Medicare Cloud — Phase 6: User Interface Development

Goal

Build intuitive, role-specific Lightning interfaces and reusable Lightning Web Components (LWC) for Medicare Cloud. This document provides step-by-step instructions for Lightning App Builder, Record Pages, Tabs, Home Page Layouts, Utility Bar, LWC development, Apex integration, events, wire adapters, imperative Apex calls, and navigation service.

Prerequisites

• Salesforce Developer Edition or Sandbox
• Salesforce CLI (SFDX) and a dev hub if using scratch orgs
• Visual Studio Code with Salesforce Extension Pack
• Basic knowledge of Salesforce Setup, Objects, and Profiles
• Git (recommended) for source control

Contents

1. Plan & Design

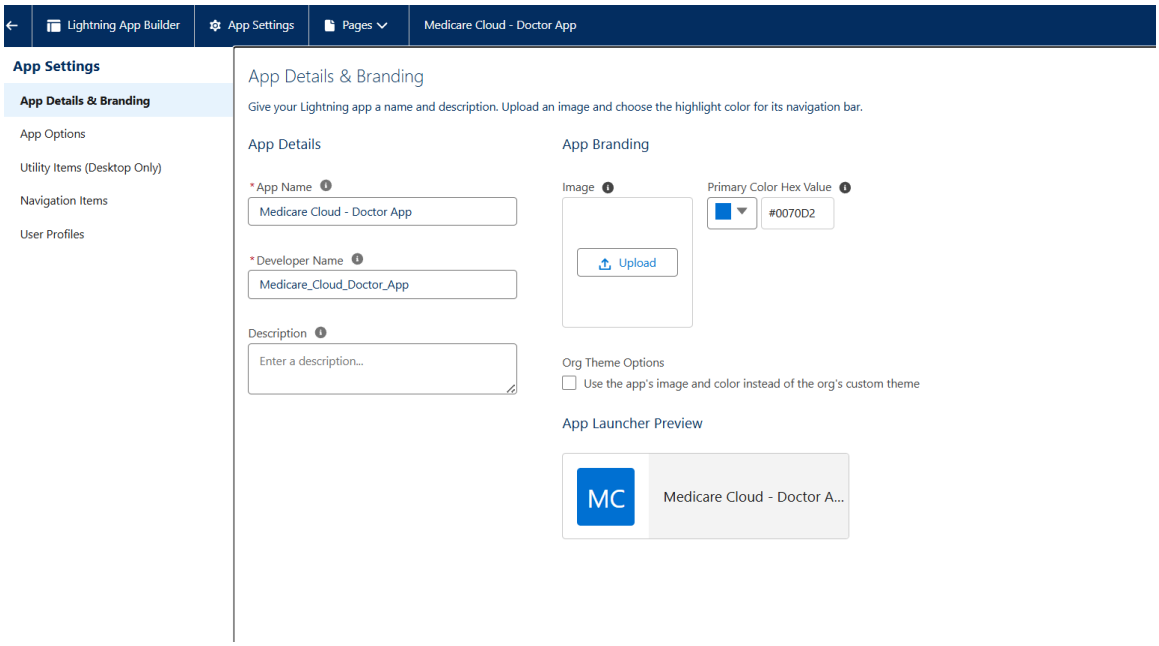
Before building UI, design the apps and components:

• Identify user personas (Doctor, Nurse, Insurance Officer, Admin).
• Define key pages: Patient record page, Appointment scheduler, Claims console.
• Decide which pages will use standard components and which require custom LWCs.
• Sketch layouts for Home, Record, and App pages.

2. Lightning App Builder (Create Lightning App)

Steps:

1. Setup → Quick Find → App Manager.
2. Click New Lightning App.
3. Enter App Name (e.g., "Medicare Cloud - Doctor App"), Description, Branding (logo/color).
4. Navigation Type: Standard Navigation.
5. Add Navigation Items (Patient, Appointments, Claims, Reports). Save & Finish.
6. Edit App → Utility Bar to add quick utilities (phone, notes, custom LWC).



