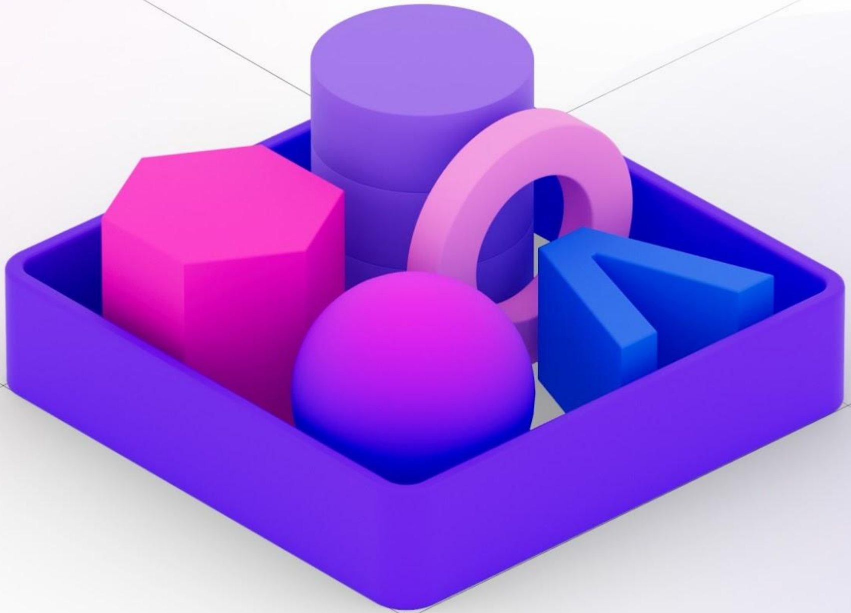
.NET Conf



WELCOME

From IL Weaving to Source Generators

the Realm story

Ferdinando Papale | (mostly) .NET Developer | @papafeit

The context: “Code generation” in .NET

IL Weaving

.NET code is compiled to IL (*Intermediate Language*) first, then converted to machine code at runtime (JIT) or ahead of time (AOT)

IL is similar to Java bytecode, “high-level assembly”

IL can be modified with *Weaving*

Possible to modify existing code in any way, “feels like magic”

Weaving happens **after** compilation

Useful to generate repetitive or optimised code

Source code

```
public class Person
{
    public string Name { get; set; }
}
```

IL

```
.method public hidebysig specialname
    instance string get_Name () cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldfld string Person::'<Name>k__BackingField'
    IL_0006: ret
} // end of method Person::get_Name

.method public hidebysig specialname
    instance void set_Name (string 'value') cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldarg.1
    IL_0002: stfld string Person::'<Name>k__BackingField'
    IL_0007: ret
} // end of method Person::set_Name
```

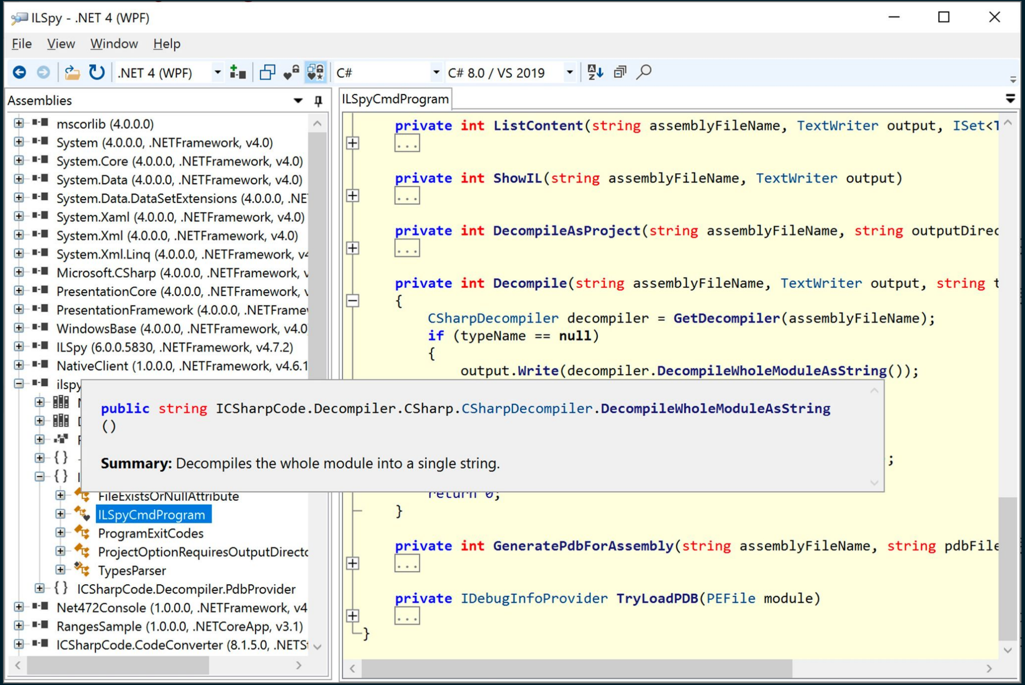
PropertyChanged.Fody

```csharp
public class Person: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public string Name { get; set; }
    public int Age { get; set; }
}
```

```csharp
public class Person: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _name;
    public string Name
    {
        get => _name;
        set
        {
            if (value != _name)
            {
                _name = value;
                OnPropertyChanged("Name");
            }
        }
    }

    private int _age;
    public int Age
    {
        get => _age;
        set
        {
            if (value != _age)
            {
                _age = value;
                OnPropertyChanged("Age");
            }
        }
    }

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this,
            new PropertyChangedEventArgs(propertyName));
    }
}
```

