essential to have `c/nuUtils` and `c/nuCSS` utility components or modules available in your org for this component to function correctly, as it relies on them for utility functions and CSS variables.
- The component makes asynchronous calls to fetch timesheet data when the `timesheetId` is defined. Proper error handling is implemented to manage any errors that occur during these asynchronous operations.

This documentation aims to provide a comprehensive understanding of the functions, methods, and use cases of the `NuTimesheetHeaderView` Lightning Web Component.

nuTimesheets

Description

The provided code appears to represent components of a complex Salesforce Lightning Web Component (LWC) named `NuTimesheets`. It is designed to manage and display timesheets in a Salesforce environment, enabling users to view, submit, and manage timesheets with various states such as "Unsubmitted," "Submitted," and "All." It supports both desktop and mobile layouts, differentiates timesheet entries based on criteria like shifts, unpaid breaks, and offers functionalities for adding, editing, removing time slots, and submitting timesheets.

Methods

- **handleSlotSelect**: Event handler for selecting a time slot.
- **handlenuTimesheetSubmitClick AND handleTimesheetSubmitClick**: Event handlers for submitting a timesheet.
- **handleTimesheetSelectionChange**: Event handler for timesheet selection change.
- **handleTimesheetUrlClick**: Event handler for clicking on a timesheet URL.
- **handleFileAttachmentsClick**: Event handler for clicking on file attachments.
- **handleAddSlotClick**: Event handler for adding a new time slot.
- **handleRemoveSlotClick**: Event handler for removing a time slot.
- **handleSaveSlotClick**: Event handler for saving a time slot.
- **handleCancelClick**: Event handler for canceling an action.
- **setRenderMode**: Sets the render mode of the component (Live or Sample).
- **findTimesheet, findEntry, findSlot**: Methods for finding a timesheet, entry, and slot respectively.
- **createSlot**: Method for creating a new time slot.
- **updateSlot**: Method for updating an existing time slot.
- **deleteSlot**: Method for deleting an existing time slot.
- **selectNextTimesheet**: Method for selecting the next timesheet following certain criteria.

Properties

- **isLoading**: Boolean indicating if the component is currently loading data.
- **timesheetLoaded**: Boolean indicating if a timesheet has been fully loaded.
- **selectedTimesheet**: Object holding the currently selected timesheet data.
- **selectedSlot**: Object holding the currently selected time slot data.
- **mobileLayout**: Boolean indicating if the component should render in mobile layout.
- **AgencyLayout, UnpaidBreakOptions, ShiftOptions, etc.**: Various properties configuring how timesheets and their entries should be displayed or interacted with.

Use Case

The component is suitable for organizations that manage employee timesheets within the Salesforce environment, allowing for detailed tracking of work hours, shifts, and the submission of timesheets for payroll or other administrative purposes. It supports multiple views and interactions in a single unified interface, easing the management of work records.

Important Notes

- **Dynamic Data Handling**: The component includes methods for handling real and sample data (`setRenderMode`), indicating it's designed for both development and production use with versatility in data handling.
- **Custom Events**: The component extensively uses custom events for interaction within the timesheet management workflow, such as slot selection, timesheet submission, and slot modification.
- **Styling**: It employs CSS styles both locally and from an external source (`@import 'c/nuCSS';`); this modularity ensures consistent theming and ease of updates.
- **Salesforce-Specific Features**: The use of Salesforce-specific modules and methods (`@salesforce/apex`, `NavigationMixin`, `lightning/*` components) underlines its deep integration within the Salesforce ecosystem.

This component represents a sophisticated use of LWCs for practical business processes within Salesforce, showcasing the capability of Salesforce Lightning to create responsive, data-driven applications.

nuTimesheetsFormatted

Description

This Salesforce Lightning Web Component (LWC) represents a complex timesheet interface for handling different functionalities related to timesheets including viewing and manipulating timesheets, entries, and slots based on various criteria like status, date ranges, classifications etc. It allows interaction with external services to fetch, display, and update timesheet-related data, and supports dynamic UI updates based on user interaction.

Methods

1. **submitTimesheetAsync**: Submits a timesheet by ID.
2. **saveAllShift**: Saves all shifts passed to it.
3. **saveShift**: Saves a single shift.
4. **removeShift**: Removes a shift based on provided shift details.
5. **loadNewShift**: Loads a new shift for a given timesheet entry.
6. **loadTimesheet**: Loads a timesheet by ID.
7. **setSlotAr**: Sets the slotAr variable based on the timesheetCollection provided.

8. **findTimesheet**: Finds a timesheet by its ID.
9. **findEntry**: Finds an entry by its ID within a given timesheet.
10. **findSlot**: Finds a slot by its ID.
11. **createSlot**: Creates a slot with given details.
12. **updateSlot**: Updates a slot with new details.
13. **deleteSlot**: Deletes a slot by its ID.
14. **selectNextTimesheet**: Selects the next appropriate timesheet based on certain criteria.
15. **testPoll**: Function to continuously poll for updates in a fixed interval.
16. **pollFunction**: A function that is called periodically in `testPoll` method to execute polling logic.

Properties

1. **timesheetCollection**: Holds the collection of timesheets loaded from the backend.
2. **filteredTimesheets**: Holds the timesheets filtered based on certain criteria.
3. **isFloatPool**: Indicates whether a given timesheet belongs to floating pool category.
4. **isLoading**: Indicates if the component is in the loading state.
5. **hideHeader**: Boolean to show/hide header based on configuration or action.
6. **startDate**, **endDate**: Start and end date filter values.
7. **statOptions**, **provOptions**, **locOptions**, **classificationOptions**: Options for different picklist filters.
8. **ttb**, **tte**, **tta**: Total bill amount, total expenses, and total amount after adjustments respectively.
9. **showSubmitButton**: Controls the visibility of the submit button.

Use Case

This component can be used in applications requiring detailed timesheet management functionalities, including viewing timesheet details, editing timesheets, filtering timesheets by different attributes like status, date, and classification, and performing operations like submission and deletion of timesheets and their corresponding entries and slots.

