Introduction to the MVVM Toolkit

Article • 01/13/2023

The `CommunityToolkit.Mvvm` package (aka MVVM Toolkit, formerly named `Microsoft.Toolkit.Mvvm`) is a modern, fast, and modular MVVM library. It is part of the .NET Community Toolkit and is built around the following principles:

- **Platform and Runtime Independent** - **.NET Standard 2.0**, **.NET Standard 2.1** and **.NET 6** 🚀 (UI Framework Agnostic)
- **Simple to pick-up and use** - No strict requirements on Application structure or coding-paradigms (outside of 'MVVM'ness), i.e., flexible usage.
- **À la carte** - Freedom to choose which components to use.
- **Reference Implementation** - Lean and performant, providing implementations for interfaces that are included in the Base Class Library, but lack concrete types to use them directly.

The MVVM Toolkit is maintained and published by Microsoft, and part of the .NET Foundation. It is also used by several first party applications that are built into Windows, such as the Microsoft Store.

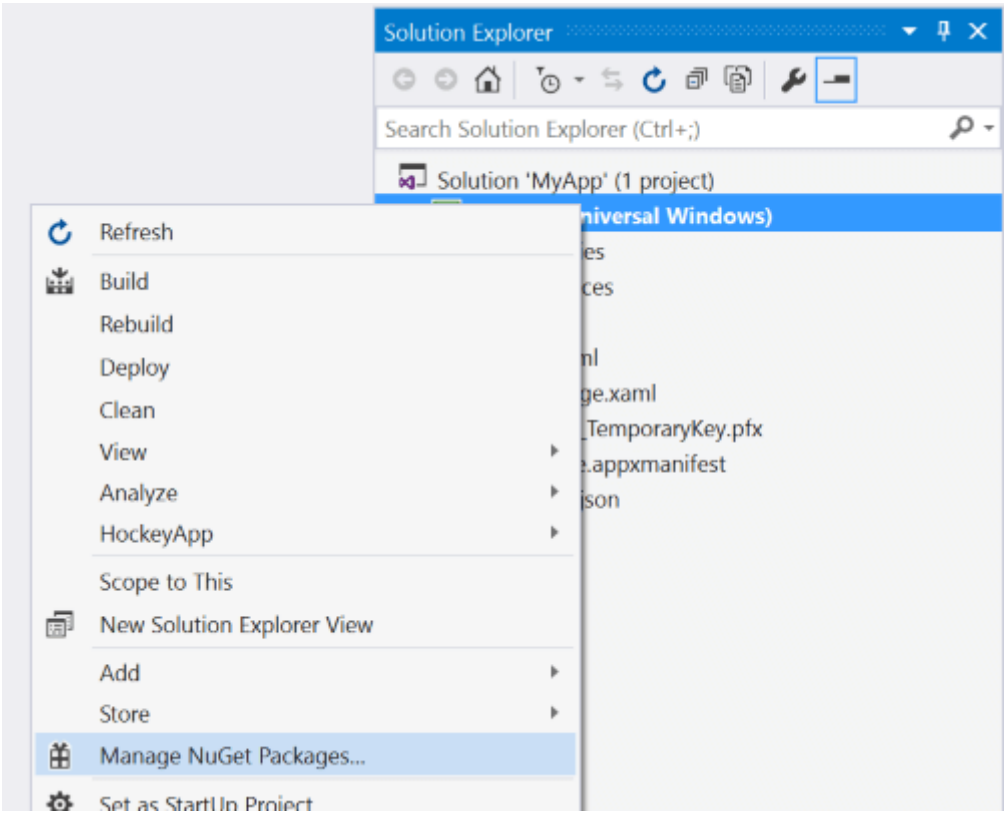
This package targets **.NET Standard** so it can be used on any app platform: UWP, WinForms, WPF, Xamarin, Uno, and more; and on any runtime: .NET Native, .NET Core, .NET Framework, or Mono. It runs on all of them. The API surface is identical in all cases, making it perfect for building shared libraries.

Additionally, the MVVM Toolkit also has a **.NET 6** target, which is used to enable more internal optimizations when running on .NET 6. The public API surface is identical in both cases, so NuGet will always resolve the best possible version of the package without consumers having to worry about which APIs will be available on their platform.

Getting started

To install the package from within Visual Studio:

1. In Solution Explorer, right-click on the project and select **Manage NuGet Packages**. Search for **CommunityToolkit.Mvvm** and install it.



2. Add a using or Imports directive to use the new APIs:

```
c#
```

```
using CommunityToolkit.Mvvm;
```

```
VB
```

```
Imports CommunityToolkit.Mvvm
```

3. Code samples are available in the other docs pages for the MVVM Toolkit, and in the unit tests for the project.

When should I use this package?

Use this package for access to a collection of standard, self-contained, lightweight types that provide a starting implementation for building modern apps using the MVVM pattern. These types alone are usually enough for many users to build apps without needing additional external references.

The included types are:

- **CommunityToolkit.Mvvm.ComponentModel**
  - ObservableObject
  - ObservableRecipient

    - ObservableValidator
- **CommunityToolkit.Mvvm.DependencyInjection**
    - Ioc
- **CommunityToolkit.Mvvm.Input**
    - RelayCommand
    - RelayCommand<T>
    - AsyncRelayCommand
    - AsyncRelayCommand<T>
    - IRelayCommand
    - IRelayCommand<T>
    - IAsyncRelayCommand
    - IAsyncRelayCommand<T>
- **CommunityToolkit.Mvvm.Messaging**
    - IMessenger
    - WeakReferenceMessenger
    - StrongReferenceMessenger
    - IRecipient<TMessage>
    - MessageHandler<TRecipient, TMessage>
- **CommunityToolkit.Mvvm.Messaging.Messages**
    - PropertyChangedMessage<T>
    - RequestMessage<T>
    - AsyncRequestMessage<T>
    - CollectionRequestMessage<T>
    - AsyncCollectionRequestMessage<T>
    - ValueChangedMessage<T>

This package aims to offer as much flexibility as possible, so developers are free to choose which components to use. All types are loosely-coupled, so that it's only necessary to include what you use. There is no requirement to go "all-in" with a specific series of all-encompassing APIs, nor is there a set of mandatory patterns that need to be followed when building apps using these helpers. Combine these building blocks in a way that best fits your needs.

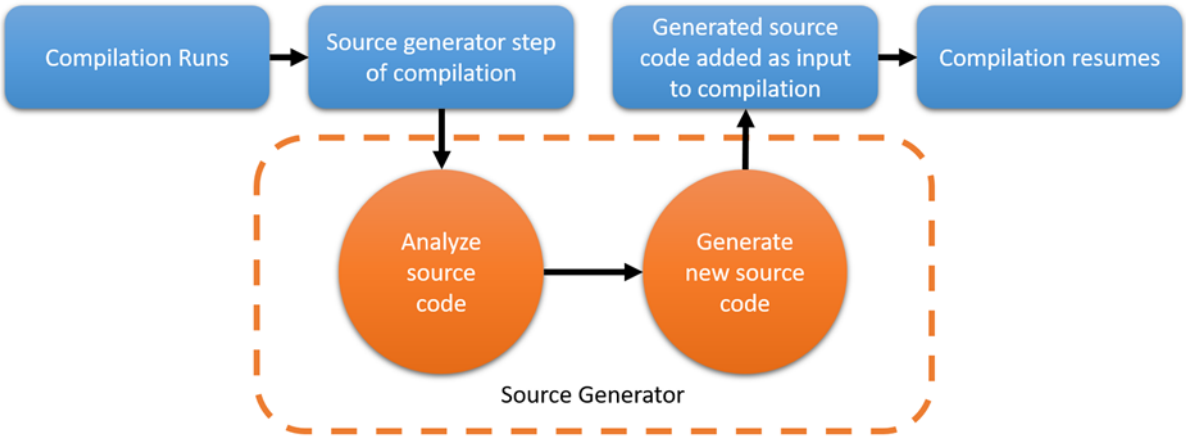
Additional resources

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

MVVM source generators

Article • 07/15/2022

Starting with version 8.0, the MVVM Toolkit includes brand new Roslyn source generators that will help greatly reduce boilerplate when writing code using the MVVM architecture. They can simplify scenarios where you need to setup observable properties, commands and more. If you're not familiar with source generators, you can read more about them here. This is a simplified view of how they work:



This means that as you're writing code, the MVVM Toolkit generator will now take care of generating additional code for you behind the scenes, so you don't have to worry about it. This code will then be compiled and included in your application, so the end result is exactly the same as if you had written all that extra code manually, but without having to do all of that extra work! 🎉

For instance, wouldn't it be great if instead of having to setup an observable property normally:

C#

```
private string? name;

public string? Name
{
    get => name;
    set => SetProperty(ref name, value);
}
```

You could just have a simple annotated field to express the same?

C#

```
[ObservableProperty]
private string? name;
```

What about creating a command:

C#

```
private void SayHello()
{
    Console.WriteLine("Hello");
}

private ICommand? sayHelloCommand;

public ICommand SayHelloCommand => sayHelloCommand ??= new
RelayCommand(SayHello);
```

What if we could just have our method, and nothing else?

C#

```
[RelayCommand]
private void SayHello()
{
    Console.WriteLine("Hello");
}
```

With the new MVVM source generators, all of this is possible, and much more! 🦁

> ⓘ **Note**
>
> Source generators can be used independently from other existing features in the MVVM Toolkit, and you're free to mix and match using source generators with previous APIs as needed. That is, you're free to just gradually start using source generators in new files and eventually migrating older files to use them to reduce verbosity, but there is no obligation to always use either approach in a whole project or application.

These docs will go over exactly what features are included with the MVVM generators and how to use them:

- **CommunityToolkit.Mvvm.ComponentModel**
  - ObservableProperty
  - INotifyPropertyChanged

- **CommunityToolkit.Mvvm.Input**
  - RelayCommand

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

ObservableProperty attribute

Article • 05/02/2023

The ObservableProperty type is an attribute that allows generating observable properties from annotated fields. Its purpose is to greatly reduce the amount of boilerplate that is needed to define observable properties.

> ⓘ **Note**
>
> In order to work, annotated fields need to be in a **partial class** with the necessary `INotifyPropertyChanged` infrastructure. If the type is nested, all types in the declaration syntax tree must also be annotated as partial. Not doing so will result in a compile errors, as the generator will not be able to generate a different partial declaration of that type with the requested observable property.

> **Platform APIs:** ObservableProperty, NotifyPropertyChangedFor, NotifyCanExecuteChangedFor, NotifyDataErrorInfo, NotifyPropertyChangedRecipients, ICommand, IRelayCommand, ObservableValidator, PropertyChangedMessage<T>, IMessenger

How it works

The `ObservableProperty` attribute can be used to annotate a field in a partial type, like so:

C#

```
[ObservableProperty]
private string? name;
```

And it will generate an observable property like this:

C#

```
public string? Name
{
    get => name;
    set => SetProperty(ref name, value);
}
```

It will also do so with an optimized implementation, so the end result will be even faster.

> ⓘ Note
>
> The name of the generated property will be created based on the field name. The generator assumes the field is named either `lowerCamel`, `_lowerCamel` or `m_lowerCamel`, and it will transform that to be `UpperCamel` to follow proper .NET naming conventions. The resulting property will always have public accessors, but the field can be declared with any visibility (`private` is recommended).

Running code upon changes

The generated code is actually a bit more complex than this, and the reason for that is that it also exposes some methods you can implement to hook into the notification logic, and run additional logic when the property is about to be updated and right after it is updated, if needed. That is, the generated code is actually similar to this:

C#

```
public string? Name
{
    get => name;
    set
    {
        if (!EqualityComparer<string?>.Default.Equals(name, value))
        {
            string? oldValue = name;
            OnNameChanging(value);
            OnNameChanging(oldValue, value);
            OnPropertyChanging();
            name = value;
            OnNameChanged(value);
            OnNameChanged(oldValue, value);
            OnPropertyChanged();
        }
    }
}

partial void OnNameChanging(string? value);
partial void OnNameChanged(string? value);

partial void OnNameChanging(string? oldValue, string? newValue);
partial void OnNameChanged(string? oldValue, string? newValue);
```

This allows you to implement any of those methods to inject additional code. The first two are useful whenever you want to run some logic that only needs to reference the new value that the property has been set to. The other two are useful whenever you

have some more complex logic that also has to update some state on both the old and new value being set.

For instance, here is an example of how the first two overloads can be used:

C#

```
[ObservableProperty]
private string? name;

partial void OnNameChanging(string? value)
{
    Console.WriteLine($"Name is about to change to {value}");
}

partial void OnNameChanged(string? value)
{
    Console.WriteLine($"Name has changed to {value}");
}
```

And here is an example of how the other two overloads can be used:

C#

```
[ObservableProperty]
private ChildViewModel? selectedItem;

partial void OnSelectedItemChanging(ChildViewModel? oldValue,
ChildViewModel? newValue)
{
    if (oldValue is not null)
    {
        oldValue.IsSelected = true;
    }

    if (newValue is not null)
    {
        newValue.IsSelected = true;
    }
}
```

You're free to only implement any number of methods among the ones that are available, or none of them. If they are not implemented (or if only one is), the entire call(s) will just be removed by the compiler, so there will be no performance hit at all for cases where this additional functionality is not required.

> ⓘ **Note**

The generated methods are **partial methods** with no implementation, meaning that if you choose to implement them, you cannot specify an explicit accessibility for them. That is, implementations of these methods should also be declared as just `partial` methods, and they will always implicitly have private accessibility. Trying to add an explicit accessibility (eg. adding `public` or `private`) will result in an error, as that is not allowed in C#.