How to see IL code / weaved code?

- SharpLab (Online)
- ILSpy or other decompilers



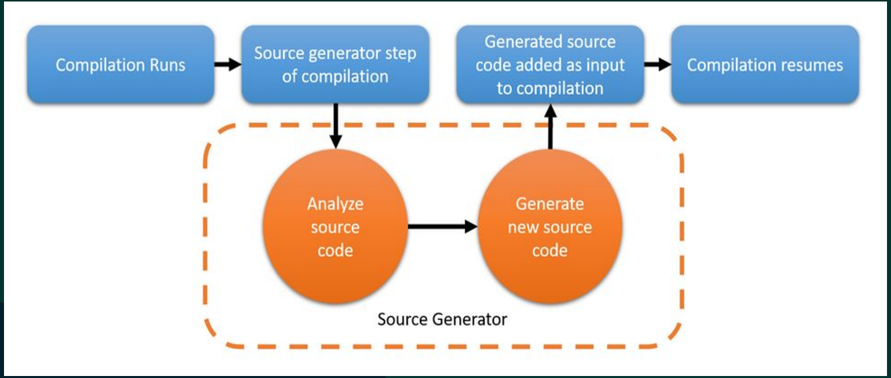
```
mscorlib (4.0.0.0)
System (4.0.0.0, .NETFramework, v4.0)
System.Core (4.0.0.0, .NETFramework, v4.0)
System.Data (4.0.0.0, .NETFramework, v4.0)
System.Data.DataSetExtensions (4.0.0.0, .NE
System.Xaml (4.0.0.0, .NETFramework, v4.0)
System.Xml (4.0.0.0, .NETFramework, v4.0)
System.Xml.Linq (4.0.0.0, .NETFramework, v4
Microsoft.CSharp (4.0.0.0, .NETFramework, v
PresentationCore (4.0.0.0, .NETFramework, v
PresentationFramework (4.0.0.0, .NETFrame
WindowsBase (4.0.0.0, .NETFramework, v4.0
ILSpy (6.0.0.5830, .NETFramework, v4.7.2)
NativeClient (1.0.0.0, .NETFramework, v4.6.1
ilspy
    FileExistsOrNullAttribute
    ILSpyCmdProgram
    ProgramExitCodes
    ProjectOptionRequiresOutputDirect
    TypesParser
  ICSharpCode.Decompiler.PdbProvider
Net472Console (1.0.0.0, .NETFramework, v4
RangesSample (1.0.0.0, .NETCoreApp, v3.1)
ICSharpCode.CodeConverter (8.1.5.0, .NETS
```

```
    private int ListContent(string assemblyFileName, TextWriter output, ISet<T
    ...

    private int ShowIL(string assemblyFileName, TextWriter output)
    ...

    private int DecompileAsProject(string assemblyFileName, string outputDire
    ...

    private int Decompile(string assemblyFileName, TextWriter output, string t
    {
        CSharpDecompiler decompiler = GetDecompiler(assemblyFileName);
        if (typeName == null)
        {
            output.Write(decompiler.DecompileWholeModuleAsString());
                                                                        ;
        return 0;
    }

    private int GeneratePdbForAssembly(string assemblyFileName, string pdbFile
    ...

    private IDebugInfoProvider TryLoadPDB(PEFile module)
    ...
}
```

```
public string ICSharpCode.Decompiler.CSharp.CSharpDecompiler.DecompileWholeModuleAsString
()
```

