Intro to Jounce

Rik Robinson
Senior Consultant
rrobinson@wintellect.com

Wintellect
Know how.

Microsoft GOLD CERTIFIED Partner

what we do

consulting ▪ training ▪ design ▪ debugging

who we are

Founded by top experts on Microsoft – Jeffrey Richter, Jeff Prosise, and John Robbins – we pull out all the stops to help our customers achieve their goals through advanced software-based consulting and training solutions.

how we do it

Consulting & Debugging

- Architecture, analysis, and design services
- Full lifecycle custom software development
- Content creation
- Project management
- Debugging & performance tuning

Training

- On-site instructor-led training
- Virtual instructor-led training
- Devscovery conferences

Design

- User Experience Design
- Visual & Content Design
- Video & Animation Production

What is Jounce?

Jounce is a reference framework for Silverlight intended to provide guidance for building modular line of business applications that follow the MVVM pattern and utilize the Managed Extensibility Framework (MEF).

- CodePlex

Jounce Notes

- Open source available on CodePlex
  - Good documentation
  - Lots of QuickStart sample applications
  - Jounce Visual Studio Project Template
- Silverlight only (not WP7 or WPF)
- Less than 1000 lines of code
- Packaged in Jounce.dll
- Lighter weight than Prism
- Direct and explicit dependency on MEF

Why MEF?

- Standardized way to expose and consume components
- Wires components together in the correct order
- Flexible discovery of components
- Metadata provides for rich querying and filtering
- Assists with lifetime management of components
- It's part of the framework!
  - System.ComponentModel.Composition

What does Jounce do for us?

- Easy wiring of Views and ViewModels
- Allows for communication between ViewModels
- Compose / Consume messages on the fly
- Seamless use of dynamic XAP files
- Simple Navigation
- Region Management
- Commands
- Logging / Tracing
- ViewModels with CRUD / Validation
- Asynchronous Workflow

Application Service

- To plug into Silverlight application lifecycle
  - Remove all of the default code in your App.Xaml.cs
  - Add this code to App.Xaml

```
<Application …
    xmlns:Services="namespace:Jounce.Framework.Services" …>

    <Application.ApplicationLifetimeObjects>
        <Services:ApplicationService>
    </Application.ApplicationLifetimeObjects>

</Application>
```

Application Service

- Creates and wires a “Jouncified” IApplicationService object to handle the Application lifecycle
- Creates a default AggregateCatalog
- Creates the CompositionContainer
- Adds MEFDebugger to help with debugging
- Creates a Logger and sets a default level of logging (configurable)
- Creates a ViewModelRouter
- Creates a Deployment Service
- Maintains a collection for Views
- Creates an EventAggregator for subscription to Exceptions
- Provides numerous hooks via Exited/Exiting/Started/Starting

demo

Getting started

Tagging a View

- Tag the View class with ExportAsView attribute
- Tags must be unique
- Use constants instead of magic strings for View tag
- Can export using typeof() instead of string
- Mark only one View as IsShell = true
- Additional properties available in ExportAsView
  - Category, MenuName, ToolTip

```
[ExportAsView("Welcome", IsShell = true)]
public partial class WelcomeView { … }

[ExportAsView(typeof(WelcomeView))]
public partial class WelcomeView { … }
```

Tagging a ViewModel

- Tag the ViewModel class with ExportAsViewModel attribute
- Tags must be unique
- Use constants instead of magic strings for ViewModel tag
- Can export using typeof() instead of string
- ViewModels can be bound to multiple Views

```
[ExportAsViewModel("WelcomeViewModel")]
public partial class WelcomeViewModel : BaseViewModel  { … }

[ExportAsViewModel(typeof(WelcomeViewModel))]
public partial class WelcomeViewModel : BaseViewModel  { … }
```

BaseViewModel

- Imports the EventAggregator, Logger, and ViewModelRouter
- Visual State Management
  - Allows setting of Visual State in Views from ViewModel
  - GoToVisualState()
  - GoToVisualStateForView()
- Design-time awareness via InDesigner property
  - Generally checked in constructor to wire design-time data
- Awareness of binding/activation events
  - Initialize() – called first time ViewModel is created
  - Activate() – called whenever a view is navigated to
  - Deactivate() – called whenever a view is navigated from

ViewModelRouter

- BaseViewModel imports a Router property
- Provides programmatic access to ViewModels

```
var viewModel = Router.ResolveViewModel("WelcomeVM");
```

- Do not call the Router from ViewModel's constructor

  Instead implement IPartImportsSatisfiedNotification and call from OnImportsSatisfied() to assure Router has been wired by MEF

Binding View to ViewModel

Export a ViewModelRoute for each combination of ViewModel and View

```
[Export]
public ViewModelRoute Binding
{
    return ViewModelRoute.Create("WelcomeVM", "Welcome");
}
```

Export a ViewModel for a View at runtime

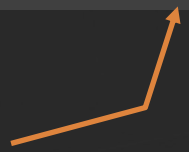
```
Router.RouteViewModelForView("WelcomeVM", "Welcome");

Router.RouteViewModelForView<WelcomeViewModel, ShellView>();
```

Non-Shared Views and VMs

ViewModelRouter provides methods for creating non-shared Views and ViewModels

View

```
var view = Router.GetNonSharedView("WelcomeView", null);
```



Pass ViewModel here and Jounce will wire

ViewModel

```
var viewModel = Router.GetNonSharedViewModel("WelcomeVM");
```

demo

Views and ViewModels

BaseEntityViewModel

- Use for view models that need CRUD operations and/or validation
- Derived from BaseViewModel
- Implements INotifyDataErrorInfo interface to allow hooks into Silverlight's built-in validation framework
- Override ValidateAll() to provide aggregate validation
- Committed boolean
  - Inverse dirty flag
  - Should be initialized to true
- CommitCommand – bind to Save buttons to automatically disable unless record is both dirty and passes all validations
- OnCommit – called when commit command is executed
- HasErrors property will be true if there are errors

