

PHASE 5: Apex Programming

Goal: It is to enhance the Burial Booking System with custom logic and automation using Apex. It uses apex classes apex triggers etc.

1. Apex Classes

- Booking Request Controller - The BookingRequestController class is a custom Apex class designed to manage all backend logic related to Booking Requests in the system. It acts as a central controller to handle creation, validation, status updates, and notifications for booking requests submitted via the Flow or Experience Cloud portal.
- It basically handles the new booking request records by the families.

The screenshot displays the Salesforce Apex Class Editor for the **BookingRequestController** class. The interface includes a top navigation bar with a "SETUP" icon and the text "Apex Classes". Below this, the class name "BookingRequestController" is shown, along with buttons for "Edit", "Delete", "Download", "Security", and "Show Dependencies".

The "Apex Class Detail" section provides metadata for the class:

- Name:** BookingRequestController
- Status:** Active
- Namespace Prefix:** (blank)
- Code Coverage:** 0% (0/15)
- Created By:** Vanshika Awasthy, 24/09/2025, 2:23 pm
- Last Modified By:** Vanshika Awasthy, 24/09/2025, 2:25 pm

The "Class Body" tab is selected, showing the following Apex code:

```
1 public with sharing class BookingRequestController {
2
3     // Create a new Booking Request
4     @AuraEnabled
5     public static Id createBooking(Map<String, Object> payload) {
6         Booking_Request__c br = new Booking_Request__c();
7
8         // Map payload values to Booking_Request__c fields
9         if(payload.containsKey('Name')) br.Name = (String)payload.get('Name');
10        if(payload.containsKey('Desired_Burial_Date')) br.Desired_Burial_Date__c = Date.valueOf((String)payload.get('Desired_Burial_Date'));
11        if(payload.containsKey('Requested_PlotId')) br.Requested_Plot__c = (Id)payload.get('Requested_PlotId');
12        if(payload.containsKey('FamilyId')) br.Family__c = (Id)payload.get('FamilyId');
13        if(payload.containsKey('Comments')) br.Comments__c = (String)payload.get('Comments');
14
15        br.Status__c = 'New';
16
17        try {
18            insert br;
19            return br.Id;
20        } catch(Exception ex){
21            throw new AuraHandledException('Unable to create booking: ' + ex.getMessage());
22        }
23    }
24
25    // Return available plots for combobox
26    @AuraEnabled(cacheable=true)
27    public static List<Plot__c> getAvailablePlots() {
28        return [SELECT Id, Name FROM Plot__c ORDER BY Name];
29    }
30 }
```

At the bottom of the editor, there are buttons for "Edit", "Delete", "Download", "Security", and "Show Dependencies".

```

1 public with sharing class BookingRequestController {
2
3     // Create a new Booking Request
4     @AuraEnabled
5     public static Id createBooking(Map<String, Object> payload) {
6         Booking_Request__c br = new Booking_Request__c();
7
8         // Map payload values to Booking_Request__c fields
9         if(payload.containsKey('Name')) br.Name = (String)payload.get('Name');
10        if(payload.containsKey('Desired_Burial_Date')) br.Desired_Burial_Date__c = Date.valueOf((String)payload.get('Desired_Burial_Date'));
11        if(payload.containsKey('Requested_PlotId')) br.Requested_Plot__c = (Id)payload.get('Requested_PlotId');
12        if(payload.containsKey('FamilyId')) br.Family__c = (Id)payload.get('FamilyId');
13        if(payload.containsKey('Comments')) br.Comments__c = (String)payload.get('Comments');
14
15        br.Status__c = 'New';
16
17        try {
18            insert br;
19        } catch {
20            // Handle error
21        }
22    }
23 }

```

2. Apex Triggers

- Generate Booking Reference Trigger :

The GenerateBookingReference trigger automatically generates a unique booking reference number for each Booking_Request__c record before it is inserted. This ensures that every booking has a standardized, unique identifier even if the Name field is left blank.

Trigger Type:

Before Insert: Executes before the record is saved to the database, allowing the Name field to be populated automatically.

```

1 trigger GenerateBookingReference on Booking_Request__c (before insert) {
2     for (Booking_Request__c br : Trigger.new) {
3         if (br.Name == null) {
4             String yy = String.valueOf(Date.today().year()).substring(2);
5             br.Name = 'BR-' + yy + '-' + String.valueOf(Math.mod(Math.abs(Crypto.getRandomInteger()), 1000000));
6         }
7     }
8 }

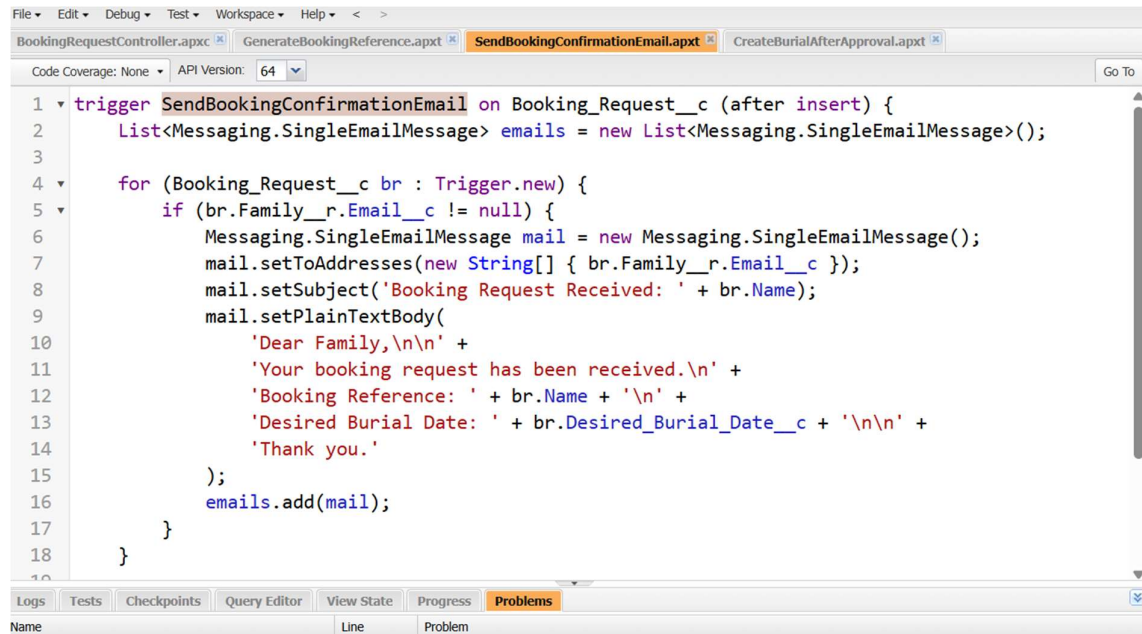
```

