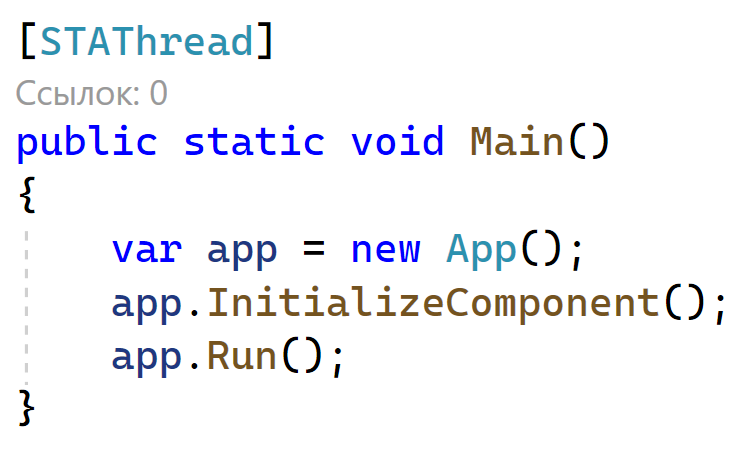
Подключение DI в проект с MVVM.

По умолчанию, когда мы создаем проект WPF, в нем нет ни поддержки конфигурации, ни DI, но мы можем подключить эти отдельные элементы.

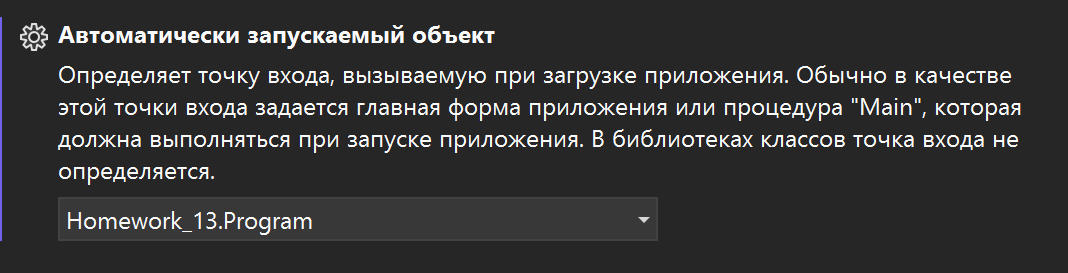
Для того, чтобы подключить данную функциональность в проект, необходимо добавить Nuget пакет Microsoft.Extensions.Hosting, он содержит внутри себя конфигурацию, DI и логирование в консоль. Данный проект по умолчанию входит в пустой проект ASP.NET Core. Данный пакет можно также использовать в .NET Framework версии после 4.7.

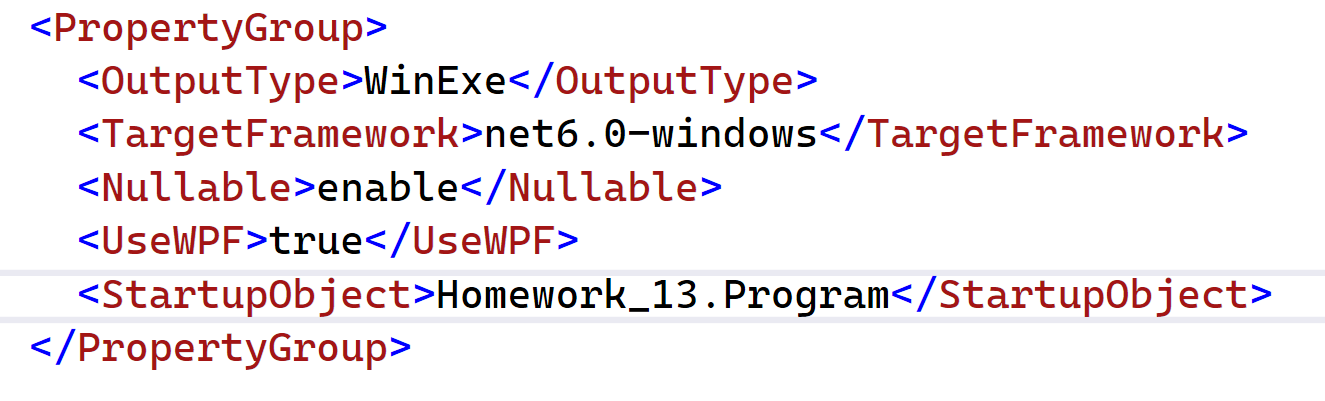
Нам понадобится создать свой Хост приложения, который находится в памяти приложения и реализует такие понятия как сервисы, но для того, чтобы создать этот хост надо создать свою точку входа. По умолчанию в WPF приложении точка входа (метод Main) спрятана в папке obj/Debug в файле App.g.cs. Для создания точки входа создаем новый класс Program в корне проекта и внутри объявляем метод Main:



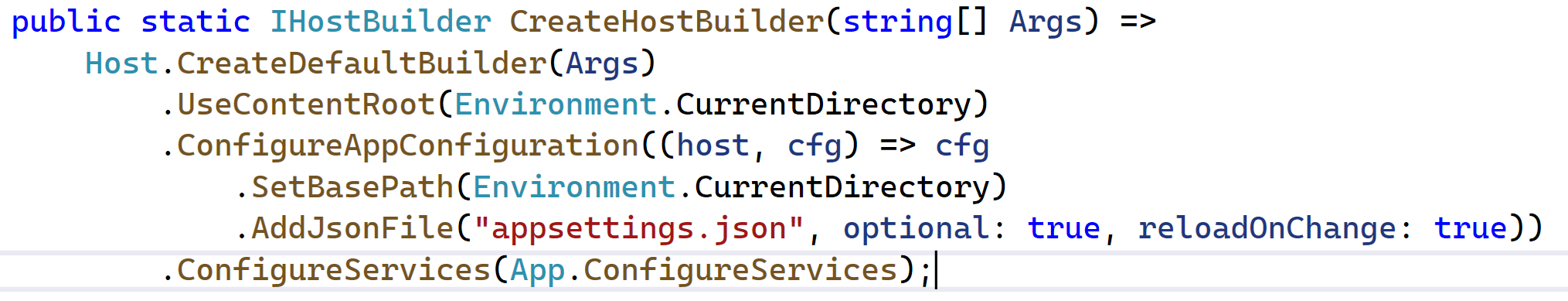
Атрибут [STAThread] при этом обязателен, иначе приложение не запустится.

При этом у нас будет две точки входа, что помешает запуску приложения. Для устранения данной неоднозначности, заходим в свойства приложения и во вкладке Автоматически запускаемый объект выставляем наш вновь созданный класс Program:



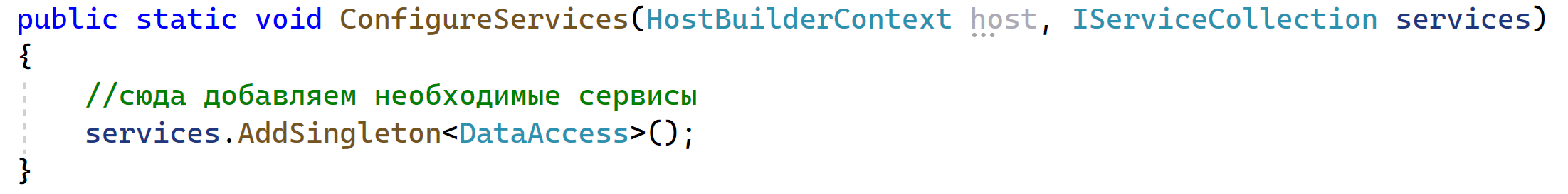
В результате в проекте появляется запись, которую мы, в принципе, могли добавить руками: 

Далее в классе Program нам необходимо создать метод:



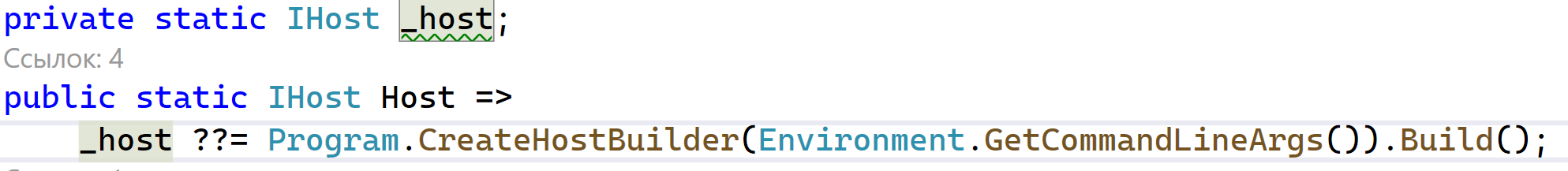
На наличие этого метода в классе Program нацелены инструменты .NET Core, такие как EF Core. appsettings.json – файл конфигурации.

Сами сервисы описываются в файле приложения App.xaml.cs:



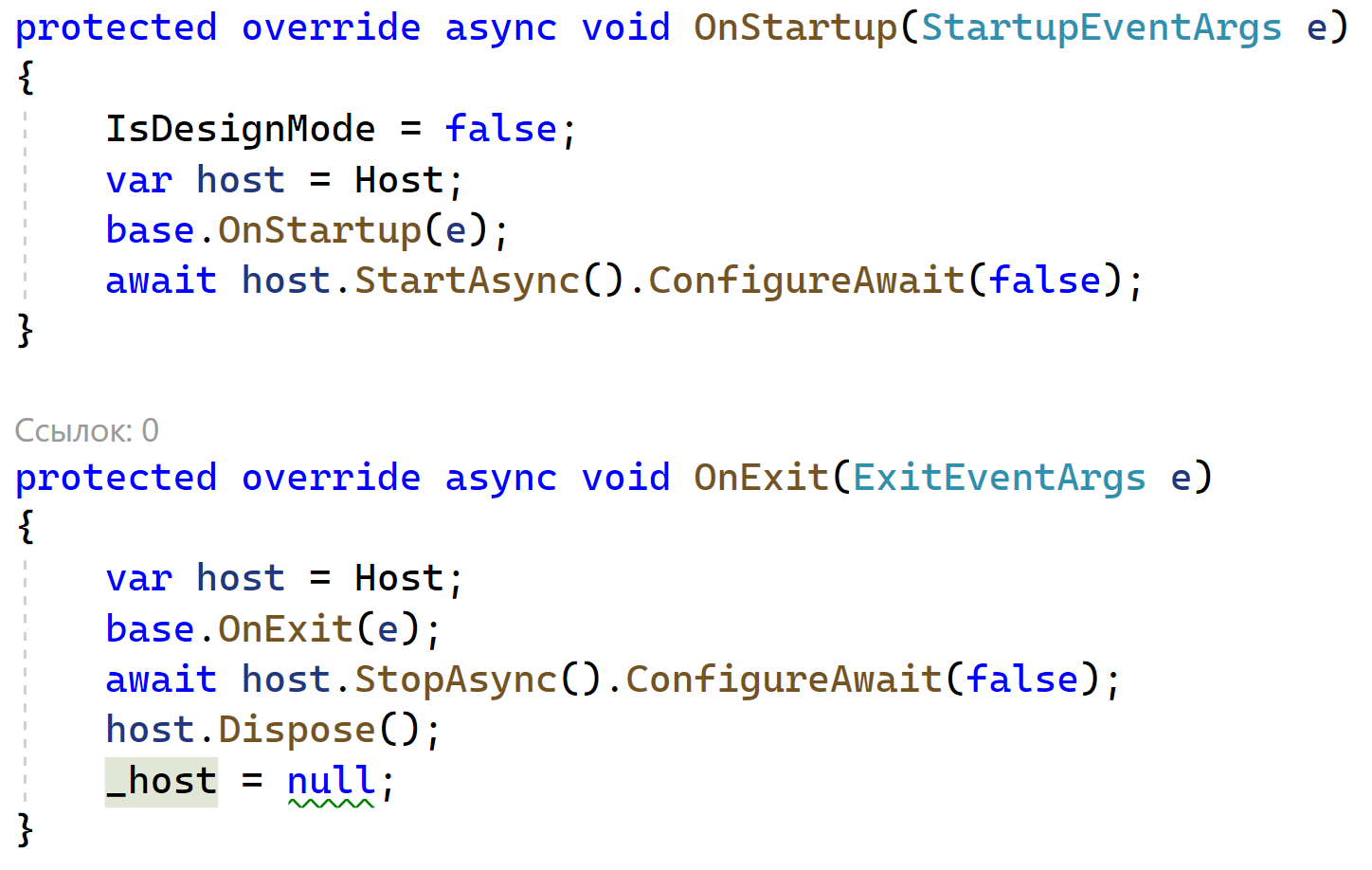
После того, как будут добавлены все необходимые сервисы, IServiceCollection будет скомпилирована в сервис менеджера, который позволит получать из него сервисы просто по их типам.