Important Notes

1. All asynchronous operations accessing backend data are managed through promises and wired services.

2. This component integrates deeply with Salesforce Apex classes for data fetching and manipulation.
3. It makes extensive use of Salesforce Lightning Design System (SLDS) for styling and layout alignment to ensure consistency with Salesforce UI.
4. Conditional rendering is used heavily to provide a dynamic UI experience based on the state of the component and user interactions.
5. Custom events are used for handling interactions in nested components and propagating actions up the component tree.
6. The component implements various LWC lifecycle hooks for initializing and updating component state based on external changes and interactions.

nuTimesheetSlotView

This documentation covers the Salesforce Lightning component, NuTimesheetSlotView, designed to manage and display timesheet records with the flexibility to handle multiple shifts, unpaid breaks, and read-only views.

Description

`NuTimesheetSlotView` is a component that manages timesheet entries for users, displaying time slots with detailed information such as date, shift, clock-in, and clock-out times, and unpaid breaks. It supports two layout modes (vertical and horizontal), allows for both read-only and data entry states, and can handle special cases such as overtime, multiday shifts, and hours-only shifts. Capability to update timeslot entries through user interactions is provided, alongside the display of messages for errors or information.

Methods

- `handleDateTimeChange(event)`: Processes changes to date or time fields, updating the timeslot record and triggering related calculations for break times and total hours.
- `handleShiftValueChange(event)`: Manages updates to shift selections, modifying related flags and calculations based on the shift type.
- `handleBlur(event)`: A generic handler for blur events on input fields, capturing field values and conducting necessary processing or validation.
- `handleSaveClick()`: Manages the "Save" button click event, validating the current timeslot details and emitting a custom event to trigger saving the data.

- `handleCancelClick()`: Handles "Cancel" button click events, emitting a custom event to signal the action.
- `computeHours()`: Calculates the total hours worked based on clock in/out times and unpaid break length.
- `computeAutoBreak()`: Automatically calculates breaks based on the duration of the shift and predefined rules.
- `fireSlotUpdatedEvent(field, value)`: Emits a custom event to inform parent components of changes to timeslot details.
- `processInput(property, value)`: A utility method for handling input changes, orchestrating validation, calculation, and state updates.

Properties

- `@api timeSlot`: Object representing the current timeslot data.
- `@api isReadonly`: Flag to toggle read-only mode for the component.
- `@api editMode`: Determines if the timeslot is in edit mode.
- `@api showShift`: Controls visibility of shift-related information.
- `@api clockInClockOut`: Indicates if times should be captured as clock-in and clock-out.
- `@api multiDay`: Notes if the timeslot spans multiple days.
- `@api showUnpaidBreak`: Toggles visibility of unpaid break duration field.
- `@api unpaidBreakOptions`: A list of options for unpaid break selection.
- Several other API properties configure display and calculation specifics according to timeslot attributes.

Use Case

This component can be utilized in applications involving timesheet management where users need to view, enter, or update their work hours, shifts, and breaks. It's particularly suited for complex scenarios with multiple shift types, overtime, and multiday coverage.

Important Notes

- While the component handles a variety of timeslot scenarios, it requires external logic (not provided) to persist changes or interact with a database.
- The custom events used for managing interactions and updates (`fireCustomLocalEvent`, `fireCustomEvent`) imply a broader application context not contained within this component.

- The separation of view modes and edit states within the same component enhances reuse but necessitates careful management of the component state to prevent inconsistencies.
- The CSS and helper utilities imported or referenced (`nuCSS`, `nuUtils`) are not defined here, implying dependence on external resources for full functionality.

```
<nu-timesheet-slot-view
    time-slot={timeSlot}
    is-readonly="{isReadonly}"
    edit-mode="{editMode}">
</nu-timesheet-slot-view>
```

This example illustrates how to embed the `NuTimesheetSlotView` in a page, binding necessary attributes for functionality.

nuTimesheetView

```
## Description

The given code outlines a Salesforce Lightning Web Component (LWC) named
**NuTimesheetView**. This component provides a user interface for
viewing and editing timesheet records, with support for both vertical
and horizontal layouts. Features of this component include the ability
to view timesheet header information, list timesheet entries with
varying shift types, and submit timesheet data. Additionally, it
supports debug logging and various customization options through
publicly exposed properties.

## Methods

### connectedCallback()
Runs when the component is inserted into the DOM. It initializes
component setup, especially for debugging purposes.

### renderedCallback()
Executes after every render of the component. It is used here for debug
logging and updating state based on timesheet changes.

### handleFieldChange(event)
Handles changes to the input fields within the component, specifically
forwarding changes to the `lightning-record-edit-form`.

### handleSubmitClick(event)
Handles the submission of the timesheet, invoking custom events and
potentially navigating to a new URL.
```

testPoll()
Starts a polling process to periodically save timesheet entries.

pollFunction()
A function called at intervals by `testPoll` to trigger the save mechanism for all changes.

handleSaveAllClick(event)
Saves all modifications to the timesheet entries and stops the polling process initiated for autosave.

handleEdit(event)
Triggers the edit mode for the timesheet, enabling modifications to be made.

Properties

@api Properties
- `debugVar` (String)
- `debugTarget` (String)
- `timesheet` (Object)
- `showHeader`, `showEntries`, `showSlotInput`, `showSubmitButton`, `showFiles` (Boolean)
- Various labels, checkboxes, and selection options related to timesheet display and functionality.

Private Properties
- `updatedSavedText` (String)
- `editMode`, `clockInClockOut`, `multiDay`, `showShift`, `showUnpaidBreak`, `showNote` (Boolean)
- `shiftOptions`, `otShifts`, `multidayShifts`, `breakShifts`, `unpaidBreakOptions` (Array)

Use Case

This component allows users to interact with timesheet records in Salesforce. It can display detailed information on timesheet entries, manage the entries, and provide inputs for submitting timesheets. It supports flexible layout arrangements and extensive customization options to cater to different user requirements. An ideal use case is for managing employee timesheets in organizations where tracking of work hours, expenses, and billable amounts is essential.

Important Notes

1. **Layout Customization**: The component supports both vertical and horizontal layouts, determined by the `layout` property.
2. **Dynamic Property Handling**: Through the combination of `@api` and private properties, along with getter methods, the component dynamically adjusts available shift options and other settings based on the context.

```
3. **Event Handling and Polling**: The component implements custom event handling, especially for save operations, and utilizes a polling mechanism to manage autosave functionality.
4. **Debugging Support**: Extensive support for debugging is provided, making it easier to trace issues during development or in production environments with debug flags.
```

nuUtils

Description

This Salesforce Lightning component is designed to interface with various Apex controllers and utilities for fetching and processing large amounts of related information from Salesforce. It is primarily focused on dashboard functionality, allowing users to interact with job orders, notifications, contacts, accounts, and more. The code dynamically generates dashboard tiles, and integrates style settings based on preferences or defaults.