- Send Booking Confirmation Email Trigger

Purpose: The SendBookingConfirmationEmail trigger automatically sends a confirmation email to the family or user whenever a new booking request is created. This ensures that users receive immediate acknowledgment of their booking submission

Trigger Type:

After Insert: Executes after the record is saved to the database, so the booking ID and other fields are available for the email.



The screenshot shows the Visual Studio IDE with the 'SendBookingConfirmationEmail.apxt' file open. The code is a trigger that runs after an insert on the 'Booking_Request__c' table. It creates a list of email messages, iterates through them, and adds them to the list. The email body contains details about the booking request.

```
1 trigger SendBookingConfirmationEmail on Booking_Request__c (after insert) {
2     List<Messaging.SingleEmailMessage> emails = new List<Messaging.SingleEmailMessage>();
3
4     for (Booking_Request__c br : Trigger.new) {
5         if (br.Family__r.Email__c != null) {
6             Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
7             mail.setToAddresses(new String[] { br.Family__r.Email__c });
8             mail.setSubject('Booking Request Received: ' + br.Name);
9             mail.setPlainTextBody(
10                 'Dear Family,\n\n' +
11                 'Your booking request has been received.\n' +
12                 'Booking Reference: ' + br.Name + '\n' +
13                 'Desired Burial Date: ' + br.Desired_Burial_Date__c + '\n\n' +
14                 'Thank you.'
15             );
16             emails.add(mail);
17         }
18     }
19 }
```



This screenshot shows the continuation of the trigger code from the previous block. It includes the closing of the for loop and the logic to send the emails using the Messaging.sendEmail method.

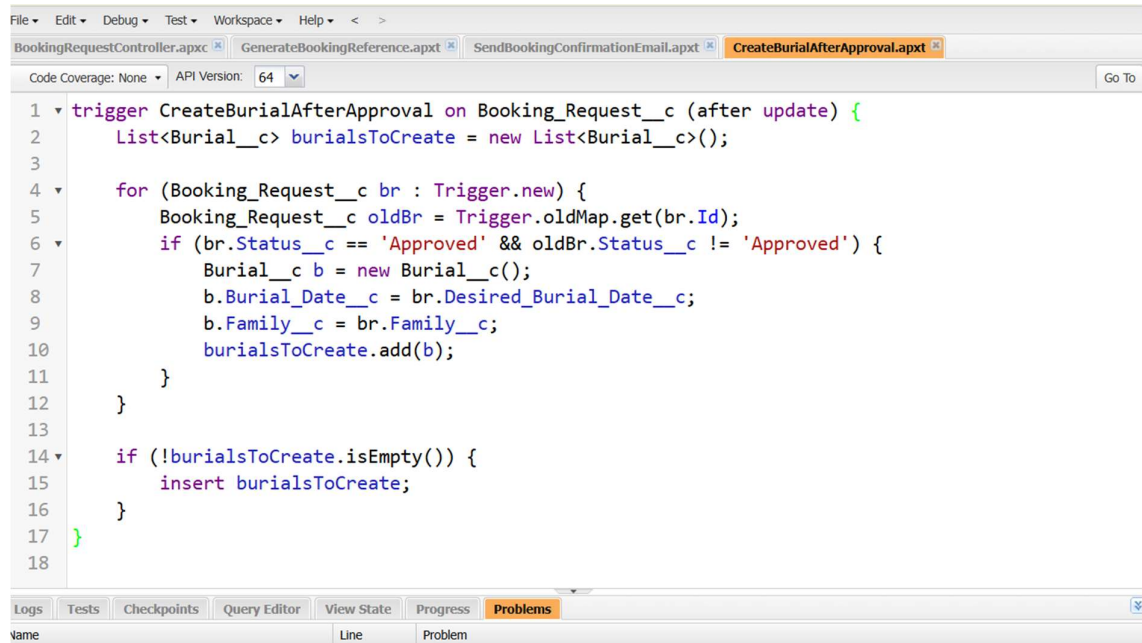
```
14         'Thank you.'
15     );
16     emails.add(mail);
17 }
18 }
19
20 if (!emails.isEmpty()) {
21     Messaging.sendEmail(emails);
22 }
23 }
24 }
```

- Create Burial After Approval Trigger

The CreateBurialAfterApproval trigger automatically creates a related Burial record when a booking request is approved. This ensures that only approved bookings result in actionable burial records, maintaining data integrity and workflow automation.

Trigger Type:

After Update: Executes after a booking request record is updated, because the trigger needs to check the updated Status field.



