TypeGen Documentation

Release 2.4.8

Jacek Burzynski

Feb 26, 2020

Contents

TypeGen is a single-class-per-file C# to TypeScript generator.

Use:

- Download from NuGet
- Use as .NET CLI tool (NuGet package)
- Browse the source code

Contents:

CHAPTER 1

Overview

1.1 About TypeGen

TypeGen is a command line tool that generates single-class-per-file TypeScript sources from C#. Its primary use case is keeping TypeScript model classes/interfaces in sync with C# models, and for this reason not all C# features are translated to TypeScript (e.g. methods or events will be ignored when generating TypeScript sources).

1.2 Requirements

For .NET Standard compatibility, see compatibility table.

Versions >= 2.0.0

- Package Manager Console TypeGen CLI: the latest version of .NET Core
- TypeGen.DotNetCli NuGet package: the latest version of .NET Core
- dotnet-typegen NuGet package: multiple .NET Core versions are supported: from .NET Core 2.1 to the latest version of .NET Core
- TypeGen.Core: .NET Standard versions: 1.3 and 2.0

Versions 1.5.7 - 1.6.7

- CLI: .NET Core 2.0
- TypeGen.Core: .NET Standard versions: 1.3 and 2.0

Versions 1.5.0 - 1.5.6

- CLI: .NET Framework 4.6
- TypeGen.Core: .NET Standard 1.3

Versions <= 1.4.x

.NET Framework 4.0

1.3 How to use it

The general idea is: first you select C# types to be generated to TypeScript, then you run the `typegen` command and the TypeScript files are generated.

The details are covered in the following subsections.

1.3.1 Installing TypeGen

To install TypeGen, add the TypeGen NuGet package to your project. After adding this package, the `TypeGen` command will be available in the Package Manager console.

You can also use TypeGen as a .NET Core CLI tool, in which case you should install it from this package.

1.3.2 Selecting C# types to generate

C# types to generate can be selected in 2 ways:

1. By specifying *attributes* on the C# types you wish to generate.
2. By creating one or more *generation specs* (a generation spec is a C# class which specifies which types should be generated) somewhere in your project. To instruct TypeGen to use your generation spec, place this content in a file named *tgconfig.json* in your project directory:

```
{
    "generationSpecs": ["MyGenerationSpec"]
}
```

Attributes offer quicker, but more functionally-restricted way of selecting types to generate (recommended for smaller projects), whereas generation specs require more initial work to do, but offer richer functionality (recommended for bigger projects).

After adding attributes or creating/changing generation spec(s), you need to build your project before running the `TypeGen` command.

1.3.3 Running the TypeGen command

To run TypeGen from the Package Manager console, open the PM console, select your project from the dropdown list and type `TypeGen generate` or `TypeGen -p "MyProjectName" generate` (depending on the current working directory of the PM Console; you might have to restart Visual Studio).

To run TypeGen as a .NET Core CLI tool, type `dotnet-typegen generate` or `dotnet-typegen -p "MyProjectName" generate` (depending on your current location) in your OS command shell.

1.4 Example

Let's say you have a *ProductDto* class that you want to export to TypeScript.

1. If you're using attributes, annotate your class with an appropriate attribute:

```
[ExportTsClass]
public class ProductDto
{
    public decimal Price { get; set; }
    public string[] Tags { get; set; }
}
```

2. If you're using a generation spec, first create your generation spec somewhere in your project:

```
public class MyGenerationSpec : GenerationSpec
{
    public MyGenerationSpec()
    {
        AddClass<ProductDto>();
    }
}
```

. . . and then create a file named *tgconfig.json* directly in your project folder and place the following content in this file:

```
{
    "generationSpecs": ["MyGenerationSpec"]
}
```

After finishing instructions described in either 1. or 2., **build your project** and type `TypeGen generate` or `TypeGen -p "MyProjectName" generate` (depending on the current working directory of the PM Console) into the Package Manager Console (you might have to restart Visual Studio). Instead of using the Package Manager Console, you can also use TypeGen as a .NET Core CLI tool by typing `dotnet-typegen generate` in your OS command shell.