Summary: Decompiles the whole module into a single string.

Source Generators



Compiler feature introduced with .NET 5

“Plugs” into the compilation pipeline

Source Generators are passed a *compilation object* that can be analyzed

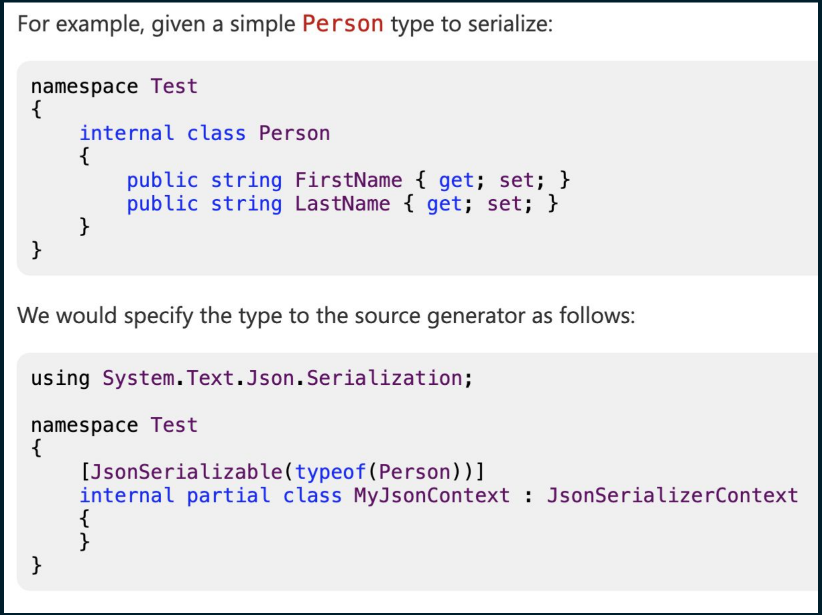
Source Generators emit source code

Only additive

Source generation happen **during** compilation

Useful to generate repetitive or optimised code

System.Text.Json

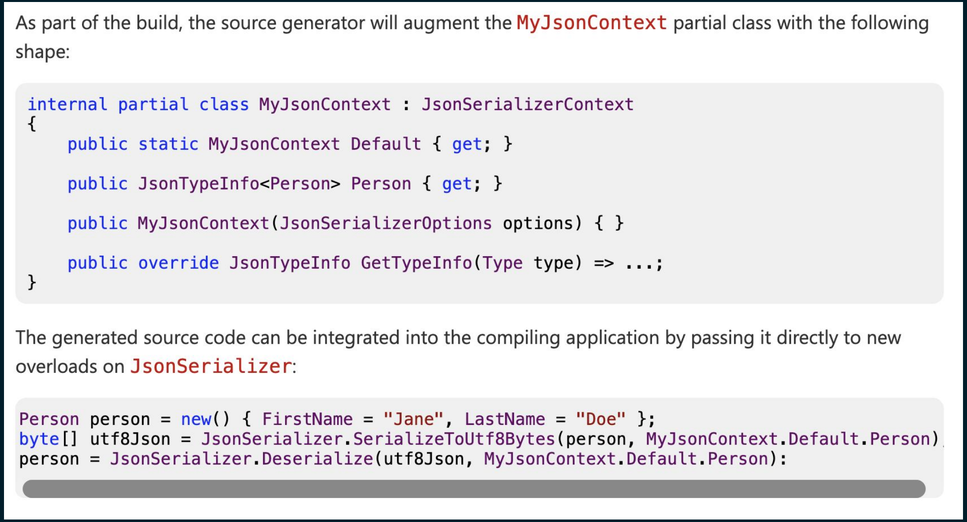


For example, given a simple `Person` type to serialize:

```
namespace Test
{
    internal class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

We would specify the type to the source generator as follows:

```
using System.Text.Json.Serialization;

namespace Test
{
    [JsonSerializable(typeof(Person))]
    internal partial class MyJsonContext : JsonSerializerContext
    {
    }
}
```



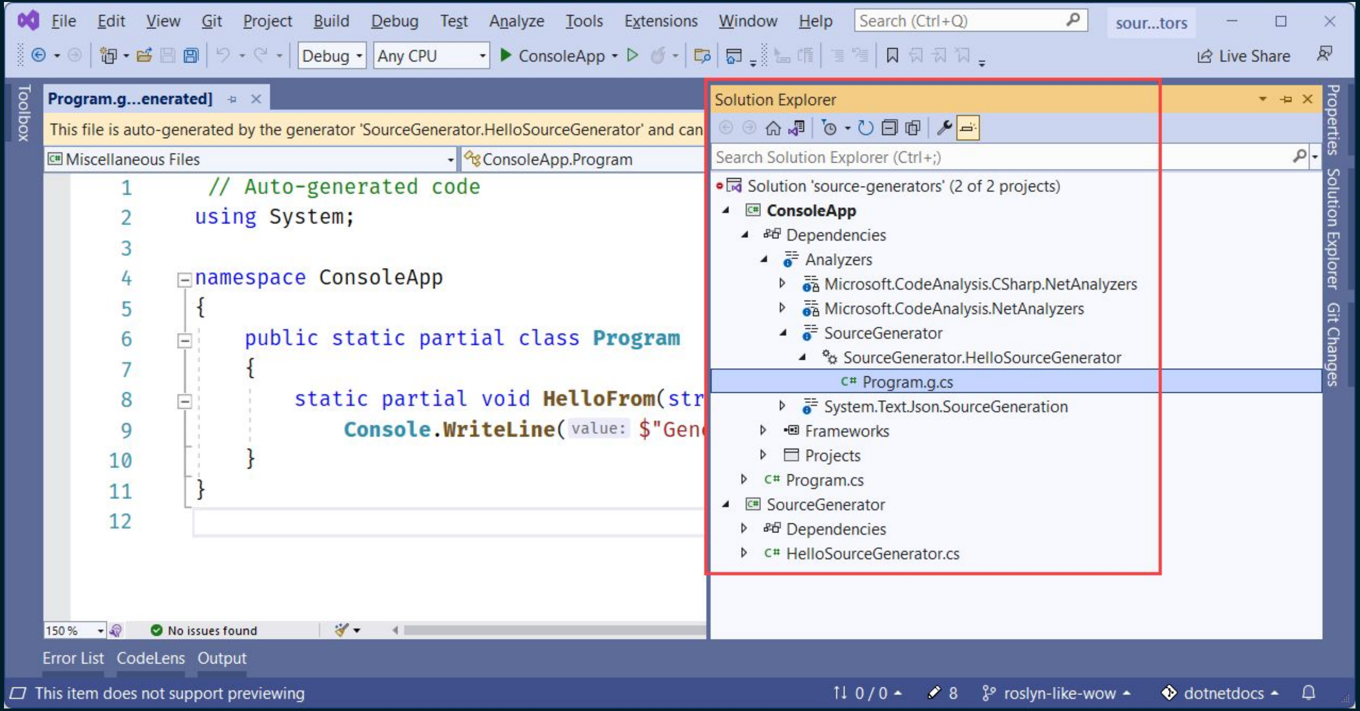
As part of the build, the source generator will augment the `MyJsonContext` partial class with the following shape:

```
internal partial class MyJsonContext : JsonSerializerContext
{
    public static MyJsonContext Default { get; }

    public JsonTypeInfo<Person> Person { get; }

    public MyJsonContext(JsonSerializerOptions options) { }

    public override JsonTypeInfo GetTypeInfo(Type type) => ...;
}
```

The generated source code can be integrated into the compiling application by passing it directly to new overloads on `JsonSerializer`:

```
Person person = new() { FirstName = "Jane", LastName = "Doe" };
byte[] utf8Json = JsonSerializer.SerializeToUtf8Bytes(person, MyJsonContext.Default.Person)
person = JsonSerializer.Deserialize(utf8Json, MyJsonContext.Default.Person);
```

How to see Source Generated code



Program.g...enerated]

This file is auto-generated by the generator 'SourceGenerator.HelloSourceGenerator' and can

```
// Auto-generated code
using System;

namespace ConsoleApp
{
    public static partial class Program
    {
        static partial void HelloFrom(str
            Console.WriteLine( value: $"Gen
        }
}
```

Solution Explorer

```
Solution 'source-generators' (2 of 2 projects)
  ConsoleApp
    Dependencies
      Analyzers
        Microsoft.CodeAnalysis.CSharp.NetAnalyzers
        Microsoft.CodeAnalysis.NetAnalyzers
        SourceGenerator
          SourceGenerator.HelloSourceGenerator
            Program.g.cs
        System.Text.Json.SourceGeneration
      Frameworks
      Projects
    Program.cs
  SourceGenerator
    Dependencies
    HelloSourceGenerator.cs
```

realm
by MongoDB

Object-oriented and cross platform database

Created with mobile in mind

Available for multiple programming languages and platforms

Lots of nice features (including Device Sync)....

Realm objects are live
Live objects always reflect the latest data stored in Realm

The past (IL Weaving)

Defined model

```
public class Person : RealmObject
{
    [PrimaryKey]
    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public IList<Dog> Dogs { get; }
}
```

Weaved model

```
public class Person : RealmObject
{
    public string Name
    {
        //Simplified
         get => GetValue("Name");
         set => SetValue(“Name”, value);
    }
    //…
}
```

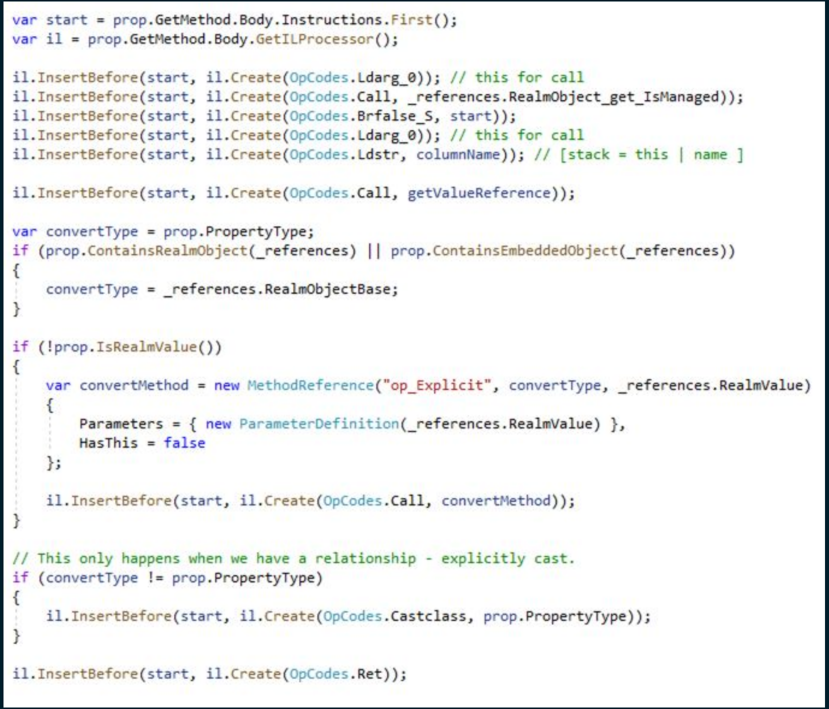
Defined model (IL)

```
.method public hidebysig specialname
    instance string get_Name () cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldfld string
Person::'<Name>k__BackingField'
    IL_0006: ret
} // end of method Person::get_Name
```

Weaved model (IL)

```
.method public hidebysig specialname
    instance string get_Name () cil managed
{
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldfld bool RealmObject::IsManaged
    IL_0006: brtrue.s IL_000f
    IL_0008: ldarg.0
    IL_0009: ldfld string Person2::_name
    IL_000e: ret
    IL_000f: ldarg.0
    IL_0010: ldstr "Name"
    IL_0015: call instance string
RealmObject::GetValue(string)
    IL_001a: ret
} // end of method Person::get_Name
```

IL Weaving drawbacks