The screenshot shows an IDE window with the following tabs: BookingRequestController.apxc, GenerateBookingReference.apxt, SendBookingConfirmationEmail.apxt, and CreateBurialAfterApproval.apxt. The 'CreateBurialAfterApproval.apxt' tab is active. Below the tabs, there is a 'Code Coverage: None' and 'API Version: 64' dropdown. The main editor area displays the following code:

```
1 trigger CreateBurialAfterApproval on Booking_Request__c (after update) {  
2     List<Burial__c> burialsToCreate = new List<Burial__c>();  
3  
4     for (Booking_Request__c br : Trigger.new) {  
5         Booking_Request__c oldBr = Trigger.oldMap.get(br.Id);  
6         if (br.Status__c == 'Approved' && oldBr.Status__c != 'Approved') {  
7             Burial__c b = new Burial__c();  
8             b.Burial_Date__c = br.Desired_Burial_Date__c;  
9             b.Family__c = br.Family__c;  
10            burialsToCreate.add(b);  
11        }  
12    }  
13  
14    if (!burialsToCreate.isEmpty()) {  
15        insert burialsToCreate;  
16    }  
17 }  
18
```

At the bottom of the IDE, there is a 'Problems' tab which is currently empty, showing columns for Name, Line, and Problem.

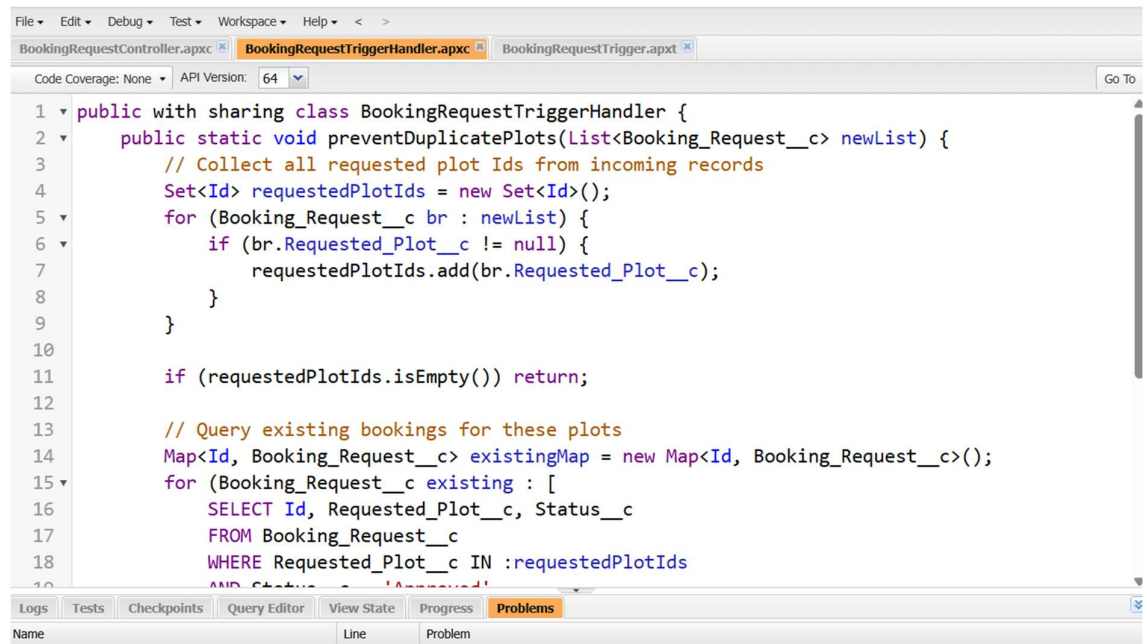
3. SOQL & SOSL

- SOQL is mainly used in triggers, controllers, and batch jobs to fetch booking or burial records.
- SOSL is helpful for search functionality in Experience Cloud, where users or admins might search bookings/families using names, references, or emails.

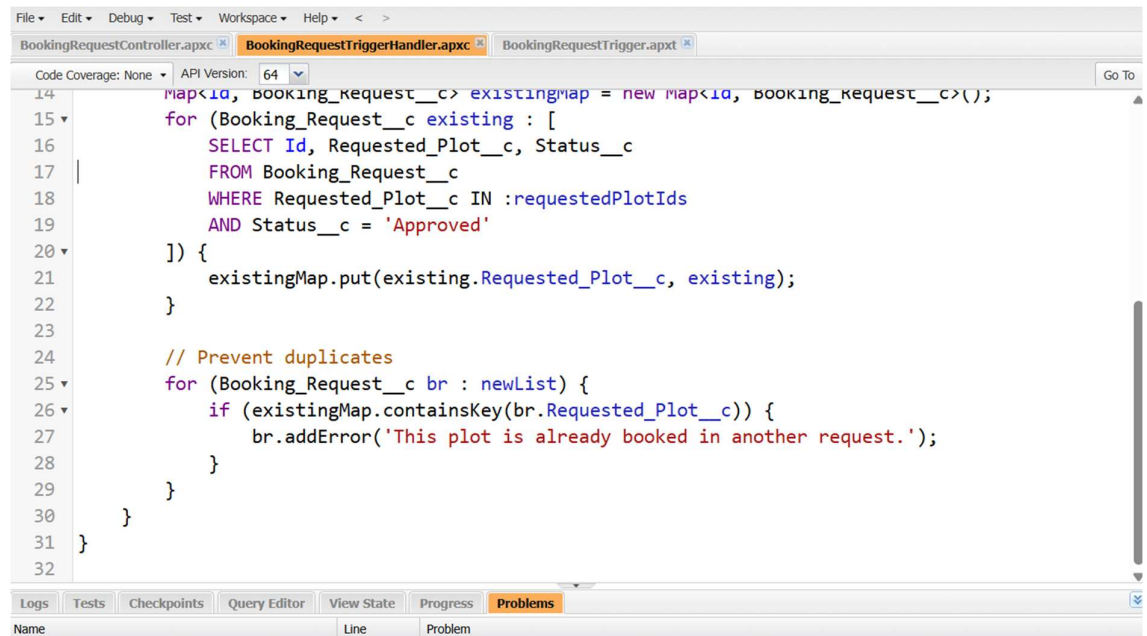
4. Collections: List, Set, Map

- List – It is an ordered collection that allows duplicates. It store multiple booking requests retrieved using SOQL.
- Set – An unordered collection of unique values (no duplicates). It store unique Family IDs from booking requests.
- Map – A collection of key–value pairs, where each key is unique. Link Booking IDs to their corresponding Booking records. Also it quickly fetches a record by ID without looping.