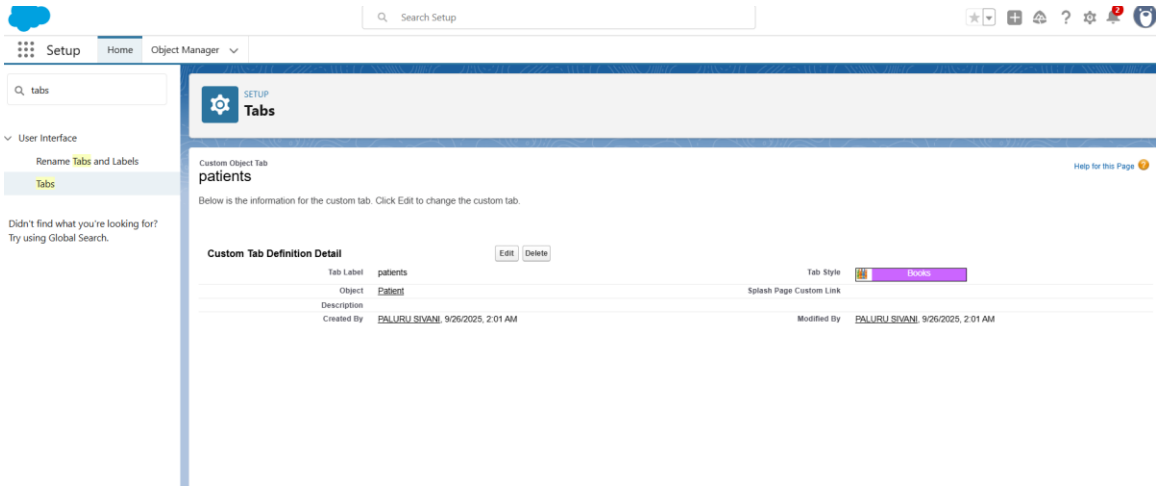
3. Tabs

Steps:

1. Setup → Quick Find → Tabs.
2. Create Custom Object Tabs for Patient__c, Appointment__c, Insurance_Claim__c.
3. Create Lightning Component Tabs for LWCs after deployment (New → Lightning Component Tab).

4. In App Manager, add tabs to the app navigation.

Naming tip: Use meaningful tab labels like "Patients", "Appointments", "Claims".



4. Home Page Layouts

Steps:

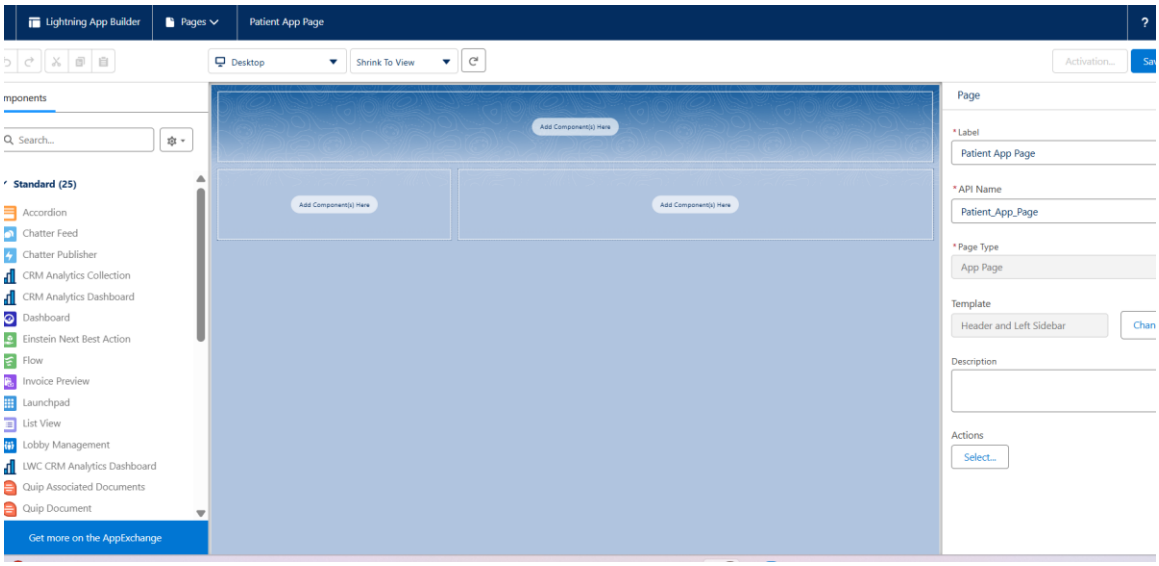
1. Setup → Quick Find → Lightning App Builder → New → Home Page.
2. Choose template (Header & 3 Columns, etc.).
3. Drag standard components (Report Chart, List Views) and custom LWCs onto the canvas.
4. Save → Activate → assign to App/Profile or Org Default.

