Step-by-Step: Crafting a Transparent Status Bar and Bottom Notch for Android and iOS Apps

When designing mobile apps, a transparent status bar and bottom notch can give your interface a cleaner, more immersive feel. Here's a quick guide on setting this up for both Android and iOS.

Sample

GitHub - VBZelin/MyMauiSamplesApp

Contribute to VBZelin/MyMauiSamplesApp development by creating an account on GitHub.

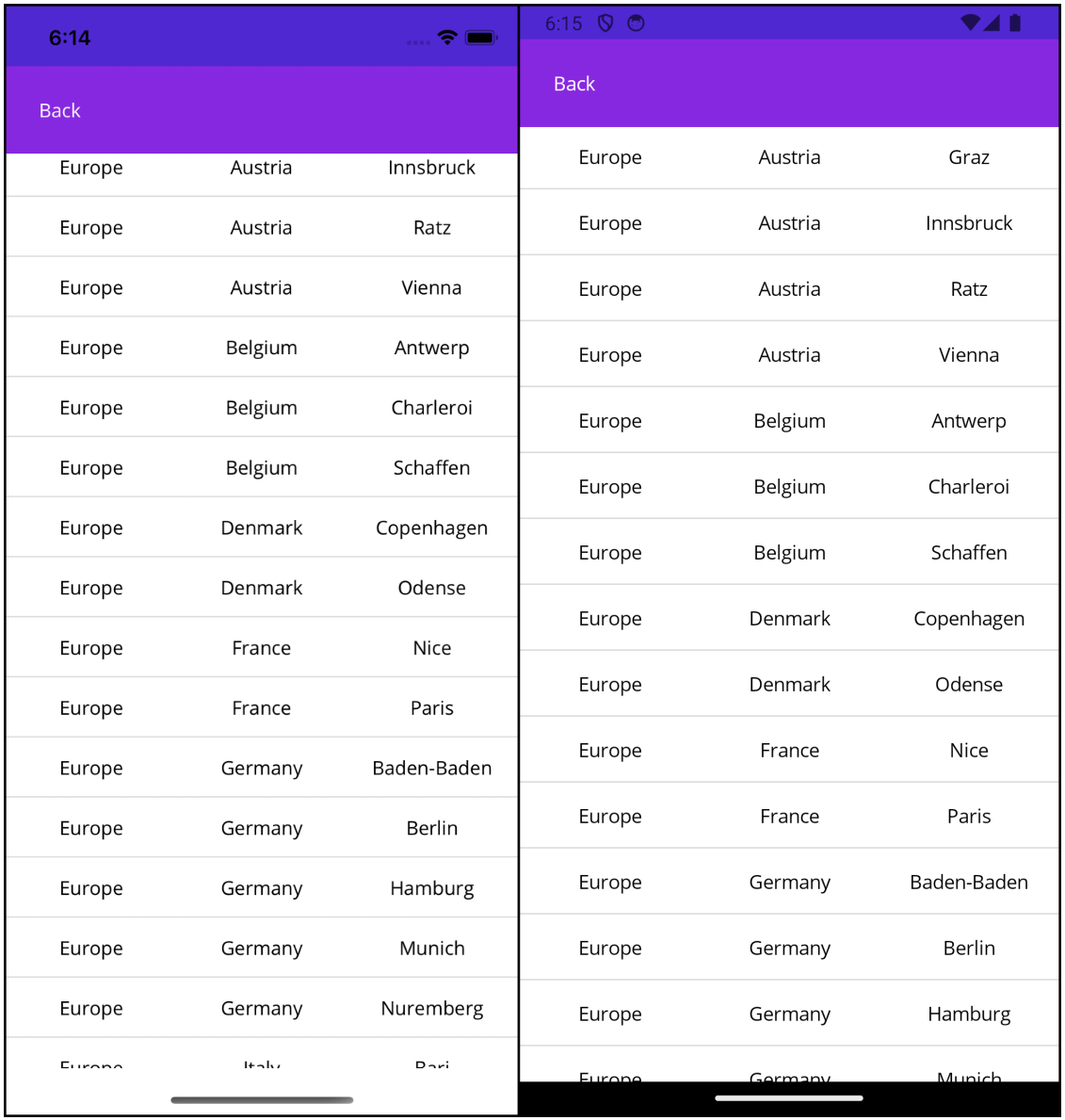
https://github.com/VBZelin/MyMauiSamplesApp

