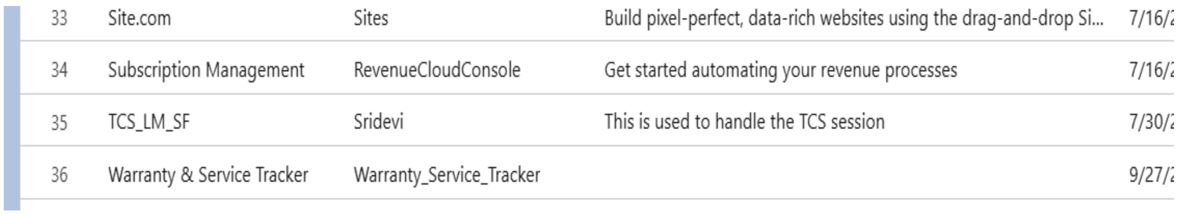
Phase 6: User Interface Development

1. Lightning App Builder

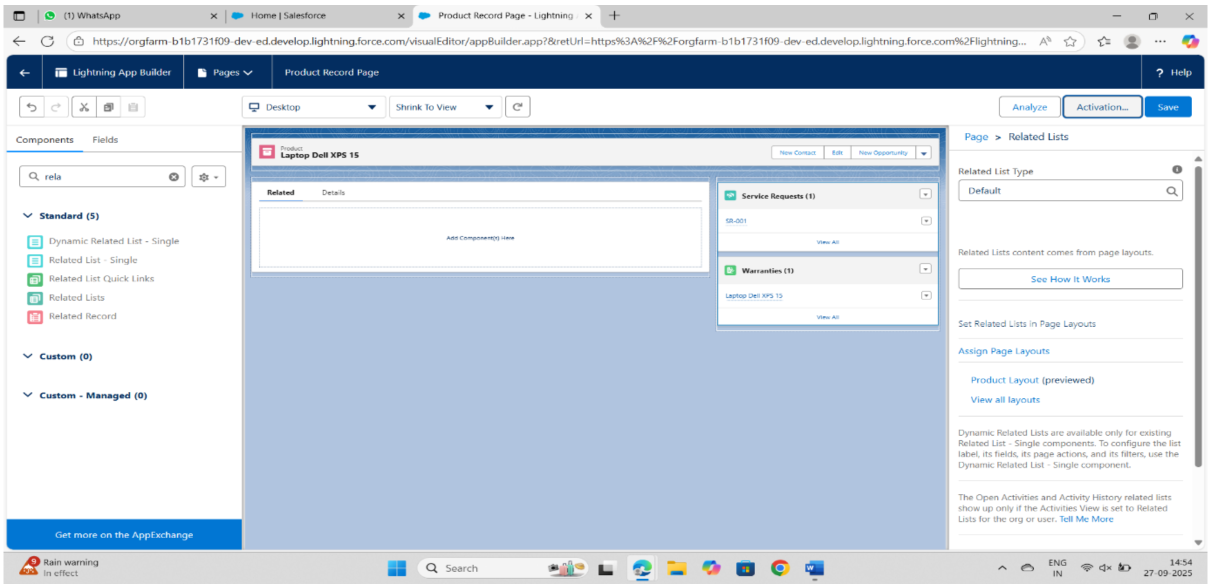
- **What we did:**
  Used Lightning App Builder to customize the app experience. Created a custom **Warranty & Service Tracker App** that includes Products, Warranties, and Service Requests in the navigation bar.
- **Steps:**
  1. Setup → App Manager → New Lightning App.
  2. Gave app name: *Warranty & Service Tracker*.
  3. Added tabs: Products, Warranties, Service Requests.
  4. Assigned to all profiles for visibility.
- **Screenshot:** App Manager showing new app.