This will generate a single TypeScript file (named *product-dto.ts*) in your project directory. The file will look like this:

```
export class ProductDto {
    price: number;
    tags: string[];
}
```

1.5 What next

More details about the available configuration options (that you can place in *tgconfig.json*) are described in the *CLI* section. You can also find out more about *attributes* or *generation specs* in their dedicated sections.

If you need to convert between different naming conventions (i.e. your C# code uses different conventions than your TypeScript code), you can utilize the *converters* functionality.

Instead of using the `TypeGen` command in the console, you can generate files directly from your code using the *TypeGen programmatical API*.

CHAPTER 2

What is generated?

2.1 C# properties and fields

Both properties and fields will be generated to TypeScript, unless marked with *TsIgnore* attribute. Fields are always declared before properties in the resulting TypeScript file.

The following class:

```
[ExportTsClass]
public class MyClass
{
    public int MyProperty { get; set; }
    public string MyField { get; set; }
    public string MyProperty2 { get; set; }
}
```

. . . will be generated to:

```
export class MyClass {
    myField: string;
    myProperty: number;
    myProperty2: string;
}
```

2.1.1 Static, readonly and const properties/fields

By default, any static, readonly or const C# property/field is translated as shown below:

- *static* property/field -> TypeScript *static* property (only for TypeScript classes)
- *readonly* field -> TypeScript *readonly* property (for both TypeScript classes and interfaces; additionally, the property's default value is generated for TS classes)

- *const* field -> for TypeScript classes: *static readonly* property with default value; for TypeScript interfaces: *readonly* property

To change the behavior described above, you can use a combination of *TsIgnore*, *TsDefaultValue*, *TsStatic/TsNonStatic* or *TsReadonly/TsNotReadonly* attributes on a property or field (more information in the *TypeGen attributes* section).

2.1.2 Default values for TypeScript properties

A default value for a TypeScript property will be generated in any of the following cases:

- The C# property/field is annotated with the *TsDefaultValue* attribute
- The C# property/field has a non-default value
- A default value is specified for the generated property's type in either the *defaultValuesForTypes* CLI parameter or the *GeneratorOptions.DefaultValuesForTypes* property

The order of checking for a default value to use is the same as listed above (first *TsDefaultValueAttribute*, then the member's value, then default value for a TS type).

2.2 Primitive property/field types

Property/field types that can be represented by TypeScript primitive types will be automatically mapped to the corresponding TypeScript types. Default mapping of C# to TypeScript files is presented below:

- *dynamic -> any*
- *all C# built-in numeric and byte types -> number*
- *string, char, Guid -> string*
- *bool -> boolean*
- *object -> object*
- *DateTime, DateTimeOffset -> Date*

Additionally, any type that implements the *IDictionary* interface (or the *IDictionary* interface itself) will be mapped to TypeScript dictionary type. For example, *Dictionary<int, string>* will be mapped to *{ [key: number]: string; }*.

There is an option to override the default mappings or create new [C# to TypeScript] type mappings by using the *customTypeMappings* option in the CLI or *GeneratorOptions.CustomTypeMappings* in the programmatical API.

2.3 Complex property/field types

If the type of a property or field is a complex type (custom object or enum), the following strategy is used:

1. If the complex type is annotated with a *ExportTs...* attribute, its TypeScript source will be generated according to the path specified in the attribute.
2. If the complex type is **not** annotated with a *ExportTs...* attribute, its TypeScript source will be generated in the same folder as the containing C# type (which is currently being exported).

Therefore, it has to be kept in mind that if a complex type is not annotated with a *ExportTs...* attribute, it may be generated more than once (for each type that uses this complex type).

Example #1

C# sources:

```
[ExportTsClass(OutputDir = "my/folder1")]
public class MyClass1
{
    public int MyProperty { get; set; }
    public MyClass2 MyComplexProperty { get; set; }
}

public class MyClass2
{
    public int SomeProperty { get; set; }
}
```

my-class1.ts (in *my/folder1*):

```
import { MyClass2 } from "./my-class2";

export class MyClass1 {
    myProperty: number;
    myComplexProperty: MyClass2;
}
```

my-class2.ts (in *my/folder1*):

```
export class MyClass2 {
    someProperty: number;
}
```