```
var start = prop.GetMethod.Body.Instructions.First();
var il = prop.GetMethod.Body.GetILProcessor();

il.InsertBefore(start, il.Create(OpCodes.Ldarg_0)); // this for call
il.InsertBefore(start, il.Create(OpCodes.Call, _references.RealmObject_get_IsManaged));
il.InsertBefore(start, il.Create(OpCodes.Brfalse_S, start));
il.InsertBefore(start, il.Create(OpCodes.Ldarg_0)); // this for call
il.InsertBefore(start, il.Create(OpCodes.Ldstr, columnName)); // [stack = this | name ]

il.InsertBefore(start, il.Create(OpCodes.Call, getValueReference));

var convertType = prop.PropertyType;
if (prop.ContainsRealmObject(_references) || prop.ContainsEmbeddedObject(_references))
{
    convertType = _references.RealmObjectBase;
}

if (!prop.IsRealmValue())
{
    var convertMethod = new MethodReference("op_Explicit", convertType, _references.RealmValue)
    {
        Parameters = { new ParameterDefinition(_references.RealmValue) },
        HasThis = false
    };

    il.InsertBefore(start, il.Create(OpCodes.Call, convertMethod));
}

// This only happens when we have a relationship - explicitly cast.
if (convertType != prop.PropertyType)
{
    il.InsertBefore(start, il.Create(OpCodes.Castclass, prop.PropertyType));
}

il.InsertBefore(start, il.Create(OpCodes.Ret));
```

Not readable
IL code is difficult to read and to reason about

Difficult to extend
Weaver requires specific knowledge and a lot of trial and error

Black box
Changes to IL are “not visible” to final user

Not debuggable
It’s not possible to step into the weaved code

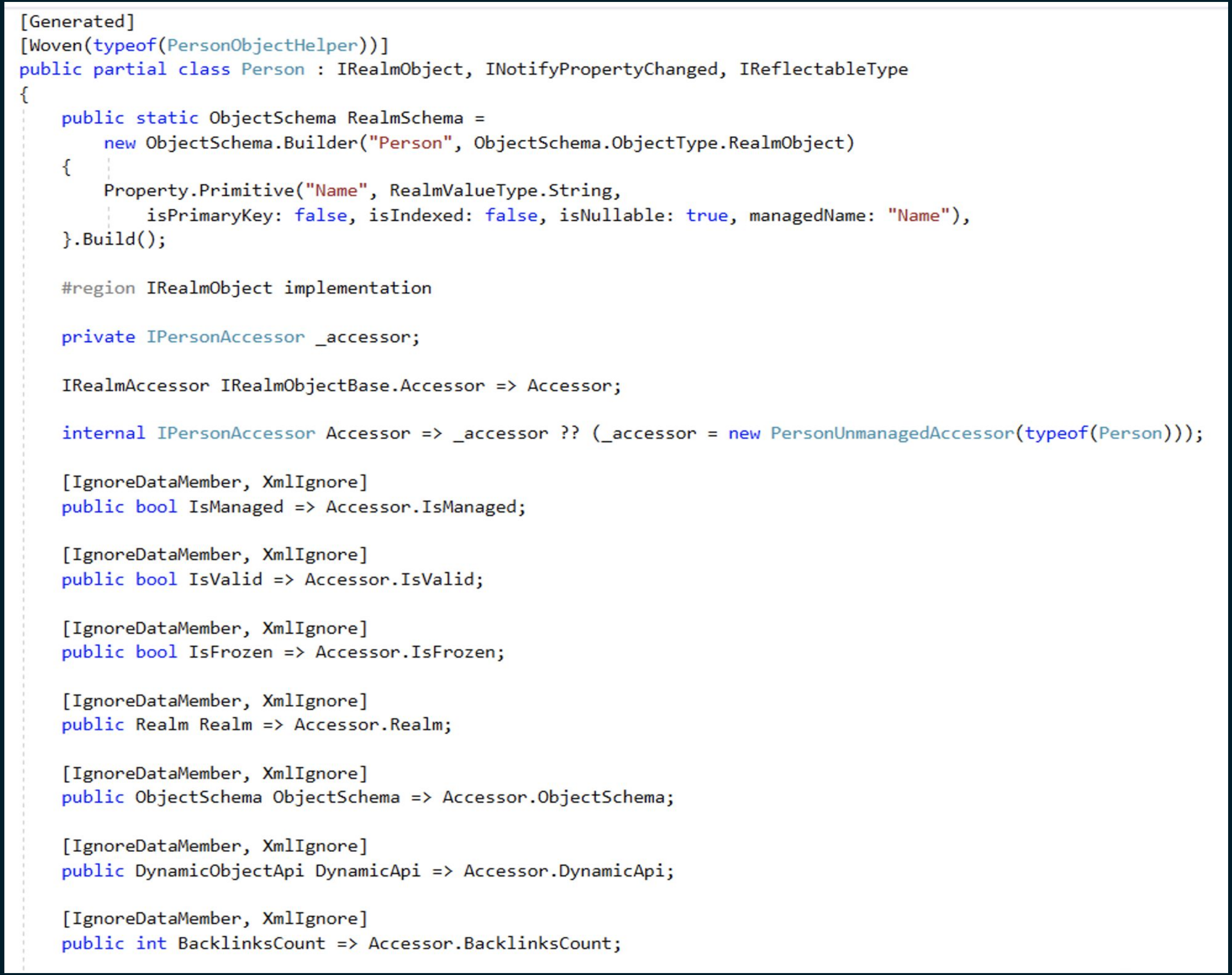
The future (Source Generators)

Classic model