33	Site.com	Sites	Build pixel-perfect, data-rich websites using the drag-and-drop Si...	7/16/
34	Subscription Management	RevenueCloudConsole	Get started automating your revenue processes	7/16/
35	TCS_LM_SF	Sridevi	This is used to handle the TCS session	7/30/
36	Warranty & Service Tracker	Warranty_Service_Tracker		9/27/

2. Record Pages

- **What we did:**
  Designed custom **record pages** for Product, Warranty, and Service Request using Lightning App Builder.
- **Steps:**
  1. Setup → Object Manager → Product__c → Lightning Record Pages.
  2. Created a new Record Page → added Highlights Panel, Related Lists, and Tabs.
  3. Assigned as Org Default.
- **Screenshot:** Product record page editor.



Page > Related Lists

Related List Type: Default

Related Lists content comes from page layouts.

See How It Works

Set Related Lists in Page Layouts

Assign Page Layouts

Product Layout (previewed)

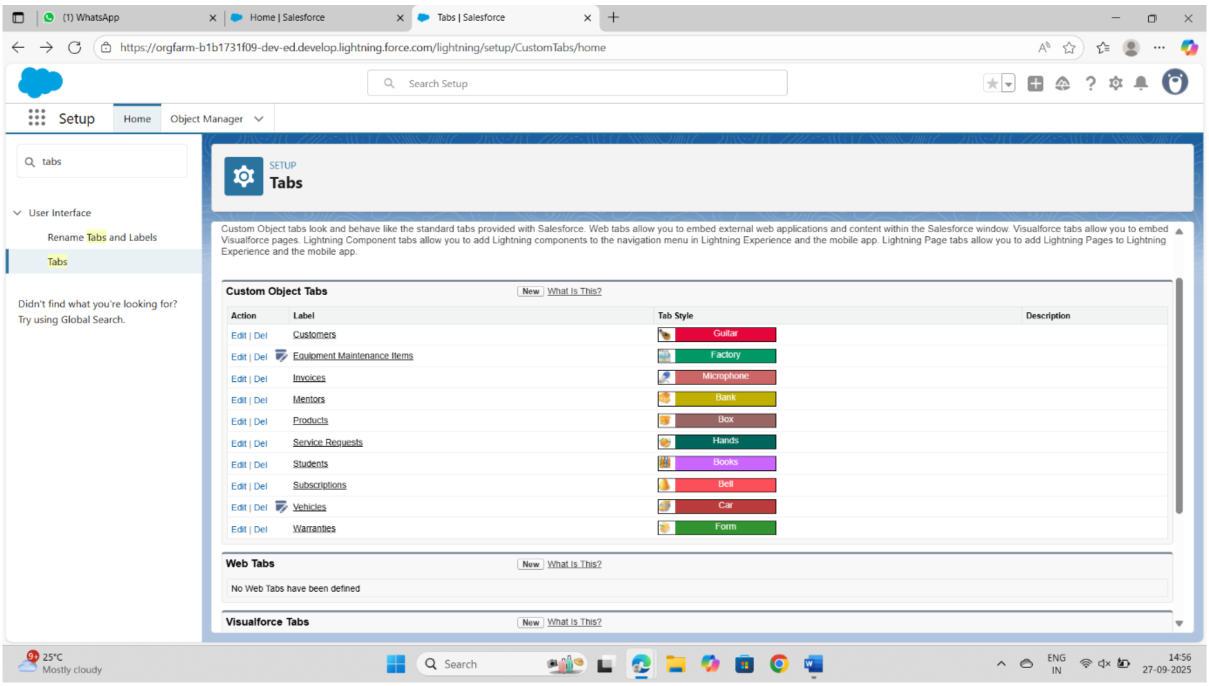
View all layouts

Dynamic Related Lists are available only for existing Related List - Single components. To configure the list label, its fields, its page actions, and its filters, use the Dynamic Related List - Single component.

The Open Activities and Activity History related lists show up only if the Activities View is set to Related Lists for the org or user. Tell Me More

3. Tabs

- **What we did:**
  Added **Custom Tabs** for each custom object.
- **Steps:**
  1. Setup → Tabs → New.
  2. Chose Object = Product__c, Warranty__c, Service_Request__c.
  3. Selected tab style (icon & color).
  4. Added them to the Warranty & Service Tracker App.