Example #2

C# sources:

```
[ExportTsClass(OutputDir = "my/folder1")]
public class MyClass1
{
    public int MyProperty { get; set; }
    public MyClass2 MyComplexProperty { get; set; }
}

[ExportTsClass(OutputDir = "my/folder2")]
public class MyClass2
{
    public int SomeProperty { get; set; }
}
```

my-class1.ts (in *my/folder1*):

```
import { MyClass2 } from "../folder2/my-class2";

export class MyClass1 {
    myProperty: number;
    myComplexProperty: MyClass2;
}
```

my-class2.ts (in *my/folder2*):

```
export class MyClass2 {
    someProperty: number;
}
```

2.4 Collection types

All collection or nested collection types will be exported by TypeGen. E.g., for a C# source looking like this:

```
[ExportTsClass]
public class MyClass
{
    public int[] IntArray { get; set; }
    public IEnumerable<int> IntEnumerable { get; set; }
    public IEnumerable<int[]> IntEnumArrayCombo { get; set; }
    public IEnumerable<IList<int[]>> IntEnumListArrayCombo { get; set; }
}
```

. . . the following TypeScript file will be generated:

```
export class MyClass {
    intArray: number[];
    intEnumerable: number[];
    intEnumArrayCombo: int[][];
    intEnumListArrayCombo: int[][][];
}
```

2.5 Base classes

Since TypeGen 1.2, base classes are automatically generated for both TypeScript classes and interfaces.

```
[ExportTsClass]
public class MyClass : BaseClass
{
    public string MyProperty { get; set; }
}

public class BaseClass
{
    public int BaseField;
}
```

For this code, two TypeScript files will be generated: one for *MyClass* and one for *BaseClass*:

```
export class MyClass extends BaseClass
{
    myProperty: string;
}

export class BaseClass
{
    baseField: number;
}
```

2.6 Generic types

TypeGen 1.2 introduces TypeScript files generation for custom generic types. Both generic type definitions and generic types (i.e. types with filled parameters) are exported to TypeScript. Additionally, generic type constraints (specified in

where) will also be exported to TypeScript (**note**: *new()* and *class* constraints will not be exported).

Example:

```
[ExportTsClass]
public class MyClass<T, U> : BaseClass<T> where T: GenericClass<string>
{
    public T GenericProperty1 { get; set; }
    public U GenericProperty2 { get; set; }
    public GenericClass<int> GenericClassProperty { get; set; }
}

public class BaseClass<T>
{
    public T BaseProperty { get; set; }
}

public class GenericClass<T>
{
    public T GenericClassField;
}
```

From this code, the following TypeScript sources will be generated:

```
export class MyClass<T extends GenericClass<string>, U> extends BaseClass<T> {
    genericProperty1: T;
    genericProperty2: U;
    genericClassProperty: GenericClass<number>;
}

export class BaseClass<T> {
    baseProperty: T;
}

export class GenericClass<T> {
    genericClassField: T;
}
```

CHAPTER 3

TypeGen attributes

Using attributes is one of the two ways of selecting C# types to be generates as TypeScript sources (the other one being *generation spec*).

Attributes have some advantages over generation specs:

- they are simpler and require less overhead
- for some people they are more readable, as you can see the configuration as you read the type definition

Compared to generation specs they also have some limitations:

- configuration is tied to a type - you can only have a single configuration for a given type
- you can only generate types that you wrote - i.e. you cannot generate types from external assemblies

Typical use case for using attributes is smaller projects or projects without complex file generation logic.

3.1 ExportTs... attributes

In order to indicate that a given type is to be exported to TypeScript, it needs to be annotated with a *ExportTs...* attribute. The available attributes are:

- *ExportTsClassAttribute*
- *ExportTsInterfaceAttribute*
- *ExportTsEnumAttribute*

Each of the attributes has one property named *OutputDir*, which can be set to indicate the output directory of the generated TypeScript file. The output directory (and all other defined paths) are resolved relatively to the generator's base output directory (*ProjectFolder* parameter in CLI).

For example, to export a class as a TypeScript interface to *my/sources* folder, the following code can be used:

```
[ExportTsInterface(OutputDir = "my/sources")]
public class MyClass
{
    public int MyProperty { get; set; }
}
```

In this case, *MyClass* (by default) will be exported to *my/sources/my-class.ts* upon generation.

