Windows 10

Hands-on lab

Lab: Handling Page Navigation and Back

August 2015

Microsoft

CONTENTS

Overview

With UWP, Microsoft has introduced tools to help you provide a seamless navigation experience across devices. While a UWP app will run on any Windows 10 device, we still must plan for scenarios where the hardware differs.

The SystemNavigationManager class lets you opt-in to display a shell-drawn back button when a hardware back button isn't available, or when you don’t want to provide your own back navigation UI on the pages of your app. The SystemNavigationManager also offers a universal BackRequested event that you can use to handle backwards navigation regardless of the device family or the manifestation of the back UI. These new abilities allow you to handle back navigation without extension SDKs or adaptive code. Previously in Windows 8 apps, a back button drawn in the app content was the only back navigation option available for tablet and desktop devices. Now you may choose whichever method works best for your app's implementation, design, and user experience.

In this lab, we will cover the basics of navigation and how to implement it in a simple UWP app. We will explore how we can pass data between pages of the application as we navigate and handle back navigation with the shell-drawn back button.

Objectives

This lab will show you how to:

- Create a new XAML page
- Navigate to a secondary page
- Pass a parameter on navigation
- Enable the shell-drawn back button when the backstack is populated
- Implement the standard back-requested pattern

System requirements

You must have the following to complete this lab:

- Microsoft Windows 10
- Microsoft Visual Studio 2015

- Windows 10 Mobile Emulator

Setup

You must perform the following steps to prepare your computer for this lab:

1. Install Microsoft Windows 10.
2. Install Microsoft Visual Studio 2015. Choose a custom install and ensure that the Universal Windows App Development Tools are selected from the optional features list.
3. Install the Windows 10 Mobile Emulator.

Exercises

This Hands-on lab includes the following exercises:

1. Page Navigation
2. Handling Back

Estimated time to complete this lab: **30 to 45 minutes**.

Exercise 1: Page Navigation

Page navigation is an important part of any non-trivial app. In this exercise, you will create a secondary view, navigate directly to the view using Frame.Navigate, and learn how to pass data on navigation.

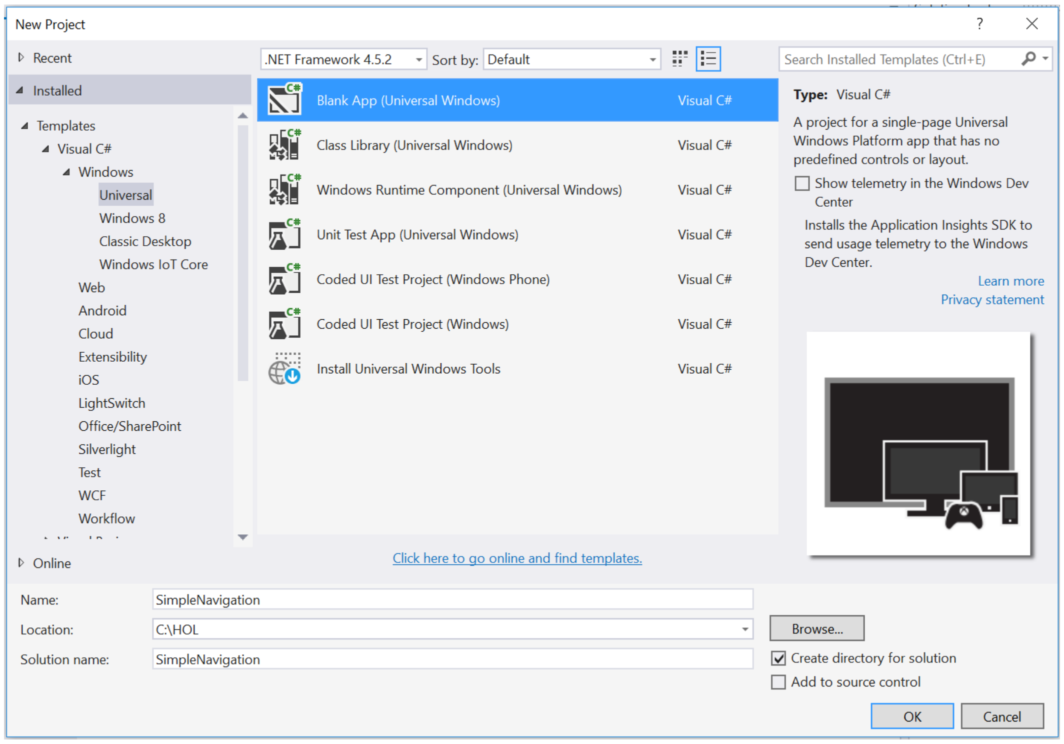
Task 1 – Create a blank Universal Windows app

We will begin by creating a project from the Blank App template.

1. In a new instance of Visual Studio 2015, choose **File > New> Project** to open the New Project dialog. Navigate to **Installed > Templates > Visual C# > Windows > Universal** and select the **Blank App (Universal Windows)** template.

2. Name your project **SimpleNavigation** and select the file system location where you will save your Hands-on Lab solutions. We have created a folder in our **C:** directory called **HOL** that you will see referenced in screenshots throughout the labs.

   Leave the options selected to **Create new solution** and **Create directory for solution**. You may deselect both **Add to source control** and **Show telemetry in the Windows Dev Center** if you don't wish to version your work or use Application Insights. Click **OK** to create the project.



Blank App (Universal Windows)	Visual C#
Class Library (Universal Windows)	Visual C#
Windows Runtime Component (Universal Windows)	Visual C#
Unit Test App (Universal Windows)	Visual C#
Coded UI Test Project (Windows Phone)	Visual C#
Coded UI Test Project (Windows)	Visual C#
Install Universal Windows Tools	Visual C#

Type: Visual C#

A project for a single-page Universal Windows Platform app that has no predefined controls or layout.

☐ Show telemetry in the Windows Dev Center

Installs the Application Insights SDK to send usage telemetry to the Windows Dev Center.

Name: SimpleNavigation

Location: C:\HOL

Solution name: SimpleNavigation

☑ Create directory for solution

☐ Add to source control

Figure 1

Create a new Blank App project in Visual Studio 2015.

3. Set your Solution Configuration to **Debug** and your Solution Platform to **x86**. Select **Local Machine** from the Debug Target dropdown menu.