```
public class Person : RealmObject
{
    [PrimaryKey]
    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public IList<Dog> Dogs { get; }
}
```

New model

```
public partial class Person : IRealmObject
{
    [PrimaryKey]
    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public IList<Dog> Dogs { get; }
}
```



```
[Generated]
[Woven(typeof(PersonObjectHelper))]
public partial class Person : IRealmObject, INotifyPropertyChanged, IReflectableType
{
    public static ObjectSchema RealmSchema =
        new ObjectSchema.Builder("Person", ObjectSchema.ObjectType.RealmObject)
    {
        Property.Primitive("Name", RealmValueType.String,
            isPrimaryKey: false, isIndexed: false, isNullable: true, managedName: "Name"),
    }.Build();

    #region IRealmObject implementation

    private IPersonAccessor _accessor;

    IRealmAccessor IRealmObjectBase.Accessor => Accessor;

    internal IPersonAccessor Accessor => _accessor ?? (_accessor = new PersonUnmanagedAccessor(typeof(Person)));

    [IgnoreDataMember, XmlIgnore]
    public bool IsManaged => Accessor.IsManaged;

    [IgnoreDataMember, XmlIgnore]
    public bool IsValid => Accessor.IsValid;

    [IgnoreDataMember, XmlIgnore]
    public bool IsFrozen => Accessor.IsFrozen;

    [IgnoreDataMember, XmlIgnore]
    public Realm Realm => Accessor.Realm;

    [IgnoreDataMember, XmlIgnore]
    public ObjectSchema ObjectSchema => Accessor.ObjectSchema;

    [IgnoreDataMember, XmlIgnore]
    public DynamicObjectApi DynamicApi => Accessor.DynamicApi;

    [IgnoreDataMember, XmlIgnore]
    public int BacklinksCount => Accessor.BacklinksCount;
```

The bright side

Readable and Debuggable
Generated code can be inspected and debugged

Easy to work with
The generated code is just “plain” code, easy to reason with

Extensible
Allow us to introduce support for new features much faster (nullability…)

Part of compilation pipeline
Integrated in build system

The less bright side

Text generation
Generating readable code is annoying

Performance
Source generators can run multiple times, even with no changes

Only Additive
Existing code cannot be modified

IL Weaving is still there 😓

Weaving

```
class Person : RealmObject
{
    [PrimaryKey]
    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public IList<Dog> Dogs { get; }
}
```

SG

```
partial class Person : IRealmObject
{
    [PrimaryKey]
    private Guid _id;

    private string _name;

    private int _age;

    private IList<Dog> _dogs;
}
```

SG + Weaving

```
partial class Person : IRealmObject
{
    [PrimaryKey]
    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public IList<Dog> Dogs { get; }
}
```

New model

