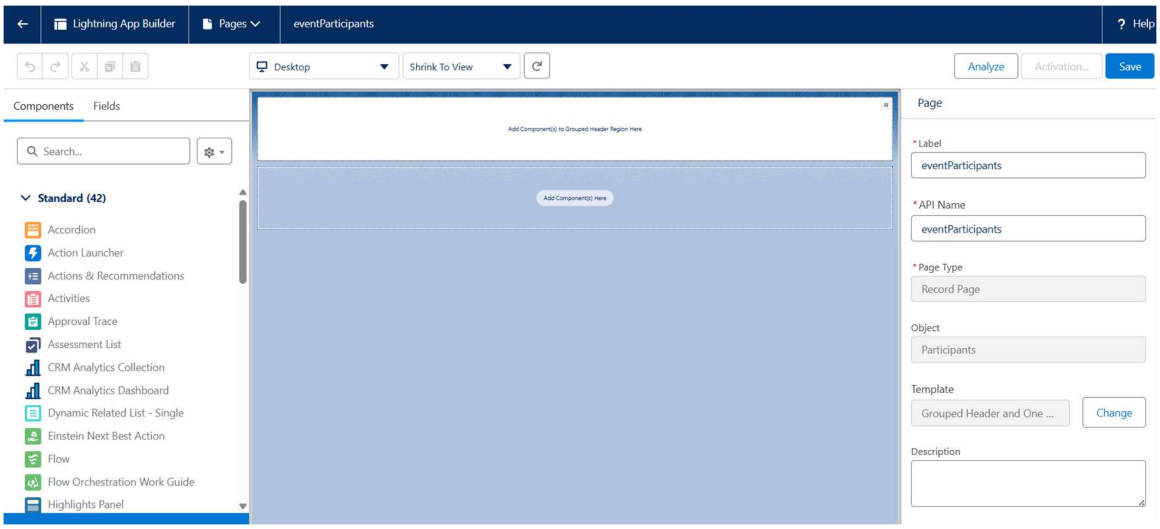
CAMPUS EVENT MANAGEMENT SYSTEM

NAME: SHAIK RAQUEEBUL ISLAM

Phase 6: User Interface Development

Stage 1: Lightning App Builder

- The Lightning App Builder was the core tool used to design the UI for our application. It provided a drag-and-drop interface to easily create pages for Events, Participants, and Feedback management. By combining standard Salesforce components with our custom LWCs, we were able to deliver a user-friendly and modular experience without complex coding.



Stage 2: Record Pages

- We customized record pages to make information more accessible and organized for end users. For Events, we used a tabbed layout to group details, participants, and feedback. For Participants, we added a sidebar LWC to display feedback directly on the record. These enhancements improved navigation and reduced the number of clicks required to view related data.

main > default > lwc > eventSearch > eventSearch.js-meta.xml

```xml
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>57.0</apiVersion>
  <isExposed>true</isExposed>
  <targets>
    <target>lightning__HomePage</target>
    <target>lightning__AppPage</target>
    <target>lightning__RecordPage</target>
  </targets>
  <targetConfigs>
    <targetConfig targets="lightning__RecordPage">
      <objects>
        <object>Event_Details__c</object>
      </objects>
    </targetConfig>
  </targetConfigs>
</LightningComponentBundle>
```

main > default > lwc > eventParticipants > eventParticipants.js > ...

```js
// force-app/main/default/lwc/eventParticipants/eventParticipants.js
import { LightningElement, api } from 'lwc';
import getParticipants from '@salesforce/apex/EventParticipantService.getParticipants';

export default class EventParticipants extends LightningElement {
  @api recordId;
  participants;

  connectedCallback(){
    this.loadParticipants();
  }

  loadParticipants(){
    getParticipants({ eventId: this.recordId })
      .then(res => this.participants = res)
      .catch(err => console.error(err));
  }
}
```

Stage 3: Home Page Layouts

- A custom home page named *Campus Events Home Page* was designed to act as the landing page for event managers and admins. It included the eventSearch LWC, allowing quick access to upcoming events. Assigning this home page to the app and key profiles made the system feel more personalized and tailored to its users.

main > default > classes > EventParticipantService.cls

```apex
public with sharing class EventParticipantService {
  @AuraEnabled(cacheable=true)
  public static List<Participant__c> getParticipants(Id eventId) {
    return [SELECT Id, Name, Participant_Email__c FROM Participant__c WHERE Event__c = :eventId ORDER BY CreatedDate DESC];
  }
}
```

Stage 4: Lightning Web Components (LWCs)

- LWCs were developed to extend standard Salesforce UI with dynamic functionality. Components like eventSearch, eventParticipants, and participantFeedback allowed us to provide features such as event searching, participant listing, and feedback collection. Each LWC was built with HTML for structure, JavaScript for logic, and metadata files to expose them to Salesforce.

main > default > lwc > eventSearch > eventSearch.html > ...

```
<template>
  <lightning-card title="Event Search">
    <div class="slds-p-around_medium">
      <lightning-input label="Search" value={searchKey} onchange={handleChange}></lightning-input>
      <lightning-button label="Search" onclick={handleSearch} class="slds-m-top_small"></lightning-button>
    </div>

    <template if:true={events}>
      <ul class="slds-p-horizontal_small">
        <template for:each={events} for:item="ev">
          <li key={ev.Id}>
            {ev.Event_Name__c} – {ev.Event_Date__c}
          </li>
        </template>
      </ul>
    </template>
  </lightning-card>
</template>
```

Stage 5: Apex with LWC

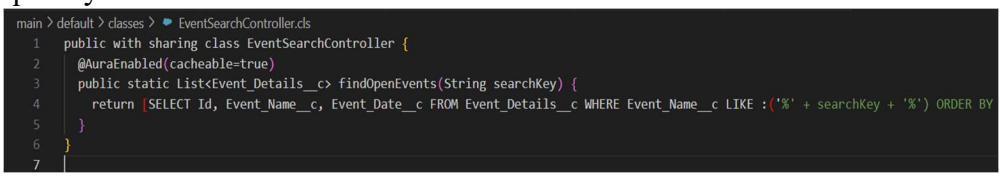
- To connect the UI with backend logic, we used Apex controller classes. These classes included methods to query open events, fetch participants, and handle feedback submissions. By using @AuraEnabled methods, our LWCs could call Apex seamlessly, ensuring that the UI always displayed real-time data from Salesforce.

main > default > lwc > eventSearch > eventSearch.js > ...

```
import { LightningElement } from 'lwc';
import findOpenEvents from '@salesforce/apex/EventSearchController.findOpenEvents';

export default class EventSearch extends LightningElement {
  searchKey = '';
  events;

  handleChange(e) {
    this.searchKey = e.target.value;
  }

  handleSearch() {
    findOpenEvents({ searchKey: this.searchKey })
      .then(res => { this.events = res; })
      .catch(err => { this.events = undefined; console.error(err); });
  }
}
```

Stage 6: Events in LWC

- We implemented JavaScript events to make components interactive and responsive. For example, onchange events captured user input such as search text, while onclick events triggered data fetching through Apex methods. This approach ensured that LWCs were engaging and responded quickly to user actions.



Stage 7: Wire Adapters

- We also leveraged wire adapters for declarative data access in LWCs. These allowed us to fetch Salesforce records automatically without writing imperative calls in all scenarios. For example, the eventSearch component could display upcoming events in real time by wiring Apex methods directly to component variables.