Notifying dependent properties

Imagine you had a `FullName` property you wanted to raise a notification for whenever `Name` changes. You can do that by using the `NotifyPropertyChangedFor` attribute, like so:

C#

```
[ObservableProperty]
[NotifyPropertyChangedFor(nameof(FullName))]
private string? name;
```

This will result in a generated property equivalent to this:

C#

```
public string? Name
{
    get => name;
    set
    {
        if (SetProperty(ref name, value))
        {
            OnPropertyChanged("FullName");
        }
    }
}
```

Notifying dependent commands

Imagine you had a command whose execution state was dependent on the value of this property. That is, whenever the property changed, the execution state of the command should be invalidated and computed again. In other words, ICommand.CanExecuteChanged should be raised again. You can achieve this by using the `NotifyCanExecuteChangedFor` attribute:

C#

```
[ObservableProperty]
[NotifyCanExecuteChangedFor(nameof(MyCommand))]
private string? name;
```

This will result in a generated property equivalent to this:

C#

```
public string? Name
{
    get => name;
    set
    {
        if (SetProperty(ref name, value))
        {
            MyCommand.NotifyCanExecuteChanged();
        }
    }
}
```

In order for this to work, the target command has to be some IRelayCommand property.

Requesting property validation

If the property is declared in a type that inherits from ObservableValidator, it is also possible to annotate it with any validation attributes and then request the generated setter to trigger validation for that property. This can be achieved with the `NotifyDataErrorInfo` attribute:

C#

```
[ObservableProperty]
[NotifyDataErrorInfo]
[Required]
[MinLength(2)] // Any other validation attributes too...
private string? name;
```

This will result in the following property being generated:

C#

```
public string? Name
{
    get => name;
    set
    {
```

```
        if (SetProperty(ref name, value))
        {
            ValidateProperty(value, "Value2");
        }
    }
}
```

That generated `ValidateProperty` call will then validate the property and update the state of the `ObservableValidator` object, so that UI components can react to it and display any validation errors appropriately.

> ⓘ **Note**
>
> By design, only field attributes that inherit from **ValidationAttribute** will be forwarded to the generated property. This is done specifically to support data validation scenarios. All other field attributes will be ignored, so it is not currently possible to add additional custom attributes on a field and have them also be applied to the generated property. If that is required (eg. to control serialization), consider using a traditional manual property instead.

Sending notification messages

If the property is declared in a type that inherits from ObservableRecipient, you can use the `NotifyPropertyChangedRecipients` attribute to instruct the generator to also insert code to send a property changed message for the property change. This will allow registered recipients to dynamically react to the change. That is, consider this code:

C#

```
[ObservableProperty]
[NotifyPropertyChangedRecipients]
private string? name;
```

This will result in the following property being generated:

C#

```
public string? Name
{
    get => name;
    set
    {
        string? oldValue = name;
```

```
        if (SetProperty(ref name, value))
        {
            Broadcast(oldValue, value);
        }
    }
}
```

That generated `Broadcast` call will then send a new PropertyChangedMessage<T> using the `IMessenger` instance in use in the current viewmodel, to all registered subscribers.

Adding custom attributes

In some cases, it might be useful to also have some custom attributes over the generated properties. To achieve that, you can simply use the `[property: ]` target in attribute lists over annotated fields, and the MVVM Toolkit will automatically forward those attributes to the generated properties.

For instance, consider a field like this:

C#

```
[ObservableProperty]
[property: JsonRequired]
[property: JsonPropertyName("name")]
private string? username;
```

This will generate a `Username` property, with those two `[JsonRequired]` and `[JsonPropertyName("name")]` attributes over it. You can use as many attribute lists targeting the property as you want, and all of them will be forwarded to the generated properties.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

RelayCommand attribute

Article • 05/02/2023

The RelayCommand type is an attribute that allows generating relay command properties for annotated methods. Its purpose is to completely eliminate the boilerplate that is needed to define commands wrapping private methods in a viewmodel.

> ⓘ **Note**
>
> In order to work, annotated methods need to be in a **partial class**. If the type is nested, all types in the declaration syntax tree must also be annotated as partial. Not doing so will result in a compile errors, as the generator will not be able to generate a different partial declaration of that type with the requested command.

> **Platform APIs:** RelayCommand, ICommand, IRelayCommand, IRelayCommand<T>, IAsyncRelayCommand, IAsyncRelayCommand<T>, Task, CancellationToken

How it works

The `RelayCommand` attribute can be used to annotate a method in a partial type, like so:

C#

```
[RelayCommand]
private void GreetUser()
{
    Console.WriteLine("Hello!");
}
```

And it will generate a command like this:

C#

```
private RelayCommand? greetUserCommand;

public IRelayCommand GreetUserCommand => greetUserCommand ??= new
RelayCommand(GreetUser);
```

> ⓘ **Note**

The name of the generated command will be created based on the method name. The generator will use the method name and append "Command" at the end, and it will strip the "On" prefix, if present. Additionally, for asynchronous methods, the "Async" suffix is also stripped before "Command" is appeneded.

Command parameters

The `[RelayCommand]` attribute supports creating commands for methods with a parameter. In that case, it will automatically change the generated command to be an IRelayCommand<T> instead, accepting a parameter of the same type:

C#

```
[RelayCommand]
private void GreetUser(User user)
{
    Console.WriteLine($"Hello {user.Name}!");
}
```

This will result in the following generated code:

C#

```
private RelayCommand<User>? greetUserCommand;

public IRelayCommand<User> GreetUserCommand => greetUserCommand ??= new
RelayCommand<User>(GreetUser);
```

The resulting command will automatically use the type of the argument as its type argument.

Asynchronous commands

The `[RelayCommand]` command also supports wrapping asynchronous methods, via the IAsyncRelayCommand and IAsyncRelayCommand<T> interfaces. This is handled automatically whenever a method returns a Task type. For instance:

C#

```
[RelayCommand]
private async Task GreetUserAsync()
{
    User user = await userService.GetCurrentUserAsync();
```

```
    Console.WriteLine($"Hello {user.Name}!");
}
```

This will result in the following code:

C#

```
private AsyncRelayCommand? greetUserCommand;

public IAsyncRelayCommand GreetUserCommand => greetUserCommand ??= new
AsyncRelayCommand(GreetUserAsync);
```

If the method takes a parameter, the resulting command will also be generic.

There is a special case when the method has a CancellationToken, as that will be propagated to the command to enable cancellation. That is, a method like this:

C#

```
[RelayCommand]
private async Task GreetUserAsync(CancellationToken token)
{
    try
    {
        User user = await userService.GetCurrentUserAsync(token);

        Console.WriteLine($"Hello {user.Name}!");
    }
    catch (OperationCanceledException)
    {
    }
}
```

Will result in the generated command passing a token to the wrapped method. This allows consumers to just call IAsyncRelayCommand.Cancel to signal that token, and to allow pending operations to be stopped correctly.

Enabling and disabling commands

It is often useful to be able to disable commands, and to then later on invalidate their state and have them check again whether they can be executed or not. In order to support this, the `RelayCommand` attribute exposes the `CanExecute` property, which can be used to indicate a target property or method to use to evaluate whether a command can be executed:

C#

```
[RelayCommand(CanExecute = nameof(CanGreetUser))]
private void GreetUser(User? user)
{
    Console.WriteLine($"Hello {user!.Name}!");
}

private bool CanGreetUser(User? user)
{
    return user is not null;
}
```

This way, `CanGreetUser` is invoked when the button is first bound to the UI (eg. to a button), and then it is invoked again every time IRelayCommand.NotifyCanExecuteChanged is invoked on the command.

For instance, this is how a command can be bound to a property to control its state:

C#

```
[ObservableProperty]
[NotifyCanExecuteChangedFor(nameof(GreetUserCommand))]
private User? selectedUser;
```

XML

```
<!-- Note: this example uses traditional XAML binding syntax -->
<Button
    Content="Greet user"
    Command="{Binding GreetUserCommand}"
    CommandParameter="{Binding SelectedUser}"/>
```

In this example, the generated `SelectedUser` property will invoke `GreetUserCommand.NotifyCanExecuteChanged()` method every time its value changes. The UI has a `Button` control binding to `GreetUserCommand`, meaning every time its `CanExecuteChanged` event is raised, it will call its `CanExecute` method again. This will cause the wrapped `CanGreetUser` method to be evaluated, which will return the new state for the button based on whether or not the input `User` instance (which in the UI is bound to the `SelectedUser` property) is `null` or not. This means that whenever `SelectedUser` is changed, `GreetUserCommand` will become enabled or not based on whether that property has a value, which is the desired behavior in this scenario.

> ⓘ **Note**

The command will **not** automatically be aware of when the return value for the `CanExecute` method or property has changed. It is up to the developer to call `IRelayCommand.NotifyCanExecuteChanged` to invalidate the command and request the linked `CanExecute` method to be evaluated again to then update the visual state of the control bound to the command.

Handling concurrent executions

Whenever a command is asynchronous, it can be configured to decide whether to allow concurrent executions or not. When using the `RelayCommand` attribute, this can be set via the `AllowConcurrentExecutions` property. The default is `false`, meaning that until an execution is pending, the command will signal its state as being disabled. If it instead is set to `true`, any number of concurrent invocations can be queued.

Note that if a command accepts a cancellation token, a token will also be canceled if a concurrent execution is requested. The main difference is that if concurrent executions are allowed, the command will remain enabled and it will start a new requested execution without waiting for the previous one to actually complete.

Handling asynchronous exceptions

There are two different ways async relay commands handle exceptions:

- **Await and rethrow (default)**: when the command awaits the completion of an invocation, any exceptions will naturally be thrown on the same synchronization context. That usually means that exceptions being thrown would just crash the app, which is a behavior consistent with that of synchronous commands (where exceptions being thrown will also crash the app).
- **Flow exceptions to task scheduler**: if a command is configured to flow exceptions to the task scheduler, exceptions being thrown will not crash the app, but instead they will both become available through the exposed IAsyncRelayCommand.ExecutionTask as well as bubbling up to the TaskScheduler.UnobservedTaskException. This enables more advanced scenarios (such as having UI components bind to the task and display different results based on the outcome of the operation), but it is more complex to use correctly.

The default behavior is having commands await and rethrow exceptions. This can be configured via the `FlowExceptionsToTaskScheduler` property:

```
C#
```

```
[RelayCommand(FlowExceptionsToTaskScheduler = true)]
private async Task GreetUserAsync(CancellationToken token)
{
    User user = await userService.GetCurrentUserAsync(token);

    Console.WriteLine($"Hello {user.Name}!");
}
```

In this case, the `try/catch` is not needed, as exceptions will not crash the app anymore. Note that this will also cause other unrelated exceptions to not be rethrown automatically, so you should carefully decide how to approach each individual scenario and configure the rest of the code appropriately.

Cancel commands for asynchronous operations

One last option for asynchronous commands is the ability to request a cancel command to be generated. This is an `ICommand` wrapping an async relay command that can be used to request the cancellation of an operation. This command will automatically signal its state to reflect whether or not it can be used at any given time. For instance, if the linked command is not executing, it will report its state as also not being executable. This can be used as follows:

C#

```
[RelayCommand(IncludeCancelCommand = true)]
private async Task DoWorkAsync(CancellationToken token)
{
    // Do some long running work...
}
```

This will cause a `DoWorkCancelCommand` property to also be generated. This can then be bound to some other UI component to easily let users cancel pending asynchronous operations.

Adding custom attributes

Just like with observable properties, the `RelayCommand` generator also includes support for custom attributes for the generated properties. To leverage this, you can simply use the `[property: ]` target in attribute lists over annotated methods, and the MVVM Toolkit will forward those attributes to the generated command properties.

For instance, consider a method like this:

C#

```
[RelayCommand]
[property: JsonIgnore]
private void GreetUser(User user)
{
    Console.WriteLine($"Hello {user.Name}!");
}
```

This will generate a `GreetUserCommand` property, with the `[JsonIgnore]` attribute over it. You can use as many attribute lists targeting the method as you want, and all of them will be forwarded to the generated properties.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

INotifyPropertyChanged attributes

Article • 08/10/2022

The INotifyPropertyChanged type is an attribute that allows inserting MVVM support code into existing types. Along with other related attributes (ObservableObject and ObservableRecipient), its purpose is to support developers in cases where the same functionality from these types was needed, but the target types were already implementing from another type. Since C# does not allow multiple inheritance, these attributes can instead be used to have the MVVM Toolkit generator add the same code right into those types, sidestepping this limitation.

> **Note**
>
> In order to work, annotated types need to be in a **partial class**. If the type is nested, all types in the declaration syntax tree must also be annotated as partial. Not doing so will result in a compile errors, as the generator will not be able to generate a different partial declaration of that type with the requested additional code.

> **Note**
>
> These attributes are only meant to be used in cases where the target types cannot just inherit from the equivalent types (eg. from `ObservableObject`). If that is possible, inheriting is the recommended approach, as it will reduce the binary size by avoiding creating duplicated code into the final assembly.

