Phase 6: User Interface Development

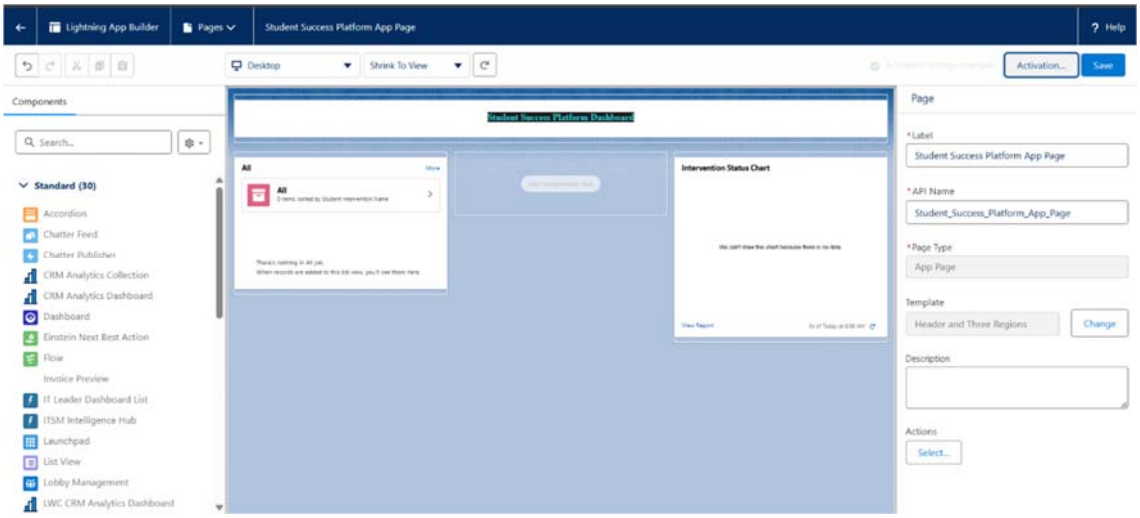
CONNECT Student Success Platform - Salesforce CRM Implementation

6.1 Lightning App Builder

Use Case: The Lightning App Builder was used to create a centralized Student Success Platform Dashboard. This dashboard provides quick access to Student Intervention records and visual status overviews, empowering users to track and manage interventions efficiently.

Implementation Summary:

- Created a new Lightning App Page named "Student Success Platform App Page" using the "Header and Three Regions" template.
- Header region displays the dashboard title using a Rich Text component.
- Left region contains a List View showing up to 5 Student Interventions for fast access.
- Right region displays a Report Chart visualizing Intervention Status statistics (based on the "Intervention Status Chart" report).
- Middle region is currently left empty (reserved for future custom components or infographics).
- The page was activated as org default for all users to make it easily accessible.

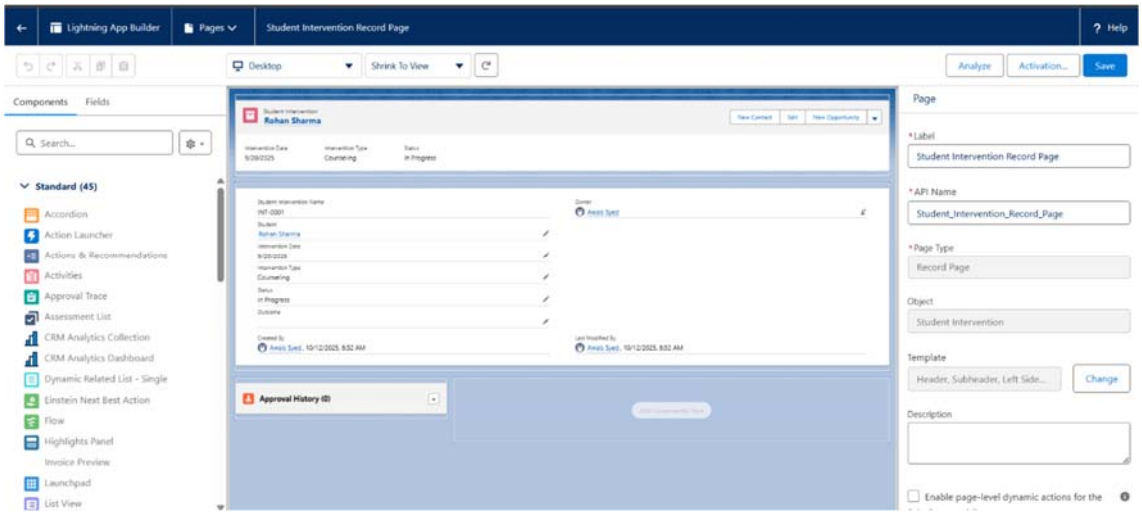


6.2 Record Pages

Use Case: A custom Lightning Record Page was designed for the "Student Intervention" object. This page provides a clean and focused view for users to manage and track individual Student Intervention details and related actions.

Implementation Summary:

- Created a new Lightning Record Page named "Student Intervention Record Page" using the "Header, Subheader, Left Sidebar" template.
- Top region displays a Highlights Panel to show key details and important actions related to the record.
- The main middle region contains the "Record Detail" component, displaying all Student Intervention fields for review and editing.
- Bottom left region is set with the "Approval History" component to show any approval actions, making workflow easier to manage.
- The right region is left empty for now, keeping the page layout simple and neat for users.
- The page was activated as the app default for the "Developer Edition" app, so all users see this design when viewing a Student Intervention record.

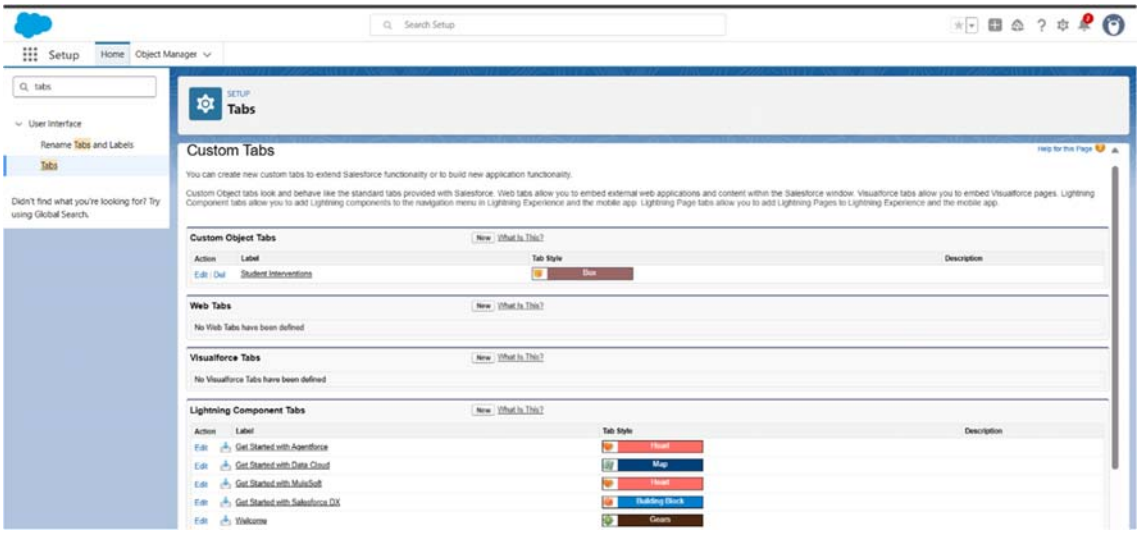


6.3 Tabs

Use Case: A custom tab was created for the Student Intervention object so users can easily find and open Student Intervention records from the Salesforce navigation bar.

Implementation Summary:

- Opened Salesforce Setup and navigated to Tabs (from the left Setup menu).
- Confirmed the existence of the "Student Interventions" tab under Custom Object Tabs, making the object accessible in the main app navigation.
- No new tab creation was needed because the tab already existed for the object.
- This tab allows quick user access to Student Intervention records directly from the Salesforce interface, supporting efficient workflow and faster record lookup.

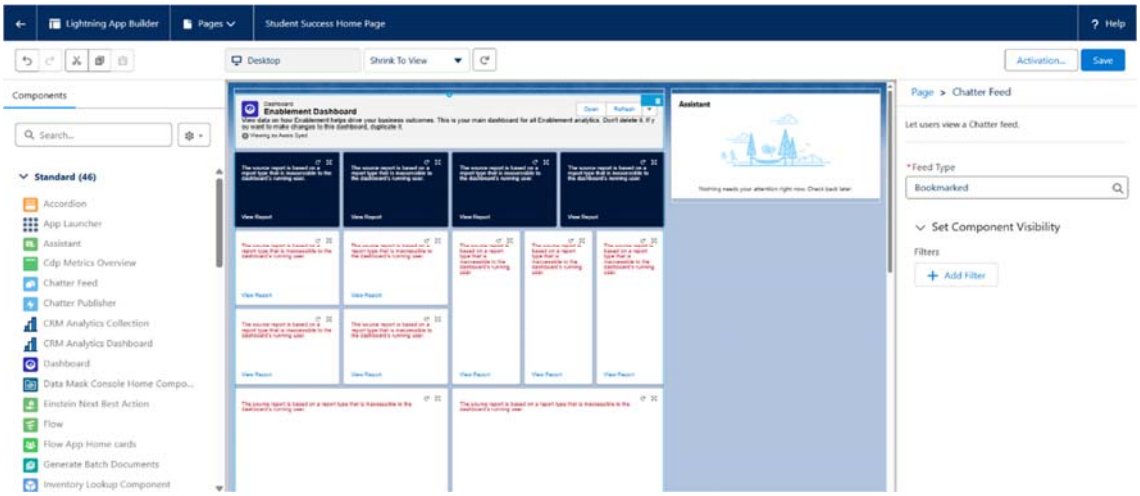


6.4 Home Page Layouts

Use Case: A custom Salesforce Home Page was created to help users quickly view the most important student success information and to assist them in taking fast action on urgent tasks as soon as they log in.

Implementation Summary:

- Opened Lightning App Builder and created a new Home Page named "Student Success Home Page" using the "Standard Home Page" template.
- The first top section was set to display a Dashboard for important statistics and visual insights.
- The second top section displays the Assistant component for reminders and suggestions.
- Bottom regions are used for components like Recent Items and (optionally) Chatter Feed or Report Chart, helping users find data and updates quickly.
- After layout design, the Home Page was saved and activated as the default for the main Salesforce app, ensuring all users see the new layout on login.



6.6. Lightning Web Components (LWC)

Use Case: A custom Lightning Web Component (LWC) was created to show a welcome message. This teaches the basics of creating, deploying, and displaying a new UI component using Salesforce's modern LWC framework.

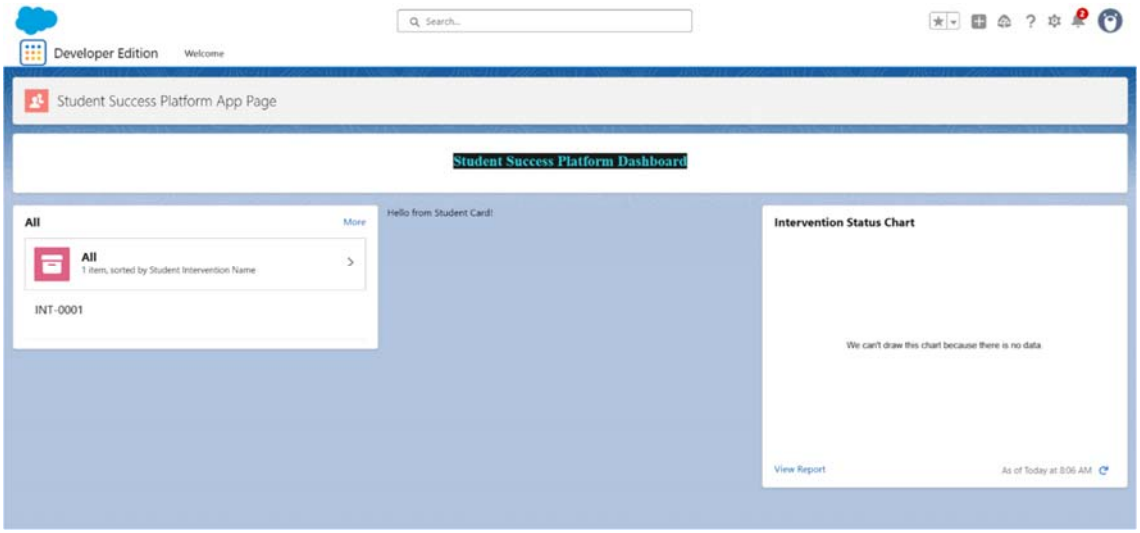
Implementation Steps:

- Opened **Visual Studio Code** and installed the Salesforce Extension Pack for LWC development.
- Used SFDX: Create Project and SFDX: Authorize an Org commands to connect to the Salesforce org from VS Code.
- Ran SFDX: Create Lightning Web Component and gave the component the name studentCard.
- Edited studentCard.html to display:

  <template>

  <h1>Hello from Student Card!</h1>

  </template>
- Set isExposed to "true" in studentCard.js-meta.xml and enabled the component for App, Record, and Home Pages.
- Deployed (SFDX: Deploy Source to Org) the new LWC to Salesforce.
- Opened the **Lightning App Builder** and selected an App Page with open sections.
- Dragged the studentCard LWC onto the empty (middle) region of the custom App Page, then saved and activated the page.
- Confirmed that the message, "Hello from Student Card!", appeared in Salesforce as expected.



6.7. Apex with LWC

Use Case: This step shows how to connect a Lightning Web Component (LWC) to a Salesforce Apex class, so LWC can display data or messages coming from Apex (the server).

Implementation Steps:

1. Create the Apex Class

- In VS Code, right-click the **classes** folder found under force-app/main/default.
- Select **SFDX: Create Apex Class** and name it StudentHelper.
- Code:

```
public with sharing class StudentHelper {
    @AuraEnabled(cacheable=true)
    public static String getWelcomeMessage() {
    return 'Welcome from Apex!';
    }}
```

- Save and deployed (right-click > "SFDX: Deploy Source to Org").

2. Add Apex Call in LWC JavaScript

- In studentCard.js, update code:

```
import { LightningElement, wire } from 'lwc';
import getWelcomeMessage from '@salesforce/apex/StudentHelper.getWelcomeMessage';
    export default class StudentCard extends LightningElement {
        welcomeMessage;
        @wire(getWelcomeMessage)
        wiredMsg({ data, error }) {
            if (data) {
            this.welcomeMessage = data;
            } else if (error) {
            this.welcomeMessage = 'Apex error';
                        } }}
```

- Save and deploy.

3. Change HTML to Display Apex Data

- In studentCard.html, use:

<template>

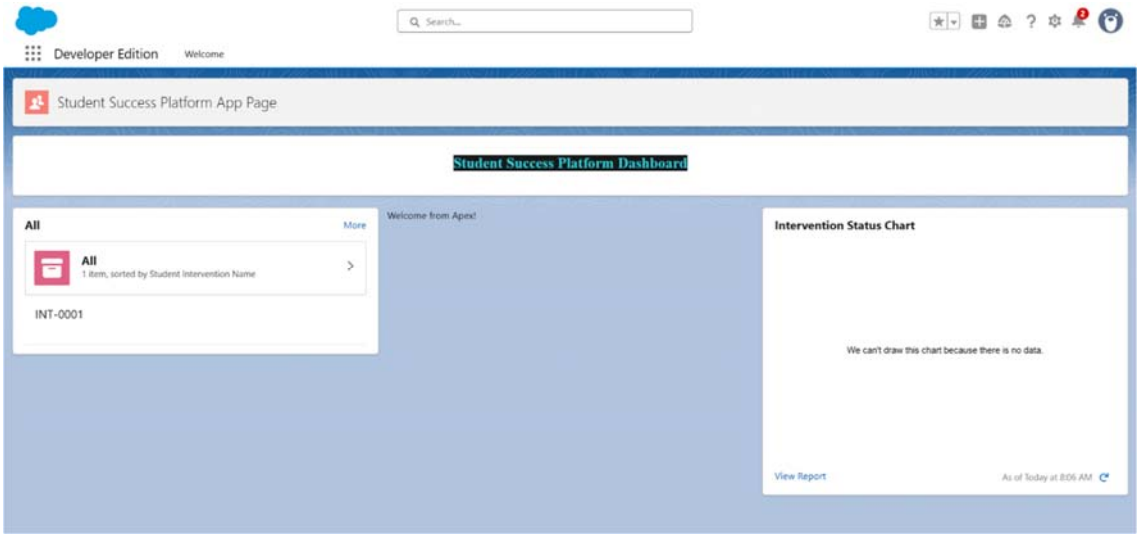
<h1>{welcomeMessage}</h1>

</template>

- Save and deploy.