SETUP
Tabs

Custom Object tabs look and behave like the standard tabs provided with Salesforce. Web tabs allow you to embed external web applications and content within the Salesforce window. Visualforce tabs allow you to embed Visualforce pages. Lightning Component tabs allow you to add Lightning components to the navigation menu in Lightning Experience and the mobile app. Lightning Page tabs allow you to add Lightning Pages to Lightning Experience and the mobile app.

Custom Object Tabs

Action	Label	Tab Style	Description
Edit | Del	Customers	Guitar	
Edit | Del	Equipment Maintenance Items	Factory	
Edit | Del	Invoices	Microphone	
Edit | Del	Mentors	Bank	
Edit | Del	Products	Box	
Edit | Del	Service Requests	Hands	
Edit | Del	Students	Books	
Edit | Del	Subscriptions	Bell	
Edit | Del	Vehicles	Car	
Edit | Del	Warranties	Form	

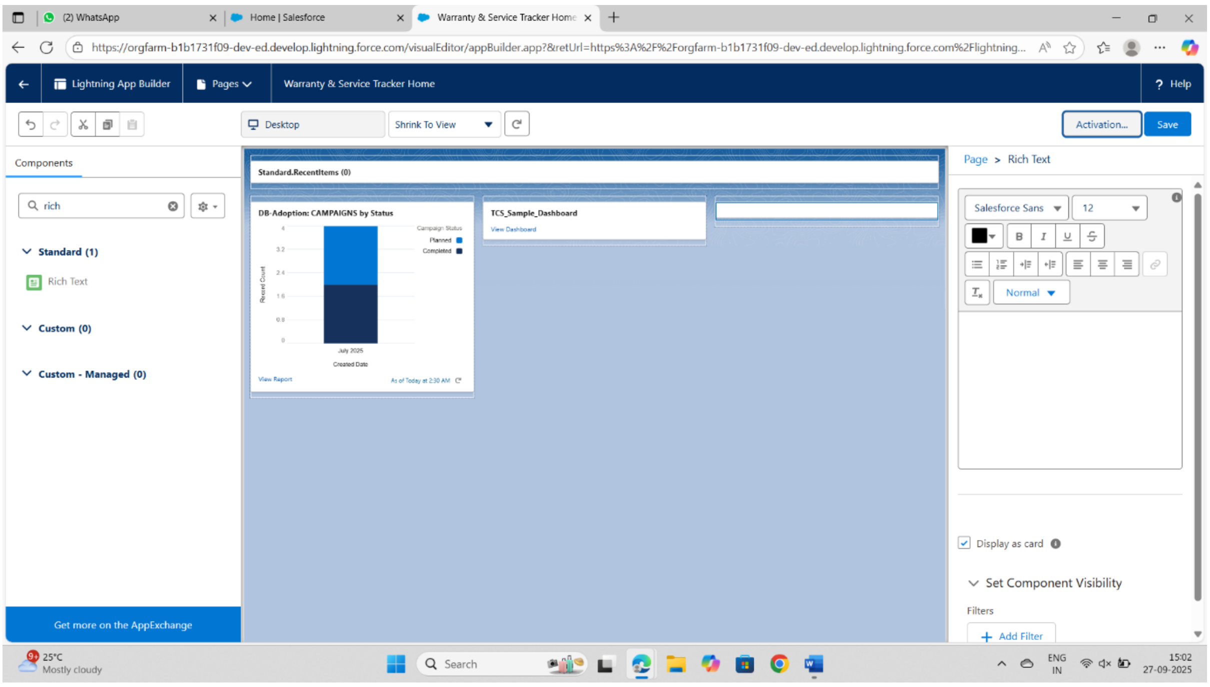
Web Tabs

No Web Tabs have been defined

Visualforce Tabs

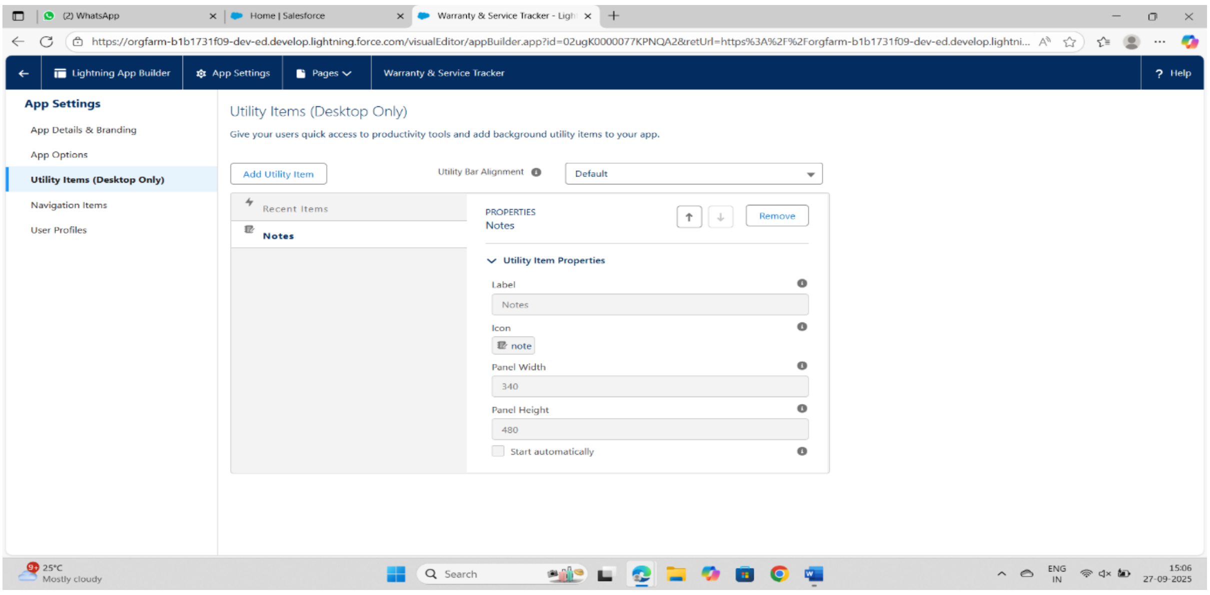
4. Home Page Layouts