> **Platform APIs:** INotifyPropertyChanged, ObservableObject, ObservableRecipient

How to use them

Using any of these attributes is pretty straightforward: just add them to a partial class and all the code from the corresponding types will automatically be generated into that type. For instance, consider this:

C#

```
[INotifyPropertyChanged]
public partial class MyViewModel : SomeOtherType
{
}
```

This will generate a complete `INotifyPropertyChanged` implementation into the `MyViewModel` type, complete with additional helpers (such as `SetProperty`) that can be used to reduce verbosity. Here is a brief summary of the various attributes:

- INotifyPropertyChanged: implements the interface and adds helper methods to set properties and raise the events.
- ObservableObject: adds all the code from the `ObservableObject` type. It is conceptually equivalent to `INotifyPropertyChanged`, with the main difference being that it also implements `INotifyPropertyChanging`.
- ObservableRecipient: adds all the code from the `ObservableRecipient` type. In particular, this can be added to a type inheriting from `ObservableValidator` to combine the two.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

ObservableObject

Article • 04/07/2022

The ObservableObject is a base class for objects that are observable by implementing the INotifyPropertyChanged and INotifyPropertyChanging interfaces. It can be used as a starting point for all kinds of objects that need to support property change notifications.

> **Platform APIs:** ObservableObject, TaskNotifier, TaskNotifier<T>

How it works

`ObservableObject` has the following main features:

- It provides a base implementation for `INotifyPropertyChanged` and `INotifyPropertyChanging`, exposing the `PropertyChanged` and `PropertyChanging` events.
- It provides a series of `SetProperty` methods that can be used to easily set property values from types inheriting from `ObservableObject`, and to automatically raise the appropriate events.
- It provides the `SetPropertyAndNotifyOnCompletion` method, which is analogous to `SetProperty` but with the ability to set `Task` properties and raise the notification events automatically when the assigned tasks are completed.
- It exposes the `OnPropertyChanged` and `OnPropertyChanging` methods, which can be overridden in derived types to customize how the notification events are raised.

Simple property

Here's an example of how to implement notification support to a custom property:

C#

```
public class User : ObservableObject
{
    private string name;

    public string Name
    {
        get => name;
        set => SetProperty(ref name, value);
    }
}
```

The provided `SetProperty<T>(ref T, T, string)` method checks the current value of the property, and updates it if different, and then also raises the relevant events automatically. The property name is automatically captured through the use of the `[CallerMemberName]` attribute, so there's no need to manually specify which property is being updated.

Wrapping a non-observable model

A common scenario, for instance, when working with database items, is to create a wrapping "bindable" model that relays properties of the database model, and raises the property changed notifications when needed. This is also needed when wanting to inject notification support to models, that don't implement the `INotifyPropertyChanged` interface. `ObservableObject` provides a dedicated method to make this process simpler. For the following example, `User` is a model directly mapping a database table, without inheriting from `ObservableObject`:

C#

```
public class ObservableUser : ObservableObject
{
    private readonly User user;

    public ObservableUser(User user) => this.user = user;

    public string Name
    {
        get => user.Name;
        set => SetProperty(user.Name, value, user, (u, n) => u.Name = n);
    }
}
```

In this case we're using the `SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string)` overload. The signature is slightly more complex than the previous one - this is necessary to let the code still be extremely efficient even if we don't have access to a backing field like in the previous scenario. We can go through each part of this method signature in detail to understand the role of the different components:

- `TModel` is a type argument, indicating the type of the model we're wrapping. In this case, it'll be our `User` class. Note that we don't need to specify this explicitly - the C# compiler will infer this automatically by how we're invoking the `SetProperty` method.
- `T` is the type of the property we want to set. Similarly to `TModel`, this is inferred automatically.

- `T oldValue` is the first parameter, and in this case we're using `user.Name` to pass the current value of that property we're wrapping.
- `T newValue` is the new value to set to the property, and here we're passing `value`, which is the input value within the property setter.
- `TModel model` is the target model we are wrapping, in this case we're passing the instance stored in the `user` field.
- `Action<TModel, T> callback` is a function that will be invoked if the new value of the property is different than the current one, and the property needs to be set. This will be done by this callback function, which receives as input the target model and the new property value to set. In this case we're just assigning the input value (which we called `n`) to the `Name` property (by doing `u.Name = n`). It is important here to avoid capturing values from the current scope and only interact with the ones given as input to the callback, as this allows the C# compiler to cache the callback function and perform a number of performance improvements. It's because of this that we're not just directly accessing the `user` field here or the `value` parameter in the setter, but instead we're only using the input parameters for the lambda expression.

The `SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string)` method makes creating these wrapping properties extremely simple, as it takes care of both retrieving and setting the target properties while providing an extremely compact API.

> ⓘ **Note**
>
> Compared to the implementation of this method using LINQ expressions, specifically through a parameter of type `Expression<Func<T>>` instead of the state and callback parameters, the performance improvements that can be achieved this way are really significant. In particular, this version is ~200x faster than the one using LINQ expressions, and does not make any memory allocations at all.

Handling `Task<T>` properties

If a property is a `Task` it's necessary to also raise the notification event once the task completes, so that bindings are updated at the right time. eg. to display a loading indicator or other status info on the operation represented by the task. `ObservableObject` has an API for this scenario:

```
C#
```

```
public class MyModel : ObservableObject
{
    private TaskNotifier<int>? requestTask;

    public Task<int>? RequestTask
    {
        get => requestTask;
        set => SetPropertyAndNotifyOnCompletion(ref requestTask, value);
    }

    public void RequestValue()
    {
        RequestTask = WebService.LoadMyValueAsync();
    }
}
```

Here the `SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>, Task<T>, string)` method will take care of updating the target field, monitoring the new task, if present, and raising the notification event when that task completes. This way, it's possible to just bind to a task property and to be notified when its status changes. The TaskNotifier<T> is a special type exposed by `ObservableObject` that wraps a target `Task<T>` instance and enables the necessary notification logic for this method. The TaskNotifier type is also available to use directly if you have a general `Task` only.

> ⓘ **Note**
>
> The `SetPropertyAndNotifyOnCompletion` method is meant to replace the usage of the `NotifyTaskCompletion<T>` type from the `Microsoft.Toolkit` package. If this type was being used, it can be replaced with just the inner `Task` (or `Task<TResult>`) property, and then the `SetPropertyAndNotifyOnCompletion` method can be used to set its value and raise notification changes. All the properties exposed by the `NotifyTaskCompletion<T>` type are available directly on `Task` instances.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

ObservableRecipient

Article • 04/07/2022

The ObservableRecipient type is a base class for observable objects that also acts as recipients for messages. This class is an extension of ObservableObject which also provides built-in support to use the IMessenger type.

> **Platform APIs:** ObservableRecipient, ObservableObject, IMessenger, WeakReferenceMessenger, IRecipient<TMessage>, PropertyChangedMessage<T>

How it works

The `ObservableRecipient` type is meant to be used as a base for viewmodels that also use the `IMessenger` features, as it provides built-in support for it. In particular:

- It has both a parameterless constructor and one that takes an `IMessenger` instance, to be used with dependency injection. It also exposes a `Messenger` property that can be used to send and receive messages in the viewmodel. If the parameterless constructor is used, the `WeakReferenceMessenger.Default` instance will be assigned to the `Messenger` property.
- It exposes an `IsActive` property to activate/deactivate the viewmodel. In this context, to "activate" means that a given viewmodel is marked as being in use, such that eg. it will start listening for registered messages, perform other setup operations, etc. There are two related methods, `OnActivated` and `OnDeactivated`, that are invoked when the property changes value. By default, `OnDeactivated` automatically unregisters the current instance from all registered messages. For best results and to avoid memory leaks, it's recommended to use `OnActivated` to register to messages, and to use `OnDeactivated` to do cleanup operations. This pattern allows a viewmodel to be enabled/disabled multiple times, while being safe to collect without the risk of memory leaks every time it's deactivated. By default, `OnActivated` will automatically register all the message handlers defined through the `IRecipient<TMessage>` interface.
- It exposes a `Broadcast<T>(T, T, string)` method which sends a `PropertyChangedMessage<T>` message through the `IMessenger` instance available from the `Messenger` property. This can be used to easily broadcast changes in the properties of a viewmodel without having to manually retrieve a `Messenger` instance to use. This method is used by the overload of the various `SetProperty`

methods, which have an additional `bool broadcast` property to indicate whether or not to also send a message.

Here's an example of a viewmodel that receives `LoggedInUserRequestMessage` messages when active:

C#

```
public class MyViewModel : ObservableRecipient,
IRecipient<LoggedInUserRequestMessage>
{
    public void Receive(LoggedInUserRequestMessage message)
    {
        // Handle the message here
    }
}
```

In the example above, `OnActivated` automatically registers the instance as a recipient for `LoggedInUserRequestMessage` messages, using that method as the action to invoke. Using the `IRecipient<TMessage>` interface is not mandatory, and the registration can also be done manually (even using just an inline lambda expression):

C#

```
public class MyViewModel : ObservableRecipient
{
    protected override void OnActivated()
    {
        // Using a method group...
        Messenger.Register<MyViewModel, LoggedInUserRequestMessage>(this,
(r, m) => r.Receive(m));

        // ...or a lambda expression
        Messenger.Register<MyViewModel, LoggedInUserRequestMessage>(this,
(r, m) =>
        {
            // Handle the message here
        });
    }

    private void Receive(LoggedInUserRequestMessage message)
    {
        // Handle the message here
    }
}
```

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

ObservableValidator

Article • 02/01/2023

The ObservableValidator is a base class implementing the INotifyDataErrorInfo interface, providing support for validating properties exposed to other application modules. It also inherits from `ObservableObject`, so it implements INotifyPropertyChanged and INotifyPropertyChanging as well. It can be used as a starting point for all kinds of objects that need to support both property change notifications and property validation.

> **Platform APIs:** ObservableValidator, ObservableObject

How it works

`ObservableValidator` has the following main features:

- It provides a base implementation for `INotifyDataErrorInfo`, exposing the `ErrorsChanged` event and the other necessary APIs.
- It provides a series of additional `SetProperty` overloads (on top of the ones provided by the base `ObservableObject` class), that offer the ability of automatically validating properties and raising the necessary events before updating their values.
- It exposes a number of `TrySetProperty` overloads, that are similar to `SetProperty` but with the ability of only updating the target property if the validation is successful, and to return the generated errors (if any) for further inspection.
- It exposes the `ValidateProperty` method, which can be useful to manually trigger the validation of a specific property in case its value has not been updated but its validation is dependent on the value of another property that has instead been updated.
- It exposes the `ValidateAllProperties` method, which automatically executes the validation of all public instance properties in the current instance, provided they have at least one [ValidationAttribute] applied to them.
- It exposes a `ClearAllErrors` method that can be useful when resetting a model bound to some form that the user might want to fill in again.
- It offers a number of constructors that allow passing different parameters to initialize the ValidationContext instance that will be used to validate properties. This can be especially useful when using custom validation attributes that might require additional services or options to work correctly.

Simple property

Here's an example of how to implement a property that supports both change notifications as well as validation:

C#

```
public class RegistrationForm : ObservableValidator
{
    private string name;

    [Required]
    [MinLength(2)]
    [MaxLength(100)]
    public string Name
    {
        get => name;
        set => SetProperty(ref name, value, true);
    }
}
```

Here we are calling the `SetProperty<T>(ref T, T, bool, string)` method exposed by `ObservableValidator`, and that additional `bool` parameter set to `true` indicates that we also want to validate the property when its value is updated. `ObservableValidator` will automatically run the validation on every new value using all the checks that are specified with the attributes applied to the property. Other components (such as UI controls) can then interact with the viewmodel and modify their state to reflect the errors currently present in the viewmodel, by registering to `ErrorsChanged` and using the `GetErrors(string)` method to retrieve the list of errors for each property that has been modified.

Custom validation methods

Sometimes validating a property requires a viewmodel to have access to additional services, data, or other APIs. There are different ways to add custom validation to a property, depending on the scenario and the level of flexibility that is required. Here is an example of how the [CustomValidationAttribute] type can be used to indicate that a specific method needs to be invoked to perform additional validation of a property:

C#

```
public class RegistrationForm : ObservableValidator
{
    private readonly IFancyService service;

    public RegistrationForm(IFancyService service)
    {
        this.service = service;
```

```
    }

    private string name;

    [Required]
    [MinLength(2)]
    [MaxLength(100)]
    [CustomValidation(typeof(RegistrationForm), nameof(ValidateName))]
    public string Name
    {
        get => this.name;
        set => SetProperty(ref this.name, value, true);
    }

    public static ValidationResult ValidateName(string name,
ValidationContext context)
    {
        RegistrationForm instance =
(RegistrationForm)context.ObjectInstance;
        bool isValid = instance.service.Validate(name);

        if (isValid)
        {
            return ValidationResult.Success;
        }

        return new("The name was not validated by the fancy service");
    }
}
```

In this case we have a static `ValidateName` method that will perform validation on the `Name` property through a service that is injected into our viewmodel. This method receives the `name` property value and the `ValidationContext` instance in use, which contains things such as the viewmodel instance, the name of the property being validated, and optionally a service provider and some custom flags we can use or set. In this case, we are retrieving the `RegistrationForm` instance from the validation context, and from there we are using the injected service to validate the property. Note that this validation will be executed next to the ones specified in the other attributes, so we are free to combine custom validation methods and existing validation attributes however we like.

Custom validation attributes

Another way of doing custom validation is by implementing a custom `[ValidationAttribute]` and then inserting the validation logic into the overridden `IsValid` method. This enables extra flexibility compared to the approach described above, as it makes it very easy to just reuse the same attribute in multiple places.