Problem Overview

One of the primary challenges is that the status bar and bottom notch are separate from the app's content. Unlike other UI components that

transition smoothly when navigating between pages, the status bar and bottom notch remains static, leading to visual inconsistencies:

1. If the status bar color isn't updated properly, it may not align with the new page's theme, creating an inconsistent look.
2. Since the status bar and bottom notch are system-level elements, app content can sometimes be cut off or hidden behind it, especially if safe area settings aren't handled correctly.



iOS Setup

Firstly, we define a `ContentPage` in XAML for our `TransparentStatusBarPage`. This page is configured to ignore the standard safe areas, allowing the content to extend beneath the system status bar, achieving an immersive effect.

```xml
<ContentPage
    x:Class="MyMauiSamplesApp.BasePage"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"
    xmlns:local="clr-namespace:MyMauiSamplesApp"
    xmlns:toolkit="http://schemas.microsoft.com/dotnet/2022/maui/toolkit"
    x:Name="basePageRoot"
    ios:Page.UseSafeArea="{Binding UseFullScreen, Source={Reference basePageRoot}, Converter={StaticResource invertedBoolConverter}}"
    Shell.FlyoutBehavior="Disabled"
    Shell.NavBarIsVisible="False"
    Shell.TabBarIsVisible="False">
    <Shell.BackButtonBehavior>
        <BackButtonBehavior IsEnabled="False" IsVisible="False" />
    </Shell.BackButtonBehavior>
    <!-- Content goes here -->
</ContentPage>
```

Here, `UseSafeArea` is dynamically bound and set to `False`, allowing the content to extend into the status bar area. Additionally, the navigation bar, tab bar and back button are hidden to maintain a clean look.

Then inside the `ContentPage`, we utilize the `StatusBarBehavior` from the MAUI Toolkit to dynamically adjust the status bar's appearance:

```xml
<ContentPage.Behaviors>
    <toolkit:StatusBarBehavior
        ApplyOn="OnPageNavigatedTo"
        StatusBarColor="Transparent"
        StatusBarStyle="{AppThemeBinding Light=DarkContent, Dark=LightContent}" />
</ContentPage.Behaviors>
```

This behavior is triggered when navigating to the page, setting the status bar color to transparent and adjusting the content color based on the theme (light or dark).

Last thing is to set `IgnoreSafeArea="True"` in body `Grid`, it allows the content to extend into the status bar and bottom notch areas. This helps the background and UI elements flow naturally under these system UI elements, making the app feel more immersive. ***However, you may need to adjust padding to prevent important content from being hidden, especially on devices with notches.***

Android Setup

In the `TransparentStatusBarPage` code-behind, platform-specific adjustments ensure that the Android status bar becomes fully transparent without any layout constraints, while iOS maintains its default handling.

