**Architecture / Design**

The initial concept that formulated in my mind upon reviewing the assignment is outlined as follows:-

Get the contact record (including email field) 🡪 get the email Id of contact record 🡪 get the user associated with this email id.

**Step 1: Get the contact record Id.**

To access the contact record, we first require a recordId. when placing the component on the record page, Salesforce provides the recordId of the current record using the public property '@api recordId'. This gives us easy access to the contact record ID.

**Step2: Get the actual contact record.**

With the contact record's ID in hand, we can proceed to retrieve all the field values associated with it. At this juncture, I had two possible approaches:

A. Develop an Apex method that takes the ID as a parameter and returns the corresponding contact sObject record.

B. Utilize the @wire adapter LDS method called 'getRecord' (imported from the 'lightning/uiRecordApi' module).

Opting for B appeared more advantageous due to its reactive nature. This means that if the data changes within Salesforce, the information in the Lightning Web Component (LWC) will automatically update. This is achieved by either re-executing the wired method or reinitializing the wired variable. Given our requirement where users can modify the record, including changing the email address, I selected option B. This ensures that our component remains up-to-date and properly rendered based on any new updates.

**Step3: Getting the user data associated with the email address.**

Now that we possess the contact record, the next hurdle is to acquire the user details linked to the email address within that contact record. For this purpose, I developed an @AuraEnabled Apex method named 'getContactUser()' (note that to utilize an Apex method in an LWC, it must be annotated with '@AuraEnabled').

The question that arises is where to invoke this Apex method. Placing the invocation within 'connectedCallback()' poses a challenge, as the contact record retrieval facilitated by the wired adapter 'getRecord()' might still be ongoing due to its asynchronous nature. Consequently, the contactRecord value might not have been obtained yet by the time the Apex method is called within 'connectedCallback()' or 'renderedCallback()'. Hence, calling the 'getContactUser()' Apex method within these methods is not appropriate.

To resolve this dilemma, I designed a JS method named 'fetchContactUserDetails(email)'. From this method, I initiate an imperative call to the 'getContactUser(String email)' Apex method. This strategic approach ensures that when we invoke the 'getContactUser(String email)' Apex method, we are guaranteed to have the contact record ready.

**The steps outlined above provide a high-level description that elucidates the solution's design.**

-------------------------------------------------------------------------------------------------------------------------------

**HTML SIDE**

On the HTML side of the template, I've employed conditional rendering via `<template if:true>` and `<template if:false>` elements. These elements rely on a JavaScript property called 'isContactUserAvailable', which is exclusively set to 'true' when we successfully retrieve the user linked to the given email address. Conversely, it triggers rendering of the `<template if:false>` branch, displaying a message stating 'no user found with this email address'.

Furthermore, I've incorporated Salesforce Lightning Design System (SLDS) classes to enhance the visual design. To organize the screen into two columns within a Lightning Card element (used to display user details), I've utilized classes like 'slds-grid' and 'slds-gutters'.

**Explanation of Apex Side Controller**

To retrieve user information based on the email address (sourced from the contact's email address), I've developed an Apex class named 'ContactUserController'. Within this class, there exists a method 'getContactUser(String email)', marked with the '@AuraEnabled' annotation.

The implementation of this method adheres to best practices and takes into account various scenarios, including:

* Addressing the scenario where the email address is blank.
* Handling cases where user information is not found.
* Managing potential exceptions that might arise.

The method is crafted to handle all these cases effectively.

**ContactUserWrapper Class**

In the primary controller class, namely 'ContactUserController', I've adhered to the instruction to use the wrapper class. Consequently, instead of returning the actual User sObject, the controller returns an instance of the 'ContactUserWrapper' wrapper object.

**ContactUserControllerTest Class**

Adhering to the best practices of Apex testing, I've established a comprehensive test setup that generates two contact records and one user record. Within this test class, I've devised three distinct test cases:

1. A test to validate the 'getContactUser()' method when provided with an email address that lacks any associated user.

2. A test to assess the 'getContactUser()' method's performance with an email address that does possess an associated user.

3. A test designed to evaluate the behavior of the 'getContactUser()' method when a blank email address is supplied.

I've managed to achieve full coverage—100%—for both the 'ContactUserController' and 'ContactWrapper' classes.

While I've concentrated on the fundamental core functionality in my test class, it's worth noting that there exists potential for further expansion. Additional test cases could be formulated to account for more intricate scenarios. However, the current test class effectively ensures the robustness of the solution.

**Reactiveness of the component**

The process begins with the wire adapter, a default reactive component, which retrieves contactRecord details. This wired method maintains its reactivity, ensuring that if a user alters the email address, the Salesforce Lightning Data Service (LDS) wire adapter automatically triggers an update by reexecuting the wired method. As a result, the solution consistently receives the latest contact data, and subsequently, the imperative call to the Apex controller method provides the most up-to-date user details or an appropriate 'no user found' message, based on the current contact data state.

**Component Visibility**

The component's accessibility is restricted solely to System Administrators. This restriction is configured during the component's placement on the record page. Within the 'component visibility' settings, a filter condition can be specified, requiring that the profile name of the current user matches 'System Administrator'.

To validate this visibility configuration, I conducted a test by creating a user named 'Test User' with the profile 'Standard platform user'. I then evaluated the component's visibility for the 'Test User' by logging in as that user.

**Component Availability**

In accordance with the specified requirements, the component must be situated exclusively on the contact record page. To enforce this constraint, I've employed the '<target>' attribute, restricting it solely to 'lightning\_\_RecordPage'. Additionally, I've defined the target configurations with '<object>Contact</object>' within the component's XML configuration file. This ensures that the component is exclusively associated with the contact object's record page.

**Component Property configuration by Admin**

For the convenience of administrators in customizing the component's title and headings, I've introduced a configurable property in the component's XML configuration file. This entails using a tag such as '<property name="componentTitle">' which is subsequently employed as a public property within the JS controller.

**Extra feature- View User Details Button**

I've introduced an additional feature to the component: a 'View User Details' button. Leveraging the NavigationMixin in LWC, I've implemented navigation functionality that directs users to the Contact User record page. While this enhancement wasn't explicitly specified in the original assignment, I've incorporated it into the component to augment interactivity and enrich the user experience.

**Important Assumption**

In specific situations, it's conceivable that a single email address could be associated with multiple user accounts. Given that the assignment did not explicitly address this scenario, I proceeded with the assumption that if multiple users share the same email address, the details of the initial user would be presented. In practical scenarios, I would have engaged in communication with the Business Analyst or the entity supplying the requirements to gain clarity on this matter. This dialogue would be vital to ascertain the appropriate course of action for accurately managing instances where multiple users are linked to the same email address.

**Note:** This document provides a concise outline of my solution's design and the thought process. However, I also have numerous insights to share at the code level.