Suppose we wanted to validate a property based on its relative value with respect to another property in the same viewmodel. The first step would be to define a custom `[GreaterThanAttribute]`, like so:

C#

```
public sealed class GreaterThanAttribute : ValidationAttribute
{
    public GreaterThanAttribute(string propertyName)
    {
        PropertyName = propertyName;
    }

    public string PropertyName { get; }

    protected override ValidationResult IsValid(object value,
ValidationContext validationContext)
    {
        object
            instance = validationContext.ObjectInstance,
            otherValue =
instance.GetType().GetProperty(PropertyName).GetValue(instance);

        if (((IComparable)value).CompareTo(otherValue) > 0)
        {
            return ValidationResult.Success;
        }

        return new("The current value is smaller than the other one");
    }
}
```

Next we can add this attribute into our viewmodel:

C#

```
public class ComparableModel : ObservableValidator
{
    private int a;

    [Range(10, 100)]
    [GreaterThan(nameof(B))]
    public int A
    {
        get => this.a;
        set => SetProperty(ref this.a, value, true);
    }

    private int b;

    [Range(20, 80)]
    public int B
```

```
    {
        get => this.b;
        set
        {
            SetProperty(ref this.b, value, true);
            ValidateProperty(A, nameof(A));
        }
    }
}
```

In this case, we have two numerical properties that must be in a specific range and with a specific relationship between each other (`A` needs to be greater than `B`). We have added the new `[GreaterThanAttribute]` over the first property, and we also added a call to `ValidateProperty` in the setter for `B`, so that `A` is validated again whenever `B` changes (since its validation status depends on it). We just need these two lines of code in our viewmodel to enable this custom validation, and we also get the benefit of having a reusable custom validation attribute that could be useful in other viewmodels in our application as well. This approach also helps with code modularization, as the validation logic is now completely decoupled from the viewmodel definition itself.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

RelayCommand and RelayCommand<T>

Article • 04/07/2022

The RelayCommand and RelayCommand<T> are `ICommand` implementations that can expose a method or delegate to the view. These types act as a way to bind commands between the viewmodel and UI elements.

> **Platform APIs:** RelayCommand, RelayCommand<T>, IRelayCommand, IRelayCommand<T>

How they work

`RelayCommand` and `RelayCommand<T>` have the following main features:

- They provide a base implementation of the `ICommand` interface.
- They also implement the IRelayCommand (and IRelayCommand<T>) interface, which exposes a `NotifyCanExecuteChanged` method to raise the `CanExecuteChanged` event.
- They expose constructors taking delegates like `Action` and `Func<T>`, which allow the wrapping of standard methods and lambda expressions.

Working with `ICommand`

The following shows how to set up a simple command:

C#

```
public class MyViewModel : ObservableObject
{
    public MyViewModel()
    {
        IncrementCounterCommand = new RelayCommand(IncrementCounter);
    }

    private int counter;

    public int Counter
    {
        get => counter;
        private set => SetProperty(ref counter, value);
    }
```

```
    public ICommand IncrementCounterCommand { get; }

    private void IncrementCounter() => Counter++;
}
```

And the relative UI could then be (using WinUI XAML):

XML

```
<Page
    x:Class="MyApp.Views.MyPage"
    xmlns:viewModels="using:MyApp.ViewModels">
    <Page.DataContext>
        <viewModels:MyViewModel x:Name="ViewModel"/>
    </Page.DataContext>

    <StackPanel Spacing="8">
        <TextBlock Text="{x:Bind ViewModel.Counter, Mode=OneWay}"/>
        <Button
            Content="Click me!"
            Command="{x:Bind ViewModel.IncrementCounterCommand}"/>
    </StackPanel>
</Page>
```

The `Button` binds to the `ICommand` in the viewmodel, which wraps the private `IncrementCounter` method. The `TextBlock` displays the value of the `Counter` property and is updated every time the property value changes.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

AsyncRelayCommand and AsyncRelayCommand<T>

Article • 07/15/2022

The AsyncRelayCommand and AsyncRelayCommand<T> are `ICommand` implementations that extend the functionalities offered by RelayCommand, with support for asynchronous operations.

> **Platform APIs:** AsyncRelayCommand, AsyncRelayCommand<T>, RelayCommand, IAsyncRelayCommand, IAsyncRelayCommand<T>

How they work

`AsyncRelayCommand` and `AsyncRelayCommand<T>` have the following main features:

- They extend the functionalities of the synchronous commands included in the library, with support for `Task`-returning delegates.
- They can wrap asynchronous functions with an additional `CancellationToken` parameter to support cancelation, and they expose a `CanBeCanceled` and `IsCancellationRequested` properties, as well as a `Cancel` method.
- They expose an `ExecutionTask` property that can be used to monitor the progress of a pending operation, and an `IsRunning` that can be used to check when an operation completes. This is particularly useful to bind a command to UI elements such as loading indicators.
- They implement the IAsyncRelayCommand and IAsyncRelayCommand<T> interfaces, which means that viewmodel can easily expose commands using these to reduce the tight coupling between types. For instance, this makes it easier to replace a command with a custom implementation exposing the same public API surface, if needed.

Working with asynchronous commands

Let's imagine a scenario similar to the one described in the `RelayCommand` sample, but a command executing an asynchronous operation:

C#

```
public class MyViewModel : ObservableObject
{
```

```
    public MyViewModel()
    {
        DownloadTextCommand = new AsyncRelayCommand(DownloadText);
    }

    public IAsyncRelayCommand DownloadTextCommand { get; }

    private Task<string> DownloadText()
    {
        return WebService.LoadMyTextAsync();
    }
}
```

With the related UI code:

XML

```
<Page
    x:Class="MyApp.Views.MyPage"
    xmlns:viewModels="using:MyApp.ViewModels"
    xmlns:converters="using:Microsoft.Toolkit.Uwp.UI.Converters">
    <Page.DataContext>
        <viewModels:MyViewModel x:Name="ViewModel"/>
    </Page.DataContext>
    <Page.Resources>
        <converters:TaskResultConverter x:Key="TaskResultConverter"/>
    </Page.Resources>

    <StackPanel Spacing="8" xml:space="default">
        <TextBlock>
            <Run Text="Task status:"/>
            <Run Text="{x:Bind
ViewModel.DownloadTextCommand.ExecutionTask.Status, Mode=OneWay}"/>
            <LineBreak/>
            <Run Text="Result:"/>
            <Run Text="{x:Bind ViewModel.DownloadTextCommand.ExecutionTask,
Converter={StaticResource TaskResultConverter}, Mode=OneWay}"/>
        </TextBlock>
        <Button
            Content="Click me!"
            Command="{x:Bind ViewModel.DownloadTextCommand}"/>
        <ProgressRing
            HorizontalAlignment="Left"
            IsActive="{x:Bind ViewModel.DownloadTextCommand.IsRunning,
Mode=OneWay}"/>
    </StackPanel>
</Page>
```

Upon clicking the `Button`, the command is invoked, and the `ExecutionTask` updated. When the operation completes, the property raises a notification which is reflected in the UI. In this case, both the task status and the current result of the task are displayed.

Note that to show the result of the task, it is necessary to use the `TaskExtensions.GetResultOrDefault` method - this provides access to the result of a task that has not yet completed without blocking the thread (and possibly causing a deadlock).

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

Ioc (Inversion of control)

Article • 07/15/2022

A common pattern that can be used to increase modularity in the codebase of an application using the MVVM pattern is to use some form of inversion of control. One of the most common solution in particular is to use dependency injection, which consists in creating a number of services that are injected into backend classes (ie. passed as parameters to the viewmodel constructors) - this allows code using these services not to rely on implementation details of these services, and it also makes it easy to swap the concrete implementations of these services. This pattern also makes it easy to make platform-specific features available to backend code, by abstracting them through a service which is then injected where needed.

The MVVM Toolkit doesn't provide built-in APIs to facilitate the usage of this pattern, as there already exist dedicated libraries specifically for this such as the `Microsoft.Extensions.DependencyInjection` package, which provides a fully featured and powerful DI set of APIs, and acts as an easy to setup and use `IServiceProvider`. The following guide will refer to this library and provide a series of examples of how to integrate it into applications using the MVVM pattern.

> **Platform APIs:** Ioc

Configure and resolve services

The first step is to declare an `IServiceProvider` instance, and to initialize all the necessary services, usually at startup. For instance, on UWP (but a similar setup can be used on other frameworks too):

C#

```
public sealed partial class App : Application
{
    public App()
    {
        Services = ConfigureServices();

        this.InitializeComponent();
    }

    /// <summary>
    /// Gets the current <see cref="App"/> instance in use
    /// </summary>
    public new static App Current => (App)Application.Current;
```

```
    /// <summary>
    /// Gets the <see cref="IServiceProvider"/> instance to resolve
application services.
    /// </summary>
    public IServiceProvider Services { get; }

    /// <summary>
    /// Configures the services for the application.
    /// </summary>
    private static IServiceProvider ConfigureServices()
    {
        var services = new ServiceCollection();

        services.AddSingleton<IFilesService, FilesService>();
        services.AddSingleton<ISettingsService, SettingsService>();
        services.AddSingleton<IClipboardService, ClipboardService>();
        services.AddSingleton<IShareService, ShareService>();
        services.AddSingleton<IEmailService, EmailService>();

        return services.BuildServiceProvider();
    }
}
```

Here the `Services` property is initialized at startup, and all the application services and viewmodels are registered. There is also a new `Current` property that can be used to easily access the `Services` property from other views in the application. For instance:

C#

```
IFilesService filesService = App.Current.Services.GetService<IFilesService>
();

// Use the files service here...
```

The key aspect here is that each service may very well be using platform-specific APIs, but since those are all abstracted away through the interface our code is using, we don't need to worry about them whenever we're just resolving an instance and using it to perform operations.

Constructor injection

One powerful feature that is available is "constructor injection", which means that the DI service provider is able to automatically resolve indirect dependencies between registered services when creating instances of the type being requested. Consider the following service:

C#

```
public class FileLogger : IFileLogger
{
    private readonly IFilesService FileService;
    private readonly IConsoleService ConsoleService;

    public FileLogger(
        IFilesService fileService,
        IConsoleService consoleService)
    {
        FileService = fileService;
        ConsoleService = consoleService;
    }

    // Methods for the IFileLogger interface here...
}
```

Here we have a `FileLogger` type implementing the `IFileLogger` interface, and requiring `IFilesService` and `IConsoleService` instances. Constructor injection means the DI service provider will automatically gather all the necessary services, like so:

C#

```
/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<IFilesService, FilesService>();
    services.AddSingleton<IConsoleService, ConsoleService>();
    services.AddSingleton<IFileLogger, FileLogger>();

    return services.BuildServiceProvider();
}

// Retrieve a logger service with constructor injection
IFileLogger fileLogger = App.Current.Services.GetService<IFileLogger>();
```

The DI service provider will automatically check whether all the necessary services are registered, then it will retrieve them and invoke the constructor for the registered `IFileLogger` concrete type, to get the instance to return.

What about viewmodels?

A service provider has "service" in its name, but it can actually be used to resolve instances of any class, including viewmodels! The same concepts explained above still

apply, including constructor injection. Imagine we had a `ContactsViewModel` type, using an `IContactsService` and an `IPhoneService` instance through its constructor. We could have a `ConfigureServices` method like this:

C#

```
/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    // Services
    services.AddSingleton<IContactsService, ContactsService>();
    services.AddSingleton<IPhoneService, PhoneService>();

    // Viewmodels
    services.AddTransient<ContactsViewModel>();

    return services.BuildServiceProvider();
}
```

And then in our `ContactsView`, we would assign the data context as follows:

C#

```
public ContactsView()
{
    this.InitializeComponent();
    this.DataContext = App.Current.Services.GetService<ContactsViewModel>();
}
```

More docs

For more info about `Microsoft.Extensions.DependencyInjection`, see here.

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

Messenger

Article • 04/07/2022

The IMessenger interface is a contract for types that can be used to exchange messages between different objects. This can be useful to decouple different modules of an application without having to keep strong references to types being referenced. It is also possible to send messages to specific channels, uniquely identified by a token, and to have different messengers in different sections of an application. The MVVM Toolkit provides two implementations out of the box: WeakReferenceMessenger and StrongReferenceMessenger: the former uses weak references internally, offering automatic memory management for recipients, while the latter uses strong references and requires developers to manually unsubscribe their recipients when they're no longer needed (more details about how to unregister message handlers can be found below), but in exchange for that offers better performance and far less memory usage.

> **Platform APIs:** IMessenger, WeakReferenceMessenger, StrongReferenceMessenger, IRecipient<TMessage>, MessageHandler<TRecipient, TMessage>, ObservableRecipient, RequestMessage<T>, AsyncRequestMessage<T>, CollectionRequestMessage<T>, AsyncCollectionRequestMessage<T>.

How it works

Types implementing `IMessenger` are responsible for maintaining links between recipients (receivers of messages) and their registered message types, with relative message handlers. Any object can be registered as a recipient for a given message type using a message handler, which will be invoked whenever the `IMessenger` instance is used to send a message of that type. It is also possible to send messages through specific communication channels (each identified by a unique token), so that multiple modules can exchange messages of the same type without causing conflicts. Messages sent without a token use the default shared channel.

There are two ways to perform message registration: either through the IRecipient<TMessage> interface, or using a MessageHandler<TRecipient, TMessage> delegate acting as message handler. The first lets you register all the handlers with a single call to the `RegisterAll` extension, which automatically registers the recipients of all the declared message handlers, while the latter is useful when you need more flexibility or when you want to use a simple lambda expression as a message handler.

Both `WeakReferenceMessenger` and `StrongReferenceMessenger` also expose a `Default` property that offers a thread-safe implementation built-in into the package. It is also

possible to create multiple messenger instances if needed, for instance if a different one is injected with a DI service provider into a different module of the app (for instance, multiple windows running in the same process).