Далее внутри приложения App.xaml.cs необходимо создать хост:



Теперь имея класс APP мы можем через статическое свойство Host мы можем обратиться к этому хосту. При первом обращении он будет создан, будут сконфигурированы все его сервисы.

Для запуска и остановки хоста понадобится методы OnStartup и OnExit:



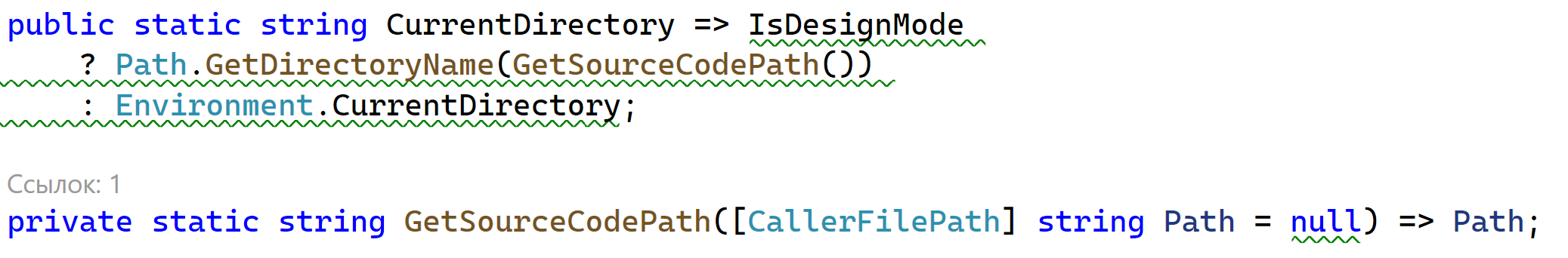
Host запускается ассинхронно, поэтому методы также делаем асинхронными. Не забываем добавить ConfigureAwait(false), иначе получим мертвую блокировку.

Теперь в любом месте программы мы можем обратиться к нашему хосту так: App.Host.Services.GetRequiredService<DataService>().GetData();.

Приложение WPF работает в двух областях. Первая и основная составляющая приложения, это когда мы его запускаем. Второе, когда приложение тоже запускается, но под капотом под отладчиком в дизайнере, когда мы открываем окно и видим дизайнер. Проблема в том, что каталогами приложения, запущенного нормальным образом и запущенного из-под дизайнера будут совершенно разные места на диске. Для решения этой проблемы добавим свойство:

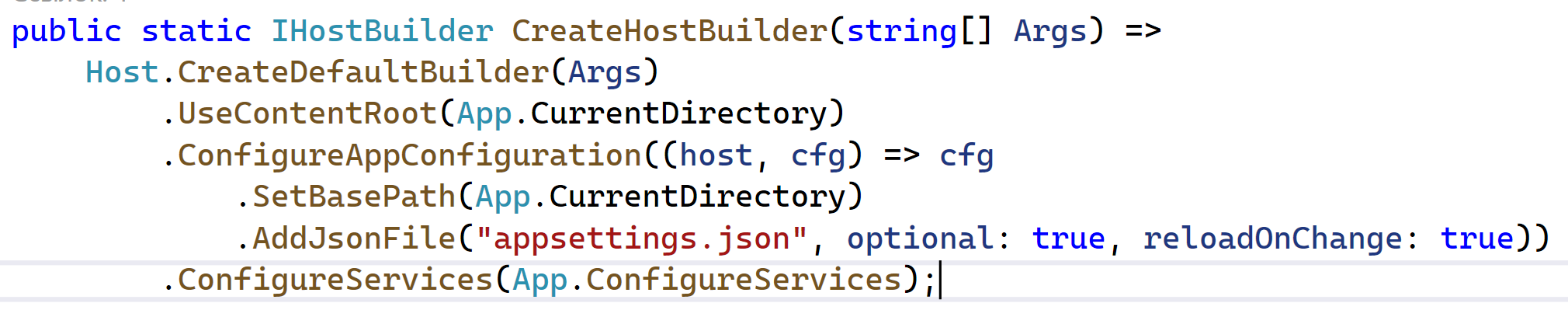
public static bool IsDesignMode { get; private set; } = true;

и метод:



Метод GetSourceCodePath позволяет вернуть путь к файлу App.xaml.cs с помощью атрибута компилятора [CallerFilePath]. Метод GetDiretoryName будет возвращать нам путь к каталогу, где находится наше приложение в режиме дизайнера и в режиме основной его работы.

После этого мы подменим Environment на App в методе CreateHostBuilder в классе Program:



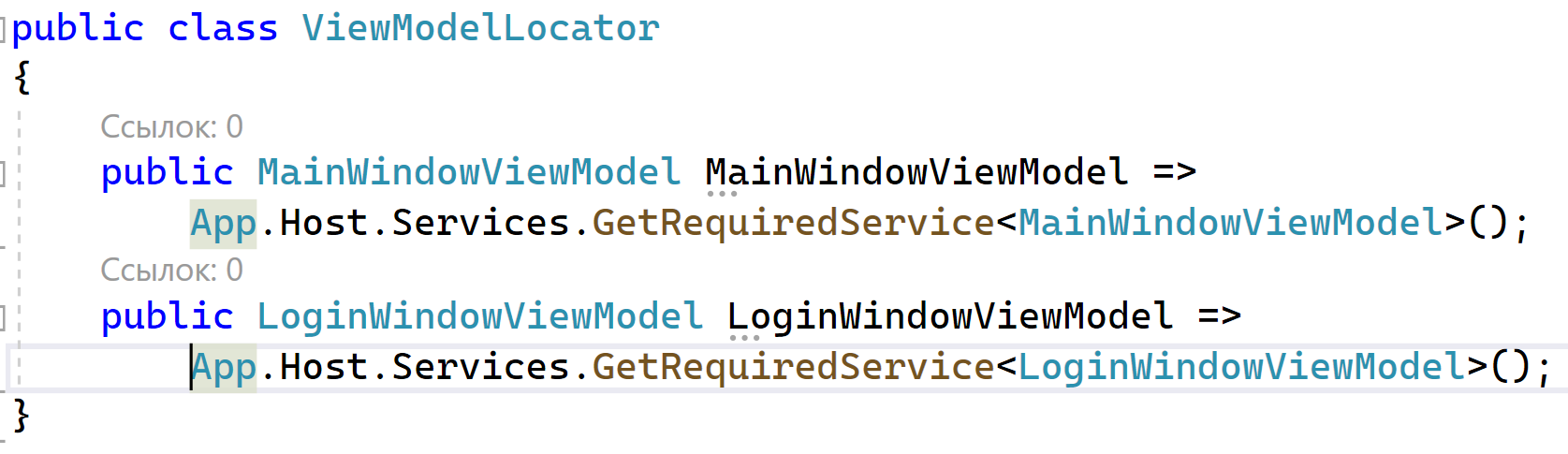
Добавляем пустой файл конфигурации appsettings.json в проект. Файл нужно проверить на предмет попадания в проект при компиляции. Для этого нужно зайти в свойства файла, где в строке «Копировать в выходной каталог» нужно отметить копировать более позднюю версию.