To apply a transparent status bar and notch area to a specific page, use the following code in its .cs file

CSharp

```csharp
protected override void OnAppearing() {
  base.OnAppearing();
  #if ANDROID
  var window = Platform.CurrentActivity?.Window;
  if (window != null) {
    window.SetFlags(Android.Views.WindowManagerFlags.LayoutNoLimits,
Android.Views.WindowManagerFlags.LayoutNoLimits);
    window.ClearFlags(Android.Views.WindowManagerFlags.TranslucentStatus);
  }
  #endif
}
protected override void OnDisappearing() {
  base.OnDisappearing();
  #if ANDROID
  var window = Platform.CurrentActivity?.Window;
  if (window != null) {
    window.ClearFlags(Android.Views.WindowManagerFlags.LayoutNoLimits);
    window.AddFlags(Android.Views.WindowManagerFlags.TranslucentStatus);
  }
  #endif
}
```

To implement a transparent status bar and notch area throughout the application lifecycle, use the following code in `MauiProgram.cs`

```csharp
#if ANDROID
builder.ConfigureLifecycleEvents(events =>
{
    events.AddAndroid(android => android.OnCreate((activity, bundle) =>
    {
        var window = activity.Window;
        if (window is not null)
        {
            window.SetFlags(Android.Views.WindowManagerFlags.LayoutNoLimits,
Android.Views.WindowManagerFlags.LayoutNoLimits);
            window.ClearFlags(Android.Views.WindowManagerFlags.TranslucentStatus);
        }
    }));
});
#endif
```

For Android, the flags `LayoutNoLimits` are set to allow the app to extend into the area where the status bar resides. The `TranslucentStatus` flag is cleared to ensure there are no semi-transparent overlays.

Notch Height

To create a full-screen experience without content getting blocked by the status bar, navigation bar, or notches, you need to calculate the notch height (safe area insets) separately for iOS and Android. You can do this by writing platform-specific code to get the correct height based on each system's safe area settings.

iOS Notch Height Calculation

The `SafeAreaInsets` provide the top and bottom padding needed to keep content from overlapping with app content.

```csharp
public static partial class SafeAreaHelper
{
    public static partial Thickness GetSafeAreaInsets()
```

```
    {
        if (UIDevice.CurrentDevice.CheckSystemVersion(11, 0))
        {
            var window = UIApplication.SharedApplication
                .ConnectedScenes
                .OfType<UIWindowScene>()
                .SelectMany(scene => scene.Windows)
                .FirstOrDefault(w => w.IsKeyWindow);

            if (window != null)
            {
                var insets = window.SafeAreaInsets;
                return new Thickness(insets.Left, insets.Top, insets.Right, insets.Bottom);
            }
        }

        return new Thickness(0);
    }
}
```

Android Notch Height Calculation

Safe area insets are calculated using system-defined dimensions.

