Requirements specification

I embarked on an intensive adaptation process that proved to be a genuine challenge. During this crucial period, I dedicated myself to the in-depth study of two tutorials previously unfamiliar to me. The complexity of these guides not only demanded swift assimilation but also a thorough understanding to be able to apply them effectively within the scope of my responsibilities for the week. This intensive immersion served as a valuable opportunity to accelerate my skill development within the team, despite the fact that I joined a week after its inception.

I found particular resonance with the tutorial on databases, as I had encountered significant challenges in that area during the preceding weeks. This specific tutorial became a focal point for me, driven by a personal need to overcome difficulties I had faced earlier. The knowledge acquired proved to be a linchpin for the task at hand during the week, offering me a newfound understanding and proficiency that directly addressed the challenges I had previously encountered. In essence, my deliberate focus on this tutorial not only contributed to overcoming past obstacles but also significantly bolstered my effectiveness in fulfilling my responsibilities for the current week.

The task I chose to treat quickly is : As a system administrator, I want to maintain reference values for alert types

At the moment I started coding, I saw that Alois had already created the database. So, I began by creating my branch in relation to his branch.

Then, I began with creating my Alert class :

```
using SQLite;

namespace UndacBlue.Models
{
    public class Alert
    {
        [PrimaryKey, AutoIncrement]
        public int Id { get; set; }

        public string Name { get; set; }

    }
}
```

I, then, departed from Alois's database.cs file to implement the functions necessary for the CRUD method :

```
   // CRUD operations for Alert

     public Task<List<Alert>> GetAlertsAsync()
     {
         return _database.Table<Alert>().ToListAsync();
     }

     public Task<Alert> GetAlertByIdAsync(int id)
     {
          return _database.Table<Alert>().Where(a => a.Id ==
 id).FirstOrDefaultAsync();
     }

     public Task<int> InsertAlertAsync(string name)
     {
         var alertType = new Alert { Name = name };
         return _database.InsertAsync(alertType);
     }

     public Task<int> UpdateAlertAsync(Alert alert)
     {
         return _database.UpdateAsync(alert);
     }

     public Task<int> DeleteAlertAsync(Alert alert)
     {
         return _database.DeleteAsync(alert);
     }
```

This code was made possible thanks to the tutorial!

To conclude with the code, I simply utilized the various functions in the code-behind of my Alert page (AlertsPage.xaml.cs) and populated the database accordingly.

```
 public partial class AlertsPage : ContentPage
     {
         private ObservableCollection<Alert> Alerts { get; set; }
         private Database Database { get; set; }

         public AlertsPage()
         {
             InitializeComponent();
```

```
            Database = new Database();
            Alerts = new ObservableCollection<Alert>();
            AlertsListView.ItemsSource = Alerts;

            LoadAlerts();
        }

        private async void LoadAlerts()
        {
            Alerts.Clear();
            var alerts = await Database.GetAlertsAsync();
            foreach (var alert in alerts)
            {
                Alerts.Add(alert);
            }
        }

        private async void OnAddAlertClicked(object sender, EventArgs e)
        {
            string alertName = AlertNameEntry.Text;
            if (!string.IsNullOrEmpty(alertName))
            {
                await Database.InsertAlertAsync(alertName);
                LoadAlerts();
                AlertNameEntry.Text = string.Empty; // Clear the entry after
adding an alert
            }
        }

        private async void OnDeleteAlertClicked(object sender, EventArgs e)
        {
            var button = (Button)sender;
            var alert = (Alert)button.CommandParameter;

            if (alert != null)
            {
                await Database.DeleteAlertAsync(alert);
                LoadAlerts();
            }
        }
    }
```

The implementation of the code was punctuated by challenges, notably the delicate decision-making regarding the use of the `async` and the determination of the access level, transitioning from `internal` to `public`. The choice of whether or not to incorporate the async keyword raised questions.

Unfortunately, I was only able to finish on Sunday, so I couldn't submit my code for review, and I didn't receive any code to review either.