> ⓘ **Note**
>
> Since the `WeakReferenceMessenger` type is simpler to use and matches the behavior of the messenger type from the `MvvmLight` library, it is the default type being used by the **ObservableRecipient** type in the MVVM Toolkit. The `StrongReferenceType` can still be used, by passing an instance to the constructor of that class.

Sending and receiving messages

Consider the following:

C#

```
// Create a message
public class LoggedInUserChangedMessage : ValueChangedMessage<User>
{
    public LoggedInUserChangedMessage(User user) : base(user)
    {
    }
}

// Register a message in some module
WeakReferenceMessenger.Default.Register<LoggedInUserChangedMessage>(this,
(r, m) =>
{
    // Handle the message here, with r being the recipient and m being the
    // input message. Using the recipient passed as input makes it so that
    // the lambda expression doesn't capture "this", improving performance.
});

// Send a message from some other module
WeakReferenceMessenger.Default.Send(new LoggedInUserChangedMessage(user));
```

Let's imagine this message type being used in a simple messaging application, which displays a header with the user name and profile image of the currently logged user, a panel with a list of conversations, and another panel with messages from the current conversation, if one is selected. Let's say these three sections are supported by the `HeaderViewModel`, `ConversationsListViewModel` and `ConversationViewModel` types respectively. In this scenario, the `LoggedInUserChangedMessage` message might be sent by the `HeaderViewModel` after a login operation has completed, and both those other

viewmodels might register handlers for it. For instance, `ConversationsListViewModel` will load the list of conversations for the new user, and `ConversationViewModel` will just close the current conversation, if one is present.

The `IMessenger` instance takes care of delivering messages to all the registered recipients. Note that a recipient can subscribe to messages of a specific type. Note that inherited message types are not registered in the default `IMessenger` implementations provided by the MVVM Toolkit.

When a recipient is not needed anymore, you should unregister it so that it will stop receiving messages. You can unregister either by message type, by registration token, or by recipient:

C#

```
// Unregisters the recipient from a message type
WeakReferenceMessenger.Default.Unregister<LoggedInUserChangedMessage>(this);

// Unregisters the recipient from a message type in a specified channel
WeakReferenceMessenger.Default.Unregister<LoggedInUserChangedMessage, int>
(this, 42);

// Unregister the recipient from all messages, across all channels
WeakReferenceMessenger.Default.UnregisterAll(this);
```

> ⚠ **Warning**
>
> As mentioned before, this is not strictly necessary when using the `WeakReferenceMessenger` type, as it uses weak references to track recipients, meaning that unused recipients will still be eligible for garbage collection even though they still have active message handlers. It is still good practice to unsubscribe them though, to improve performances. On the other hand, the `StrongReferenceMessenger` implementation uses strong references to track the registered recipients. This is done for performance reasons, and it means that each registered recipient should manually be unregistered to avoid memory leaks. That is, as long as a recipient is registered, the `StrongReferenceMessenger` instance in use will keep an active reference to it, which will prevent the garbage collector from being able to collect that instance. You can either handle this manually, or you can inherit from `ObservableRecipient`, which by default automatically takes care of removing all the message registrations for recipient when it is deactivated (see docs on `ObservableRecipient` for more info about this).

It is also possible to use the `IRecipient<TMessage>` interface to register message handlers. In this case, each recipient will need to implement the interface for a given message type, and provide a `Receive(TMessage)` method that will be invoke when receiving messages, like so:

C#

```
// Create a message
public class MyRecipient : IRecipient<LoggedInUserChangedMessage>
{
    public void Receive(LoggedInUserChangedMessage message)
    {
        // Handle the message here...
    }
}

// Register that specific message...
WeakReferenceMessenger.Default.Register<LoggedInUserChangedMessage>(this);

// ...or alternatively, register all declared handlers
WeakReferenceMessenger.Default.RegisterAll(this);

// Send a message from some other module
WeakReferenceMessenger.Default.Send(new LoggedInUserChangedMessage(user));
```

Using request messages

Another useful feature of messenger instances is that they can also be used to request values from a module to another. In order to do so, the package includes a base RequestMessage<T> class, which can be used like so:

C#

```
// Create a message
public class LoggedInUserRequestMessage : RequestMessage<User>
{
}

// Register the receiver in a module
WeakReferenceMessenger.Default.Register<MyViewModel,
LoggedInUserRequestMessage>(this, (r, m) =>
{
    // Assume that "CurrentUser" is a private member in our viewmodel.
    // As before, we're accessing it through the recipient passed as
    // input to the handler, to avoid capturing "this" in the delegate.
    m.Reply(r.CurrentUser);
});

// Request the value from another module
```

```
User user = WeakReferenceMessenger.Default.Send<LoggedInUserRequestMessage>
();
```

The `RequestMessage<T>` class includes an implicit converter that makes the conversion from a `LoggedInUserRequestMessage` to its contained `User` object possible. This will also check that a response has been received for the message, and throw an exception if that's not the case. It is also possible to send request messages without this mandatory response guarantee: just store the returned message in a local variable, and then manually check whether a response value is available or not. Doing so will not trigger the automatic exception if a response is not received when the `Send` method returns.

The same namespace also includes base requests message for other scenarios: AsyncRequestMessage<T>, CollectionRequestMessage<T> and AsyncCollectionRequestMessage<T>. Here's how you can use an async request message:

C#

```
// Create a message
public class LoggedInUserRequestMessage : AsyncRequestMessage<User>
{
}

// Register the receiver in a module
WeakReferenceMessenger.Default.Register<MyViewModel,
LoggedInUserRequestMessage>(this, (r, m) =>
{
    m.Reply(r.GetCurrentUserAsync()); // We're replying with a Task<User>
});

// Request the value from another module (we can directly await on the
request)
User user = await
WeakReferenceMessenger.Default.Send<LoggedInUserRequestMessage>();
```

Examples

- Check out the sample app (for multiple UI frameworks) to see the MVVM Toolkit in action.
- You can also find more examples in the unit tests.

Putting things together

Article • 02/07/2023

Now that we've outlined all the different components that are available through the `CommunityToolkit.Mvvm` package, we can look at a practical example of them all coming together to build a single, larger example. In this case, we want to build a very simple and minimalistic Reddit browser for a select number of subreddits.

What do we want to build

Let's start by outlining exactly what we want to build:

- A minimal Reddit browser made up of two "widgets": one showing posts from a subreddit, and the other one showing the currently selected post. The two widget need to be self contained and without strong references to one another.
- We want users to be able to select a subreddit from a list of available options, and we want to save the selected subreddit as a setting and load it up the next time the sample is loaded.
- We want the subreddit widget to also offer a refresh button to reload the current subreddit.
- For the purposes of this sample, we don't need to be able to handle all the possible post types. We'll just assign a sample text to all loaded posts and display that directly, to make things simpler.

Setting up the viewmodels

Let's start with the viewmodel that will power the subreddit widget and let's go over the tools we need:

- **Commands:** we need the view to be able to request the viewmodel to reload the current list of posts from the selected subreddit. We can use the `AsyncRelayCommand` type to wrap a private method that will fetch the posts from Reddit. Here we're exposing the command through the `IAsyncRelayCommand` interface, to avoid strong references to the exact command type we're using. This will also allow us to potentially change the command type in the future without having to worry about any UI component relying on that specific type being used.
- **Properties:** we need to expose a number of values to the UI, which we can do with either observable properties if they're values we intend to completely replace, or

with properties that are themselves observable (eg. `ObservableCollection<T>`). In this case, we have:
    - `ObservableCollection<object> Posts`, which is the observable list of loaded posts. Here we're just using `object` as a placeholder, as we haven't created a model to represent posts yet. We can replace this later on.
    - `IReadOnlyList<string> Subreddits`, which is a readonly list with the names of the subreddits that we allow users to choose from. This property is never updated, so it doesn't need to be observable either.
    - `string SelectedSubreddit`, which is the currently selected subreddit. This property needs to be bound to the UI, as it'll be used both to indicate the last selected subreddit when the sample is loaded, and to be manipulated directly from the UI as the user changes the selection. Here we're using the `SetProperty` method from the `ObservableObject` class.
    - `object SelectedPost`, which is the currently selected post. In this case we're using the `SetProperty` method from the `ObservableRecipient` class to indicate that we also want to broadcast notifications when this property changes. This is done to be able to notify the post widget that the current post selection is changed.
- **Methods:** we just need a private `LoadPostsAsync` method which will be wrapped by our async command, and which will contain the logic to load posts from the selected subreddit.

Here's the viewmodel so far:

C#

```
public sealed class SubredditWidgetViewModel : ObservableRecipient
{
    /// <summary>
    /// Creates a new <see cref="SubredditWidgetViewModel"/> instance.
    /// </summary>
    public SubredditWidgetViewModel()
    {
        LoadPostsCommand = new AsyncRelayCommand(LoadPostsAsync);
    }

    /// <summary>
    /// Gets the <see cref="IAsyncRelayCommand"/> instance responsible for
loading posts.
    /// </summary>
    public IAsyncRelayCommand LoadPostsCommand { get; }

    /// <summary>
    /// Gets the collection of loaded posts.
    /// </summary>
    public ObservableCollection<object> Posts { get; } = new
```

```
ObservableCollection<object>();

    /// <summary>
    /// Gets the collection of available subreddits to pick from.
    /// </summary>
    public IReadOnlyList<string> Subreddits { get; } = new[]
    {
        "microsoft",
        "windows",
        "surface",
        "windowsphone",
        "dotnet",
        "csharp"
    };

    private string selectedSubreddit;

    /// <summary>
    /// Gets or sets the currently selected subreddit.
    /// </summary>
    public string SelectedSubreddit
    {
        get => selectedSubreddit;
        set => SetProperty(ref selectedSubreddit, value);
    }

    private object selectedPost;

    /// <summary>
    /// Gets or sets the currently selected subreddit.
    /// </summary>
    public object SelectedPost
    {
        get => selectedPost;
        set => SetProperty(ref selectedPost, value, true);
    }

    /// <summary>
    /// Loads the posts from a specified subreddit.
    /// </summary>
    private async Task LoadPostsAsync()
    {
        // TODO...
    }
}
```

Now let's take a look at what we need for viewmodel of the post widget. This will be a much simpler viewmodel, as it really only needs to expose a `Post` property with the currently selected post, and to receive broadcast messages from the subreddit widget to update the `Post` property. It can look something like this:

```
C#
```

```
public sealed class PostWidgetViewModel : ObservableRecipient,
IRecipient<PropertyChangedMessage<object>>
{
    private object post;

    /// <summary>
    /// Gets the currently selected post, if any.
    /// </summary>
    public object Post
    {
        get => post;
        private set => SetProperty(ref post, value);
    }

    /// <inheritdoc/>
    public void Receive(PropertyChangedMessage<object> message)
    {
        if (message.Sender.GetType() == typeof(SubredditWidgetViewModel) &&
            message.PropertyName ==
nameof(SubredditWidgetViewModel.SelectedPost))
        {
            Post = message.NewValue;
        }
    }
}
```

In this case, we're using the `IRecipient<TMessage>` interface to declare the messages we want our viewmodel to receive. The handlers for the declared messages will be added automatically by the `ObservableRecipient` class when the `IsActive` property is set to `true`. Note that it is not mandatory to use this approach, and manually registering each message handler is also possible, like so:

C#

```
public sealed class PostWidgetViewModel : ObservableRecipient
{
    protected override void OnActivated()
    {
        // We use a method group here, but a lambda expression is also valid
        Messenger.Register<PostWidgetViewModel,
PropertyChangedMessage<object>>(this, (r, m) => r.Receive(m));
    }

    /// <inheritdoc/>
    public void Receive(PropertyChangedMessage<object> message)
    {
        if (message.Sender.GetType() == typeof(SubredditWidgetViewModel) &&
            message.PropertyName ==
nameof(SubredditWidgetViewModel.SelectedPost))
        {
            Post = message.NewValue;
```

```
        }
    }
}
```

We now have a draft of our viewmodels ready, and we can start looking into the services we need.

Building the settings service

> **Note**
>
> The sample is built using the dependency injection pattern, which is the recommended approach to deal with services in viewmodels. It is also possible to use other patterns, such as the service locator pattern, but the MVVM Toolkit does not offer built-in APIs to enable that.

Since we want some of our properties to be saved and persisted, we need a way for viewmodels to be able to interact with the application settings. We shouldn't use platform-specific APIs directly in our viewmodels though, as that would prevent us from having all our viewmodels in a portable, .NET Standard project. We can solve this issue by using services, and the APIs in the `Microsoft.Extensions.DependencyInjection` library to setup our `IServiceProvider` instance for the application. The idea is to write interfaces that represent all the API surface that we need, and then to implement platform-specific types implementing this interface on all our application targets. The viewmodels will only interact with the interfaces, so they will not have any strong reference to any platform-specific type at all.