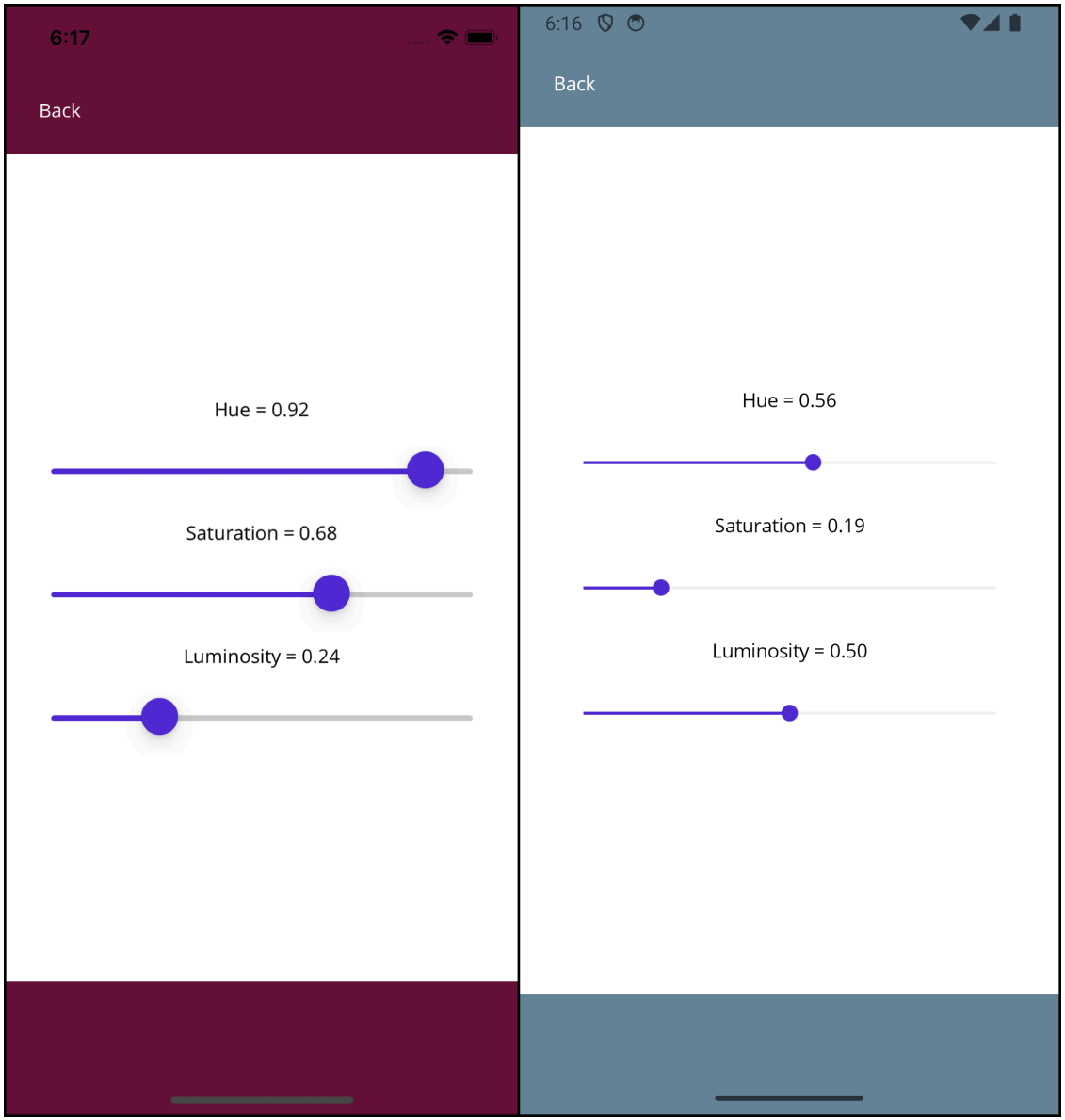
CSharp

```
public static partial class SafeAreaHelper
{
    public static partial Thickness GetSafeAreaInsets()
    {
        var safeAreaInsets = new Thickness(0, GetStatusBarHeight(), 0,
GetNavigationBarHeight());
        return safeAreaInsets;
    }

    public static double GetStatusBarHeight()
    {
        return GetSystemBarHeight("status_bar_height");
    }

```

```
    public static double GetNavigationBarHeight()
    {
        return GetSystemBarHeight("navigation_bar_height");
    }

    private static double GetSystemBarHeight(string resourceName)
    {
        var context = Platform.CurrentActivity ?? throw new NullReferenceException("Activity is null");

        var resources = context.Resources;
        if (resources == null)
            return 0;

        int resourceId = resources.GetIdentifier(resourceName, "dimen", "android");
        if (resourceId > 0)
        {
            var displayMetrics = resources.DisplayMetrics;
            if (displayMetrics == null)
                return 0;

            return resources.GetDimensionPixelSize(resourceId) / displayMetrics.Density;
        }

        return 0;
    }
}
```

Updated Result

After implementing the transparent status bar and handling the bottom notch area, the UI now provides a smoother and more immersive experience. The background extends under the status bar and bottom area. Here is the result from my demo app:



Key improvements:

✔ **Status bar color updates correctly** before page transitions, preventing flickers.

✔ **Content no longer gets cut off** by the status bar or bottom notch.

✔ **Smooth integration** of UI elements into system areas, creating a modern, edge-to-edge design.