4. Test and Validate

- Open the App Page in Salesforce.
- Confirm that the LWC displays **"Welcome from Apex!"** from your Apex class, proving that the connection works.



6.8. Events in LWC

Use Case: Show how Lightning Web Components handle user events (like button clicks) to update what the user sees on the page.

Implementation Steps:

1. **Edit LWC HTML to Add Button**
    - Open studentCard.html and update:

      <template>

        <h1>{welcomeMessage}</h1>

        <button onclick={handleButtonClick}>Click Me!</button>

        {buttonMessage}

      </template>

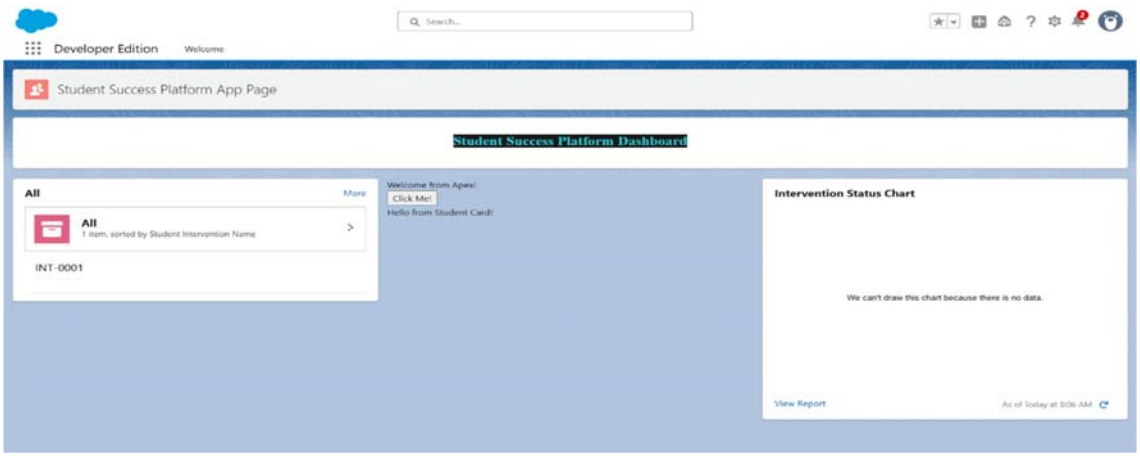
- This creates a button and a text area for feedback when the button is clicked.

2. **Add Event Handling in JS**
    - Open studentCard.js and update:

```
import { LightningElement, wire } from 'lwc';
import getWelcomeMessage from '@salesforce/apex/StudentHelper.getWelcomeMessage';
export default class StudentCard extends LightningElement {
    welcomeMessage;
    buttonMessage = '';
    @wire(getWelcomeMessage)
    wiredMsg({ data, error }) {
    if (data) {
    this.welcomeMessage = data;
    } else if (error) {
    this.welcomeMessage = 'Apex error';
        }}
    handleButtonClick() {
    this.buttonMessage = 'Hello from Student Card!';
        }}
```

    - This code sets a new message when the button is clicked.

3. **Deploy to Salesforce**
    - Right-click the studentCard folder in VS Code and choose **SFDX: Deploy Source to Org** to update the component.



6.9. Wire Adapters

Use Case: Wire adapters are used to pull live Salesforce data into Lightning Web Components without writing any Apex. This step makes the UI dynamic and personal by fetching information directly, such as the logged-in user's name.