Label suggestions: "Medicare Cloud Home", "Doctor Dashboard Home".

5. Record Pages

Steps:

1. Setup → Object Manager → Patient__c → Lightning Record Pages → New.
2. Choose page template (Header & Tabs recommended for healthcare).
3. Add components: Record Detail, Related Lists, Highlights Panel, and custom LWCs (e.g., CarePlanTimeline).
4. Configure component visibility filters (e.g., show Telemedicine widget only for Doctors).
5. Save → Activate → assign by App/Profile/Record Type as needed.



6. Utility Bar

Steps:

1. App Manager → Edit your Lightning App → Utility Bar.
2. Add utilities: Recent Items, Notes, Path, Custom LWC.
3. Configure size, initial state (open/closed), and other properties.
4. Save the app. Utility items appear at the bottom of the Lightning interface.

Use case: quick patient search, click-to-call, or a persistent notes widget.

7. LWC — Build & Deploy

Recommended workflow: develop locally with VS Code + SFDX, commit to Git, deploy to dev org.

Steps to create a basic LWC (PatientList):

patientList.html (LWC template):

```xml
<template>
  <lightning-card title="Patients">
    <div class="slds-p-around_small">
      <lightning-input type="search" placeholder="Search patients..." onchange={handleSearch}></lightning-input>
      <template if:true={patients.data}>
        <ul>
          <template for:each={patients.data} for:item="p">
            <li key={p.Id} class="slds-m-top_small">
              <a href="javascript:void(0)" data-id={p.Id} onclick={handleSelect}>{p.FullName}</a>
            </li>
          </template>
        </ul>
      </template>
      <template if:true={patients.error}>
        <c-error-panel errors={patients.error}></c-error-panel>
      </template>
    </div>
  </lightning-card>
</template>
```

patientList.js (LWC JavaScript):

```
scode > JS PatientList.js > ...
import { LightningElement, track, wire } from 'lwc';
import getPatients from '@salesforce/apex/PatientController.getPatients';

export default class PatientList extends LightningElement {
  @track searchKey = '';

  @wire(getPatients, { searchKey: '$searchKey' })
  patients;

  handleSearch(event) {
    this.searchKey = event.target.value;
  }

  handleSelect(evt) {
    const id = evt.currentTarget.dataset.id;
    // dispatch custom event to parent or navigate (example below)
    const selection = new CustomEvent('patientselected', { detail: { id } });
    this.dispatchEvent(selection);
  }
}
```

patientList.js-meta.xml (meta configuration):

```
.vscode > patientList.js-meta.xml
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>59.0</apiVersion>
  <isExposed>true</isExposed>
  <targets>
    <target>lightning__RecordPage</target>
    <target>lightning__AppPage</target>
    <target>lightning__HomePage</target>
  </targets>
</LightningComponentBundle>
```

Deployment (VS Code):
1. sfdx force:auth:web:login -a DevOrg
2. sfdx force:source:push (scratch org) OR sfdx force:source:deploy -p force-app/main/default
3. Add the LWC to a Lightning Page via Lightning App Builder.