Validation – Field Level

```
public string FirstName
{
    get { return _firstName; }
    set
    {
        //  other setter stuff goes here
        ValidateName(ExtractPropertyName(() => FirstName, value));
    }
}
```

```
private void ValidateName(string propertyName, string value)
{
    ClearErrors(propertyName);

    if (string.IsNullOrEmpty(value))
        SetError(propertyName, “The field is required”);
}
```

Don’t forget to clear errors

Validation – Aggregate

- Override the ValidateAll method
- Place validations in ValidateAll that involve multiple fields or are server-based
- ValidateAll is called by Jounce before any commit command is processed

```
protected override void ValidateAll()
{
   ValidatePhoneNumber();
   ValidateEmail();
   PingServer();
}
```

Commands

- Loosely coupled way to bind the UI to the logic that performs the action
- Command is exposed as a property of the ViewModel
- Jounce provides the ActionCommand
- CanExecute
  - Automatic enable/disable for buttons

```
public IActionCommand<MyClass> SaveCommand { get; private set; }

// to create
SaveCommand = new ActionCommand<MyClass>(
                        entity => Service.Save(entity),
                        entity => IsValid(entity)
                );
```

demo

BaseEntityViewModel and Validation

Event Aggregation

- Jounce provides EventAggregator for messaging
- Single source that can publish and provide subscriptions
- Any message T can be sent
- To publish a null, specify the type and send null

```
EventAggregator.Publish<MyMessage>(null);
```

- For subscriptions
  - The entity that handles message must implement IEventSink<T>

Event Aggregation

Import the EventAggregator

```
[Import]
public IEventAggregator EventAggregator { get; set; }
```

Publish

```
EventAggregator.Publish("A message of type string");
```

Subscribe

```
EventAggregator.Subscribe<string>(this);
```

Simple Navigation

```
EventAggregator.Publish(new ViewNavigationArgs("MyView"));
```

- Finds or creates the View
- Binds the View to the ViewModel
- Calls Initialize and/or Activate on the ViewModel
- Publishes the ViewNavigatedArgs message once complete
- Use NavigationTrigger behavior to fire navigation events in XAML
- AddNamedParameter extension method allows passing of payload on Publish
- AsNavigationArgs extension method available for Views

Simple Navigation

- ViewNavigationArgs
  - Raised to notify Jounce that a View is changing its View status
  - Just publish a ViewNavigationArgs message to fire a navigation
  - Set Deactivate flag to indicate activate/deactivate
- ViewNavigatedArgs
  - Fired at the conclusion of a Navigation event
  - Subscribe to ViewNavigatedArgs message to be notified of Navigation
  - Used by Region Manager for region management

Regions

- Regions can be ContentControl, ItemsControl, TabControl
- Create custom Region types via RegionAdapterBase
- Regions indicated by RegionName attached property

MyView.xaml

```
<ContentControl Regions:ExportAsRegion.RegionName="MainRegion" />
```

MyView.xaml.cs

```
[ExportAsView("MyView")]
[ExportViewToRegion("MyView", "MainRegion")]
public partial class MyView
{
}
```

demo

Simple Navigation

Binding Views to XAP files

- ViewXapRoute - tells Jounce that a particular class can be found in a specific XAP
- Jounce dynamically loads the XAP the first time the View is requested
- To explicitly load a XAP use Deployment.RequestXap()
- Set Copy Local = false for Jounce reference in dynamic modules
- Implement IModuleInitializer to hook into Initialized/Initialize events

```
[Export]
public ViewXapRoute ExtrasRoute
{
    get
    {
        return ViewXapRoute.Create("Extras", "MyApp.Extras.xap");
    }
}
```

demo

Dynamic XAP

Logging

- Jounce provides ILogger
- MEFDebugger class implements ILogger and writes to the debug console
- By default, ApplicationService creates a MEFDebugger
- ILogger.Log allows various levels of SeverityLevel
  - Verbose, Information, Warning, Error, Critical
- To implement ILogger, declare a class and export the ILogger

```
[Export(typeof(ILogger))]
public MyLogger : ILogger
{
}
```

Design-Time Data

- BaseViewModel provides InDesigner boolean to test
- Test InDesigner property before making calls to services or other dependencies that would not be available at design time
- Numerous ways to keep the solution "Blendable"
  - Create separate DesignTimeViewModel and use d:DataContext
  - Conditional compilation directives that use separate classes and/or methods to create sample data
  - Blend can create some basic sample data, but doesn't hold up with nested collections of complex objects

Workflows

- Allows firing of asynchronous processes in a sequential fashion
- Defined by the IWorkflow interface
- Several IWorkFlow implementations included
  - WorkflowAction, WorkflowBackgroundWorker, etc.
- Relies on C# iterators
- To create a Workflow, chain together a series of IWorkflow nodes using an enumerable method
- Use WorkflowController.Begin to start the process

demo

Workflows

Jounce Summary

- Easy wiring of Views and ViewModels
- Allows for communication between ViewModels
- Compose / Consume messages on the fly
- Seamless use of dynamic XAP files
- Simple Navigation
- Region Management
- Commands
- Tracing / Logging
- ViewModels with CRUD / Validation
- Asynchronous Workflow

Resources Slide

- Jounce on CodePlex
  - jounce.codeplex.com
- Jeremy Likness (Author of Jounce)
  - @JeremyLikness
  - csharperimage.jeremylikness.com

Questions?

Rik Robinson
Senior Consultant
rrobinson@wintellect.com