3.2 TsCustomBaseAttribute

The TsCustomBaseAttribute allows for specifying a custom declaration of the base type. Can be used on classes and interfaces. If no base class is specified, base class declaration will be removed. When TsCustomBaseAttribute is used on a class/interface, its base type will **still be generated**. To avoid base type generation, use TsIgnoreBaseAttribute placed before TsCustomBaseAttribute.

```
[ExportTsClass]
[TsCustomBase("CustomBase")]
public class MyClass : MyBase
{
    public string MyProperty { get; set; }
}
```

resulting TypeScript file:

```
export class MyClass extends CustomBase {
    myProperty: string;
}
```

If the used base type requires an *import* statement to be present, the import path can be specified as a second constructor argument. Additionally, if the type is used as an alias, the original type name can be specified as the third constructor argument.

Example:

```
[ExportTsClass(OutputDir = "my/sources")]
[TsCustomBase("CustomBase", "../some/path/custom-base")]
public class MyClass
{
    // ...
}
```

This will result in the following being generated:

```
import { CustomBase } from "../some/path/custom-base";

export class MyClass extends CustomBase {
    // ...
}
```

3.3 TsDefaultExportAttribute

Used to specify if the generated TypeScript type should use a default export. There is also a possibility to enable default exports globally in the CLI (*useDefaultExport* parameter) or in the generator options (*GeneratorOp-*

tions.UseDefaultExport).

Opt-in:

```
[ExportTsInterface]
[TsDefaultExport]
public class MyInterface
{
    //...
}
```

translates to:

```
interface MyInterface {
    //...
}
export default MyInterface;
```

Opt-out:

```
[ExportTsInterface]
[TsDefaultExport(false)]
public class MyInterface
{
    //...
}
```

translates to:

```
export interface MyInterface {
    //...
}
```

3.4 TsDefaultTypeOutputAttribute

Since TypeGen 1.2, there is an option to specify a default output path for a member's type, which will be used if no *ExportTs...* attribute is present for this type. The path is relative to the project's folder (when using CLI) or generator's base directory (when generating programmatically).

```
[ExportTsClass(OutputDir = "my/sources")]
public class MyClass
{
    [TsDefaultTypeOutput("custom/output/path")]
    public CustomType CustomTypeProperty { get; set; }
}
```

In this example, type *CustomType* will be generated in *custom/output/path* directory if no *ExportTs...* attribute is specified in *CustomType* definition.

3.5 TsDefaultValueAttribute

To indicate a default value for a TypeScript property, the *TsDefaultValue* attribute can be used:

```
[ExportTsClass(OutputDir = "my/sources")]
public class MyClass
{
    [TsDefaultValue("3.14")]
    public int MyProperty { get; set; }

    [TsDefaultValue("\"A string\"")]
    public string MyStringProperty { get; set; }
}
```

The parameter passed in *TsDefaultValue*'s constructor is a string, which is directly copied to the TypeScript file as a default value for a property. Therefore, default values for string properties have to be wrapped with " (double quote) or ' (single quote) characters, in order to generate a valid TypeScript file.

Feature for versions >= 2.0.0: If you want to specify default values to be generated for all members with a given TypeScript type, you can do so by using the *defaultValuesForTypes* option in the CLI or *GeneratorOptions.DefaultValuesForTypes* in the programmatical API.

3.6 TsIgnoreAttribute

To ignore a property or field when generating TypeScript source, *TsIgnore* attribute can be used:

```
[ExportTsClass(OutputDir = "my/sources")]
public class MyClass
{
    public int MyProperty { get; set; }

    [TsIgnore]
    public string IgnoredProperty { get; set; }
}
```

In this case, the generated TypeScript class will only contain *MyProperty* property declaration.