BookingRequestTriggerHandler.cls (Apex Class)

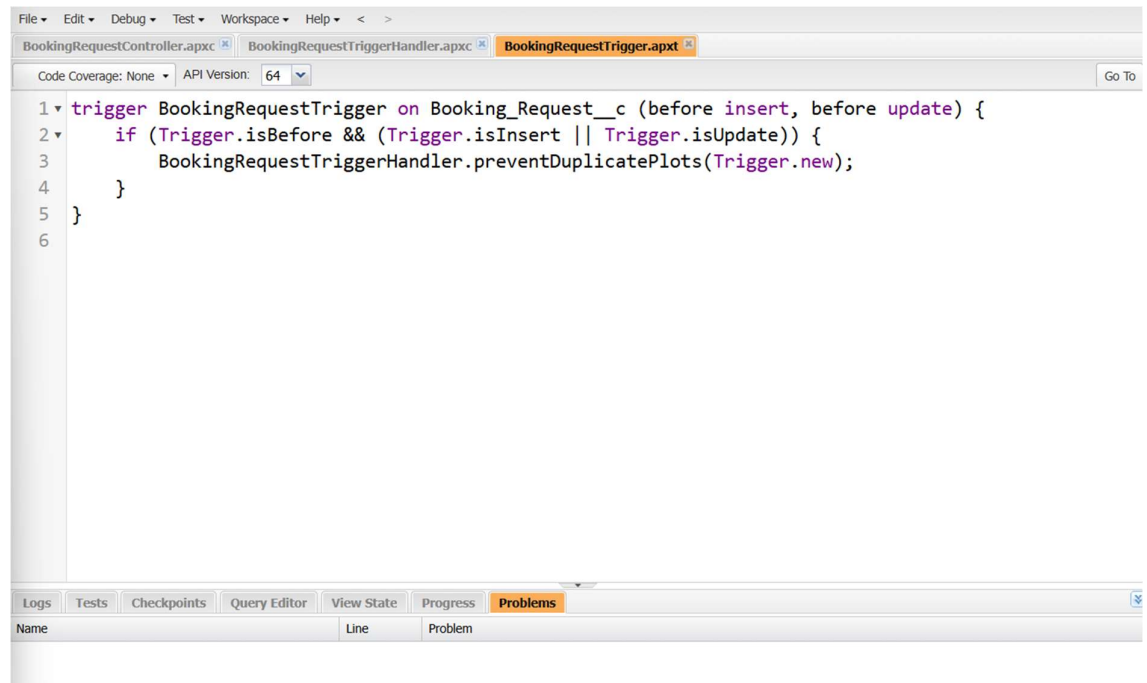


```
1 public with sharing class BookingRequestTriggerHandler {
2     public static void preventDuplicatePlots(List<Booking_Request__c> newList) {
3         // Collect all requested plot Ids from incoming records
4         Set<Id> requestedPlotIds = new Set<Id>();
5         for (Booking_Request__c br : newList) {
6             if (br.Requested_Plot__c != null) {
7                 requestedPlotIds.add(br.Requested_Plot__c);
8             }
9         }
10
11         if (requestedPlotIds.isEmpty()) return;
12
13         // Query existing bookings for these plots
14         Map<Id, Booking_Request__c> existingMap = new Map<Id, Booking_Request__c>();
15         for (Booking_Request__c existing : [
16             SELECT Id, Requested_Plot__c, Status__c
17             FROM Booking_Request__c
18             WHERE Requested_Plot__c IN :requestedPlotIds
19             AND Status__c = 'Approved'
```



```
14         Map<Id, Booking_Request__c> existingMap = new Map<Id, Booking_Request__c>();
15         for (Booking_Request__c existing : [
16             SELECT Id, Requested_Plot__c, Status__c
17             FROM Booking_Request__c
18             WHERE Requested_Plot__c IN :requestedPlotIds
19             AND Status__c = 'Approved'
20         ]) {
21             existingMap.put(existing.Requested_Plot__c, existing);
22         }
23
24         // Prevent duplicates
25         for (Booking_Request__c br : newList) {
26             if (existingMap.containsKey(br.Requested_Plot__c)) {
27                 br.addError('This plot is already booked in another request.');
```

BookingRequestTrigger.trigger (Apex Trigger)



```
1 trigger BookingRequestTrigger on Booking_Request__c (before insert, before update) {
2     if (Trigger.isBefore && (Trigger.isInsert || Trigger.isUpdate)) {
3         BookingRequestTriggerHandler.preventDuplicatePlots(Trigger.new);
4     }
5 }
6
```

PHASE 6: User Interface Development

Goal: The goal of this phase is to design and implement an intuitive, user-friendly interface for the Booking & Family Portal system. This includes using Lightning App Builder, Lightning Web Components (LWC), Flows, and record pages to enable families and admins to submit, view, and manage booking requests efficiently.