Here's a simple interface for a settings service:

C#

```
public interface ISettingsService
{
    /// <summary>
    /// Assigns a value to a settings key.
    /// </summary>
    /// <typeparam name="T">The type of the object bound to the key.
</typeparam>
    /// <param name="key">The key to check.</param>
    /// <param name="value">The value to assign to the setting key.</param>
    void SetValue<T>(string key, T value);

    /// <summary>
    /// Reads a value from the current <see cref="IServiceProvider"/>
```

```
instance and returns its casting in the right type.
    /// </summary>
    /// <typeparam name="T">The type of the object to retrieve.</typeparam>
    /// <param name="key">The key associated to the requested object.
</param>
    [Pure]
    T GetValue<T>(string key);
}
```

We can assume that platform-specific types implementing this interface will take care of dealing with all the logic necessary to actually serialize the settings, store them to disk and then read them back. We can now use this service in our `SubredditWidgetViewModel`, in order to make the `SelectedSubreddit` property persistent:

C#

```
/// <summary>
/// Gets the <see cref="ISettingsService"/> instance to use.
/// </summary>
private readonly ISettingsService SettingsService;

/// <summary>
/// Creates a new <see cref="SubredditWidgetViewModel"/> instance.
/// </summary>
public SubredditWidgetViewModel(ISettingsService settingsService)
{
    SettingsService = settingsService;

    selectedSubreddit = settingsService.GetValue<string>
(nameof(SelectedSubreddit)) ?? Subreddits[0];
}

private string selectedSubreddit;

/// <summary>
/// Gets or sets the currently selected subreddit.
/// </summary>
public string SelectedSubreddit
{
    get => selectedSubreddit;
    set
    {
        SetProperty(ref selectedSubreddit, value);

        SettingsService.SetValue(nameof(SelectedSubreddit), value);
    }
}
```

Here we're using dependency injection and constructor injection, as mentioned above. We've declared an `ISettingsService SettingsService` field that just stores our settings

service (which we're receiving as parameter in the viewmodel constructor), and then we're initializing the `SelectedSubreddit` property in the constructor, by either using the previous value or just the first available subreddit. Then we also modified the `SelectedSubreddit` setter, so that it will also use the settings service to save the new value to disk.

Great! Now we just need to write a platform specific version of this service, this time directly inside one of our app projects. Here's what that service might look like on UWP:

C#

```
public sealed class SettingsService : ISettingsService
{
    /// <summary>
    /// The <see cref="IPropertySet"/> with the settings targeted by the
current instance.
    /// </summary>
    private readonly IPropertySet SettingsStorage =
ApplicationData.Current.LocalSettings.Values;

    /// <inheritdoc/>
    public void SetValue<T>(string key, T value)
    {
        if (!SettingsStorage.ContainsKey(key)) SettingsStorage.Add(key,
value);
        else SettingsStorage[key] = value;
    }

    /// <inheritdoc/>
    public T GetValue<T>(string key)
    {
        if (SettingsStorage.TryGetValue(key, out object value))
        {
            return (T)value;
        }

        return default;
    }
}
```

The final piece of the puzzle is to inject this platform-specific service into our service provider instance. We can do this at startup, like so:

C#

```
/// <summary>
/// Gets the <see cref="IServiceProvider"/> instance to resolve application
services.
/// </summary>
public IServiceProvider Services { get; }
```

```
/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<ISettingsService, SettingsService>();
    services.AddTransient<PostWidgetViewModel>();

    return services.BuildServiceProvider();
}
```

This will register a singleton instance of our `SettingsService` as a type implementing `ISettingsService`. We are also registering the `PostWidgetViewModel` as a transient service, meaning every time we retrieve an instance, it will be a new one (you can imagine this being useful if wanted to have multiple, independent post widgets). This means that every time we resolve an `ISettingsService` instance while the app in use is the UWP one, it will receive a `SettingsService` instance, which will use the UWP APIs behind the scene to manipulate settings. Perfect!

Building the Reddit service

The last component of the backend that we're missing is a service that is able to use the Reddit REST APIs to fetch the posts from the subreddits we're interested in. To build it, we're going to use refit, which is a library to easily build type-safe services to interact with REST APIs. As before, we need to define the interface with all the APIs that our service will implement, like so:

C#

```
public interface IRedditService
{
    /// <summary>
    /// Get a list of posts from a given subreddit
    /// </summary>
    /// <param name="subreddit">The subreddit name.</param>
    [Get("/r/{subreddit}/.json")]
    Task<PostsQueryResponse> GetSubredditPostsAsync(string subreddit);
}
```

That `PostsQueryResponse` is a model we wrote that maps the JSON response for that API. The exact structure of that class is not important - suffice to say that it contains a collection of `Post` items, which are simple models representing our posts, like this:

C#

```
public class Post
{
    /// <summary>
    /// Gets or sets the title of the post.
    /// </summary>
    public string Title { get; set; }

    /// <summary>
    /// Gets or sets the URL to the post thumbnail, if present.
    /// </summary>
    public string Thumbnail { get; set; }

    /// <summary>
    /// Gets the text of the post.
    /// </summary>
    public string SelfText { get; }
}
```

Once we have our service and our models, we can plug them into our viewmodels to complete our backend. While doing so, we can also replace those `object` placeholders with the `Post` type we've defined:

C#

```
public sealed class SubredditWidgetViewModel : ObservableRecipient
{
    /// <summary>
    /// Gets the <see cref="IRedditService"/> instance to use.
    /// </summary>
    private readonly IRedditService RedditService =
Ioc.Default.GetRequiredService<IRedditService>();

    /// <summary>
    /// Loads the posts from a specified subreddit.
    /// </summary>
    private async Task LoadPostsAsync()
    {
        var response = await
RedditService.GetSubredditPostsAsync(SelectedSubreddit);

        Posts.Clear();

        foreach (var item in response.Data.Items)
        {
            Posts.Add(item.Data);
        }
    }
}
```

We have added a new `IRedditService` field to store our service, just like we did for the settings service, and we implemented our `LoadPostsAsync` method, which was previously empty.

The last missing piece now is just to inject the actual service into our service provider. The big difference in this case is that by using `refit` we don't actually need to implement the service at all! The library will automatically create a type implementing the service for us, behind the scenes. So we only need to get an `IRedditService` instance and inject it directly, like so:

C#

```
/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<ISettingsService, SettingsService>();
    services.AddSingleton(RestService.For<IRedditService>
("https://www.reddit.com/"));
    services.AddTransient<PostWidgetViewModel>();

    return services.BuildServiceProvider();
}
```

And that's all we need to do! We now have all our backend ready to use, including two custom services that we created specifically for this app! 🥕

Building the UI

Now that all the backend is completed, we can write the UI for our widgets. Note how using the MVVM pattern let us focus exclusively on the business logic at first, without having to write any UI-related code until now. Here we'll remove all the UI code that's not interacting with our viewmodels, for simplicity, and we'll go through each different control one by one. The full source code can be found in the sample app.

Before going through the various controls, here's how we can resolve viewmodels for all the different views in our application (eg. the `PostWidgetView`):

C#

```
public PostWidgetView()
{
```

```
    this.InitializeComponent();
    this.DataContext = App.Current.Services.GetService<PostWidgetViewModel>
();
}

public PostWidgetViewModel ViewModel => (PostWidgetViewModel)DataContext;
```

We're using our `IServiceProvider` instance to resolve the `PostWidgetViewModel` object we need, which is then assigned to the data context property. We're also creating a strongly-typed `ViewModel` property that simply casts the data context to the correct viewmodel type - this is needed to enable `x:Bind` in the XAML code.

Let's start with the subreddit widget, which features a `ComboBox` to select a subreddit, a `Button` to refresh the feed, a `ListView` to display posts and a `ProgressBar` to indicate when the feed is loading. We'll assume that the `ViewModel` property represents an instance of the viewmodel we've described before - this can be declared either in XAML or directly in code behind.

Subreddit selector:

XML

```
<ComboBox
    ItemsSource="{x:Bind ViewModel.Subreddits}"
    SelectedItem="{x:Bind ViewModel.SelectedSubreddit, Mode=TwoWay}">
    <interactivity:Interaction.Behaviors>
        <core:EventTriggerBehavior EventName="SelectionChanged">
            <core:InvokeCommandAction Command="{x:Bind
ViewModel.LoadPostsCommand}"/>
        </core:EventTriggerBehavior>
    </interactivity:Interaction.Behaviors>
</ComboBox>
```

Here we're binding the source to the `Subreddits` property, and the selected item to the `SelectedSubreddit` property. Note how the `Subreddits` property is only bound once, as the collection itself sends change notifications, while the `SelectedSubreddit` property is bound with the `TwoWay` mode, as we need it both to be able to load the value we retrieve from our settings, as well as updating the property in the viewmodel when the user changes the selection. Additionally, we're using a XAML behavior to invoke our command whenever the selection changes.

Refresh button:

XML

```
<Button Command="{x:Bind ViewModel.LoadPostsCommand}"/>
```

This component is extremely simple, we're just binding our custom command to the `Command` property of the button, so that the command will be invoked whenever the user clicks on it.

Posts list:

XML

```
<ListView
    ItemsSource="{x:Bind ViewModel.Posts}"
    SelectedItem="{x:Bind ViewModel.SelectedPost, Mode=TwoWay}">
    <ListView.ItemTemplate>
        <DataTemplate x:DataType="models:Post">
            <Grid>
                <TextBlock Text="{x:Bind Title}"/>
                <controls:ImageEx Source="{x:Bind Thumbnail}"/>
            </Grid>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Here we have a `ListView` binding the source and selection to our viewmodel property, and also a template used to display each post that is available. We're using `x:DataType` to enable `x:Bind` in our template, and we have two controls binding directly to the `Title` and `Thumbnail` properties of our post.

Loading bar:

XML

```
<ProgressBar Visibility="{x:Bind ViewModel.LoadPostsCommand.IsRunning,
Mode=OneWay}"/>
```

Here we're binding to the `IsRunning` property, which is part of the `IAsyncRelayCommand` interface. The `AsyncRelayCommand` type will take care of raising notifications for that property whenever the asynchronous operation starts or completes for that command.

The last missing piece is the UI for the post widget. As before, we've removed all the UI-related code that was not necessary to interact with the viewmodels, for simplicity. The

full source code is available in the sample app.

XML

```
<Grid>

    <!--Header-->
    <Grid>
        <TextBlock Text="{x:Bind ViewModel.Post.Title, Mode=OneWay}"/>
        <controls:ImageEx  Source="{x:Bind ViewModel.Post.Thumbnail,
Mode=OneWay}"/>
    </Grid>

    <!--Content-->
    <ScrollViewer>
        <TextBlock Text="{x:Bind ViewModel.Post.SelfText, Mode=OneWay}"/>
    </ScrollViewer>
</Grid>
```

Here we just have a header, with a `TextBlock` and an `ImageEx` control binding their `Text` and `Source` properties to the respective properties in our `Post` model, and a simple `TextBlock` inside a `ScrollViewer` that is used to display the (sample) content of the selected post.

Sample Application

Sample application available here.

Good to go! 🚀

We've now built all our viewmodels, the necessary services, and the UI for our widgets - our simple Reddit browser is completed! This was just meant to be an example of how to build an app following the MVVM pattern and using the APIs from the MVVM Toolkit.

As stated above, this is only a reference, and you're free to modify this structure to fit your needs and/or to pick and choose only a subset of components from the library. Regardless of the approach you take, the MVVM Toolkit should provide a solid foundation to hit the ground running when starting a new application, by letting you focus on your business logic instead of having to worry about manually doing all the necessary plumbing to enable proper support for the MVVM pattern.

Migrating from MvvmLight

Article • 01/18/2023

This article outlines some of the key differences between the MvvmLight Toolkit and the MVVM Toolkit to ease your migration.

While this article specifically focuses on the migrations from MvvmLight to the MVVM Toolkit, note that there are additional improvements that have been made within the MVVM Toolkit, so it is highly recommend taking a look at the documentation for the individual new APIs.

> **Platform APIs:** ObservableObject, ObservableRecipient, RelayCommand, RelayCommand<T>, AsyncRelayCommand, AsyncRelayCommand<T>, IMessenger, WeakReferenceMessenger, StrongReferenceMessenger, IRecipient<TMessage>, MessageHandler<TRecipient, TMessage>, IMessengerExtensions

Installing the MVVM Toolkit

To take advantage of the MVVM Toolkit, you'll first need to install the latest NuGet package to your existing .NET application.

Install via .NET CLI

```
dotnet add package CommunityToolkit.Mvvm --version 8.1.0
```

Install via PackageReference

XML

```
<PackageReference Include="CommunityToolkit.Mvvm" Version="8.1.0" />
```

Migrating ObservableObject

The following steps focus on migrating your existing components which take advantage of the `ObservableObject` of the MvvmLight Toolkit. The MVVM Toolkit provides an ObservableObject type that is similar.

The first change here will be swapping using directives in your components.

```
C#

// MvvmLight
using GalaSoft.MvvmLight;

// MVVM Toolkit
using CommunityToolkit.Mvvm.ComponentModel;
```

Below are a list of migrations that will need to be performed if being used in your current solution.

ObservableObject methods

`Set<T>(Expression, ref T, T)`

`Set(Expression, ref T, T)` does not have a like-for-like method signature replacement.

However, `SetProperty(ref T, T, string)` provides the same functionality with additional performance benefits.