3.7 TsIgnoreBaseAttribute

TsIgnoreBaseAttribute causes the base class/interface declaration to be empty **and** the base type to not be generated (unless the base type itself is marked with an *ExportTs...* attribute).

```
[ExportTsClass]
[TsIgnoreBase]
public class MyClass : MyBase
{
    public string MyProperty { get; set; }
}
```

generated TypeScript (MyBase is not generated if it doesn't have an *ExportTs...* attribute):

```
export class MyClass {
    myProperty: string;
}
```

3.8 TsMemberNameAttribute

The TsMemberName attribute allows to override the generated TypeScript property name.

```
[ExportTsClass(OutputDir = "my/sources")]
public class MyClass
{
    [TsMemberName("customProperty")] // will generate as customProperty: string;
    public string MyProperty { get; set; }
}
```

3.9 TsNull, TsNotNull, TsUndefined, TsNotUndefined attributes

These attributes are used to indicate an opt-in/out *null* or *undefined* type union (used in TypeScript null checking mode). Negative (*not*) attributes have precedence over positive attributes. E.g. this definition:

```
[ExportTsClass]
public class MyClass
{
    [TsNull]
    public string MyProperty { get; set; }
}
```

will be translated to:

```
export class MyClass {
    myProperty: string | null;
}
```

If string initializers are enabled, the above opt-in example will produce the following TypeScript:

```
export enum MyEnum
{
    A = "A",
    B = "B"
}
```

To specify custom logic for changing (converting) a C# enum value name to an enum initializer string, you can specify enum string initializers converters in the CLI (*enumStringInitializersConverters* parameter) or in the generator options (*GeneratorOptions.EnumStringInitializersConverters*).

3.10 TsOptionalAttribute

Marks an interface property as optional.

```
[ExportTsInterface]
public class MyInterface
{
    [TsOptional]
    public string MyProperty { get; set; }
}
```

translates to:

```
export interface MyInterface {
    myProperty?: string;
}
```

3.11 TsReadonly, TsNotReadonly attributes

These attributes are used to mark a property or a field as readonly / not readonly.

C# code:

```
[ExportTsClass]
public class MyClass
{
    public readonly string ReadonlyField = "value";

    [TsNotReadonly]
    [TsDefaultValue(null)]
    public readonly string ReadonlyOptOut;

    [TsReadonly]
    public string ReadonlyOptIn = "value";

    [TsNotReadonly]
    public const string NotReadonlyConst = "value";
}
```

generated TypeScript:

```
export class MyClass {
    readonly readonlyField: string = "value";
    readonlyOptOut: string;
    readonly readonlyOptIn: string = "value";
    static notReadonlyConst: string = "value";
}
```

3.12 TsStatic, TsNotStatic attributes

These attributes are used to mark a property or a field as static / not static.

C# code:

```
[ExportTsClass]
public class MyClass
{
    public static string StaticField;

    [TsNotStatic]
    public static string StaticOptOut;

    [TsStatic]
    public string StaticOptIn;

    [TsNotStatic]
```

(continues on next page)

(continued from previous page)

```
    public const string NotStaticConst = "value";
}
```

generated TypeScript:

```
export class MyClass {
    static staticField: string;
    staticOptOut: string;
    static staticOptIn: string;
    readonly notStaticConst: string = "value";
}
```

3.13 TsStringInitializersAttribute

Used to specify if TypeScript string initializers should be used for an enum. There is also a possibility to enable enum string initializers globally in the CLI (*enumStringInitializers* parameter) or in the generator options (*GeneratorOptions.EnumStringInitializers*).

Opt-in:

```
[ExportTsEnum]
[TsStringInitializers]
public enum MyEnum
{
    A,
    B
}
```

Opt-out:

```
[ExportTsEnum]
[TsStringInitializers(false)]
public enum MyEnum
{
    A,
    B
}
```

3.14 TsTypeAttribute

There is a possibility to override the generated TypeScript type for a property or field. To do so, the *TsType* attribute can be used:

```
[ExportTsClass(OutputDir = "my/sources")]
public class MyClass
{
    public int MyProperty { get; set; }

    [TsType(TsType.String)]
    public int StringProperty { get; set; }
```

(continues on next page)

(continued from previous page)

```
    [TsType("CustomType")]
    public string CustomTypeProperty { get; set; }
}
```

The attribute's constructor allows for 2 methods of specifying the TypeScript type:

- explicitly - by typing the string value that will be used as a TypeScript type
- by using the *TsType* enum - the *TsType* enum contains values representing the built-in TypeScript types (object, boolean, string, number, Date and any)

If the used type requires an *import* statement to be present, the import path can be specified as a second constructor argument. Additionally, if the type is used as an alias, the original type name can be specified as the third constructor argument.