1. Lightning App Builder

The Lightning App Builder was used in our project to design and customize pages for the Family Portal and internal users. It allowed us to drag and drop components, such as Flows for booking requests, record detail pages, and related lists, without writing code. This provided a user-friendly interface for families to submit bookings and for admins to manage approvals, ensuring a smooth and intuitive experience

Lightning App Builder

The Lightning App Builder provides an easy to use graphical interface for creating custom Lightning pages for Salesforce Lightning Experience and mobile app. Lightning pages are built using Lightning components—compact, configurable, and reusable elements that you can drag and drop into regions of the page in the Lightning App Builder.

View: **All** [Create New View](#)

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Other | **All**

Action	Label ↑	Name	Namespace Prefix	Description	Type	Created By	Last Modified By
Edit Clone Del	Account Record Page	Account_Record_Page			Record Page	VAwas, 12/09/2025, 11:25 am	VAwas, 12/09/2025, 11:25 am
Edit Clone Del	Booking Request Page	Booking_Request_Page			Record Page	VAwas, 23/09/2025, 4:19 pm	VAwas, 24/09/2025, 2:44 pm
Edit Clone Del	Booking Request Record Page	Booking_Request_Record_Page			Record Page	VAwas, 24/09/2025, 3:16 pm	VAwas, 24/09/2025, 3:16 pm
Edit Clone Del	Booking Request Record Page	Booking_Request_Record_Page1			Record Page	VAwas, 24/09/2025, 3:19 pm	VAwas, 24/09/2025, 3:19 pm
Edit Clone Del	Plot Record Page	Plot_Record_Page			Record Page	VAwas, 17/09/2025, 1:21 pm	VAwas, 17/09/2025, 1:21 pm

2. Record Pages

- **Booking Request Record Page**
The Booking Request Record Page was customized using Lightning App Builder to display all key details of a booking request in one place.

Booking Request Record Page

Label: Booking Request Record Page

API Name: Booking_Request_Record_Page1

Page Type: Record Page

Object: Booking Request

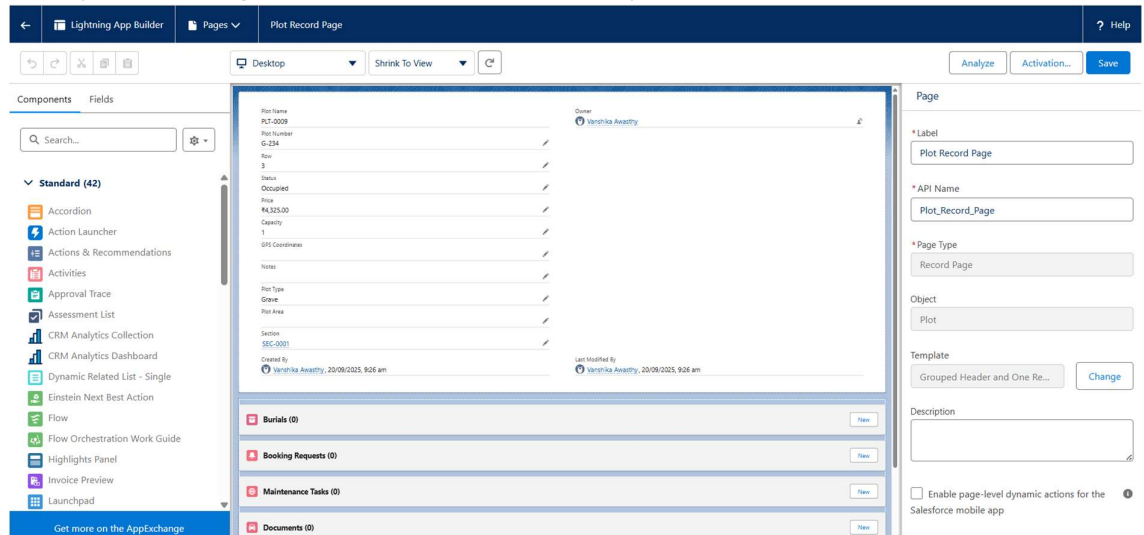
Template: Header and Right Sidebar

Description:

Enable page-level dynamic actions for the Salesforce mobile app

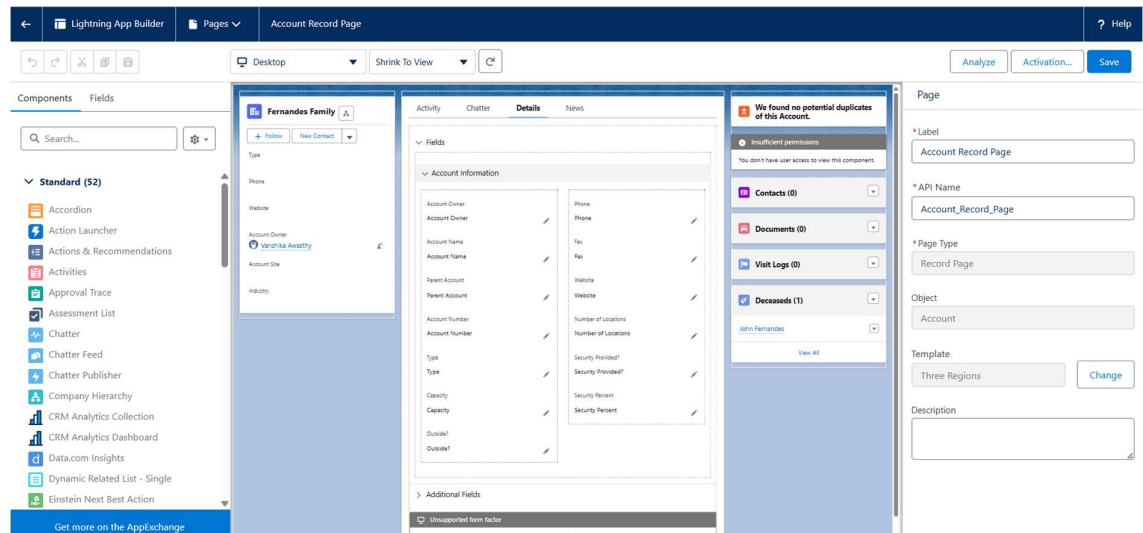
- **Plot Record Page**

The Plot Record Page was designed in Lightning App Builder to display all information related to burial plots, including Plot Number, Location, Availability Status, and linked Burial records.