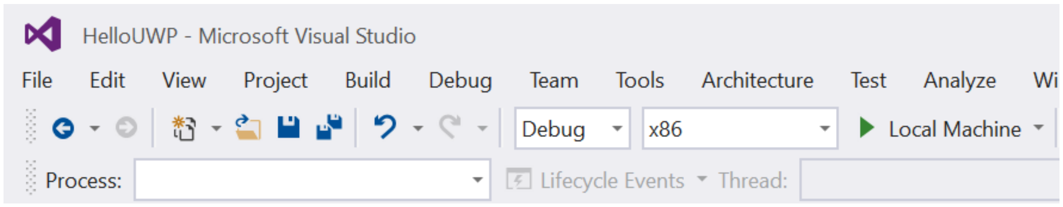


Figure 2
Configure your app to run on the Local Machine.

4. Build and run your app. You will see a blank app window with the frame rate counter enabled by default for debugging.

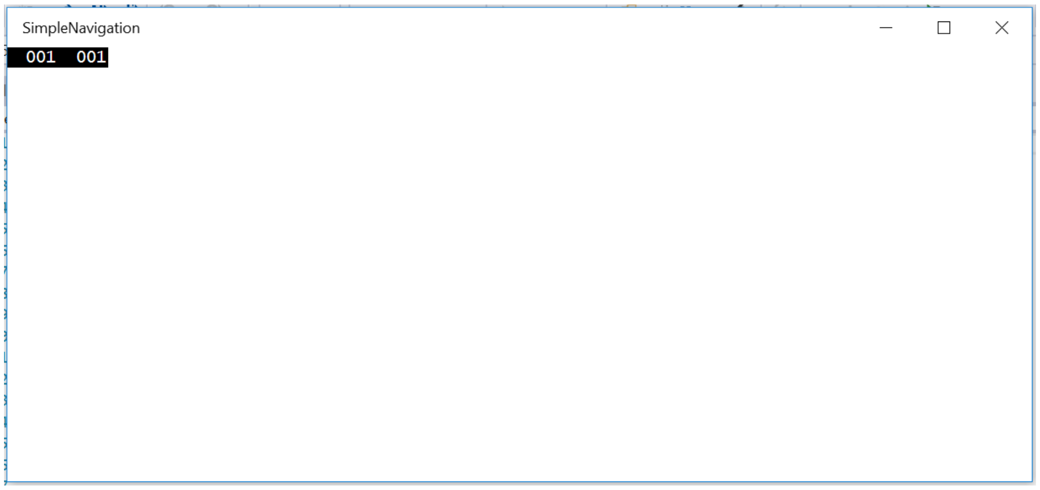


Figure 3
The blank universal app running in Desktop mode.

Note: The frame rate counter is a debug tool that helps to monitor the performance of your app. It is useful for apps that require intensive graphics processing but unnecessary for the simple apps you will be creating in the Hands-on Labs.

In the Blank App template, the preprocessor directive to enable or disable the frame rate counter is in **App.xaml.cs**. The frame rate counter may overlap or hide your app content if you leave it on. For the purposes of the Hands-on Labs, you may turn it off by setting **this.DebugSettings.EnableFrameRateCounter** to **false**.

5. Return to Visual Studio and stop debugging.

Task 2 - Create a view

Before you can introduce navigation, you will need to create a secondary view to which you can navigate from the MainPage of the app.

1. In your SimpleNavigation solution, right-click on the project name in the Solution Explorer and choose **Add > New Item**.

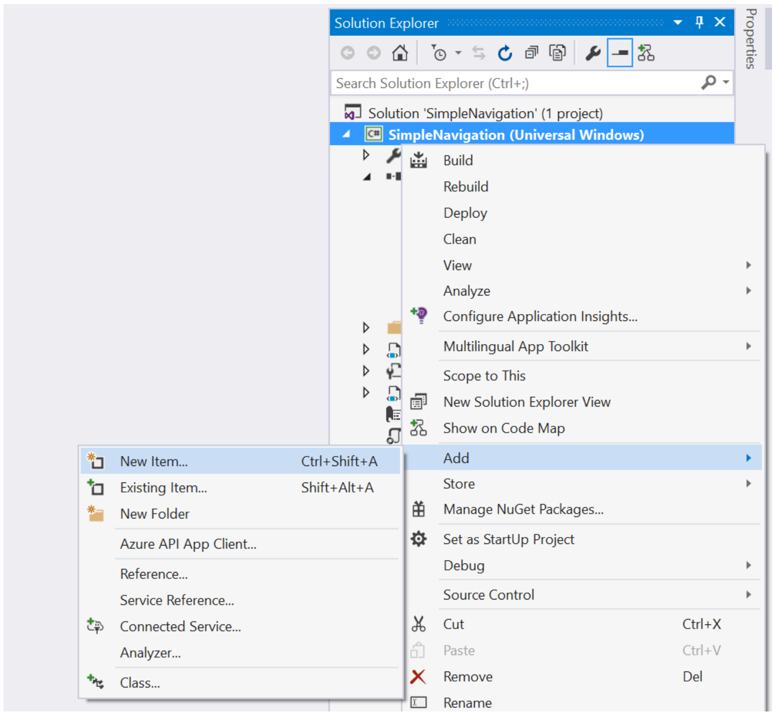
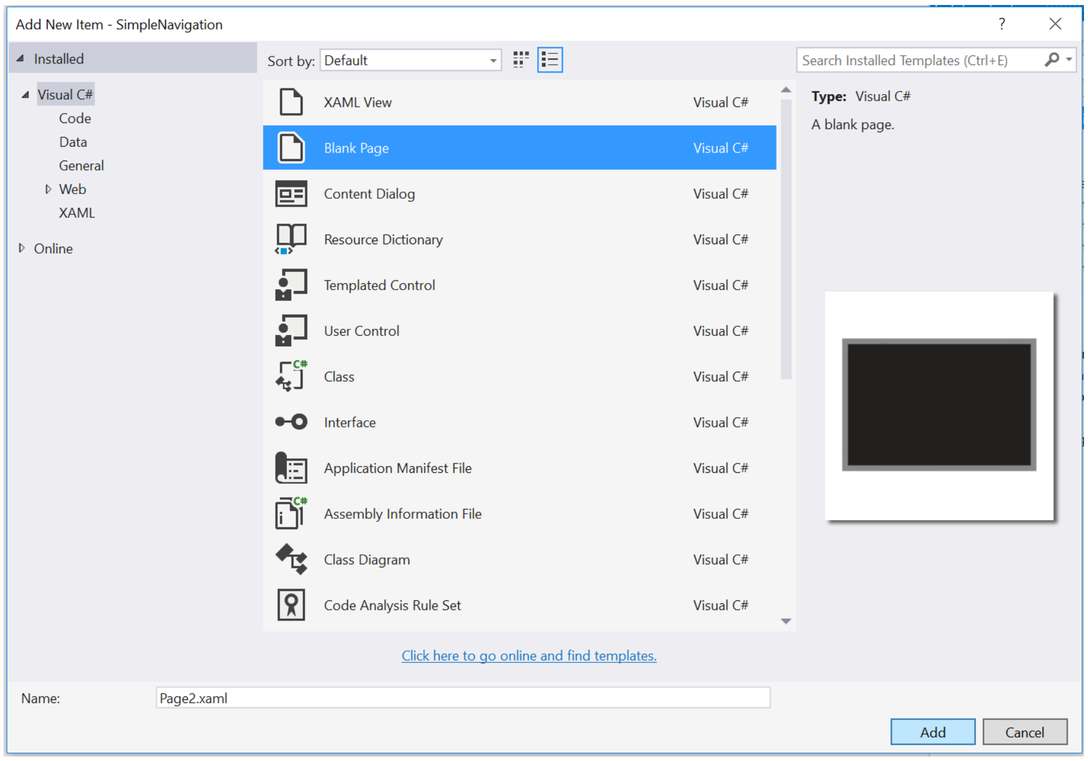