Вышеуказанным образом создается структура для запуска системы инверсии управления DI.

Далее будет описано использование данной структуры DI в приложении MVVM.

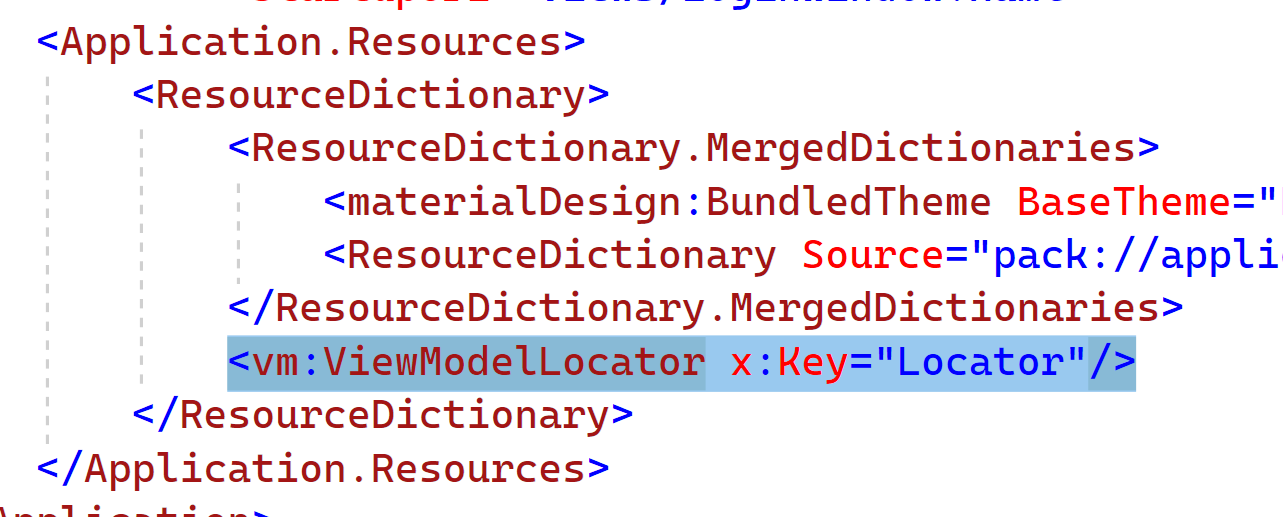
Начинаем с вью моделей.

Добавляем в папку ViewModels специальный класс, который назовем ViewModelLocator:



Этот класс представляет собой сборище свойств, через которые мы осуществляем доступ к конкретным вью моделям. Мы обращаемся с сервисам и просим выдать нам необходимую вью модель.

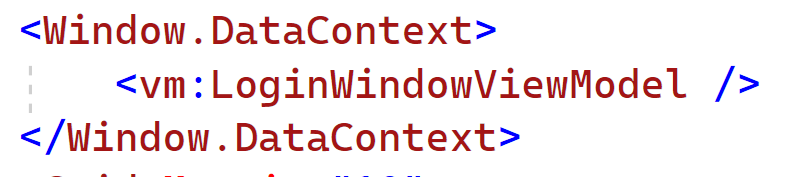
Далее размещаем объект этого класса в ресурсах на уровне всего приложения в файле App.xaml:



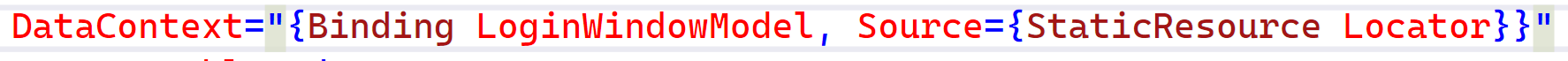
Теперь через ключ “Locator” он будет нам доступен во всем приложении.