- **Account Record Page**

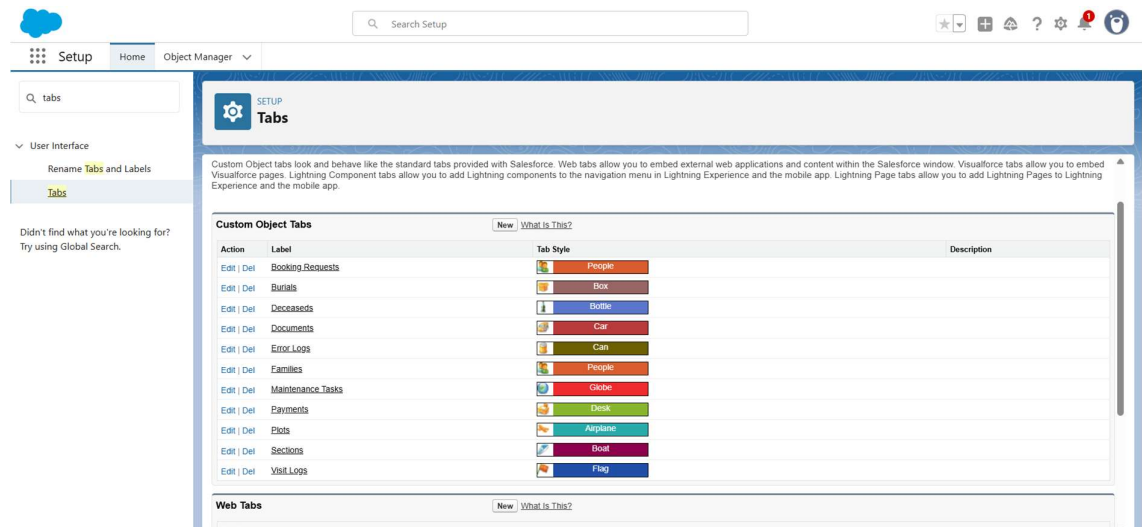
The Account Record Page was customized to act as the central hub for managing families and their related records.



3. Tabs

Tabs are navigation items that allow users to access specific objects, records, or functionality quickly. Each tab acts like a shortcut that opens a particular type of data or application page. Tabs can represent standard objects (like Accounts, Contacts), custom objects (like Booking Requests, Burials, Plots), or even Visualforce/Aura/LWC pages.

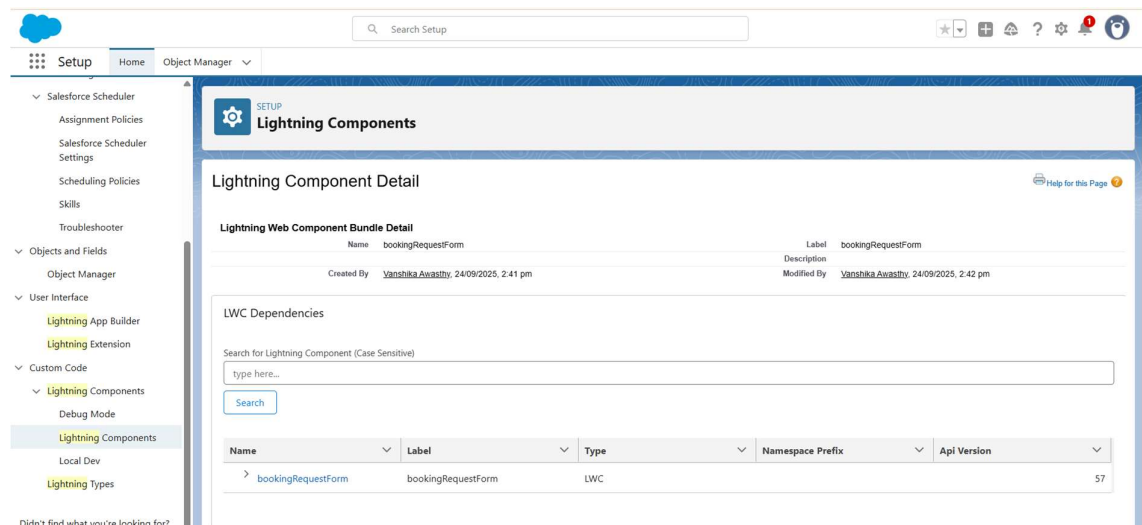
In our project, tabs were created for Booking Requests, Burials, Deceaseds, Documents, Error Logs, Families, Maintenance Task, Payments, Plots, Section and Visit Logs.



4. Lightning Web Components (LWC)

Lightning Web Components (LWC) is Salesforce's modern framework for building fast, reusable, and responsive web components using standard web technologies like HTML, JavaScript, and CSS

In our project, an LWC was developed for the Booking Request Form. This component allowed families to enter booking details in a dynamic and user-friendly interface. It improved the booking experience by making the form interactive, responsive, and mobile-friendly, compared to traditional page layouts or Visualforce pages.



Booking Request Family Portal Form

Home Support

Search...

User17576565044869652946

Burial Booking Flows

* Deceased Name

* Family Option

☐ Existing

☐ New

New Family Phone

Requested Plot


Search...

Comments

Upload Documents

[Upload Files](#) Or drop files

Next



Lightning Web Component (LWC) Files:

- **bookingRequestForm.html (Template / UI)**
Defines the form layout: Name, Desired Burial Date, Plot selection, Comments, Submit button.
Displays success and error messages dynamically.
Uses Salesforce Lightning base components for consistent UI and accessibility.

```

File Edit Selection View Go Run Terminal Help
Digital Cemetery
EXPLORER
  DIGITAL CEMETERY
    .husky
    .sf
    .sfdx
    .vscode
    config
    force-app/main/default/lwc
      applications
      aura
      classes
      contentassets
      flexipages
      layouts
      lwc
        bookingRequestForm.html
        bookingRequestForm.js
        bookingRequestForm.js-meta.xml
      tests
    jsconfig.json
    objects
    permissionsets
    staticresources
    tabs
    triggers
    scripts
    .forceignore
    .gitignore
    .prettiignore
    .prettierrc

force-app/main/default/lwc/bookingRequestForm/bookingRequestForm.html
1 <template>
2   <lightning-card title="Booking Request">
3     <div class="slds-p-around_small">
4       <lightning-input label="Name" value={name} onchange={handleChange} data-id="name"></lightning-input>
5       <lightning-input type="date" label="Desired Burial Date" value={desiredDate} onchange={handleChange} data-id="desiredDate"></lightning-input>
6       <lightning-combobox label="Plot" options={plotOptions} value={selectedPlot} onchange={handlePlotChange}></lightning-combobox>
7       <lightning-textarea label="Comments" value={comments} onchange={handleChange} data-id="comments"></lightning-textarea>
8       <div class="slds-m-top_small">
9         <lightning-button variant="brand" label="Submit Request" onclick={submit}></lightning-button>
10      </div>
11
12     <template if:true={successMessage}>
13       <div class="slds-m-top_small slds-text-color_success">{successMessage}</div>
14     </template>
15     <template if:true={errorMessage}>
16       <div class="slds-m-top_small slds-text-color_error">{errorMessage}</div>
17     </template>
18   </div>
19 </lightning-card>
20 </template>
21
PROBLEMS OUTPUT DEBUG CONSOLE
Salesforce CLI

```

- bookingRequestForm.js (Controller / Logic)
Handles component behavior : tracks field values, responds to user input, validates form.
Calls Apex (BookingRequestController) to create Booking Request records.
Updates UI with success or error messages.

```

1  import { LightningElement, track } from 'lwc';
2  import createBooking from '@salesforce/apex/BookingRequestController.createBooking';
3  import getAvailablePlots from '@salesforce/apex/BookingRequestController.getAvailablePlots';
4
5  export default class BookingRequestForm extends LightningElement {
6      @track name = '';
7      @track desiredDate = '';
8      @track comments = '';
9      @track plotOptions = [];
10     @track selectedPlot = '';
11     @track successMessage = '';
12     @track errorMessage = '';
13
14     connectedCallback() {
15         getAvailablePlots()
16             .then(result => {
17                 this.plotOptions = result.map(p => ({ label: p.Name, value: p.Id }));
18             })
19             .catch(err => {
20                 this.errorMessage = 'Unable to load plots';
21             });
22     }
23
24     handleChange(e) {
25         const id = e.target.dataset.id;
26         if(id === 'name') this.name = e.target.value;
27         if(id === 'date') this.desiredDate = e.target.value;
28         if(id === 'comments') this.comments = e.target.value;
29     }
30
31     handlePlotChange(e) {
32         this.selectedPlot = e.target.value;
33     }
34
35     submit() {
36         this.errorMessage = '';
37         this.successMessage = '';
38
39         if (!this.name || !this.desiredDate || !this.selectedPlot) {
40             this.errorMessage = 'Please fill all required fields';
41             return;
42         }
43
44         const payload = {
45             Name: this.name,

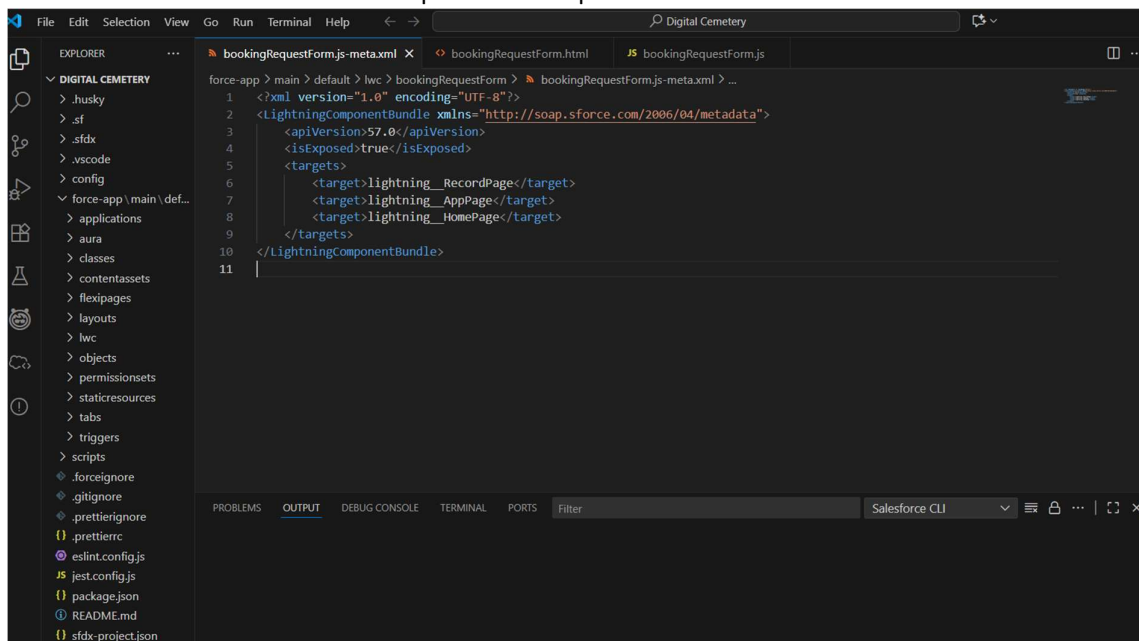
```

```
const payload = {
  Name: this.name,
  Desired_Burial_Date: this.desiredDate,
  Requested_PlotId: this.selectedPlot,
  Comments: this.comments
};

createBooking({ payload })
  .then(id => {
    this.successMessage = 'Booking request submitted – reference: ' + id;
    this.name = '';
    this.desiredDate = '';
    this.selectedPlot = '';
    this.comments = '';
  })
  .catch(err => {
    this.errorMessage = err.body ? err.body.message : 'Unexpected error';
  });
}
```