Example:

```
[ExportTsClass(OutputDir = "my/sources")]
public class MyClass
{
    [TsType("CT", "../some/path/custom-type", "CustomType")]
    public string CustomTypeProperty { get; set; }
}
```

This will result in the following being generated:

```
import { CustomType as CT } from "../some/path/custom-type";

export class MyClass {
    customTypeProperty: CT;
}
```

3.15 TsTypeUnionsAttribute

The *TsTypeUnions* attribute specifies additional TypeScript type unions for a member.

Example:

```
[ExportTsClass]
public class MyClass
{
    [TsTypeUnions("null")]
    public int IntProperty { get; set; }

    [TsTypeUnions("null", "undefined")]
    public string StringProperty { get; set; }

    [TsTypeUnions("string")]
    public DateTime DateTimeProperty { get; set; }
}
```

The above will produce the following TypeScript file:

```
export class MyClass {
    intProperty: number | null;
```

(continues on next page)

(continued from previous page)

```
    stringProperty: string | null | undefined;
    dateTimeProperty: Date | string;
}
```

CHAPTER 4

Generation spec

Generation spec is one of the two ways of selecting C# types to be generates as TypeScript sources (the other one being *attributes*).

Using a generation spec has a few advantages over attributes:

- you can generate types from external assemblies
- configuration is not tied up to a type - you can make different configurations for the same type in different generation specs
- you can create reusable configurations and combine them together

Using generation specs is typically good for bigger projects or projects with more complex file generation logic.

4.1 Overview

Generation spec is a class that specifies which types should be generated to TypeScript and how barrels (if any) should be generated. All generation specs must derive from *TypeGen.Core.SpecGeneration.GenerationSpec*. Inside a generation spec, it is possible to select types for generation or add barrels by invoking *AddClass*, *AddInterface*, *AddEnum* or *AddBarrel* methods in 3 different places:

- **OnBeforeGeneration method** - a virtual method invoked before files are generated. It's advised to add all C# types to generate in this method, as it exposes the currently used *GeneratorOptions* instance (available through *OnBeforeGenerationArgs*), which can be helpful in specifying additional logic based on the currently used generation options. This is not the optimal place to put *AddBarrel* calls, as the directory structure for TypeScript sources may not yet have been created in the file system.
- **OnBeforeBarrelGeneration method** - a virtual method invoked after C# types are translated and saved to TypeScript files, but before any barrels are generated. This is the place to put any *AddBarrel* method calls. Because *OnBeforeBarrelGeneration* is invoked after TypeScript types are saved in the file system, it's possible to add barrel files based on the directory structure of the generated TypeScript files (*GeneratorOptions.BaseOutputDirectory* can be used to access the "global" output directory for the generated TypeScript sources).

- **class constructor** - class constructor is invoked before files are generated, and therefore it's possible to add C# types for generation in the constructor. However, this should be avoided in favor of adding types in the *OnBeforeGeneration* method, as *OnBeforeGeneration* gives access to an instance of *GeneratorOptions* (i.e. options used for file generation), which can be further used to customize the logic of adding C# types.

It is possible to express everything that can be written with *attributes* using generation specs, because each attribute has an equivalent in the generation spec configuration. For example, an equivalent of `[TsTypeAttribute(TsType.String)]` would e.g. be `AddClass<...>().Member("MemberName").Type(TsType.String)`.

To perform file generation from one or more generation specs, their class names should be passed to the *generationSpecs* *config option*.

If any generation specs are present in the *generationSpecs* config option, TypeGen will perform file generation **only** from generation specs (i.e. not from attributes in the specified assemblies). To perform file generation from both attributes (in the specified assemblies) and generation specs, you can use the *generateFromAssemblies* config option.

