So let’s get started! I have something setup already. It has a lot of stuff already built in but not yet active.

Step 1: Bring in the nuget packages. Just do a search for prism.unity.

* Currently it is at 6.1.1 for uwp
* Bringing in this will bring in all of the dependent assemblies as well, one-stop shopping
* Notice for this, we are using the Unity Container. There are other choices, this just happens to be the one that I like to use and am familiar with

Step 2: Override app from PrismUnityApplication

* Very first thing we are going to do in the code is to change the application object
  + This is the entry point of UWP applications, think of it as Main
  + There is a lot of boiler plate code that you will need for any app that you write, and prism can do that for you.
  + Things like navigation, session state service, gestures and others
  + You will also notice that in the app.xaml that there is already some stuff added for supporting hamburger menus, but this isn’t really active yet
* Let’s start coding
* In App.xaml, change the class declaration
  + Add prism.unity.windows namespace so we can reference the base class
  + <prism:PrismUnityApplication ….
* Go to the app.xaml.cs code behind
* Get rid of everything except for the constructor
* In the constructor, get rid of the onsuspending
* A couple of things you always have to do is override:
  + OnLaunchApplicationAsync: main entry point. One of the most important things that this method is going to do is show the correct starting page. For the purposes of this demo, we are just going to navigate to the main page.
  + OnInitializeAsync: this is where you initialize your services, repositories etc. For this app, we are going to start with giving it a resource loader so we can localize later if we want. Notice that it is being registered with the container so that it can be instantiated with any object that needs it.
  + Also, a little gotcha: the container namespace needs to be added for this code to be compiled propertly.
    - Using Microsoft.Practices.Unity;
* Let’s compile and run. First thing that happens is we get a crash when we try to have it navigate to the main page.

Step 3: Talk a bit about navigation:

* Navigation in a UWP app is handled by the main frame object. In the default app template, that frame is the frame around the entire window, and all pages display with in the entirety of the window.
  + Normally with navigation, you need to know the type of the page that you want to navigate to and you provide that to the Frame.Navigate method
  + The above method will try and create the page based on that type and pop it into the frame.
  + Right away, you can see that you need a reference to the frame object everywhere you perform any kind of navigation. Your business logic will need to know about the UI objects.
  + This makes your app tightly coupled and harder to maintain.
* Since Prism creates the frame in the PrismUnityApplication class, it also takes care of wrapping up all the navigation calls associated with the frame into an abstracted service and registering it in the container. It was designed to work with the viewmodels and the session state service as well, but we will cover those a bit later. You can see the result of that work as the “NavigationService” object in the OnLaunch method.
  + Instead, the Prism NavigationService uses a string value to determine what page to navigate to. If you tell it to navigate to “Main”, it will use reflection to look in the “Views” namespace for a page of type “MainPage”.
  + With this type of system, your business logic has no idea of what it is navigating to and is not coupled to it. What’s more, you can easily do things like use a completely different page by just changing the value of the string token and the rest of the app will never know.
  + Since the NavigationService is in the container, it can be instantiated pretty much anywhere. Most importantly, it can be instantiated with your app “business” logic and the business logic will have no concept of how that navigation is happening.
* Getting back to those conventions, it can do the same thing with the viewmodel for the newly created page.
  + Using the above example, if you specify “Main” it will look in the Views namespace for the MainPage. It will also look in the “ViewModels” namespace for the MainPageViewModel and inject it into the data context of the page.
* Compile and run:
  + Add regular blank page called MainPage to “Views”
  + Run again, yay!

Recap: we have created a Prism UWP app and shown how we can navigate. Next we are going to do our first viewmodel

Step 4: First view model

* So now we can attach a view model to the page
* In keeping with the Prism convention, we will create it in the ViewModels namespace.
* Create a MainPageViewModel class
  + Copy it in
  + We can see we are deriving from AppViewModelBase (already in the project).
    - This just sets up a number of app specific helpers and services.
    - For example, every view model gets a reference to the navigation service so that it can move to a different page; a service for loading resource strings; a couple of other services that most of the viewmodels will need.
    - You can see that our base class inherits from ViewModelBase. Follow it down far enough and you will see that it derives from INotifyPropertyChanged.
* Modify the MainPage to use the view model so that it can get the viewmodel automatically
  + Derive from SessionStateAwarePage
    - Prism defined page.
    - Provides some default behaviour like delegating navigation events to the attached view model
  + Setup the AutoWireViewModel
    - This is an attached property. It acts like it is part of the class, but it is actually something prism provides.
    - When the page is being constructed, this property is checked. If it set to true, it tries to determine the correct ViewModel and inject it into the DataContext of the page.
    - You get this for free!
  + Put in the code-behind
  + Recompile and run
  + Show how the viewmodel works, put a breakpoint in the setter for TestText, change the text value
    - Viewmodel knows nothing about presentation, only knows the data
* Recap: we have setup our application, and without doing much of anything, we already have a good architecture setup, we have a navigation system setup, and though we haven’t really used these yet, we can handle some gestures, session state, localization and loose communication between objects.