Implementation Steps:

1. **Open and Edit studentCard.js**

- Add these imports at the top:

```
import { LightningElement, wire } from 'lwc';
import { getRecord } from 'lightning/uiRecordApi';
import USER_ID from '@salesforce/user/Id';
import getWelcomeMessage from '@salesforce/apex/StudentHelper.getWelcomeMessage
';
```

- Specify fields to retrieve:

```
const FIELDS = ['User.Name'];
```

- Update the class to use wire with the record API and add handle for button event:

```
export default class StudentCard extends LightningElement {
 welcomeMessage;
  buttonMessage = '';
   userName;
 @wire(getWelcomeMessage)
   wiredMsg({ data, error }) {
        if (data) {
        this.welcomeMessage = data;
          } else if (error) {
                this.welcomeMessage = 'Apex error';
        }}
  @wire(getRecord, { recordId: USER_ID, fields: FIELDS })
   userRecord({ error, data }) {
         if (data) {
         this.userName = data.fields.Name.value;
```

```
} else if (error) {
    this.userName = 'Error loading user';
  } }
handleButtonClick() {
  this.buttonMessage = 'Hello from Student Card!';
}}
```

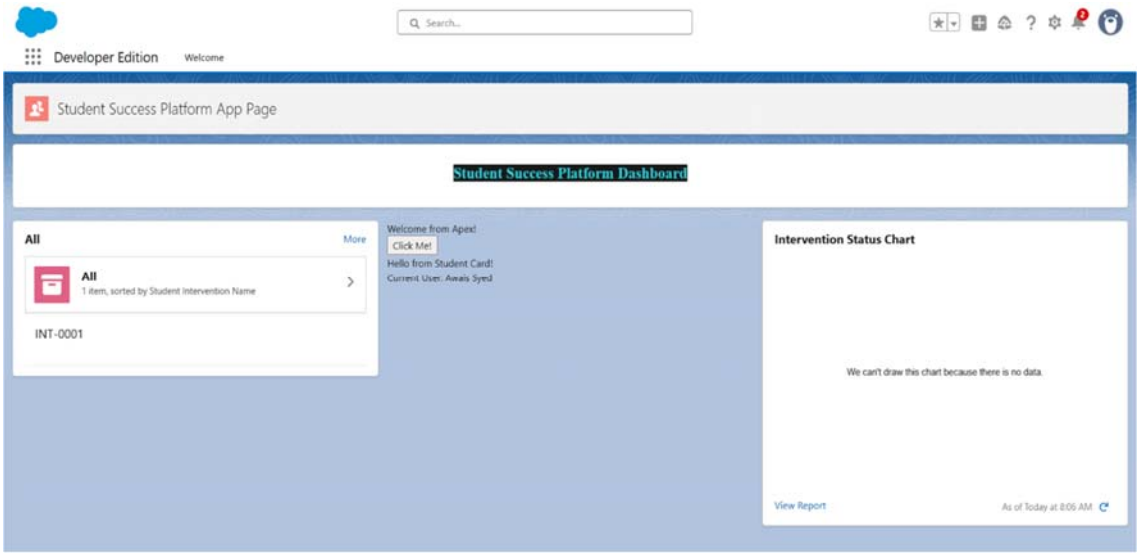
2. **Edit studentCard.html to Display Data**

- Use this HTML to show all values:

```
<template>
  <h1>{welcomeMessage}</h1>
  <button onclick={handleButtonClick}>Click Me!</button>
  <p>{buttonMessage}</p>
  <p>Current User: {userName}</p>
</template>
```

3. **Deploy and Test**

- Save and deploy your LWC to Salesforce.
- Refresh the App Page; you should see your welcome message, a button, the event message, and now the current user's name pulled live from Salesforce.



6.10. Imperative Apex Calls

Use Case: This step shows how to execute Apex code only when a button is clicked (instead of auto-refresh). Useful when you want to control exactly when server interactions happen.

Implementation Steps:

1. **Add to Apex Class:**

   @AuraEnabled

   public static String getHelloImperative() {

   return 'Hello from Imperative Apex!';

   }

2. **Add to LWC JS:**

import getHelloImperative **from** '@salesforce/apex/StudentHelper.getHelloImperative';

// in your export default class:

imperativeMessage = '';

async handleImperativeClick() {

try {

const result = **await** getHelloImperative();

this.imperativeMessage = result;