Figure 4

Add a new item in the Solution Explorer.

2. Select the **Blank Page** item type in the Visual C# item list. Name the item **Page2.xaml** and click **Add** to create it.


Add New Item - SimpleNavigation
Installed
Visual C#
Code
Data
General
Web
XAML
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
XAML View Visual C#
Blank Page Visual C#
Content Dialog Visual C#
Resource Dictionary Visual C#
Templated Control Visual C#
User Control Visual C#
Class Visual C#
Interface Visual C#
Application Manifest File Visual C#
Assembly Information File Visual C#
Class Diagram Visual C#
Code Analysis Rule Set Visual C#
Click here to go online and find templates.
Add
Cancel

Type: Visual C#

A blank page.

Name: Page2.xaml

Figure 5

Add an item of type Blank Page to the SimpleNavigation project.

Note: Do not select a XAML View by mistake! Although they sound similar, Blank Pages and XAML Views are not the same. A Blank Page includes both a XAML file and an associated code-behind file. A XAML View does not include code-behind. We will cover XAML Views in the Adaptive UI lab.

3. Open **Page2.xaml** and add a TextBlock to display the page title.

XAML

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Text="Page 2" FontWeight="Light" FontSize="24" Margin="12" />
</Grid>
```

Note: ThemeResource is a XAML markup extension that allows you to reference XAML styles defined elsewhere in a XAML Resource Dictionary. ThemeResource can dynamically use different resource dictionaries to reflect the user's active system theme at runtime. A StaticResource reference differs in that it does not update at runtime.

For more information on using the ThemeResource extension, visit https://msdn.microsoft.com/en-us/library/windows/apps/dn263118.aspx

4. Open **MainPage.xaml** and add a page title there as well. Our pages are simple, so it will be helpful to know where we are once we introduce navigation.

XAML

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Text="Page Navigation" FontWeight="Light" FontSize="24" />
</Grid>
```

Task 3 – Introduce navigation

Now that you have two views in your project, you can navigate between them. In-app navigation takes place in a frame, which acts as a container for your pages. When your app starts up, the root frame is built in App.xaml.cs and attaches to the window. The frame is important in the sense that it manages navigation between the pages. In this task, you will create a button on the main page to trigger navigation to Page2.

1. Open **MainPage.xaml** and add a button that will trigger navigation to Page2. Enclose the page title and the button in a StackPanel to improve the layout of the page.

XAML

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel HorizontalAlignment="Left" Margin="12">
        <TextBlock Text="Page Navigation" FontWeight="Light" FontSize="24" />
        <Button Content="Go to Page 2" Margin="0,12,0,0" />
    </StackPanel>
</Grid>
```

2. Add a click event to the button. You will create the **Button_Click** method to handle navigation in the next step.

XAML

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel HorizontalAlignment="Left" Margin="12">
        <TextBlock Text="Page Navigation" FontWeight="Light" FontSize="24" />
        <Button Content="Go to Page 2" Margin="0,12,0,0"
                Click="Button_Click" />
    </StackPanel>
</Grid>
```

3. Now let's add the **Button_Click** event handler to **MainPage.xaml.cs**. When you call **Frame.Navigate**, the frame loads the content of the specified page. It accepts as a parameter the type of page object to which you wish to navigate, and an optional second parameter to pass to that page. We will pass a parameter later on in this exercise, but for now we will navigate without it.

C#

```
public sealed partial class MainPage : Page
```

```
{
    public MainPage()
    {
        this.InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
       _Frame.Navigate(typeof(Page2));
    }
}
```

4. Build and run your app on the Local Machine. When you click the **Go to Page 2** button on the main page, your frame will navigate to Page 2. We haven't enabled back behavior, so you won't be able to return to MainPage yet. You will learn how to handle back behavior in the next exercise.

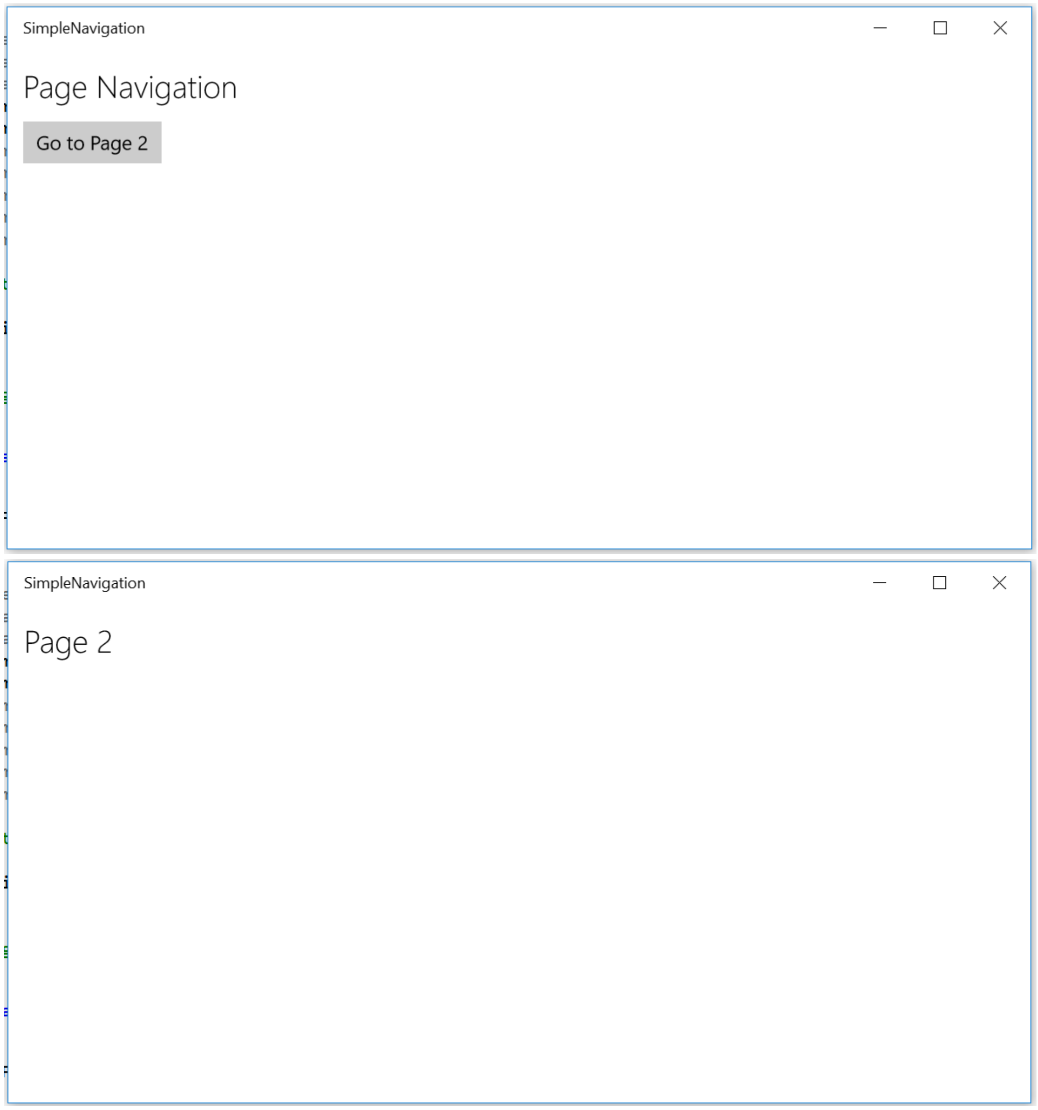


Figure 6
The ***Go to Page 2*** *button triggers navigation to another page in the app.*

5. Stop debugging and return to Visual Studio.

Task 4 – Pass a parameter to Page 2

You have successfully navigated between pages in your app. It is often useful to pass information to the new page on navigation. In this task, you will pass a parameter from the main page to Page2.

1. Add a **TextBox** to **MainPage.xaml**. This TextBox will accept user input that you will pass to Page2 for display later on.

XAML

```
<StackPanel HorizontalAlignment="Left" Margin="12">
    <TextBlock Text="Page Navigation" FontWeight="Light" FontSize="24" />
    <TextBox x:Name="Message" Header="Enter a parameter to send to Page 2"
Width="300" Margin="0,12,0,0" />
    <Button Content="Go to Page 2" Margin="0,12,0,0"
            Click="Button_Click" />
</StackPanel>
```

2. In your **Button_Click** handler in the code-behind, add the optional second parameter to **Frame.Navigate** to pass the TextBox text to Page2.

C#
```
private void Button_Click(object sender, RoutedEventArgs e)
{
   Frame.Navigate(typeof(Page2), Message.Text);
}
```

Note: The optional parameter you pass into **Frame.Navigate** does not have to be a string (it is an object in the API), but it must be serializable. An object that is serializable can be converted to a stream of bytes for storage in order to save its state and recreate that state later on. The frame keeps track of the app history and navigation parameters, which it needs to resume from suspend, for instance, and it does so by serializing the parameters. If you wish to pass a parameter that is not serializable, you may write code to serialize it yourself for advanced use cases.

Task 5 – Display the message passed to Page2

Your message will now be passed as a parameter when you navigate to Page2. However, we haven't done anything to handle the message in Page2, so it won't yet display. Let's add a pop-up dialog to display the message.

1. Open **Page2.xaml.cs**. Create an override for the **OnNavigatedTo()** method.

C#
```
public Page2()
{
    this.InitializeComponent();
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
}
```

Note: The page constructor may not be called every time you navigate to a page if the page has already been loaded. The **OnNavigatedTo()** method is called every time you navigate to a page, so we can use it to consistently trigger the display of the incoming message.

2. Add the **Windows.UI.Popups** namespace to the **Page2** code-behind.

C#

```
using Windows.UI.Popups;
```

3. Await a message dialog in the **OnNavigatedTo()** override and pass in the incoming parameter. You will need to add the **async** keyword to the method to handle the await.

C#

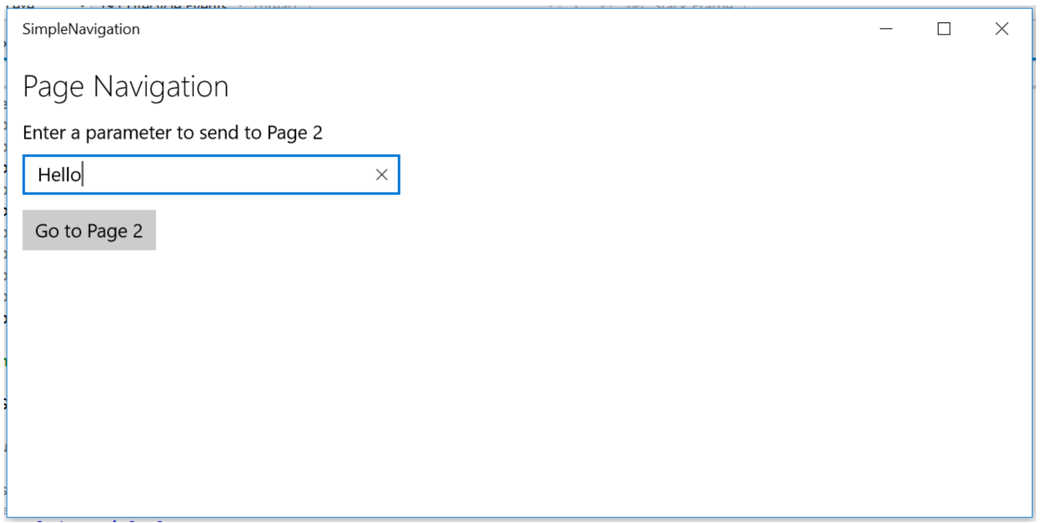
```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    await new MessageDialog("You sent: " + e.Parameter).ShowAsync();

    base.OnNavigatedTo(e);
}
```

Note: Asynchronous methods allow the application to continue execution without having to wait for operations that may potentially block the UI thread. The UI will continue to respond to the user while the async operation runs in the background.

