



Driving IT

From IL Weaving to Source Generators

the Realm story



Ferdinando Papale | .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

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

IL can be modified with *Weaving*

Weaving happens **after** compilation

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

*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: ldflld 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



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

    public string GivenNames { get; set; }
    public string FamilyName { get; set; }
    public string FullName => $"{GivenNames} {FamilyName}";
}
```

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

    string givenNames;
    public string GivenNames
    {
        get => givenNames;
        set
        {
            if (value != givenNames)
            {
                givenNames = value;
                OnPropertyChanged("GivenNames");
                OnPropertyChanged("FullName");
            }
        }
    }

    string familyName;
    public string FamilyName
    {
        get => familyName;
        set
        {
            if (value != familyName)
            {
                familyName = value;
                OnPropertyChanged("FamilyName");
                OnPropertyChanged("FullName");
            }
        }
    }

    public string FullName => $"{GivenNames} {FamilyName}";

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

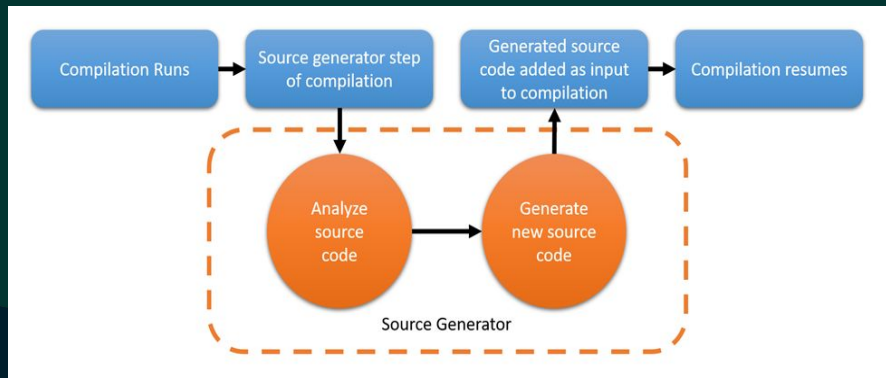
Source Generators

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

Source generation happens **during** compilation

Only additive

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);
```



Realm is an
object-oriented and
cross-platform
database

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; }
}
```

Compiled 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)



Broader re-think of
the SDK associated
with the move to
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...)



The less bright side

Tooling

There are various tooling issues appearing while working with Source Generators

No central “knowledge base”

Lots of googling

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
just been published in beta

Planning to support nullability

Planning to add incremental
generator

Completely remove IL Weaving (?)

Conclusion

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



IL Weaving is powerful but difficult

Source Generators are a good alternative

They have their own quirks/limitations

The switch was worth it, it will allow to introduce new features in an easier way

Thank you for
your time.