Methods

1. **fireCustomEvent, fireCustomLocalEvent, fireToast:** These methods are used to dispatch custom events within the Salesforce platform. `fireCustomEvent` and `fireCustomLocalEvent` allow bubbling to vary, while `fireToast` is specifically for showing toast messages with customizable titles, messages, and variants.

   ```
   fireCustomEvent(component, customEventName, eventValue)
   fireCustomLocalEvent(component, customEventName, eventValue)
   fireToast(component, title, message, type)
   ```

2. **parseBool, isStringMissing, isMobile, getUrlParameter:** Utility methods for parsing boolean values from various types, checking if a string is missing, detecting if the device is mobile, and fetching URL parameters respectively.

   ```
   parseBool(value)
   isStringMissing(val)
   isMobile()
   getUrlParameter(name)
   ```

3. **round, logIt, debugIt, elapsedSeconds, deepCopy, getEventStyle, formatPhoneNumber, fetchContactAndRelatedData, getPageStyles, getISODate, getMidnightISO, getLocalISO,**

getWeekDay, buildDashboard: These methods serve various purposes ranging from rounding numbers, logging, deep copying objects, formatting phone numbers, fetching related data for contacts, getting page styles, formatting dates, and building the dynamic dashboard based on settings.

```
round(value, decimals)
logIt(name, obj)
debugIt(target, debugVar)
elapsedSeconds(startTime)
deepCopy(inObject)
getEventStyle(eventType)
formatPhoneNumber(phone)
fetchContactAndRelatedData()
getPageStyles(accountRecordType)
```

Properties

- **API_NAMES:** This is an external reference to constants that map API field names, used throughout the component to access specific fields dynamically.

Use Case

This component is designed to be used in a customized Salesforce Lightning dashboard. It fetches and displays information related to job orders, contacts, accounts, and other entities based on the logged-in user's roles and permissions. The component also personalizes the dashboard with styling settings and dynamically generated content including notifications and actionable items.

Important Notes

- The component relies heavily on promises and asynchronous Apex calls to fetch data, which means it handles a considerable amount of information and may need to be adjusted based on the specific Salesforce org's limits and performance considerations.
- Style customization is supported, and component behavior changes based on the account's RecordType and other settings. This dynamic nature should be thoroughly tested to ensure all users see the correct data and style.
- The large number of imported Apex methods indicates that the backend logic is complex, and any changes to these methods should be carefully coordinated with updates to this component.

nuWorklogsRelatedList

Description

This Lightning Web Component (LWC) named `NuWorklogsRelatedList` is designed to display related work records in a datatable format within Salesforce. It dynamically retrieves worklogs either from an agency or a company based on what's relevant to the provided invoice ID. The component utilizes the Salesforce Lightning Design System (SLDS) for consistent styling and the `lightning-datatable` component for displaying the records in a table format. It also incorporates navigation capabilities to generate URLs for individual worklog records.

Methods

- **haddlePageRef(result):** This wired method handles the current page reference. It uses the `NavigationMixin.GenerateUrl` to generate base URLs for worklog and assignment links, adjusting them to point directly to specific records. Based on the `invoiceId`, it makes calls to either `getInvoiceRelatedCompanyWorklogs` or `getInvoiceRelatedAgencyWorklogs` Apex methods to fetch the worklogs. It then formats the worklogs data to ensure that the `lightning-datatable` can display it properly, including creating URLs for navigation to the worklog and assignment records.
- **connectedCallback():** This lifecycle hook runs when the component is inserted into the DOM. In the provided code, this method does not perform any action, as its body is empty.

Properties

- **invoiceId:** (API property) This public property is intended to be set by the parent component to specify the invoice ID relevant to fetching the worklogs.
- **columns:** This property holds the configurations for the columns displayed in the `lightning-datatable`, including setting up fields for 'Worklog Name', 'Status', 'Week Start', 'Week End', and 'Assignment', with 'Worklog Name' and 'Assignment' as navigable URLs to the respective records.
- **worklogs:** (track) This reactive property stores the formatted worklogs fetched based on the provided `invoiceId`. The component's template uses this property to render the datatable based on current data.

Use Case

This component can be used in Salesforce applications where there's a need to display related work records tied to a particular invoice in a list format. It's particularly useful in situations where easy navigation to individual worklog or assignment records is needed directly from the list.

Important Notes

- Ensure that the `@salesforce/apex/nuController_RelatedLists.getInvoiceRelatedAgencyWorklogs` and `@salesforce/apex/nuController_RelatedLists.getInvoiceRelatedCompanyWorklogs` Apex methods are properly implemented and accessible by this component. These methods should return lists of worklog records with at least the fields required by the component (e.g., `timesheetId`, `placementId`, etc.).
- Customize the `COLUMNS` constant as necessary to match the actual fields present in your worklog records and to meet any specific display requirements you have for the datatable.
- The component uses a combination of SLDS classes and custom CSS for styling. Adjustments to the CSS should be made with consideration to maintain consistency with the SLDS guidelines.
- The empty `connectedCallback()` method indicates that lifecycle actions might be planned for future enhancements. Implement logic within this method if you need the component to perform actions upon being added to the DOM.

```
import { LightningElement, api, wire, track } from 'lwc';
// Import other necessary modules and Apex methods
```

This provides a basic structured way to understand the implementation and functioning of the `NuWorklogsRelatedList` Lightning Web Component.

nuWorkRecordList

Description

This Salesforce Lightning Web Component (LWC) named `NuWorkRecordList` is designed to manage and display work records for contacts associated with the user. It dynamically showcases records in two modes based on the selected `statusFilter`: *Unsubmitted* and *Rejected*. The component allows users to submit unsubmitted timesheets and view detailed reasons for any rejections. It employs Salesforce Apex to fetch timesheet data and leverages the

Lightning Design System (LDS) for styling. The dynamic theming based on account data provides a customized user experience.

Methods

populateContactId

Fetches the contact ID associated with the current user.

```
@wire(fetchContactId)
populateContactId({error,data}){ ... };
```

populateAccountData

Fetches account data by the contact ID and dynamically sets the component's styling based on the account's theme.

```
@wire(fetchAccountByContactId,{contactId : "$contactId"})
populateAccountData({error, data}){ ... };
```

fetchTimesheets

Fetches timesheets for the contact based on the `statusFilter`. It filters timesheets for either 'Unsubmitted' or 'Rejected' statuses and prepares them for display.

```
fetchTimeSheets() { ... };
```

doViewTs

Handles navigation to the view page of the selected timesheet.

```
doViewTs(event) { ... };
```

doSubmitTimesheet

Submits the selected timesheet for processing and updates its status to 'Submitted'.

```
doSubmitTimesheet(event) { ... };
```

Properties

- `@api statusFilter`: External filter status that determines which timesheets to display (either 'Unsubmitted' or 'Rejected').
- `@track timesheets`: The list of timesheets fetched based on the contact and status filter.
- `@track recordCount`: The count of timesheets currently available to display.
- `contactId`: Stores the contact ID associated with the current user.
- `accountId`: Stores the account ID associated with the contact.
- `@track containerStyle`: Dynamic styling applied to the component container based on account data.
- `@track sectionHeadingStyle`: Dynamic styling applied to section headings within the component based on account data.
- `showSpinner`: Controls the visibility of the loading spinner.

Use Case

- **For a Recruiting Agency:** This component can be used by recruitment or staffing agencies on their Salesforce platform to manage work records for their contractors or employees. It provides an efficient way to monitor the submission and rejection of timesheets, facilitating timely payroll processing and dispute resolution.

HTML Structure

The HTML template dynamically displays a loading spinner, a header with the total number of items, and a table containing timesheet data. It conditions rendering elements based on `isSubmitMode` and `statusFilter`.

Important Notes

- The component uses the `NavigationMixin` for navigating to record view pages, making it compatible with Salesforce's Single Page Application (SPA) architecture.
- The dynamic styling (`containerStyle` and `sectionHeadingStyle`) relies on custom fields `Use_Portal_Theme__c`, `Tertiary_Color__c`, `Secondary_Color__c`, and `Primary_Text_Color__c` from the account object, which must exist and be populated for the styling to apply.
- Error handling and logging are implemented for Apex calls and update operations, ensuring easier debugging and user feedback.
- The component employs Salesforce's Lightning Data Service (LDS) functionalities (`updateRecord`) for performing CRUD operations directly from the client-side without

custom Apex controllers for updating records, enhancing performance and leveraging Salesforce's built-in security features.

- The CSS file imports styles from a custom static resource which allows for additional customization and theming beyond what is dynamically set from the account settings.

nuWorkRecordsAndExpenses

Description

The Salesforce Lightning Web Component (LWC) named `NuWorkRecordsAndExpenses` is designed to manage and display various tabs related to work records and expenses. It displays different aspects of timesheets and expenses, segmenting timesheets into categories like Action Required, Submitted, and Approved, while also offering a view for managing expenses. The component makes use of child components to present detailed views and functionalities for each tab, showcasing a structured and organized approach to handling work records and expenses within a Salesforce organization.

Methods

1. **connectedCallback**: This lifecycle hook is used for performing operations after the component gets inserted into the DOM. It initializes variables like `startDate`, `endDate`, `defaultTab`, and `childTab`, while leveraging `localStorage` to remember user's tab preferences.

Properties

- **@api Properties**
  - **testContactId**: Id of the contact to test the component with, if necessary.
  - **renderMode, positionLabel, weekStartLabel, weekEndLabel, totalHoursLabel, guaranteedHoursLabel**: Configurable labels and modes for rendering timesheet information.
  - **fileAttachmentsPageName**: Specifies the page name for navigating to file attachments.
  - **title**: Title of the timesheets section.
  - **statusFilter, showFiles, showNote, showShift, showPosition, showWeekStart, showWeekEnd, clockInClockOut, showUnpaidBreak, showTotalHours,**

showGuaranteedHours, showWorkLocation: Boolean flags and filters for controlling what elements to display on the UI.
    - **workLocationLabel, autoBreakValue, mode**: Additional configurations regarding work location label, automatic break value calculation, and component mode.
    - **unsubmittedTimesheets**: Array to keep track of unsubmitted timesheets.
- **@track Properties**
    - **contactId**: Stores the contact ID resolved for the current user.
    - **defaultTab, childTab**: Dynamically managed properties to control active tab based on user interaction or saved states.

Use Case

This component could be used in a Salesforce application where managing and monitoring employee timesheets and expenses is crucial. It allows employees to navigate through their work records categorically, submit their timesheets, and manage expense records, all in one place. HR personnel or managers could use this component to overview and approve timesheets or to review expense submissions efficiently.

Important Notes

1. **Lightning Data Service (LDS) and Apex**: The component utilizes LDS and Apex calls to fetch relevant data, such as the contact ID associated with the current user and timesheets for the employee. It's important to handle these calls with care to ensure data security and performance.
2. **Front-End Handling**: It heavily relies on front-end logic to dynamically switch between tabs and manage the state of the displayed content. This approach necessitates careful state management and UI responsiveness concerns.
3. **Accessibility and Internationalization**: While the code presents basic functionalities, attention should be given to enhancing accessibility features and internationalization, ensuring the component can be used effectively by a global audience across different locales.
4. **Caching and Performance Optimization**: The component makes use of `localStorage` to remember user preferences for tabs. Developers should consider caching strategies and performance optimization, especially when dealing with large sets of timesheet data or complex user interactions.
5. **Security Considerations**: Ensure that the component adheres to Salesforce's security best practices, especially in relation to exposing sensitive employee data like timesheets and

expenses. Use of field-level security, object-level security, and sharing rules should be meticulously planned.

```
// Key snippet illustrating usage of @wire service to fetch data based
on dynamic properties
@wire(fetchTimesheetsForEmployee,{ contactId: '$contactId',
statusFilters: '$unsubfilters', startDate: '$startDate', endDate:
'$endDate' })
wiredTimesheets(wireResult) { /* ... */ }
```

This component exemplifies a complex integration of Salesforce LWC with Apex backed services for handling real-world HR and employee management workflows within Salesforce.

orderApprovalActionPanel

Description

The `OrderApprovalActionPanel` component provides a user interface for approving or rejecting job orders within the Salesforce Lightning Experience. It leverages the Salesforce Lightning Design System (SLDS) for styling, making it visually consistent with the Salesforce environment. It has a combination of buttons, a combobox for selection, and textarea fields to capture user inputs and comments. When certain criteria are met, it allows users to approve or reject job orders, providing an option to input comments for both actions. This component also displays a modal window for rejection comments.

Methods

1. **approveJobOrderAsync(recordId)**: Asynchronously approves a job order by calling the `approveJobOrder` Apex method. It displays a success or error toast message based on the action's result.

```
async approveJobOrderAsync(recordId) { /* implementation */ }
```

2. **rejectJobOrderAsync(recordId, rejectComment)**: Asynchronously rejects a job order by calling the `rejectJobOrder` Apex method. It too, displays a success or error toast message based on the action's result.

```
async rejectJobOrderAsync(recordId, rejectComment) { /* implementation
*/ }
```

Properties

1. **recordId (@api)**: The Salesforce record ID of the job order being processed.
2. **comment (@track)**: The approval comment typed by the user.
3. **rejectComment (@track)**: The rejection comment typed by the user in the modal window.
4. **openModal (@track)**: A boolean flag that controls the visibility of the rejection comment modal window.
5. **canApprove (@track)**: A boolean flag indicating if the user has the permission to approve or reject the job order.
6. **availableTo (@track)**: An array of options for the 'Select Availability' combobox, populated based on Salesforce Picklist values.
7. **selectedOption (@track)**: Stores the currently selected option from the 'Select Availability' combobox. Defaults to "Internal".

Use Case

This component could be used in a Salesforce Lightning record page specifically designed for job order management. It provides a convenient way for users to approve or reject job orders based on the availability and other criteria direct from within the record's UI.

Important Notes

- This component relies on custom Apex methods (`canApprove`, `approveJobOrder`, `rejectJobOrder`) and utility methods (like `fireToast`), which need to be defined in your Salesforce Org.
- Proper permissions should be configured to ensure the correct visibility of the component based on the user's role or profile.
- The component uses a combination of Lightning Base Components (`lightning-button`, `lightning-combobox`, `lightning-textarea`, etc.) and SLDS classes for styling and structure, ensuring a cohesive look and feel within the Salesforce environment.
- Error handling is crucial for both Apex calls and UI feedback. Make sure to thoroughly test for edge cases and unexpected behaviors.

ordersList

Description

This Salesforce Lightning Web Component (LWC) `OrdersList` is designed to display a filtered list of job orders based on various criteria like status, work location, specialty, and priority. It employs a combination of imperative Apex methods to fetch data, child components for multi-select picklists, and lightning elements for data input and display. The component showcases dynamic styling based on fetched organization theme settings and provides interactive sorting and filtering functionalities on the displayed job orders list.

Properties

- `@api testAccountId`: External Account ID that can be passed to the component for testing or specific data fetching.
- `@api status`: The initial status filter to apply to the job orders fetched.
- `@api title`: The title to display on the component, typically used to denote the type of data being presented.

Methods

1. `connectedCallback()`: Lifecycle method that runs when the component is inserted into the DOM. It initializes data fetching operations and sets initial states.
2. `renderedCallback()`: Lifecycle method that runs after every render of the component. It is used for post-render operations or initialization that requires the DOM to be present.
3. `loadMoreData(event)`: Method attached to an infinite loading mechanism for the data table. It fetches additional data when the user scrolls to the bottom of the table.
4. `updateColumnSorting(event)`: Handles the logic for sorting data based on the selected column in the datatable.
5. `sortBy(field, reverse, primer)`: Utility method that returns a comparator function based on the field to sort by, whether it should be reversed and an optional primer function for preprocessing values.
6. `handleNameFilterChange(event)`, `handleStatusFilterChange(event)`, `handleSpecialtyFilterChange(event)`, `handlePriorityFilterChange(event)`, `handleCompanyFilterChange(event)`: Event handlers for filtering based on various fields. They update the internal state and trigger recalculation of the filtered dataset.
7. `doFilter()`: Applies all active filters to the job orders and updates the displayed data.

Use Case

This component can be used in a Salesforce community or as part of an application where job orders need to be displayed, sorted, and filtered dynamically based on various criteria. It shows how to dynamically apply CSS styles, handle sorting and pagination in LWC, and utilize custom child components for enhanced input mechanisms like multi-select picklists.

Important Notes

- The component makes extensive use of Salesforce Apex to fetch and manipulate data, tied closely to the schema of the Salesforce org it's deployed in. As such, the Apex class names and methods should exist and be accessible from the component.
- Custom child components (`c-nu-searchable-multi-picklist`) are used for multi-select filtering options. These components need to be defined and made compatible with the main component for it to function correctly.
- It leverages CSS from an external source or defined in another component (`c/nuCSS`) and conditionally applies styles based on the fetched account settings.
- The component also demonstrates a pattern for integrating Lightning Data Table with custom filtering and sorting functionalities that are not out-of-the-box features of the component.
- Data fetching and operations in `connectedCallback()` could potentially lead to lengthy operations and should be optimized or chunked for performance improvements, especially in large datasets. Error handling in Apex calls is present but minimal; for production use, consider adding more comprehensive feedback or retry mechanisms.

ordersToApprove

Description

This Lightning Web Component (LWC) named `OrdersToApprove` is designed for Salesforce Lightning Experience. The component retrieves and displays a list of job orders that require approval. It shows these orders in a `lightning-datatable` with pagination, using an infinite loading strategy to load additional data on demand. The component also demonstrates the use of conditional rendering, dynamic styles based on fetched data, Apex methods to fetch data from Salesforce, and integration with custom styles and utilities.

Methods

1. **connectedCallback()**

- Triggered when the component is inserted into the DOM. It initiates the fetching of contactId, accountId, and related theme styles or custom styles. It also starts the process to fetch the initial set of job orders to approve.

2. **renderedCallback()**
   - Called after every render of the component. It ensures data loading continues as needed after the initial render.
3. **loadMoreData(event)**
   - Triggered by an event such as scrolling or explicitly by code attempting to load more orders. It fetches the next set of job orders based on the current offset and updates the data table.