Step 5 – hamburger time

* Great we have the start of an app, and this is a good spot to setup the hamburger menu.
* Bit of a bigger topic, and some of the stuff I have already setup in the app:
  + Resources for the presentation of the hamburger button, presentation of the menu items, and the splitview
  + AppShell – this will be the new main window for the app, contains the splitview
  + MenuItems and MenuViewModel, supports the commands in hamburger menu
* Take a look at resources first
  + SplitViewStyles.xaml: just a bunch of setup that sets up how the splitview toggle button and menu text styles look
  + All presentation stuff
  + How the ui looks under different states: pressed, pointer over, disabled, etc.
* Take a look at AppShell
  + If you have been at any of Mark’s presentations, you will recognize the responsive xaml
    - Responsive stuff is just used to define whether the menu is hidden, buttons only or buttons and text
  + You can see we setup a user control in the pane for hosting the menu items
  + And we setup an empty Frame as the content portion of the split view.
    - This is going to be main frame of the app where all navigation happens
    - In the code behind, setup a couple of helper functions for the menu pane and content.
* Take a look at the MenuView
  + Split into two sections, commands, secondary commands
    - I put the main navigation app functionality in the top collection and things like settings and about in the bottom section.
  + Just items controls. Obviously use item controls so you can dynamically add menu items.
  + Look at the xaml, we can see we have the viewmodel being automatically injected like we did with the main page. Talk a bit more about it:
    - AutoWireViewModel property controls it.
    - DependencyProperty that calls the ViewModelLocationProvider. Unless you override the behavior, it will use a convention based approach to find the appropriate viewmodel for a given view.
    - Unsurprisingly, the ViewModelLocationProvider is setup to use the Container to actually create the viewmodel objects. All of this happens for free just for using the application.
  + DataTemplate is used to define the menu item control. This is the presentation of the buttons in the hamburger menu.
  + Code behind has some of the databinding setups in there as well. All of this stuff does is make sure that all of the bindings in the UI are updated with values from the view model once the viewmodel is injected.
* Look at menuviewmodel
  + Creates all of the menu items that are stored in the items control.
  + Interestingly, these are all viewmodels in their own right
* Look at MenuItemViewModel
  + Has all the parameters to support navigation
    - Knows the page to go to (PageToken)
    - Knows what text to display (loaded from the resources)
    - Knows the icon
    - WE saw databinding from the very first view model that we made: we bound a string object to the text property of a text box.
    - Buttons have a command property that contain an implementation of the ICommand interface. The command property is like the “Click” or “Tapped” event in the code behind.
    - Prism has a nice class that implements the ICommand interface and that is what you are seeing where the commands are created and added to the collections.
    - In the case of each of the buttons, we just navigate to the page specified by the page token.
    - It also keeps track of the state of the UI and if the current page and the button page token match, the button is disabled.
    - These are managed by helper functions so that it is easy to add more.
      * NavigateInternal uses the navigation service to navigate to the appropriate page and updates the UI
      * CanNavigateInternal checks to see whether the button should be enabled or not.
* So now we have the basics of the hamburger menu in place. Now we have to create it and instantiate it into the prism system.
* We are going to head back to the PrismUnityApplication object
* Much of the functionality in prism is set to some default behavior, but pretty much everything can be overridden as needed, and we are going to redefine how it creates the shell for the application
* Override CreateShell
  + This function creates the main shell of the application.
  + Normally just creates a frame object and your app is happy to navigate to the first page
  + Instead, we create a complete page using the AppShell
  + Inject the rootFrame object into the content portion of the splitview using the helper function
  + The reason we did this, is because normally the Frame object is the entire window area. But instead we want all navigation to happen within the context of the splitview content pane. Injecting the frame into the splitview content area keeps the hamburger menu constant regardless of the sizing of the app.
  + Compile and Run
    - All of the hamburger commands are there
    - Resize the window for responsiveness
    - You can see that we are still showing the MainPage object that was setup in the OnLaunch
    - Can also see that the button that represents the mainpage is disabled since we are on that page
    - Toggle the hamburger
    - Try to navigate : CRASH: no page

Step 6 – lets add some simple pages

* First we are going to add a couple of simple pages.
* Let’s add the about and settings page
* What I like to do is get the View Models in place first. Let’s do the about screen and settings screen as they are just place holders.
* Already have page tokens setup for them so check that: remember, these are the values that the navigation system uses. Add a SettingsPageViewModel and AboutPageViewModel
* Just add Title property to each to provide some data binding
* Let’s create an AboutPage and a SettingsPage in the Views folder
  + Addin the SessionStateAwarePage and INotifyPropertyChanged for the pages
  + Implement INotifyPropertyChanged
  + Add in the ConcreteDataContext and DataContextChanged event handlers
  + All the same code that we have created before.
* Compile, Run, Yay!
* Use the menu buttons to navigate between the main, about and settings
* Use the back button to navigate back through the stack
  + Everything looks great … but wait!
  + When you use the back button, the MenuViewModel isn’t getting notified because the back button is handled at a much lower and automatic level.
  + Need to notify the MenuViewModel somehow
  + Show the MenuViewModel class
  + Here is where we can use the IEventAggregator object
  + Subscribe to the UpdateNavigationMessage and create a handler for the message
    - Show the message object, it just makes a message out of a string, in this case the page token.
  + Whenever the message is received, just update the button status with the value of the page token
    - Sets the current page in the menu
    - And updates each of the buttons by looping through the list
      * Update is performed by each button calling its CanExecute method which was defined when each button was constructed.
      * It just checks to see if the button corresponds to the current page: yes, returns false, can’t use the button, no, returns true, can use the button
  + So that’s great, how do we get the message?
  + Since we derived from SessionStateAwarePage, it has some helper functions for knowing when pages are navigated from and to
  + It delegates these events to the baseviewmodel in prism, and we can override those events.
  + Part of the parameter is the page type. All we do is get the type name of the page and subtract “Page” from the type name and we have the token that we have been using for all of our navigation.
  + Then we use the UpdateNavigationMessage to publish the new page token and ….
  + Some of you might have noticed that the update call is doubled up when using the buttons: called once when the button is tapped, and once when the navigation is handled in the viewmodelbase. Just do a quick check to see if the target page and current page are the same so that we don’t raise the menu stuff again.

Recap: we have created a hamburger menu, added buttons and even some pages. The menu can navigate between pages and automatically updates itself based on its state.

Home stretch

Step 7– let’s add some more stuff

* Just copy in the list, add, delete and update pages and viewmodels
* Copy in the services
  + Todostorage service is there
* Compile, run, go to list page … CRASH
  + Unresolved dependency
  + In the view model, we have the ITodoStorageService being injected, but we have never registered it
  + Let’s do that now
  + App.xaml.cs
    - Declare the storage service
    - In OnInitializeAsync register with container
      * Container.RegisterInstance<
      * Also do IMessageDialogService … I know we need it. In this case, we will ask the container to create a new instance each time it is needed using Container.RegisterType<
      * Compile run
  + Now we are getting a bit of functionality in the app
  + List and delete aren’t showing anything because we haven’t created anything
  + Other pages are showing ui

Step 8 – let’s dive in

* Create a todo click on add,
* Show how add button is disabled if headline is blank
* Add text, it’s enabled, remove text, disabled.
* Add some date, click on add
* goes back to list screen
* let’s look
  + if we look at the xaml, you can see all the ui there
  + Look in the code-behind, there is really nothing. No click or text changed events that you normally see in winforms. Functionality is in the viewmodel
  + Storage service is injected
  + You can see the that the headline property is there, notice that when the property is set, it updates the buttons in the ui by calling RaiseCanExecuteChanged
    - All this does is get the UI to call the CanExecute function that was defined
    - It just checks to see if there is a value in the headline
  + Once you can click the button, it just calls the Execute method
    - Bundles the data and calls the storage service
    - If successful, navigates back to the main page
    - NavigationService was injected into the viewmodel
    - Viewmodel doesn’t know anything about how the navigation happens

Step 9 – Small step, let’s load and save data

* Right now the data isn’t being saved anywhere.
* In the OnInitializeAsync method, load from storage
* Override OnSuspendingApplicationAsync to save to storage

Step 10 – Finally let’s look at state

* Prism has it built in (mostly)
* Want data to be restored when moving back and forth from the different pages
* Let’s try it out in the add page
* Go to the add page viewmodel
* Add restorablestate attribute to the headline property
* Run the app, add some text to the headline property
* Go to another page, use the BACK button to navigate back
* Yay
* Now let’s use the menu buttons to navigate and we see that the restorestate doesn’t seem to work, but use the BACK button to navigate, eventually it is restored.
* Revolves around the way the navigation system works. It treats page navigation like a stack of cards and the BACK button is used to pop the state off of their stack
  + Prism uses a forward navigation style system
  + Even though we are going to the same page each time, Prism views it as a completely new page.
* So, going to change the navigation service to handle the restore state in a more hamburger friendly style
* Make a copy of the FrameNavigationService
* Make some changes to the NavigateToCurrentViewModel, NavigateFromCurrentViewModel
  + State is stored in a dictionary object where the key is just a sequential index
  + Change it so that the key is name of the viewmodel class instead.
* Override OnCreateNavigationService in app.xaml.cs
  + Use the default GetPageType function, SessionStateService
  + Run the app, show the sessionstate now working