- **What we did:**
  Customized Home Page to show dashboard components and quick links.
- **Steps:**
  1. **Setup → Lightning App Builder → Home Page → New.**
  2. **Dragged standard components like Recent Items, Reports, and Dashboard Snapshot.**
  3. **Activated for Warranty & Service Tracker App.**



5. Utility Bar

- **What we did:**
  Added a **Utility Bar** for quick access to Notes and Recent Items.
- **Steps:**
  1. Setup → App Manager → Edit App → Utility Bar.
  2. Added “Notes” and “History”.


Utility Items (Desktop Only)

Give your users quick access to productivity tools and add background utility items to your app.

Utility Bar Alignment: Default

Utility items: Recent Items, Notes

PROPERTIES
Notes

Utility Item Properties

Label: Notes

Icon: note

Panel Width: 340

Panel Height: 480

Start automatically: ☐

6. Lightning Web Components (LWC)

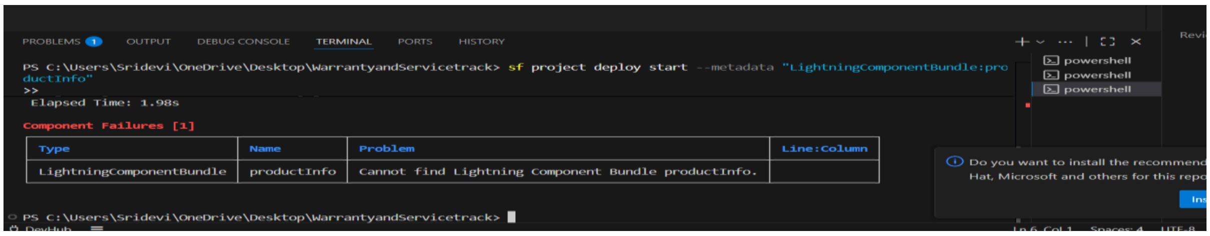
- **What we did:**
  Created a sample LWC to display Product details.

- **Code:**

```
import { LightningElement, api } from 'lwc';
export default class ProductInfo extends LightningElement {
    @api recordId;
}
<template>
    <lightning-card title="Product Info">
        <p>Product Record Id: {recordId}</p>
    </lightning-card>
</template>
```

- **Deploy:**

```
sf project deploy start --metadata "LightningComponentBundle:productInfo"
```


```
PS C:\Users\Sridevi\OneDrive\Desktop\WarrantyandServicetrack> sf project deploy start --metadata "LightningComponentBundle:pro
ductInfo"
>>
 Elapsed Time: 1.98s

Component Failures [1]
┌──────────────────────────┬─────────────┬─────────────────────────────────────────────────┬─────────────┐
│ Type                     │ Name        │ Problem                                         │ Line:Column │
├──────────────────────────┼─────────────┼─────────────────────────────────────────────────┼─────────────┤
│ LightningComponentBundle │ productInfo │ Cannot find Lightning Component Bundle productInfo. │         │
└──────────────────────────┴─────────────┴─────────────────────────────────────────────────┴─────────────┘

PS C:\Users\Sridevi\OneDrive\Desktop\WarrantyandServicetrack>
```

7. Apex with LWC

- **What we did:**
  Connected LWC to Apex to fetch Products.

  Product.js

```
public with sharing class ProductController {
  @AuraEnabled(cacheable=true)
 public static List<Product__c> getProducts() {
  return [SELECT Id, Name, Serial_Number__c, Price__c FROM Product__c LIMIT
5];
    }
}
```

```
import { LightningElement, wire } from 'lwc';
import getProducts from '@salesforce/apex/ProductController.getProducts';


export default class ProductList extends LightningElement {
   @wire(getProducts) products;
}
```

productList.html

```
<template>
   <lightning-card title="Products">
      <template if:true={products.data}>
         <template for:each={products.data} for:item="p">
            <p key={p.Id}>{p.Name} - {p.Serial_Number__c} - {p.Price__c}</p>
         </template>
      </template>
   </lightning-card>
</template>
```

- **Screenshot:** Product list displayed on page.

8. Events in LWC

- **What we did:**
  Implemented simple child → parent event communication.
- **Code:**
  Child Component → dispatch event.
- this.dispatchEvent(new CustomEvent('notify', { detail: 'Hello from child' }));

Parent Component → handle event.

```
<c-child onnotify={handleMessage}></c-child>
```

9. Wire Adapters

- **What we did:**
  Used @wire with UI API.

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

import { getRecord } from 'lightning/uiRecordApi';

import NAME_FIELD from '@salesforce/schema/Product__c.Name';


export default class ProductWire extends LightningElement {

   @wire(getRecord, { recordId: 'a01XX00000XXXX', fields: [NAME_FIELD] })

   product;

}
```

10. Imperative Apex Calls

- **What we did:**
  Called Apex explicitly from JS.

```
import getProducts from '@salesforce/apex/ProductController.getProducts';

handleClick() {

   getProducts()

      .then(result => { this.products = result; })

     .catch(error => { this.error = error; });

}
```

11. Navigation Service

- **What we did:**
  Used NavigationMixin to open record pages.

```
    import { NavigationMixin } from 'lightning/navigation';
    export default class ProductNavigator extends NavigationMixin(LightningElement) {
       navigateToProduct() {
        this[NavigationMixin.Navigate]({
          type: 'standard__recordPage',
      attributes: {
            recordId: 'a01XX00000XXXX',
             objectApiName: 'Product__c',
               actionName: 'view'
            }
          });
   }
}
```