Properties

1. **@api testOriginalActorId**
   - Optional. Allows for testing the component with a specified Salesforce actor ID.
2. **@api testContactId**
   - Optional. Allows for setting a test contact ID, circumventing the initial fetch from the user context.
3. **@track contactId**
   - The ID of the contact linked to the current user.
4. **@track accountId**
   - The ID of the account associated with the contact.
5. **@track wiresLoaded**
   - A flag indicating whether initial data has been loaded.
6. **columns**
   - Defines the structure and type of columns shown in the `lightning-datatable`.
7. **ordersToApprove**
   - Holds the set of job orders loaded for approval.
8. **recordCount**
   - Tracks the number of records currently displayed.
9. **totalNumberOfRows**
   - The total count of job orders available for approval.
10. **loadMoreStatus**
    - Textual status indicating the loading state of additional records.

Use Case

A typical use case for this component might be in a Salesforce org where job orders are managed and require approval processes. A user such as a manager or an admin can use this component to review, sort, and approve job orders directly from a custom Salesforce page or application. The component provides an efficient way to manage large sets of data by loading them incrementally as the user scrolls through the list.

Important Notes

- Ensure that the Apex classes `nuService_Contact` and `nuService_nuJobOrder` are accessible and have the appropriate `@AuraEnabled` methods needed by this component.
- The datatable uses `enable-infinite-loading` and a custom offset management strategy to handle large sets of data without loading everything at once, improving performance.
- Dynamic styling based on the account's configuration or default styles demonstrates how to tailor the UI programmatically.
- This component relies on external utilities and styles (e.g., `getPageStyles` from `c/nuUtils`, and styles from `c/nuCSS`) which need to be defined and available in your org.

poolingCandidateList

Description

The `PoolingCandidateList` component is designed for Salesforce Lightning Web Component (LWC) framework. It displays a list of candidate pools related to a specific account and provides the capability to create new candidate pools. It features dynamic styling, infinite scrolling for data tables, and conditional rendering based on the component's state.

Methods

- **handleEnableContactCreation**: This method toggles the flag `enableContactPollsCreation` to show or hide the form for creating new contact polls.
- **connectedCallback**: Lifecycle hook which fires when the component is inserted into the DOM. It initializes component data by fetching the associated account ID, updating styles based on account information, and loading initial candidate pool data.
- **renderedCallback**: Lifecycle hook which fires after every render of the component. It ensures the initial loading of more data into the data table.

- **loadMoreData**: This method is invoked to load more data into the data table as the user scrolls, implementing an infinite scrolling functionality.

Properties

- **testAccountId (@api)**: An API property that allows you to set an Account ID for testing purposes.
- **title (@api)**: An API property to set the title displayed on the component.
- **error**: To store error messages from server-side calls.
- **columns**: Defines the columns for the `lightning-datatable`.
- **candidatData**: An array to store the data fetched for the candidate pools.
- **recordCount**: Keeps track of the number of records displayed in the datatable.
- **initialized**: A flag to ensure certain actions are only performed once after the component is rendered.
- **wiresLoaded (@track)**: A tracked property to manage the rendering flow based on whether the asynchronous operations have completed.
- **loadMoreStatus**: Indicates the current status of loading more data into the datatable (e.g., "Loading" or "No more data to load").
- **totalNumberOfRows**: Stores the total number of candidate pool records available.
- **enableContactPollsCreation**: A boolean flag to show or hide the form for creating new candidate pools based on user interaction.

Use Case

This component is ideal for Salesforce Communities or Lightning Applications where managing and creating account-related candidate pools is required. It can be used on account detail pages or as part of a recruitment module within a Salesforce application to streamline the process of managing candidate pools.

Important Notes

- Before deploying this component, make sure the Apex classes (`nuService_Contact` and `nuService_nuJobOrder`) and their respective methods (`fetchContactIdFromUser`, `fetchAccountByContactId`, `fetchAccountCandidatePool`, `countAccountCandidatePool`) are available and properly configured in your Salesforce org. These classes are responsible for fetching necessary data from Salesforce.

- The component makes use of the `@salesforce/community/basePath` module to construct URLs for candidate pool details. Ensure that your Salesforce community is correctly configured for this component to function as expected.
- Dynamic styling based on account information (e.g., `Tertiary_Color__c`, `Secondary_Color__c`) suggests the component is designed to adapt visually depending on specific account settings. Ensure these fields are available and populated on the Account object in your org.
- The functionality to load more data as the user scrolls (infinite scrolling) requires proper indexing and performance tuning on the Salesforce side to ensure a smooth user experience, especially for large datasets.
- Custom CSS is imported from 'c/nuCSS', and utility methods from 'c/nuUtils'. Ensure these resources exist and are accessible in your Salesforce org.

providerLookup

Description

This Salesforce Lightning Web Component (LWC) named `CustomLookup` is designed as a custom lookup component, providing a dynamic and reusable search capability. It allows users to search for records by typing in a search key, displaying matching results in a dropdown list, from which users can select. Once a selection is made, the component displays the chosen record in a pill format, with the option to clear the selection. The component is styled using Salesforce Lightning Design System (SLDS) classes for a consistent look and feel with the Salesforce experience.

Methods

- **connectedCallback()**: Initializes the component by fetching a default record if `defaultRecordId` is provided.
- **handleKeyChange(event)**: Triggers when the user types in the search input. It performs a debounced search operation and fetches matching records based on the input.
- **toggleResult(event)**: Toggles the visibility of the search results dropdown based on user interaction (e.g., clicking outside the component).
- **handleRemove()**: Clears the currently selected record and resets the component to its initial state, showing the search input.
- **handelSelectedRecord(event)**: Handles the selection of a record from the search results, updating the component to display the selected record as a pill.

- **handelSelectRecordHelper()**: A helper method used to switch the UI view from the search input to the selected record display.
- **lookupUpdatehandler(value)**: Fires a custom event named 'lookupupdate' with the detail of the selected record, allowing parent components to react to the selection.

Properties

- **@api label**: Public property for setting the label of the lookup field.
- **@api placeholder**: Public property for setting the placeholder text of the search input. Default value is 'search...'.
- **@api iconName**: Public property to specify the icon displayed in the search results and selected record pill. Default is 'standard:account'.
- **@api [location, providerType, specialty, agency]**: Public properties to capture additional filter criteria for the search operation.
- **@api defaultRecordId**: Public property for setting a default record ID to pre-populate the component.
- **@api recordType**: Public property to specify the type of records to search for.
- **@api hideClearButton**: Public property to control the visibility of the clear button.
- **@api selectedId**: Public property to hold the ID of the selected record.