} **catch** (error) {

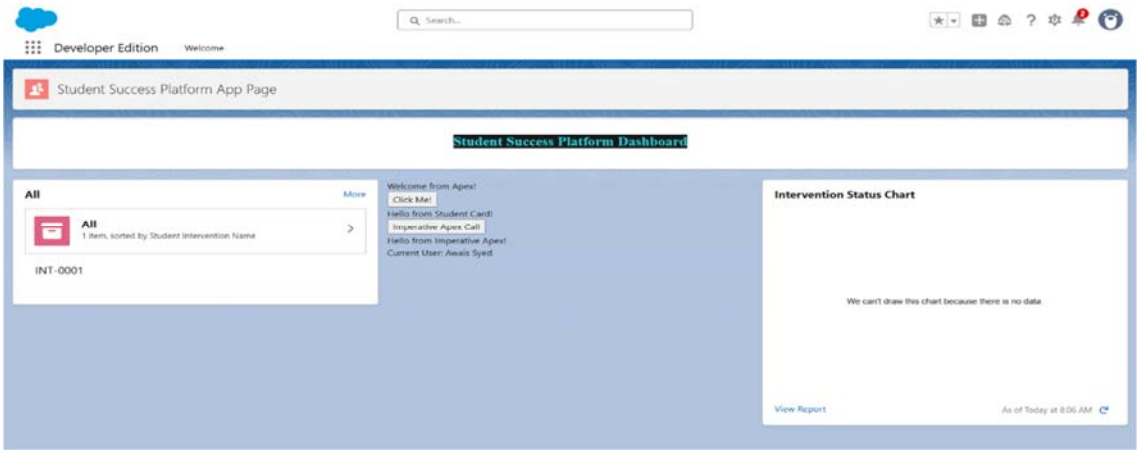
this.imperativeMessage = 'Error from imperative call';

}}

3. **Add to LWC HTML:**

<button onclick={handleImperativeClick}>Imperative Apex Call</button>

{imperativeMessage}



6.11. Navigation Service

Use Case: When the button is clicked, the user jumps straight to their User record, showing how to add navigation power to any LWC in Salesforce.

Implementation Steps:

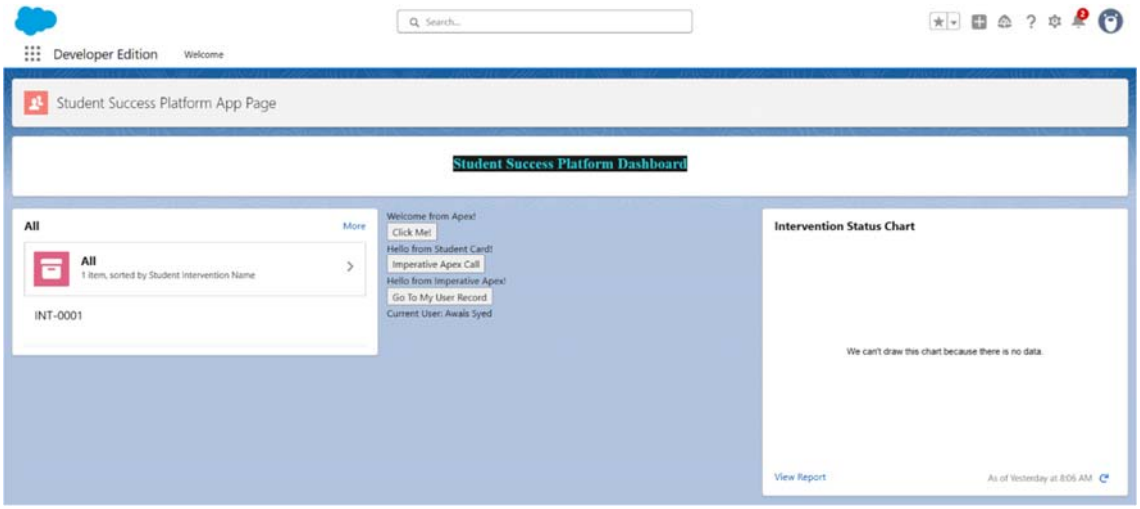
1. Add Navigation Service Code in JS:

```
import { NavigationMixin } from 'lightning/navigation';

// Inside your class...

handleNavigate() {

    this[NavigationMixin.Navigate]({

    type: 'standard__recordPage',

     attributes: {

    recordId: USER_ID,

    objectApiName: 'User',

    actionName: 'view'

  }

});

}
```

2. **Add Button for Navigation in HTML:**

```
<button onclick={handleNavigate}>Go To My User Record</button>
```

3. **Deploy and Test**
   Clicking the button sends the user to their User record in Salesforce, using the Navigation Service.



Naviagted to User record,