```
Public partial class Person : IRealmObject
{
    [PrimaryKey]
    public Guid Id { get; set; }

    public string Name { get; set; }

    public int Age { get; set; }

    public IList<Dog> Dogs { get; }
}
```

Compiled model

```
public partial class Person : IRealmObject
{
    public string Name
    {
        get => Accessor.Name;
        set => Accessor.Name = value;
    }
    //…
}
```

Status and Future Work

In the pipeline for about 6 months

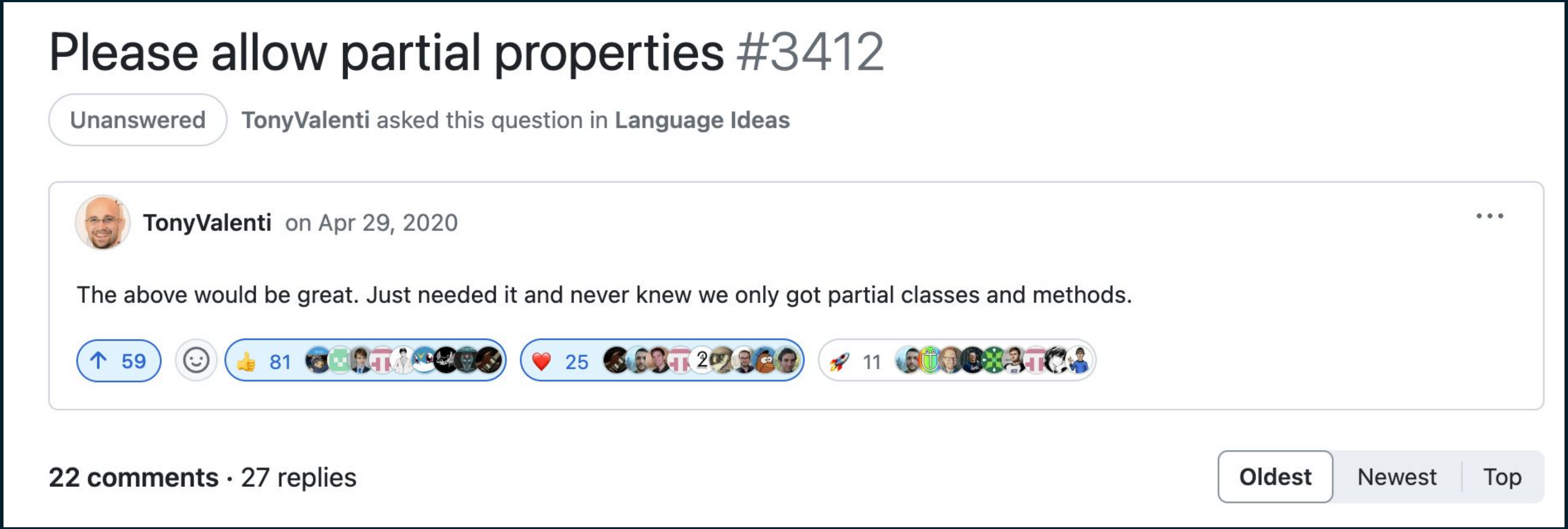
`Realm.SourceGenerator` has been published in Nov 2022 (v 10.18.0)

“Classic” model definition is still supported but pushing for source generated classes

Added support for nullability in model definition

Planning to add incremental generator

Completely remove IL Weaving (?)



Please allow partial properties #3412

Unanswered TonyValenti asked this question in Language Ideas

TonyValenti on Apr 29, 2020

The above would be great. Just needed it and never knew we only got partial classes and methods.

22 comments · 27 replies

Conclusion

Code generation is useful to hide complexity in the .NET Realm SDK

IL Weaving is powerful but difficult

Source Generators are the “modern” alternative

They have their own quirks/limitations

Overall, the switch was worth it

Thank you for your time!

Resources

Download .NET 8
aka.ms/get-dotnet-8

Realm
github.com/realm/realm-dotnet

What does Realm.Fody do?
papafe.dev/posts/realm-fody/

Source Generators
docs.microsoft.com/source-generators

Download .NET 8

https://aka.ms/get-dotnet-8