Use Case

This component is ideal for scenarios where users need to select a specific record from a potentially large set of data, such as choosing a contact, account, or any custom object record. It can be particularly useful in forms where a lookup to related records is required but with a customizable UI and functionality beyond what is provided by the standard Salesforce lookup fields.

Important Notes

1. **Async Operations**: The component makes asynchronous calls to Apex methods to fetch data. It's crucial to handle these operations properly to ensure the component remains responsive and the UI is updated correctly.
2. **Debouncing**: To optimize performance and reduce the number of server calls, search input handling is debounced. This means the component waits for a fixed amount of time (defined by `DELAY`) after the user stops typing before performing the search operation.

3. **Error Handling**: While this code snippet does not explicitly include error handling logic for apex calls, it is important to implement error handling to provide feedback to the user in the event of an issue.
4. **Custom Events**: The component dispatches a `lookupupdate` custom event when a record is selected, allowing for easy integration and interaction with parent components.

```
this.dispatchEvent(new CustomEvent('lookupupdate', { 'detail':
{selectedRecord: value} }));
```

5. **Security**: Ensure that the Apex methods called by this component enforce proper sharing and security checks to prevent unauthorized data access.

pubsub

Description

This JavaScript module implements a basic publish-subscribe (pub-sub) mechanism focused on sibling component communication within Salesforce Lightning components. Its primary function is to allow for the decoupled interaction between components, where events can be published by one component and subscribed to by another, without requiring a direct relationship between them. This approach enhances the modularity and reusability of components by adhering to a loosely coupled architecture.

Methods

`registerListener`

Registers a callback function to be invoked when a specific event is published.

```
registerListener(eventName, callback, thisArg)
```

- **Parameters:**
  - `eventName`: The name of the event to listen to.
  - `callback`: The function to be called when the event occurs.
  - `thisArg`: The context in which to invoke the callback.

`unregisterListener`

Unregisters a callback function, removing it from the list of listeners for a specific event.

```
unregisterListener(eventName, callback, thisArg)
```

- **Parameters:**
  - `eventName`: The name of the event to stop listening to.
  - `callback`: The function to be removed from the list of listener callbacks.
  - `thisArg`: The context in which the callback was to be invoked.

`unregisterAllListeners`

Unregisters all event listeners associated with a given context.

```
unregisterAllListeners(thisArg)
```

- **Parameters:**
  - `thisArg`: The context whose listeners should be removed.

`fireEvent`

Publishes an event, invoking all registered callbacks for that event.

```
fireEvent(pageRef, eventName, payload)
```

- **Parameters:**
  - `pageRef`: The page reference that indicates the scope of the event.
  - `eventName`: The name of the event to be published.
  - `payload`: The data to be passed to the listener callbacks.

Properties

While this module doesn't define properties in the traditional sense, it internally manages the `events` object, a dictionary where each key is an event name and its value is an array of listener objects. Each listener object contains a `callback` function and the `thisArg` context in which to call it.

Use Case

This pub-sub mechanism can be utilized in a Salesforce Lightning application where multiple components need to communicate or share data without being directly coupled. For example, it can be used in a dashboard where different components (e.g., charts, filters, lists) must react to changes made in one of the components (e.g., a filter change) without needing a direct reference to each other.

Important Notes

- Before using `registerListener`, ensure the component has a `@wire(CurrentPageReference) pageRef` property, as it's critical for filtering events to the correct listener scope.
- The `samePageRef` function ensures that only components on the same page reference react to the published event, preventing unintended cross-talk between pages or components that should not be listening to the event.

relatedFilesTable

Description

The `RelatedFilesTable` component, designed for Salesforce Lightning Experience, beautifully presents a table of related document files for a given record. It fetches related files based on specific criteria and allows users to review these documents by navigating to their URLs. This component elegantly encapsulates the functionality using Salesforce Lightning Web Component (LWC) standards, Apex controller calls for data retrieval, and utilizes Lightning Design System (LDS) for styling to ensure a consistent user experience that aligns with Salesforce's design aesthetics.

Methods

renderedCallback()

This lifecycle hook is invoked when the component has been inserted into the DOM and after every render of the component. It calls the `getRelatedFiles()` method to fetch the related files every time the component renders.

```
renderedCallback() {
    this.getRelatedFiles();
}
```

getRelatedFiles()

An `@api` decorated method, making it publicly available for invocation from other components or for compositional purposes. It calls the `getRelatedFiles` Apex method with parameters `recordId` and `withoutCv`. It updates the component's `data` property with the fetched results or logs an error in case of failure.

```
@api
getRelatedFiles() {
    getRelatedFiles({ recordId: this.recordId, withoutCv: this.withoutCv
})
    .then(result => {
        this.data = result;
    })
    .catch(error => {
        console.error('nuQuickActionPresentCandidate get files error',
JSON.stringify(error));
    })
}
```

previewHandler(event)

Handles clicks on the "Review" button within the data table. It uses the `NavigationMixin.Navigate` method to navigate the user to the document's URL specified in the button's `data-url` attribute.

```
previewHandler(event) {
    this[NavigationMixin.Navigate]({
        type:'standard__webPage',
        attributes:{
            url: event.target.dataset.url
        }
    })
}
```

Properties

- `recordId`: Decorated with `@api`, it allows the parent component to pass the record ID for which the related files are to be fetched.
- `withoutCv`: Another `@api` decorated property indicating whether to exclude certain files based on criteria (e.g., CV files).

- `data`: An array that holds the fetched related files data after it is retrieved from the Apex controller.

Use Case

This component is ideal for use cases where a Salesforce record is associated with various document files, and there's a need to display these files in a structured manner within the Lightning Experience. Specifically, it can serve as a component on a record page, showing all related documents that a user can review directly by clicking a button.