После того, как подключен ViewModelLocator мы можем добывать вью модели в любом окне, вью модели сначала надо зарегистрировать в наших сервисах.

До использования DI контекстом окон устанавливаются наши вью модели через возможность использования вью моделей через расширение разметки:

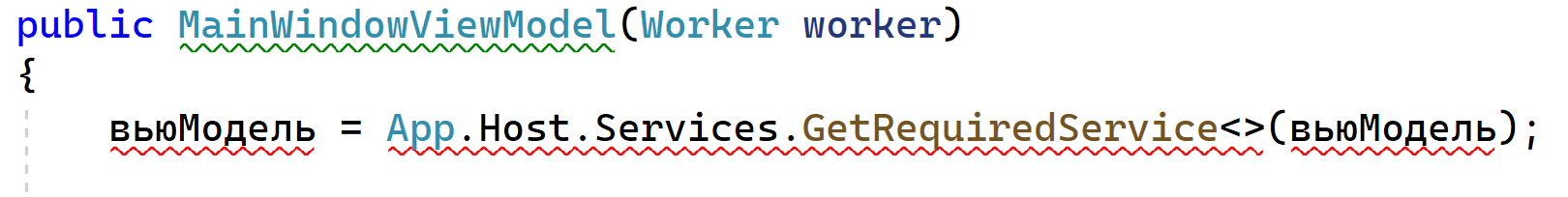


Для использования DI необходимо заменить данную запись на использование привязки к ресурсу, который берется из статических ресурсов приложения с названием Locator и из этого ресурса мы берем соответствующую WindowModel:



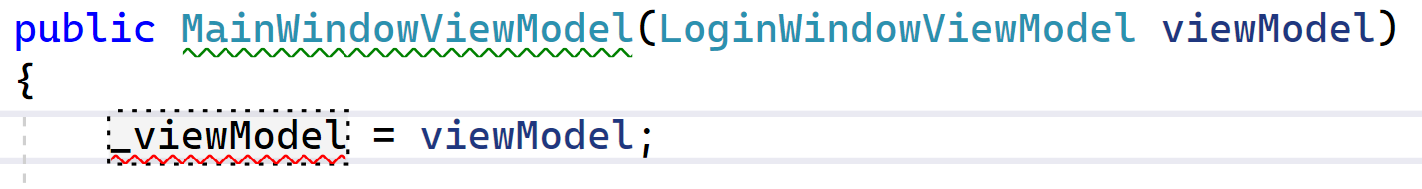
Таким образом теперь через ViewModelLocator мы осуществляем доступ к вью моделям, которые мы в нем определим.

Но данный шаблон называют антипаттерном, потому что он напрямую использует контейнер сервисов, и мы напрямую обращаемся к сервису, который хранится там. Получается, что наше окно имеет неявную зависимость и снаружи этого окна мы не можем контролировать, что будет вью моделью этого окна. Это аналогично тому, как если бы мы напрямую получали сервисы в конструкторе типа того:



Это бы также извлекло вью модель из контейнера, но при этом тем, кто будет использовать вью модель, не понятно, что происходит внутри. А внутри происходит неявная связь. Вот это и является антипаттерном.

По идее мы все зависимости должны вынести в параметры конструктора:



Но при таком подходе мы получаем циклическую зависимость, когда главную вью модель будет создавать контейнер сервисов, он посмотрит какие параметры она принимает, чтобы ее создать. Он увидит, что чтобы ее создать надо сперва создать объект класса, передаваемого в параметрах, он полезет за этим классом и попытается его создать. При попытке создать данный класс, он увидит, что в параметрах у него находится главная вью модель. Он будет пытаться воздать главную вью модель и так по кругу.

Для устранения циклической зависимости зависимых вью моделей необходимо ее разорвать. Ее можно разорвать удалив зависимость в параметрах конструктора в одной из вью моделей, в свойстве вью модели сеттер сделать с internal:

