Contents

- What's the Problem?
- Enabling automatic session restore
- Saving and loading state
    - Page local state and globally shared state
- Serialization requirements
- Limitations

App suspension, termination and resuming is a necessary evil in Windows Store apps. Punch attempts to make the task of storing state information in local storage and restoring it as transparent as possible to the developer and end user.

What's the Problem?

The app lifecycle for Windows Store apps is fundamentally different from traditional Desktop apps. From the Desktop world, we are used to multiple applications running simultaneously and staying in memory until they get terminated by the end user.

This is different for Windows Store apps. In order to extend battery life, apps get suspended when the user switches to another app. Suspension means that the app no longer gets any CPU cycles, but stays in memory. This is important to understand as suspension alone is not a big deal unless your application is expected to do work in the background while the user is doing something else in a different app. Windows Store apps have background tasks to deal with this, but we are not going to explain background tasks in this section.

Besides suspending an app, Windows when resources run low may also decide to terminate the app and evict it from memory. This is where things become difficult for the developer. If termination goes unhandled, users will lose all their pending work and instead of continuing where they left off, they will have to start over next time they activate the app. I think we can all agree that this is not a good user experience.

So, we need to be prepared to save the user's pending work somewhere and restore it after the app is loaded back into memory. That's easier said than done as in larger apps, state information is generally spread across the entire app. Each view model has state, the app itself has state such as the logged in user, navigation history etc.

Enabling automatic session restore

This feature requires Punch v2.4 or higher.

Now that we understand the challenge, what can we do about it? The good news is that Punch takes care of all the heavy lifting associated with saving session state and restoring it when necessary. In this section we are going to explain how to take advantage of this feature.

Automatic session restore is turned off by default and must be enabled in the Application class in order to function. To enable it, simply make a call to *EnableAutomaticSessionRestore()*. The call is best made in the constructor as demonstrated in the following example.

```
sealed partial class App : CocktailMefWindowsStoreApplication
{
  public App() : base(typeof (ListPageViewModel))
   {
      InitializeComponent();
      EnableAutomaticSessionRestore();
   }
  protected override void StartRuntime()
   {
     base.StartRuntime();
      IdeaBladeConfig.Instance.ObjectServer.RemoteBaseUrl = "http://localhost";
      IdeaBladeConfig.Instance.ObjectServer.ServerPort = 57209;
      IdeaBladeConfig.Instance.ObjectServer.ServiceName = "EntityService.svc";
   }
}
```

Once automatic session restore is enabled, Punch automatically saves the navigation history, current page, authentication context, page state and shared state, and restores it when necessary.

Saving and loading state

Punch takes care of the mechanics of saving and restoring state, however the developer is responsible for determining what state information should be saved and loaded for each view model and implement the necessary logic to do so.

In order for a view model to load and save state, it must implement *INavigationTarget*. *INavigationTarget* provides several navigation related methods as well as methods to load and save state.

```
public partial interface INavigationTarget
{
  /// <summary>
  /// Uniquely identifies the current view model instance. When implementing INavigationTarget,
  /// do not modify PageKey. PageKey is automatically maintained by the <see cref="INavigator"/>
  /// implementation.
  /// </summary>
  string PageKey { get; set; }
  /// <summary>
  /// Invoked to populate a view model with content passed during navigation. Any saved state
  /// is also provided when recreating a view model from a prior session.
  /// </summary>
  /// <param name="navigationParameter">The parameter value passed to
  /// <see cref="INavigator.NavigateToAsync(System.Type, object)"/>.
  /// </param>
  /// <param name="pageState">A dictionary of state preserved by this view model during an earlier
  /// session. This will be null the first time a view model is visited.</param>
  /// <param name="sharedState">A dictionary of shared state preserved by any view model during
  /// an earlier session.</param>
  void LoadState(object navigationParameter, Dictionary<string, object> pageState,
                 Dictionary<string, object> sharedState);
  /// <summary>
  /// Preserves state associated with this view model in case the application is suspended or the
  /// view model is discarded from the navigation cache.
  /// </summary>
  /// <param name="pageState">An empty dictionary to be populated with serializable state.</param>
  /// <param name="sharedState">A dictionary to be populated with shared state. The dictionary
  /// may already contain state that can be overwritten.</param>
  void SaveState(Dictionary<string, object> pageState, Dictionary<string, object> sharedState);
}
```

Page local state and globally shared state

The above *LoadState* and *SaveState* methods are both provided with two dictionaries containing state data. The *pageState* dictionary is for state information that is local to the current view model, where as the *sharedState* dictionary is for globally shared state. Each view model has the opportunity to overwrite existing shared state with updated information. A good example for shared state is the EntityManager cache if several view models share a common unit of work.

A word of caution. Shared state lives for the duration of the app, so be careful how much and what is put into the shared state dictionary.