8. Apex with LWC

Create Apex controllers to supply data to LWCs. Use @AuraEnabled(cacheable=true) for read-only methods and @AuraEnabled for DML methods.

PatientController.cls (Apex controller example):

Saving: PatientController.apxc *

Code Coverage: None API Version: 64

```
public with sharing class PatientController {
  @AuraEnabled(cacheable=true)
  public static List<Patient__c> getPatients(String searchKey) {
    String sk = '%' + String.escapeSingleQuotes(searchKey == null ? '' : searchKey) + '%';
    return [SELECT Id, Full_Name__c, Date_of_Birth__c, Phone__c FROM Patient__c
            WHERE Full_Name__c LIKE :sk ORDER BY Full_Name__c LIMIT 100];
  }

  @AuraEnabled
  public static Id savePatient(Patient__c p) {
    upsert p;
    return p.Id;
  }
}
```

9. Events in LWC

• Child-to-parent: dispatch CustomEvent (e.g., patientselected) and handle in parent via onpatientselected attribute.
• Cross-component: use Lightning Message Service (LMS) for pub/sub across DOM boundaries.
• Platform events: for async server-to-client notifications.

Example (dispatching event):
this.dispatchEvent(new CustomEvent("patientselected", { detail: { id: recordId } }));

10. Wire Adapters

Common adapters:
• getRecord, getFieldValue from lightning/uiRecordApi
• getObjectInfo and getPicklistValues from lightning/uiObjectInfoApi
• Use @wire for reactive data and cacheable=true Apex methods for efficiency.

Example getRecord usage:

```
import { getRecord, getFieldValue } from 'lightning/uiRecordApi';
import FULLNAME_FIELD from '@salesforce/schema/Patient__c.Full_Name__c';
@wire(getRecord, { recordId: '$recordId', fields: [FULLNAME_FIELD] })
rec;
```

11. Imperative Apex Calls

Use imperative calls for user triggered actions (save buttons). Handle loading state and errors.

Example:

```
import savePatient from '@salesforce/apex/PatientController.savePatient';

handleSave() {
  this.isLoading = true;
  savePatient({ p: this.patientRecord })
    .then(result => { /* handle success */ })
    .catch(error => { /* handle error */ })
    .finally(() => { this.isLoading = false; });
}
```

12. Navigation Service (NavigationMixin)

Use NavigationMixin to navigate to record pages, object home, or web URLs programmatically.

Example:

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

13. Testing, Debugging & Deployment

• Use local development server (lwc.dev or SFDX with local development) and Jest for unit tests.
• Use browser console and Salesforce Lightning Inspector for debugging.
• Run Apex tests and 75%+ code coverage for production deployment.
• Use Git and CI/CD (GitHub Actions/Bitbucket Pipelines) to run deployments via

SFDX.
• Deployment steps: push to scratch/dev org → QA sandbox → run tests → deploy to Production via CI or Change Sets.

14. Checklist & Deliverables

• Create Lightning App(s) and assign navigation items

• Create Custom Tabs for key objects and LWCs

• Design and activate Home Page layouts

• Build Record Pages with tailored components and visibility rules

• Add Utility Bar items for quick access

• Develop and unit-test LWCs (wire + imperative patterns)

• Create Apex controllers with cacheable methods and DML entry points

• Implement custom events and LMS where needed

• Test navigation and mobile behavior

• Document UI components and deployment steps (SFDX/Git)

Deliverables:
• Deployed LWCs and Apex controllers in dev org
• Lightning App, Record & Home pages configured
• Documentation and screenshots for handoff
• CI/CD pipeline for deployments (recommended)