Important Notes

- Ensure the Apex class `nuController_RelatedLists` and its method `getRelatedFilesWithCriteria` are properly defined and set up with the required permissions. The Apex method should be capable of accepting `recordId` and `withoutCv` parameters and return the related files.
- Customize the component by adjusting CSS classes as per the Salesforce Lightning Design System (SLDS) to maintain design consistency across your Salesforce org.
- The component employs Lightning Data Service (LDS) and Navigation service. Make sure to test the navigation functionality thoroughly as URL redirection behavior might vary based on Salesforce's setup and version.
- As with any component that fetches data from the server, consider handling large data volumes and implement pagination or lazy loading if necessary to enhance performance.
- Review and adhere to Salesforce's security best practices, especially concerning Apex controller methods used in Lightning Web Components.

releaseTiersList

Description

The `OrdersList` component is a Salesforce Lightning Web Component (LWC) designed to display a list of "Release Tiers" related to a specific Account in a data table format. It dynamically fetches and displays data, including the capability to incrementally load more data as the user scrolls, leveraging Salesforce's `lightning-datatable` component. The component's appearance is customized based on specific Account settings or global themes.

Methods

connectedCallback()

The `connectedCallback` lifecycle hook is used to perform initial data fetching operations. This includes retrieving the Contact ID associated with the current user, fetching Account details based on this Contact ID, and then loading the initial batch of "Release Tiers" data and the total number of rows available. It conditions styling based on Account-specific settings or global themes.

```
connectedCallback() {
    // Implementation details...
}
```

renderedCallback()

The `renderedCallback` lifecycle hook is used to initiate the loading of more data once the component has rendered. This is particularly useful for initially populating the data table.

```
renderedCallback() {
    // Implementation details...
}
```

loadMoreData(event)

This method is triggered when more data needs to be loaded into the data table. It specifically handles the infinite loading capability by fetching the next batch of "Release Tiers" records and updating the UI accordingly.

```
loadMoreData(event) {
    // Implementation details...
}
```

Properties

- `@api testAccountId`: Externally set Account ID used for testing purposes.
- `@api title`: The title to be displayed above the data table.
- `@track accountId`: Stores the Account ID determined either from `testAccountId` or fetched based on the user's Contact ID.

- `@track wiresLoaded`: Boolean indicating whether the initial data loading operations have completed.
- `columns`: Defines the columns to be displayed in the data table.
- `releaseTiers`: An array that stores the fetched "Release Tiers" records to be displayed.
- `recordCount`: Keeps track of the number of records currently displayed to the user.
- `totalNumberOfRows=0`: Stores the total number of "Release Tier" rows that are available.
- `loadMoreStatus`: Indicates the current status of data loading operations, e.g., "No more data to load".

Use Case

The `OrdersList` component is intended for use in scenarios where there is a need to display a list of "Release Tiers" associated with an Account in a paginated, easily navigable format within a Salesforce Community or Lightning Experience. It's suitable for handling large datasets by loading additional records on-demand as the user scrolls through the data table.

Important Notes

- The component's styling and behavior are dynamically adjusted based on Account-specific settings or global themes, providing a customizable user experience.
- Data fetching is performed using Apex controller methods, making efficient use of server-side resources and reducing the amount of data transferred over the network.
- The component makes use of Salesforce's `lightning-datatable` for displaying data, leveraging its built-in functionality such as infinite loading and column sorting.
- Error handling is not explicitly detailed in the provided code excerpts, but it's important to robustly handle potential exceptions in both the Apex methods and the LWC to ensure a smooth user experience.

synchScheduler

Description

This Salesforce Lightning Web Component (LWC) named `SynchScheduler` is designed to create a complex scheduling and filtering system. The component features a spinner while loading data, a selection sidebar for filtering shifts based on type, location, candidate type, and specialty, a main content area that lists available shifts in a calendar format, and modals for

adding availability, claiming shifts, confirming shifts, and cancelling shifts. Additionally, it supports dynamic styling based on account-specific settings.

Methods

Public Methods

No public methods are exposed by this component.

Private Methods

- `handleShiftTypeListboxChange`: Invoked when the shift type selection changes.
- `handleLocationListboxChange`: Invoked when the location selection changes.
- `handleProviderTypeListboxChange`: Invoked when the provider type selection changes.
- `handleSpecialtyListboxChange`: Invoked when the specialty selection changes.
- `handleProviderListboxChange`: Invoked when the provider selection changes.
- `handleClearFiltersClick`: Clears all selected filters.
- `handleColorButtonClick`: Toggles the color state of the scheduler.
- `handlePreviousButtonClick`: Shifts the calendar view to the previous period.
- `handleNextButtonClick`: Shifts the calendar view to the next period.
- `handleRefreshButtonClick`: Refreshes the calendar based on current filters.
- `handleTodayButtonClick`: Sets the calendar view to today.
- `computeTotalHours`: Computes total hours between start and end times.
- `subtractDays`: Subtracts a specified number of days from a date.
- `setEventsMap`: Stores events in the component's internal state.
- `getEventsMap`: Retrieves events from the component's internal state based on a key.

Properties

- `selectedRateType`: Tracks the selected rate type.
- `selectedShiftType`: Tracks the selected shift type.
- `selectedLocation`: Tracks the selected location.
- `selectedLocations`: An array of selected locations.
- `selectedProviderType`: Tracks the selected provider type.
- `selectedSpecialty`: Tracks the selected specialty.
- `selectedProvider`: Tracks the selected provider.
- `selectedProviders`: An array of selected providers.

- `selectedEvents`: An array of selected events.
- `colorState`: Boolean indicating the state of the color toggle.
- `locationOptions`: Contains options for the location filter.
- `shiftTypeOptions`: Contains options for shift types.
- `providerTypeOptions`: Contains options for provider types.
- `specialtyOptions`: Contains options for specialties.
- `autoSelectRate`: Boolean to auto-select the rate.

Use Case

The component is designed to be used in scheduling scenarios where users need to filter and select shifts based on various criteria such as shift type, location, candidate type, and specialty. It provides a comprehensive scheduler interface for managing availability, claiming shifts, and handling other shift-related actions.

Important Notes

- It uses a combination of custom events and wire service to fetch data and respond to user interactions.
- The component's layout and appearance can be customized based on account-specific settings like theme color and text color.
- Dynamic imports are used for CSS and utility JavaScript modules to enhance performance and modularity.
- The component makes extensive use of SLDS (Salesforce Lightning Design System) for styling.
- Modals are used for critical actions such as adding availability, claiming, confirming, cancelling, and releasing shifts, enhancing the UX by providing contextual actions within the same interface.