An asynchronous method typically has a name that ends in Async by convention and returns **Task** or **Task<T>**. When called using the **await** keyword, the runtime suspends the containing method and returns control to the caller along with a task value. Any methods that call functions using the await keyword must be marked with the **async** keyword. A task typically runs asynchronously on a thread pool thread instead of the main application thread. When an async method is complete, its associated task is marked as complete, and any return values can be accessed through the task.

For more information and examples of asynchronous programming patterns, visit https://msdn.microsoft.com/en-us/library/hh191443.aspx

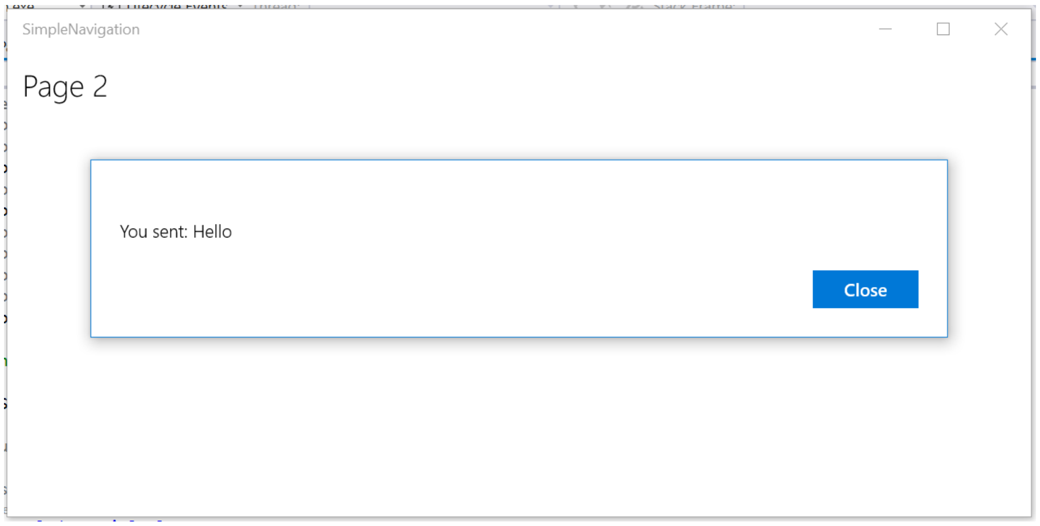
4. Build and run your app. Type a message into the TextBox on the MainPage and navigate to Page2. You will see a pop-up appear on Page2 with your message.


SimpleNavigation

Page Navigation

Enter a parameter to send to Page 2

Hello


SimpleNavigation

Page 2

You sent: Hello

Figure 7

The user-generated message is passed to Page2 as a parameter.

5. Stop debugging and return to Visual Studio.

Exercise 2: Handling Back

UWP apps run on a range of devices which differ in how they handle back navigation. Whereas Windows 10 Mobile devices typically provide a hardware back button, tablet and desktop devices usually do not. There are several ways to handle back navigation within your app. In this exercise, you will explore the differences in navigation between device families and use the shell-drawn back button to return to the main page from a secondary page.

Task 1 – Test the hardware back behavior

Let’s take a look at the existing behavior of the hardware back button on Mobile.

1. Change your Debug Target to a Mobile Emulator. We chose the **Mobile Emulator 10.0.10240.0 720p 5 inch 1GB**.
2. Build and run the SimpleNavigation app in the emulator. Navigate to Page2. While on Page2, click or tap the emulator’s hardware back button. Instead of returning to the main page, the app will exit and return you to your last position in the app back stack. In this case, if you just started the emulator, the last position will be the start screen.
3. Open the Photos app from the start screen.
4. Click and hold the hardware back button on the emulator to view open apps. Use this feature to return to your Simple Navigation app.

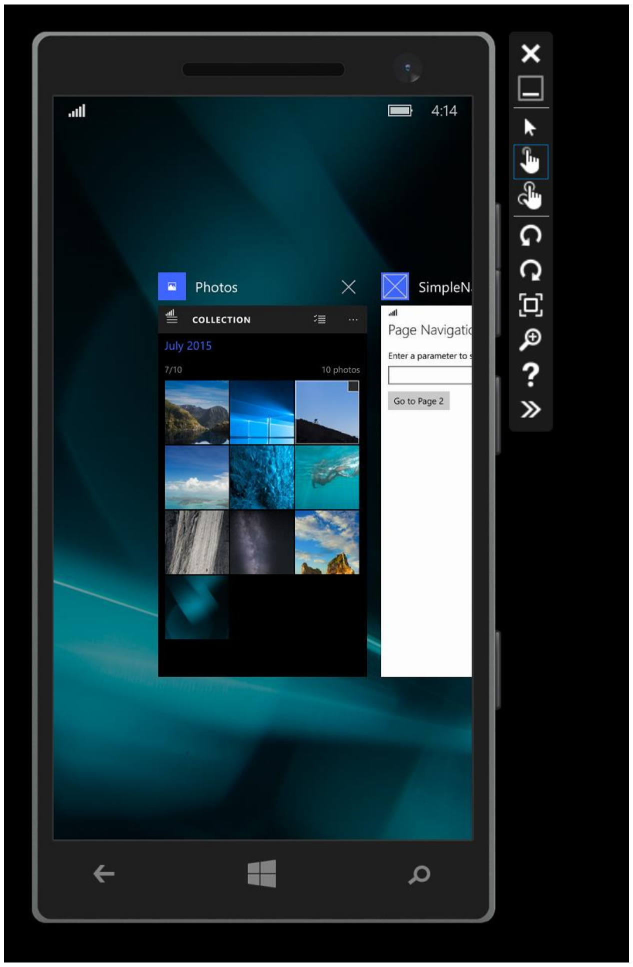


Figure 8
Click and hold the back button to navigate between open apps.

5. From the Simple Navigation app, click the back button again. You will return to the previous app in the app stack, which is the Photos app, and not to the main page of the Simple Navigation app as you might expect. Unless you implement custom back behavior, the hardware back button will navigate through the app stack by default.
   You will implement code to handle custom back behavior in **Task 3** of this exercise.

Task 2 – Enable the shell-drawn back button

Windows 10 provides a Back button in the task bar when in Tablet mode, but by default, no Back button is provided when running in Desktop mode. The **SystemNavigationManager** provides options to enable the shell-drawn **AppViewBackButton** for apps running in Desktop mode. This shell-drawn back button displays in the app's title bar when in Desktop mode. If you include code to show the shell-drawn back button, it is ignored on Mobile, because a hardware back button is expected, and is ignored when running in Tablet mode. In this task, you will display the system-provided back button in desktop mode when the user is on a page within the app where backwards navigation is possible.

1. Open **App.xaml.cs**. Add the **Windows.UI.Core** namespace.

C#

```csharp
using Windows.UI.Core;
```

2. Add a **rootFrame.Navigated** event handler at the end of the **OnLaunched** override. This event handler will fire every time there is navigation on the root frame and will display the app view back button if the app back stack isn't empty.

C#

```csharp
    // Ensure the current window is active
    Window.Current.Activate();

    rootFrame.Navigated += RootFrame_Navigated;
}

private void RootFrame_Navigated(object sender, NavigationEventArgs e)
{
      Frame rootFrame = Window.Current.Content as Frame;
      SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility
          = rootFrame.CanGoBack ? AppViewBackButtonVisibility.Visible :
                                  AppViewBackButtonVisibility.Collapsed;
}
```

Note: We only wish to show the shell back button when there is something in the back stack. This code sets the AppViewBackButtonVisibility to AppViewBackButtonVisibility.Visible if rootFrame.CanGoBack is true; otherwise, it sets it to AppViewBackButtonVisibility.Collapsed.

3. Change your Debug Target to **Local Machine**. Build and run your app and navigate to Page2. The back button will appear in the title bar in Desktop mode.

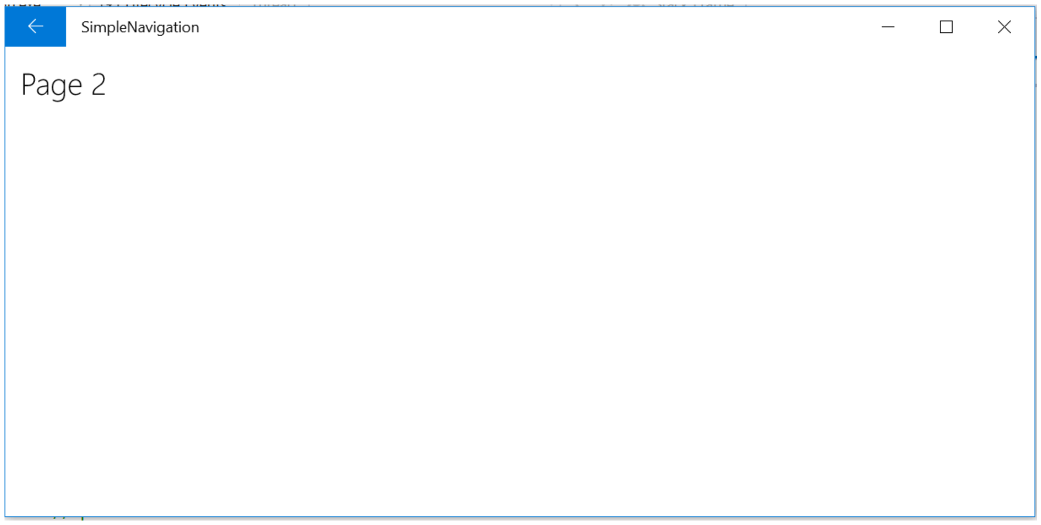


Figure 9
The AppViewBackButton is visible in the title bar in Desktop mode.

4. Click the back button in the title bar. Nothing will happen. Although you enabled its visibility, it is not yet hooked up to handle back navigation.

Note: The color of the shell-drawn back button is defined by the user's selected accent color, which the user can change in **Windows 10 Settings > Personalization > Colors**. If you wish to control the placement or style of the back button, you can instead implement your own back button control within the app, as you will do later in this lab.

5. With your app still running, use the Windows 10 Notifications panel to switch into **Tablet** mode. The back button will appear in the task bar.

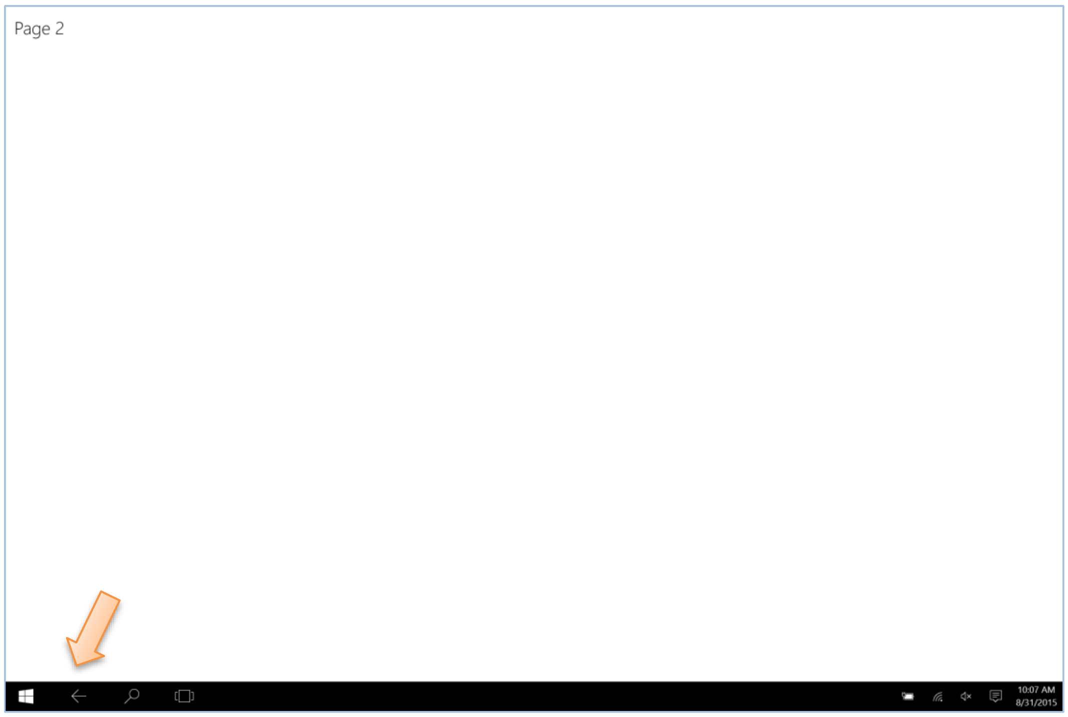


Figure 10
The back button is visible in the task bar in Tablet mode.

6. Click the back button in the task bar. You will be taken back in the app stack. Unlike the button in the title bar, the back button in tablet mode has similar behavior to the hardware back button on Mobile. This back button displays even if the shell back button in the title bar is not enabled.

Note: In Split Screen, there is a back stack available for each side of the screen.

7. Use the Notifications panel to return to Desktop mode. Stop debugging and return to Visual Studio.
8. Run the app again, this time in the Mobile emulator. When you navigate to Page2, the shell back button will not appear, because it is ignored on Mobile.

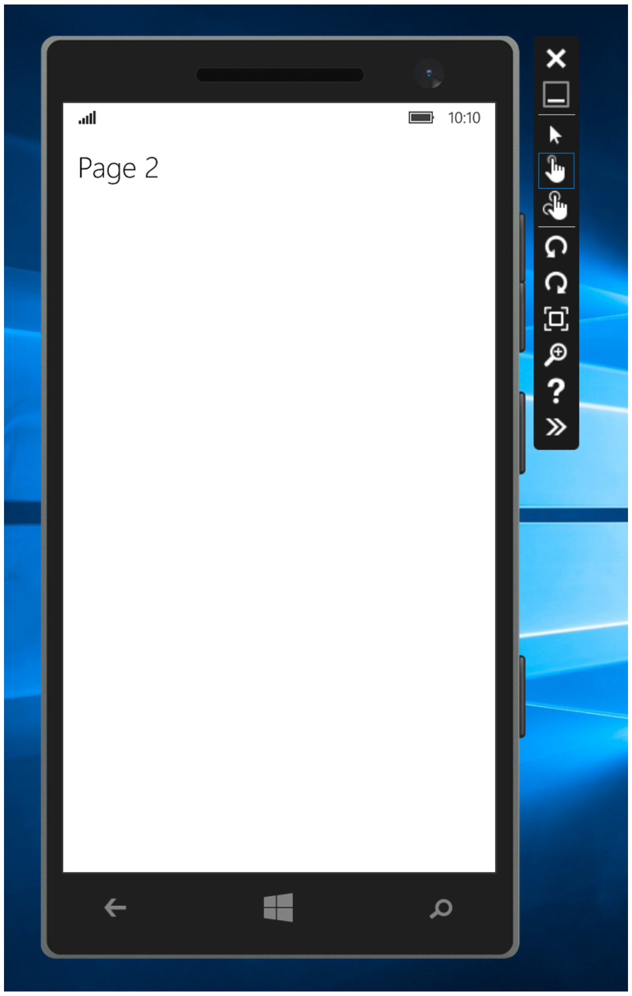


Figure 11

The AppViewBackButton is hidden on Mobile.

9. Stop debugging and return to Visual Studio.

Task 3 - Define the standard back-requested pattern

We now have Back UI available when your app is running on Windows 10 Mobile, and on Windows 10 Desktop in both desktop and tablet modes. Now let's define the standard back requested pattern to handle back requests.

1. In **App.xaml.cs**, create and subscribe to an **App_BackRequested** event.

Note: The subscription to **App_BackRequested** must take place after the view has been created. If you try to execute this code before the view has been created, SystemNavigationManager.**GetForCurrentView()** returns **null**.

C#

```
    SystemNavigationManager.GetForCurrentView().BackRequested +=
        App_BackRequested;

    rootFrame.Navigated += RootFrame_Navigated;
}

private void RootFrame_Navigated(object sender, NavigationEventArgs e)
{
   _Frame rootFrame = Window.Current.Content as Frame;
    SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility =
        rootFrame.CanGoBack ?
           AppViewBackButtonVisibility.Visible :
           AppViewBackButtonVisibility.Collapsed;
};

private void App_BackRequested(object sender, BackRequestedEventArgs e)
{
}
```

2. In your **App_BackRequested** event handler, check to see if the BackRequested event has already been handled. If not, set the default behavior to navigate back within the frame. Make sure to set **e.Handled** to **true** when done.

C#

```
private void App_BackRequested(object sender, BackRequestedEventArgs e)
{
    // Check that no one has already handled this
    if (!e.Handled)
    {
        // Default is to navigate back within the Frame
        Frame frame = Window.Current.Content as Frame;
        if (frame.CanGoBack)
        {
            frame.GoBack();
            // Signal handled so the system doesn't navigate back
            // through the app stack
            e.Handled = true;
        }
    }
}
```

3. Build and run your app. Navigate to **Page2**, then use the shell back button to return to the main page.

4. Stop debugging and return to Visual Studio.

Task 4 – In-app back button

If you would like to have more control over the styling and behavior of the back button, you can create your own button to handle back navigation.

1. Comment out the visibility of the shell back button **in App.xaml.cs**.

   C#

   ```
   //rootFrame.Navigated += RootFrame_Navigated;
   ```

2. Add a button to **Page2.xaml** and assign its style to the StaticResource **NavigationBackButtonNormalStyle**. Enclose the button and the page title in a horizontal StackPanel for alignment. Replace the margin on the title TextBlock with **VerticalAlignment="Top".**

   XAML

   ```
   <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
       <StackPanel Orientation="Horizontal">
           <Button Style="{StaticResource NavigationBackButtonNormalStyle}"
            VerticalAlignment="Top" />
           <TextBlock Text="Page 2" FontWeight="Light"
                      FontSize="24" VerticalAlignment="Top" />
       </StackPanel>
   </Grid>
   ```

3. Add a click event to the button and bind it to a handler called **GoBack**. You will create the **GoBack** handler in the next step.

   XAML

   ```
   <Button Style="{StaticResource NavigationBackButtonNormalStyle}"
           Click="GoBack" VerticalAlignment="Top" />
   ```

4. In the Page2 code behind, add the GoBack() function to handle the back event.

   C#

   ```
   private void GoBack(object sender, RoutedEventArgs e)
   {
        if (Frame.CanGoBack)
           Frame.GoBack();
   }
   ```

5. Build and run your app. When you navigate to Page2, you will see your custom back button instead of the shell back button. The custom button will return you to the main page.

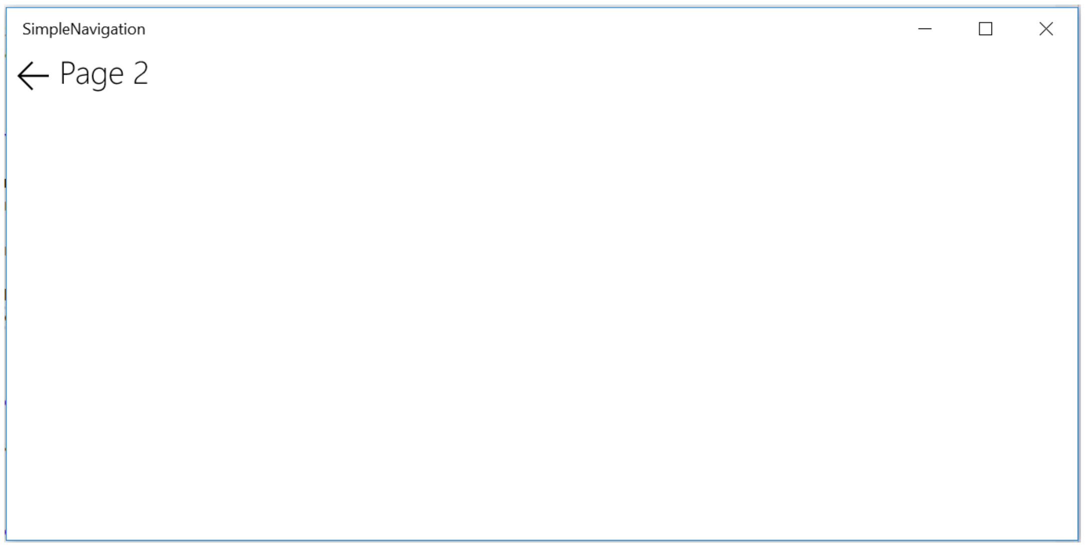


Figure 12

The custom back button on Page2.

6. Stop debugging and return to Visual Studio.

Task 5 – Control the visibility of the in-app back button

Let’s make the custom back button visible only when there is something in the back stack.

1. Add a private **Visibility** field to the Page2 code-behind.

C#

```
public sealed partial class Page2: Page
{
    public Visibility CanGoBack;
```

2. In the Page2 **OnNavigatedTo** override, check if the frame can go back and set the Visibility field accordingly.

C#

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    if (Frame.CanGoBack)
        CanGoBack = Visibility.Visible;
    else
        CanGoBack = Visibility.Collapsed;

    await new MessageDialog("You sent: " + e.Parameter).ShowAsync();
}
```

3. Bind the visibility of your back button to the **CanGoBack** field in **Page2.xaml**.

XAML

```
<Button Style="{StaticResource NavigationBackButtonNormalStyle}"
        Click="GoBack" Visibility="{x:Bind CanGoBack}"
        VerticalAlignment="Top" />
```

4. In **App.xaml.cs**, set the start up page for your app to **Page2** instead of **MainPage**.

C#

```
if (rootFrame.Content == null)
{
    // When the navigation stack isn’t restored navigate to the first page,
    // configuring the new page by passing required information as a
    // navigation parameter
    rootFrame.Navigate(typeof(Page2)), e.Arguments);
}
```

5. Build and run your app. The custom back button will not appear on Page2, because there is nothing in the back stack.

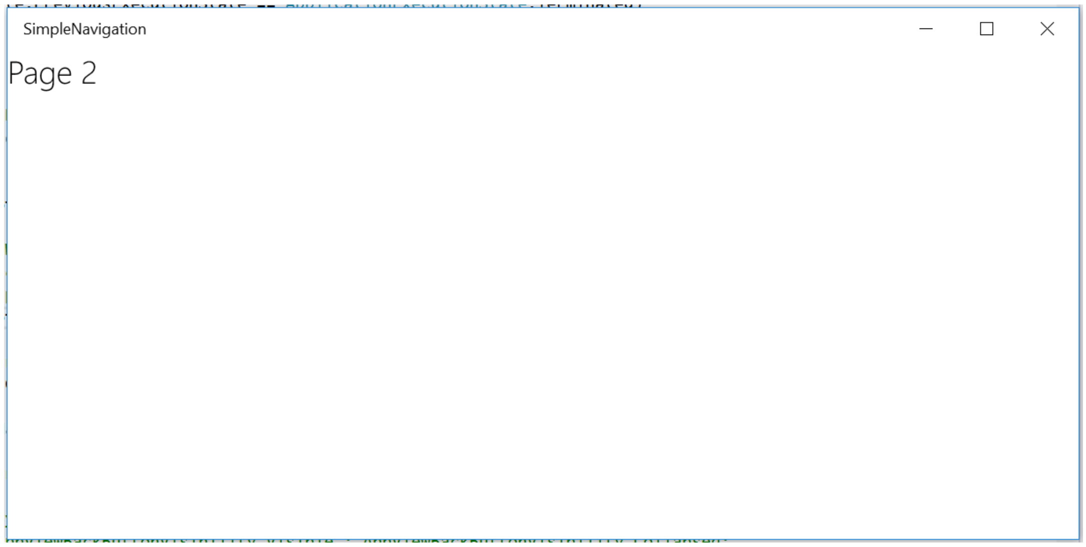


Figure 13

The custom back button does not appear when the back stack is empty.

6. Stop debugging and return to **App.xaml.cs**. Change the start up page back to **MainPage**.

C#

```
if (rootFrame.Content == null)
{
    // When the navigation stack isn’t restored navigate to the first page,
    // configuring the new page by passing required information as a
    // navigation parameter
    rootFrame.Navigate(typeof(MainPage)), e.Arguments);
```

```
}
```

7. Build and run your app. Navigate to Page2. The custom back button will appear again now that the back stack is populated.

8. Stop debugging and return to Visual Studio.

Summary

In this lab, you learned about frames, page navigation, and back behavior within a UWP app. Although navigation options vary across devices, you created solutions to provide a consistent navigation experience on Desktop, Tablet, and Mobile.