EMS OUTPUT DEBUG CONSOLE ... Filter

- bookingRequestForm.js-meta.xml (Metadata / Visibility)
Makes the LWC available in Lightning App Builder.
Defines page targets: Record Page, App Page, Home Page.
Controls where and how admins can place the component.

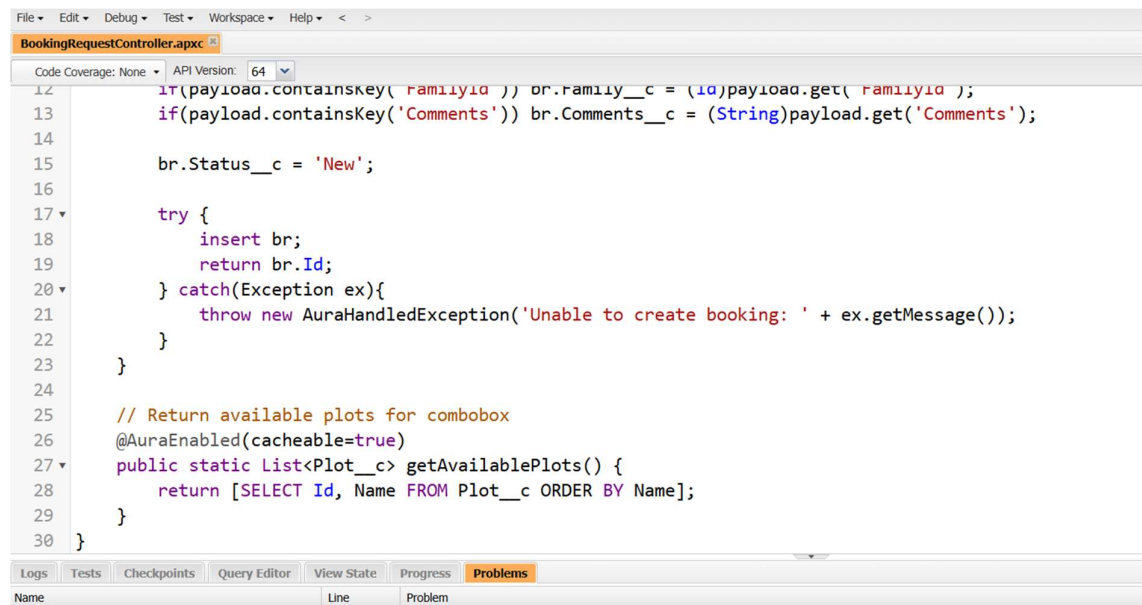


5. Apex with LWC

Booking Request Controller Class



```
1 public with sharing class BookingRequestController {
2
3     // Create a new Booking Request
4     @AuraEnabled
5     public static Id createBooking(Map<String, Object> payload) {
6         Booking_Request__c br = new Booking_Request__c();
7
8         // Map payload values to Booking_Request__c fields
9         if(payload.containsKey('Name')) br.Name = (String)payload.get('Name');
10        if(payload.containsKey('Desired_Burial_Date')) br.Desired_Burial_Date__c = Date.valueOf((String)payload.get('Desired_Burial_Date'));
11        if(payload.containsKey('Requested_PlotId')) br.Requested_Plot__c = (Id)payload.get('Requested_PlotId');
12        if(payload.containsKey('FamilyId')) br.Family__c = (Id)payload.get('FamilyId');
13        if(payload.containsKey('Comments')) br.Comments__c = (String)payload.get('Comments');
14
15        br.Status__c = 'New';
16
17        try {
18            insert br;
19        } catch (Exception ex) {
20            // Handle exception
21        }
22    }
23 }
```



```
12        if(payload.containsKey('FamilyId')) br.Family__c = (Id)payload.get('FamilyId');
13        if(payload.containsKey('Comments')) br.Comments__c = (String)payload.get('Comments');
14
15        br.Status__c = 'New';
16
17        try {
18            insert br;
19            return br.Id;
20        } catch (Exception ex){
21            throw new AuraHandledException('Unable to create booking: ' + ex.getMessage());
22        }
23    }
24
25    // Return available plots for combobox
26    @AuraEnabled(cacheable=true)
27    public static List<Plot__c> getAvailablePlots() {
28        return [SELECT Id, Name FROM Plot__c ORDER BY Name];
29    }
30 }
```

PHASE 7 : Integration & External Access

In the project Digital Cemetery , features such as Named Credentials, External Services, REST/SOAP APIs, Callouts, Platform Events, Change Data Capture, and Salesforce Connect were not implemented. The Booking & Family Portal system is entirely self-contained within Salesforce, with all processes handled using Apex, Flows, LWC, and Experience Cloud components. These integration features can be considered for future enhancements if external system connectivity is required.