```
C#

// MvvmLight
Set(() => MyProperty, ref this.myProperty, value);

// MVVM Toolkit
SetProperty(ref this.myProperty, value);
```

Note that the `string` parameter is not required if the method is being called from the property's setter as it is inferred from the caller member name, as can be seen here. If you want to invoke `SetProperty` for a property that is different from the one where the method is being invoked, you can do so by using the `nameof` operator, which can be useful to make the code less error prone by not having hardcoded names. For instance:

```
C#

SetProperty(ref this.someProperty, value, nameof(SomeProperty));
```

`Set<T>(string, ref T, T)`

`Set<T>(string, ref T, T)` does not have a like-for-like method signature replacement.

However, `SetProperty<T>(ref T, T, string)` provides the same functionality with re-ordered parameters.

C#

```
// MvvmLight
Set(nameof(MyProperty), ref this.myProperty, value);

// MVVM Toolkit
SetProperty(ref this.myProperty, value);
```

`Set<T>(ref T, T, string)`

`Set<T>(ref T, T, string)` has a renamed direct replacement, `SetProperty<T>(ref T, T, string)`.

C#

```
// MvvmLight
Set(ref this.myProperty, value, nameof(MyProperty));

// MVVM Toolkit
SetProperty(ref this.myProperty, value);
```

`RaisePropertyChanged(string)`

`RaisePropertyChanged(string)` has a renamed direct replacement, `OnPropertyChanged(string)`.

C#

```
// MvvmLight
RaisePropertyChanged(nameof(MyProperty));

// MVVM Toolkit
OnPropertyChanged();
```

As with `SetProperty`, the name of the current property is automatically inferred by the `OnPropertyChanged` method. If you want to use this method to manually raise the `PropertyChanged` event for another property, you can also manually specify the name of that property by using the `nameof` operator again. For instance:

C#

```
OnPropertyChanged(nameof(SomeProperty));
```

`RaisePropertyChanged<T>(Expression)`

`RaisePropertyChanged<T>(Expression)` does not have a direct replacement.

It is recommended for improved performance that you replace `RaisePropertyChanged<T>(Expression)` with the Toolkit's `OnPropertyChanged(string)` using the `nameof` keyword instead (or with no parameters, if the target property is the same as the one calling the method, so the name can be inferred automatically as mentioned above).

C#

```
// MvvmLight
RaisePropertyChanged(() => MyProperty);

// MVVM Toolkit
OnPropertyChanged(nameof(MyProperty));
```

`VerifyPropertyName(string)`

There is no direct replacement for the `VerifyPropertyName(string)` method and any code using this should be altered or removed.

The reason for the omission from the MVVM Toolkit is that using the `nameof` keyword for a property verifies that it exists. When MvvmLight was built, the `nameof` keyword was not available and this method was used to ensure that the property existed on the object.

C#

```
// MvvmLight
VerifyPropertyName(nameof(MyProperty));

// MVVM Toolkit
// No direct replacement, remove
```

ObservableObject properties

`PropertyChangedHandler`

`PropertyChangedHandler` does not have a direct replacement.

To raise a property changed event via the `PropertyChanged` event handler, you need to call the `OnPropertyChanged` method instead.

C#

```
// MvvmLight
PropertyChangedEventHandler handler = PropertyChangedHandler;

// MVVM Toolkit
OnPropertyChanged();
```

Migrating ViewModelBase

The following steps focus on migrating your existing components which take advantage of the `ViewModelBase` of the MvvmLight Toolkit.

The MVVM Toolkit provides an ObservableRecipient type that provides similar functionality.

Below are a list of migrations that will need to be performed if being used in your current solution.

ViewModelBase methods

`Set<T>(string, ref T, T, bool)`

`Set<T>(string, ref T, T, bool)` does not have a like-for-like method signature replacement.

However, `SetProperty<T>(ref T, T, bool, string)` provides the same functionality with re-ordered parameters.

C#

```
// MvvmLight
Set(nameof(MyProperty), ref this.myProperty, value, true);

// MVVM Toolkit
SetProperty(ref this.myProperty, value, true);
```

Note, the value and broadcast boolean parameters are not optional in the MVVM Toolkit's implementation and must be provided to use this method. The reason for this change is that by omitting the broadcast parameter when calling this method, it will by default call the ObservableObject's `SetProperty` method.

Also, the `string` parameter is not required if the method is being called from the property's setter as it is inferred from the caller member name, just like with the methods in the base `ObservableObject` class.

`Set<T>(ref T, T, bool, string)`

`Set<T>(ref T, T, bool, string)` has a renamed direct replacement, `SetProperty<T>(ref T, T, bool, string)`.

C#

```
// MvvmLight
Set(ref this.myProperty, value, true, nameof(MyProperty));

// MVVM Toolkit
SetProperty(ref this.myProperty, value, true);
```

`Set<T>(Expression, ref T, T, bool)`

`Set<T>(Expression, ref T, T, bool)` does not have a direct replacement.

It is recommended for improved performance that you replace this with the MVVM Toolkit's `SetProperty<T>(ref T, T, bool, string)` using the `nameof` keyword instead.

C#

```
// MvvmLight
Set<MyObject>(() => MyProperty, ref this.myProperty, value, true);

// MVVM Toolkit
SetProperty(ref this.myProperty, value, true);
```

`Broadcast<T>(T, T, string)`

`Broadcast<T>(T, T, string)` has a direct replacement which doesn't require a rename.

C#

```
// MvvmLight
Broadcast<MyObject>(oldValue, newValue, nameof(MyProperty));

// MVVM Toolkit
Broadcast(oldValue, newValue, nameof(MyProperty));
```

Note, the message sent via the `Messenger` property when calling the `Broadcast` method has a direct replacement for `PropertyChangedMessage` within the MVVM Toolkit library.

`RaisePropertyChanged<T>(string, T, T, bool)`

There is no direct replacement for the `RaisePropertyChanged<T>(string, T, T, bool)` method.

The simplest alternative is to call `OnPropertyChanged` and subsequently call `Broadcast` to achieve this functionality.

C#

```
// MvvmLight
RaisePropertyChanged<MyObject>(nameof(MyProperty), oldValue, newValue,
true);

// MVVM Toolkit
OnPropertyChanged();
Broadcast(oldValue, newValue, nameof(MyProperty));
```

`RaisePropertyChanged<T>(Expression, T, T, bool)`

There is no direct replacement for the `RaisePropertyChanged<T>(Expression, T, T, bool)` method.

The simplest alternative is to call `OnPropertyChanged` and subsequently call `Broadcast` to achieve this functionality.

C#

```
// MvvmLight
RaisePropertyChanged<MyObject>(() => MyProperty, oldValue, newValue, true);

// MVVM Toolkit
OnPropertyChanged(nameof(MyProperty));
Broadcast(oldValue, newValue, nameof(MyProperty));
```

`ICleanup.Cleanup()`

There is no direct replacement for the `ICleanup` interface.

However, the `ObservableRecipient` provides an `OnDeactivated` method which should be used to provide the same functionality as `Cleanup`.

`OnDeactivated` in the MVVM Toolkit will also unregister all of the registered messenger events when called.

C#

```
// MvvmLight
Cleanup();

// MVVM Toolkit
OnDeactivated();
```

Note, the `OnActivated` and `OnDeactivated` methods can be called from your existing solution as with `Cleanup`.

However, the `ObservableRecipient` exposes an `IsActive` property that also controls the call to these methods when it is set.

ViewModelBase properties

`MessengerInstance`

`MessengerInstance` has a renamed direct replacement, `Messenger`.

C#

```
// MvvmLight
IMessenger messenger = MessengerInstance;

// MVVM Toolkit
IMessenger messenger = Messenger;
```

> ⓘ Note
>
> The default value of the `Messenger` property will be the `WeakReferenceMessenger.Default` instance, which is the standard weak reference

messenger implementation in the MVVM Toolkit. This can be customized by just injecting a different `IMessenger` instance into the `ObservableRecipient` constructor.

`IsInDesignMode`

There is no direct replacement for the `IsInDesignMode` property and any code using this should be altered or removed.

The reason for the omission from the MVVM Toolkit is that the `IsInDesignMode` property exposed platform-specific implementations. The MVVM Toolkit has been designed to be platform agnostic.

C#

```
// MvvmLight
var isInDesignMode = IsInDesignMode;

// MVVM Toolkit
// No direct replacement, remove
```

ViewModelBase static properties

`IsInDesignModeStatic`

There is no direct replacement for the `IsInDesignModeStatic` property and any code using this should be altered or removed.

The reason for the omission from the MVVM Toolkit is that the `IsInDesignMode` property exposed platform-specific implementations. The MVVM Toolkit has been designed to be platform agnostic.

C#

```
// MvvmLight
var isInDesignMode = ViewModelBase.IsInDesignModeStatic;

// MVVM Toolkit
// No direct replacement, remove
```

Migrating RelayCommand

The following steps focus on migrating your existing components which take advantage of the `RelayCommand` of the MvvmLight Toolkit.

The MVVM Toolkit provides a RelayCommand type that provides like-for-like functionality taking advantage of the `ICommand` System interface.

Below are a list of migrations that will need to be performed if being used in your current solution. Where a method or property isn't listed, there is a direct replacement with the same name in the MVVM Toolkit and there is no change required.

The first change here will be swapping using directives in your components.

C#

```
// MvvmLight
using GalaSoft.MvvmLight.Command;
using Galasoft.MvvmLight.CommandWpf;

// MVVM Toolkit
using CommunityToolkit.Mvvm.Input;
```

> ⓘ Note
>
> MvvmLight uses weak references to establish the link between the command and the action called from the associated class. This is not required by the MVVM Toolkit implementation and if this optional parameter has been set to `true` in any of your constructors, this will be removed.

Using RelayCommand with asynchronous actions

If you are currently using the MvvmLight `RelayCommand` implementation with asynchronous actions, the MVVM Toolkit exposes an improved implementation for these scenarios.

You can simply replace your existing `RelayCommand` with the `AsyncRelayCommand` which has been built for asynchronous purposes.

C#

```
// MvvmLight
var command = new RelayCommand(() => OnCommandAsync());
var command = new RelayCommand(async () => await OnCommandAsync());
```

```
// MVVM Toolkit
var asyncCommand = new AsyncRelayCommand(OnCommandAsync);
```

RelayCommand methods

`RaiseCanExecuteChanged()`

The functionality of `RaiseCanExecuteChanged()` can be achieved with the MVVM Toolkit's `NotifyCanExecuteChanged()` method.

C#

```
// MvvmLight
var command = new RelayCommand(OnCommand);
command.RaiseCanExecuteChanged();

// MVVM Toolkit
var command = new RelayCommand(OnCommand);
command.NotifyCanExecuteChanged();
```

Migrating `RelayCommand<T>`

The following steps focus on migrating your existing components which take advantage of the `RelayCommand<T>` of the MvvmLight Toolkit.

The MVVM Toolkit provides a RelayCommand<T> type that provides like-for-like functionality taking advantage of the `ICommand` System interface.

Below are a list of migrations that will need to be performed if being used in your current solution. Where a method or property isn't listed, there is a direct replacement with the same name in the MVVM Toolkit and there is no change required.

The first change here will be swapping using directives in your components.

C#

```
// MvvmLight
using GalaSoft.MvvmLight.Command;
using Galasoft.MvvmLight.CommandWpf;

// MVVM Toolkit
using CommunityToolkit.Mvvm.Input;
```

Using `RelayCommand` with asynchronous actions

If you are currently using the MvvmLight `RelayCommand<T>` implementation with asynchronous actions, the MVVM Toolkit exposes an improved implementation for these scenarios.

You can simply replace your existing `RelayCommand<T>` with the `AsyncRelayCommand<T>` which has been built for asynchronous purposes.

C#

```
// MvvmLight
var command = new RelayCommand<string>(async () => await OnCommandAsync());

// MVVM Toolkit
var asyncCommand = new AsyncRelayCommand<string>(OnCommandAsync);
```

`RelayCommand<T>` Methods

`RaiseCanExecuteChanged()`

The functionality of `RaiseCanExecuteChanged()` can be achieved with the MVVM Toolkit's `NotifyCanExecuteChanged()` method.

C#

```
// MvvmLight
var command = new RelayCommand<string>(OnCommand);
command.RaiseCanExecuteChanged();

// MVVM Toolkit
var command = new RelayCommand<string>(OnCommand);
command.NotifyCanExecuteChanged();
```

Migrating `SimpleIoc`

The IoC implementation in the MVVM Toolkit doesn't include any built-in logic to handle dependency injection on its own, so you're free to use any 3rd party library to retrieve an `IServiceProvider` instance that you can then pass to the `Ioc.ConfigureServices` method. In the examples below, the `ServiceCollection` type from the `Microsoft.Extensions.DependencyInjection` library will be used.

This is the biggest change between MvvmLight and the MVVM Toolkit.

This implementation will feel familiar if you've implemented dependency injection with ASP.NET Core applications.

Registering your dependencies

With MvvmLight, you may have registered your dependencies similar to these scenarios using `SimpleIoc`.

C#

```
public void RegisterServices()
{
  SimpleIoc.Default.Register<INavigationService, NavigationService>();

  SimpleIoc.Default.Register<IDialogService>(() => new DialogService());
}
```

With the MVVM Toolkit, you would achieve the same as follows.

C#

```
public void RegisterServices()
{
  Ioc.Default.ConfigureServices(
    new ServiceCollection()
    .AddSingleton<INavigationService, NavigationService>()
    .AddSingleton<IDialogService>(new DialogService())
    .BuildServiceProvider());
}
```

Resolving dependencies

Once initialized, services can be retrieved from the `Ioc` class just like with `SimpleIoc`:

C#

```
IDialogService dialogService = SimpleIoc.Default.GetInstance<IDialogService>
();
```

Migrating to the MVVM Toolkit, you will achieve the same with:

C#

```
IDialogService dialogService = Ioc.Default.GetService<IDialogService>();
```

Removing dependencies

With `SimpleIoc`, you would unregister your dependencies with the following method call.

```
C#

SimpleIoc.Default.Unregister<INavigationService>();
```

There is no direct replacement for removing dependencies with the MVVM Toolkit `Ioc` implementation.

Preferred constructor

When registering your dependencies with MvvmLight's `SimpleIoc`, you have the option in your classes to provide a `PreferredConstructor` attribute for those with multiple constructors.

This attribute will need removing where used, and you will need to use any attributes from the 3rd party dependency injection library in use, if supported.

Migrating `Messenger`

The following steps focus on migrating your existing components which take advantage of the `Messenger` of the MvvmLight Toolkit.

The MVVM Toolkit provides two messenger implementations (`WeakReferenceMessenger` and `StrongReferenceMessenger`, see docs here) that provides similar functionality, with some key differences detailed below.

Below are a list of migrations that will need to be performed if being used in your current solution.

The first change here will be swapping using directives in your components.