The following example demonstrates an approach to preserving the unit of work in the shared state dictionary. The unit of work keeps track of whether it has previously been restored to avoid being restored over and over. Only the current view model should restore the unit of work if necessary.

```
[Export(typeof(ICustomerUnitOfWork))]
[Shared]
public class CustomerUnitOfWork : UnitOfWork<Customer>, ICustomerUnitOfWork
{
   [ImportingConstructor]
  public CustomerUnitOfWork(IEntityManagerProvider<NorthwindIBEntities> entityManagerProvider)
     : base(entityManagerProvider)
   {
   }
  public bool IsRestored { get; private set; }
  public void Restore(EntityCacheState cacheState)
   {
     EntityManager.CacheStateManager.RestoreCacheState(cacheState);
     IsRestored = true;
   }
  public EntityCacheState GetCacheState()
   {
     IsRestored = true;
    return EntityManager.CacheStateManager.GetCacheState();
   }
}
public class DetailPageViewModel : Screen, INavigationTarget
{
```

```
  // .... Snip: Code removed for clarity ....
  private bool _restored;
  public async void Start(Guid customerId)
   {
     try
      {
        if (_restored)
            Customer = _unitOfWork.Entities.WithIdFromCache(customerId);
        else
            Customer = await _unitOfWork.Entities.WithIdAsync(customerId);
      }
     catch (Exception e)
      {
        _errorHandler.Handle(e);
      }
   }
  public void OnNavigatedTo(NavigationArgs args)
   {
     Start((Guid)args.Parameter);
   }
  public void LoadState(object navigationParameter, Dictionary<string, object> pageState,
                        Dictionary<string, object> sharedState)
   {
     if (!_unitOfWork.IsRestored && sharedState.ContainsKey("uow"))
         _unitOfWork.Restore((EntityCacheState)sharedState["uow"]);
     _restored = pageState != null;
   }
  public void SaveState(Dictionary<string, object> pageState, Dictionary<string, object> sharedState)
   {
     sharedState["uow"] = _unitOfWork.GetCacheState();
   }
  public string PageKey { get; set; }
}
```

Serialization requirements

Session state is written to local storage every time the app is being suspended, which means that all state data must be serializable and of known types. Suspension will fail if any of the data can't be serialized and Punch will be unable to restore the data when the app resumes after having been terminated by the OS.

Limitations

With automatic session restore enabled, the following Navigator methods are no longer supported. An exception will be thrown if a call is attempted to any of these methods.

```
/// <summary>
///   Asynchronously navigates to an instance of the provided ViewModel type. The navigation will be cancelled if
///   the current active ViewModel cannot be closed or the target type is not authorized.
/// </summary>
/// <param name="viewModelType"> The target ViewModel type. </param>
/// <param name="prepare"> An action to initialize the target ViewModel before it is activated. </param>
/// <returns> A <see cref="Task" /> to await completion. </returns>
Task NavigateToAsync(Type viewModelType, Func<object, Task> prepare);
/// <summary>
///   Asynchronously navigates to an instance of the provided ViewModel type. The navigation will be cancelled if
///   the current active ViewModel cannot be closed or the target type is not authorized.
/// </summary>
/// <param name="viewModelType"> The target ViewModel type. </param>
/// <param name="prepare"> An action to initialize the target ViewModel before it is activated. </param>
/// <returns> A <see cref="Task" /> to await completion. </returns>
Task NavigateToAsync(Type viewModelType, Action<object> prepare);
/// <summary>
///   Asynchronously navigates to an instance of the provided ViewModel type. The navigation will be cancelled if
///   the current active ViewModel cannot be closed or the target type is not authorized.
/// </summary>
/// <param name="prepare"> An action to initialize the target ViewModel before it is activated. </param>
/// <typeparam name="T"> The target ViewModel type. </typeparam>
/// <returns> A <see cref="Task" /> to await completion. </returns>
Task NavigateToAsync<T>(Func<T, Task> prepare);
/// <summary>
```

```
///   Asynchronously navigates to an instance of the provided ViewModel type. The navigation will be cancelled if
///   the current active ViewModel cannot be closed or the target type is not authorized.
/// </summary>
/// <param name="prepare"> An action to initialize the target ViewModel before it is activated. </param>
/// <typeparam name="T"> The target ViewModel type. </typeparam>
/// <returns> A <see cref="Task" /> to await completion. </returns>
Task NavigateToAsync<T>(Action<T> prepare);
```

Use the following methods instead, however be aware that the optional navigation parameter **must be of a basic type like string, char, numeric and GUID**. Otherwise Punch will not be able to serialize the navigation history and an exception will be thrown during suspension. This is not a Punch limitation, but a limitation of Frame.GetNavigationState

```
/// <summary>
///   Asynchronously navigates to an instance of the provided ViewModel type. The navigation will be cancelled if
///   the current active ViewModel cannot be closed or the target type is not authorized.
/// </summary>
/// <param name="viewModelType"> The target ViewModel type. </param>
/// <param name="parameter">An optional parameter to be sent to the target view model.
///     See <see cref="INavigationTarget"/></param>
/// <returns> A <see cref="Task" /> to await completion. </returns>
Task NavigateToAsync(Type viewModelType, object parameter = null);
/// <summary>
///   Asynchronously navigates to an instance of the provided ViewModel type. The navigation will be cancelled if
///   the current active ViewModel cannot be closed or the target type is not authorized.
/// </summary>
/// <typeparam name="T"> The target ViewModel type. </typeparam>
/// <param name="parameter">An optional parameter to be sent to the target view model.
///     See <see cref="INavigationTarget"/></param>
/// <returns> A <see cref="Task" /> to await completion. </returns>
Task NavigateToAsync<T>(object parameter = null);
```