```
C#

// MvvmLight
using GalaSoft.MvvmLight.Messaging;

// MVVM Toolkit
using CommunityToolkit.Mvvm.Messaging;
```

Messenger methods

`Register<TMessage>(object, Action<TMessage>)`

The functionality of `Register<TMessage>(object, Action<TMessage>)` can be achieved with the MVVM Toolkit's `IMessenger` extension method `Register<TRecipient, TMessage>(object, MessageHandler<TRecipient, TMessage>)`.

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage>(this, static (r, m) =>
r.OnMyMessageReceived(m));
```

The reason for this signature is that it allows the messenger to use weak references to properly track recipients and to avoid creating closures to capture the recipient itself. That is, the input recipient is passed as an input to the lambda expression, so it doesn't need to be captured by the lambda expression itself. This also results in more efficient code, as the same handler can be reused multiple times with no allocations. Note that this is just one of the supported ways to register handlers, and it is possible to also use the `IRecipient<TMessage>` interface instead (detailed in the messenger docs), which makes the registration automatic and less verbose.

> ⓘ **Note**
>
> The `static` modifier for lambda expressions requires C# 9, and it is optional. It is useful to use it here to ensure you're not accidentally capturing the recipient or some other member, hence causing the allocation of a closure, but it is not mandatory. If you can't use C# 9, you can just remove `static` here and just be careful to ensure the code is not capturing anything.

Additionally, this example and the ones below will just be using the `Messenger` property from `ObservableRecipient`. If you want to just statically access a messenger instance from anywhere else in your code, the same examples apply as well, with the only difference being that `Messenger` needs to be replaced with eg. `WeakReferenceMessenger.Default` instead.

`Register<TMessage>(object, bool, Action<TMessage>)`

There is no direct replacement for this registration mechanism which allows you to support receiving messages for derived message types also. This change is intentional as the `Messenger` implementation aims to not use reflection to achieve its performance benefits.

Alternatively, there are a few options that can be done to achieve this functionality.

- Create a custom `IMessenger` implementation.
- Register the additional message types using a shared handler than then checks the type and invokes the right method.

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, true, this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage>(this, static (r, m) =>
r.OnMyMessageReceived(m));
Messenger.Register<MyViewModel, MyOtherMessage>(this, static (r, m) =>
r.OnMyMessageReceived(m));
```

`Register<TMessage>(object, object, Action<TMessage>)`

The functionality of `Register<TMessage>(object, object, Action<TMessage>)` can be achieved with the MVVM Toolkit's `Register<TRecipient, TMessage, TToken>(object, TToken, MessageHandler<TRecipient, TMessage>)` method.

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, nameof(MyViewModel),
this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage, string>(this,
nameof(MyViewModel), static (r, m) => r.OnMyMessageReceived(m));
```

`Register<TMessage>(object, object, bool, Action<TMessage>)`

There is no direct replacement for this registration mechanism which allows you to support receiving messages for derived message types also. This change is intentional as

the `Messenger` implementation aims to not use reflection to achieve its performance benefits.

Alternatively, there are a few options that can be done to achieve this functionality.

- Create a custom `IMessenger` implementation.
- Register the additional message types using a shared handler than then checks the type and invokes the right method.

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, nameof(MyViewModel), true,
this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage, string>(this,
nameof(MyViewModel), static (r, m) => r.OnMyMessageReceived(m));
Messenger.Register<MyViewModel, MyOtherMessage, string>(this,
nameof(MyViewModel), static (r, m) => r.OnMyMessageReceived(m));
```

`Send<TMessage>(TMessage)`

The functionality of `Send<TMessage>(TMessage)` can be achieved with the MVVM Toolkit's `IMessenger` extension method `Send<TMessage>(TMessage)`.

C#

```
// MvvmLight
Messenger.Default.Send<MyMessage>(new MyMessage());
Messenger.Default.Send(new MyMessage());

// MVVM Toolkit
Messenger.Send(new MyMessage());
```

In the above scenario where the message being sent has a parameterless constructor, the MVVM Toolkit has a simplified extension to send a message in this format.

C#

```
// MVVM Toolkit
Messenger.Send<MyMessage>();
```

`Send<TMessage>(TMessage, object)`

The functionality of `Send<TMessage>(TMessage, object)` can be achieved with the MVVM Toolkit's `Send<TMessage, TToken>(TMessage, TToken)` method.

C#

```
// MvvmLight
Messenger.Default.Send<MyMessage>(new MyMessage(), nameof(MyViewModel));
Messenger.Default.Send(new MyMessage(), nameof(MyViewModel));

// MVVM Toolkit
Messenger.Send(new MyMessage(), nameof(MyViewModel));
```

`Unregister(object)`

The functionality of `Unregister(object)` can be achieved with the MVVM Toolkit's `UnregisterAll(object)` method.

C#

```
// MvvmLight
Messenger.Default.Unregister(this);

// MVVM Toolkit
Messenger.UnregisterAll(this);
```

`Unregister<TMessage>(object)`

The functionality of `Unregister<TMessage>(object)` can be achieved with the MVVM Toolkit's `IMessenger` extension method `Unregister<TMessage>(object)`.

C#

```
// MvvmLight
Messenger.Default.Unregister<MyMessage>(this);

// MVVM Toolkit
Messenger.Unregister<MyMessage>(this);
```

`Unregister<TMessage>(object, Action<TMessage>)`

There is no direct replacement for the `Unregister<TMessage>(object, Action<TMessage>)` method in the MVVM Toolkit.

The reason for the omission is that a message recipient can only have a single registered handler for any given message type.

We recommend achieving this functionality with the MVVM Toolkit's `IMessenger` extension method `Unregister<TMessage>(object)`.

```
C#

// MvvmLight
Messenger.Default.Unregister<MyMessage>(this, OnMyMessageReceived);

// MVVM Toolkit
Messenger.Unregister<MyMessage>(this);
```

`Unregister<TMessage>(object, object)`

The functionality of `Unregister<TMessage>(object, object)` can be achieved with the MVVM Toolkit's `Unregister<TMessage, TToken>(object, TToken)` method.

```
C#

// MvvmLight
Messenger.Default.Unregister<MyMessage>(this, nameof(MyViewModel));

// MVVM Toolkit
Messenger.Unregister<MyMessage, string>(this, nameof(MyViewModel));
```

`Unregister<TMessage>(object, object, Action<TMessage>)`

There is no direct replacement for the `Unregister<TMessage>(object, object, Action<TMessage>)` method in the MVVM Toolkit.

The reason for the omission is that a message recipient can only have a single registered handler for any given message type.

We recommend achieving this functionality with the MVVM Toolkit's `Unregister<TMessage, TToken>(object, TToken)` method.

```
C#

// MvvmLight
Messenger.Default.Unregister<MyMessage>(this, nameof(MyViewModel),
OnMyMessageReceived);
```

```
// MVVM Toolkit
Messenger.Unregister<MyMessage, string>(this, nameof(MyViewModel));
```

`Cleanup()`

The `Cleanup` method has a direct replacement with the same name in the MVVM Toolkit. Note that this method is only useful when a messenger using weak references is being used, while the `StrongReferenceMessenger` type will simply do nothing when this method is called, as the internal state is already trimmed automatically as the messenger is being used.

C#

```
// MvvmLight
Messenger.Default.Cleanup();

// MVVM Toolkit
Messenger.Cleanup();
```

`RequestCleanup()`

There is no direct replacement for the `RequestCleanup` method in the MVVM Toolkit. In the context of MvvmLight, `RequestCleanup` is used to initiate a request to remove registrations which are no longer alive as the implementation takes advantage of weak references.

Any calls to the `RequestCleanup` method can be removed or replaced with `Cleanup`.

C#

```
// MvvmLight
Messenger.Default.RequestCleanup();

// MVVM Toolkit
// No direct replacement, remove
```

`ResetAll()`

The functionality of `ResetAll()` can be achieved with the MVVM Toolkit's `Reset()` method.

Unlike MvvmLight's implementation which nulls out the instance, the MVVM Toolkit clears the registered maps.

C#

```
// MvvmLight
Messenger.Default.ResetAll();

// MVVM Toolkit
Messenger.Reset();
```

Messenger static methods

`OverrideDefault(IMessenger)`

There is no direct replacement for the `OverrideDefault(IMessenger)` method in the MVVM Toolkit.

To use a custom implementation of the `IMessenger`, either registered the custom implementation in the service registrations for dependency injection or manually construct a static instance and pass this where required.

C#

```
// MvvmLight
Messenger.OverrideDefault(new Messenger());

// MVVM Toolkit
// No direct replacement
```

`Reset()`

There is no direct replacement for the static `Reset` method in the MVVM Toolkit.

The same functionality can be achieved by calling the `Reset` method of the static `Default` instance of one of the messenger types.

C#

```
// MvvmLight
Messenger.Reset();

// MVVM Toolkit
WeakReferenceMessenger.Default.Reset();
```

Messenger static properties

`Default`

`Default` has a direct replacement, `Default`, requiring no change to your existing implementation.

C#

```
// MvvmLight
IMessenger messenger = Messenger.Default;

// MVVM Toolkit
IMessenger messenger = WeakReferenceMessenger.Default;
```

Migrating message types

The message types provided in the MvvmLight toolkit are designed as a base for you as a developer to work with if needed.

While the MVVM Toolkit provides some alternatives, there are no direct replacement for these message types. We recommend looking at our available message types.

Alternatively, if your solution takes advantage of the MvvmLight message types, these can easily be ported into your own codebase.

Migrating platform-specific components

In the current MVVM Toolkit implementation, there are no replacements for platform-specific components which exist in the MvvmLight toolkit.

The following components and their associated helpers/extension methods do not have a replacement and will need considering when migrating to the MVVM Toolkit.

Android/iOS/Windows specific

- `DialogService`
- `DispatcherHelper`
- `NavigationService`

Android/iOS specific

- `ActivityBase`
- `Binding`
- `BindingMode`
- `PropertyChangedEventManager`
- `UpdateTriggerMode`

Android specific

- `CachingViewHolder`
- `ObservableAdapter`
- `ObservableRecyclerAdapter`

iOS specific

- `ObservableCollectionViewSource`
- `ObservableTableViewController`
- `ObservableTableViewSource`

Helpers

- `Empty`
- `WeakAction`
- `WeakFunc`

Migrating from MVVM Basic

Article • 01/18/2023

This article explains how to migrate apps built with the MVVM Basic option in Windows Template Studio to use the MVVM Toolkit library instead. It applies to both UWP and WPF apps created with Windows Template Studio.

> **Platform APIs:** ObservableObject, RelayCommand

This article focuses exclusively on migration and does not cover how to use the additional functionality that the library provides.

Installing the MVVM Toolkit

To use the MVVM Toolkit, you must install the NuGet package into your existing application.

Install via .NET CLI

```
dotnet add package CommunityToolkit.Mvvm --version 8.1.0
```

Install via PackageReference

XML

```
<PackageReference Include="CommunityToolkit.Mvvm" Version="8.1.0" />
```

Updating a project

There are four steps to migrate the code generated by Windows Template Studio.

1. Delete old files.
2. Replace use of `Observable`.
3. Add new namespace references.
4. Update methods with different names.

1. Delete old files

MVVM Basic is comprised of two files

```
C#

\Helpers\Observable.cs
\Helpers\RelayCommand.cs
```

Delete both of these files.

If you try and build the project at this point you will see lots of errors. These can be useful for identifying files that require changes.

2. Replace use of `Observable`

The `Observable` class was used as a base class for ViewModels. The MVVM Toolkit contains a similar class with additional functionality that is called ObservableObject.

Change all classes that previously inherited from `Observable` to inherit from `ObservableObject`.

For example

```
C#

    public class MainViewModel : Observable
```

will become

```
C#

    public class MainViewModel : ObservableObject
```

3. Add new namespace references

Add a reference to the `CommunityToolkit.Mvvm.ComponentModel` namespace in all files where there is a reference to `ObservableObject`.

You can either add the appropriate directive manually, of move the cursor to the `ObservableObject` and press `Ctrl+.` to access the Quick Action menu to add this for you.

```
C#

using CommunityToolkit.Mvvm.ComponentModel;
```

Add a reference to the `CommunityToolkit.Mvvm.Input` namespace in all files where there is a reference to RelayCommand.

You can either add the appropriate directive manually, of move the cursor to the `RelayCommand` and press `Ctrl+.` to access the Quick Action menu to add this for you.

```
C#

using CommunityToolkit.Mvvm.Input;
```

4. Update methods with different names

There are two methods that must be updated to allow for different names for the same functionality.

All calls to `Observable.Set` must be replaced with calls to `ObservableObject.SetProperty`.

So,

```
C#

    set { Set(ref _elementTheme, value); }
```

will become

```
C#

    set { SetProperty(ref _elementTheme, value); }
```

All calls to `RelayCommand.OnCanExecuteChanged` must be replaced with calls to `RelayCommand.NotifyCanExecuteChanged`.

So,

```
C#

    (UndoCommand as RelayCommand)?.OnCanExecuteChanged();
```

will become

C#

```
    (UndoCommand as RelayCommand)?.NotifyCanExecuteChanged();
```

The app should now work with the same functionality as before.