4.2 Examples

```
public class MyProjectGenerationSpec : GenerationSpec
{
    public override void OnBeforeGeneration(OnBeforeGenerationArgs args)
    {
        AddClass<ProductDto>();

        AddInterface<CarDto>("output/directory");

        AddClass<PersonDto>()
            .Member(nameof(PersonDto.Id))  // specifying member options
            .Ignore()
            .Member(x => nameof(x.Age))    // you can specify member name with lambda
            .Type(TsType.String);

        AddInterface<SettingsDto>()
            .IgnoreBase();                 // specifying type options

        AddClass(typeof(GenericDto<>));    // specifying types by Type instance

        AddEnum<ProductType>("output/dir") // specifying an enum

        // generate everything from an assembly

        foreach (Type type in GetType().Assembly.GetLoadableTypes())
        {
            AddClass(type);
        }

        // generate types by namespace

        IEnumerable<Type> types = GetType().Assembly.GetLoadableTypes()
            .Where(x => x.FullName.StartsWith("MyProject.Web.Dtos"));
        foreach (Type type in types)
        {
            AddClass(type);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    public override void OnBeforeBarrelGeneration(OnBeforeBarrelGenerationArgs args)
    {
        AddBarrel(".", BarrelScope.Files); // adds one barrel file in the global␣
↪TypeScript output directory containing only files from that directory

        AddBarrel(".", BarrelScope.Files | BarrelScope.Directories); // equivalent to␣
↪AddBarrel("."); adds one barrel file in the global TypeScript output directory␣
↪containing all files and directories from that directory


        // the following code for each directory creates a barrel file containing all␣
↪files and directories from that directory

        IEnumerable<string> directories = GetAllDirectoriesRecursive(args.
↪GeneratorOptions.BaseOutputDirectory)
            .Select(x => GetPathDiff(args.GeneratorOptions.BaseOutputDirectory, x));

        foreach (string directory in directories)
        {
            AddBarrel(directory);
        }

        AddBarrel(".");
    }

    private string GetPathDiff(string pathFrom, string pathTo)
    {
        var pathFromUri = new Uri("file:///" + pathFrom?.Replace('\\', '/'));
        var pathToUri = new Uri("file:///" + pathTo?.Replace('\\', '/'));

        return pathFromUri.MakeRelativeUri(pathToUri).ToString();
    }

    private IEnumerable<string> GetAllDirectoriesRecursive(string directory)
    {
        var result = new List<string>();
        string[] subdirectories = Directory.GetDirectories(directory);

        if (!subdirectories.Any()) return result;

        result.AddRange(subdirectories);

        foreach (string subdirectory in subdirectories)
        {
            result.AddRange(GetAllDirectoriesRecursive(subdirectory));
        }

        return result;
    }
}
```

CHAPTER 5

Command Line Interface

The TypeGen Command Line Interface (CLI) can be used either from the Package Manager Console, after installing TypeGen NuGet package, or as a .NET CLI tool.

You may need to **restart Visual Studio** before you can start using TypeGen in the Package Manager Console

Note: This issue has been resolved in Visual Studio 2017 RC

There is a known issue with using TypeGen from Package Manager Console with ASP.NET Core projects, **under Visual Studio 2015**: Package Manager Console doesn't see TypeGen.ps1. If this happens to you, you can copy TypeGen.ps1 from your NuGet packages folder to your solution/project directory and use it from PowerShell or CMD. Alternatively, you may also consider using TypeGen .NET CLI tool instead of TypeGen Package Manager Console tool.

After running the *[dotnet-]typegen* command, the following set of actions is performed for each of the specified project folders:

1. The CLI reads the assembly file (.dll or .exe). By default (if no assembly path is specified in the config), the assembly file is searched recursively in the project folder's *bin* directory. The name of the assembly must match the name of the .csproj or .xproj file present in the project folder.
2. File generation is performed based on the options specified in the configuration file

5.1 New syntax (>= 2.0.0)

```
[dotnet-]typegen [options] [command]

Options:
-h|--help               Show help information
-v|--verbose            Show verbose output
-p|--project-folder     Set project folder path(s)
-c|--config-path        Set config path(s) to use

Commands:
```

(continues on next page)

(continued from previous page)

```
generate     Generate TypeScript files
getcwd       Get current working directory
```

5.2 Old syntax (1.x.x)

```
TypeGen ProjectFolder1[|ProjectFolder2|(...)] [-Config-Path path1[|path2|(...)]] [Get-
↪Cwd] [-h | -Help] [-v | -Verbose]
```

Arguments/options

ProjectFolderN	The path to project(s) that TypeScript sources will be generated for. Package Manager Console runs from the solution directory by default, so usually ProjectFolderN would be the name of the project.
-Config-Path	Paths to config files for the specified projects (the order of config paths must match the order of projects). If a project doesn't have its config specified, *tgconfig.json* file from ProjectFolderN will be used. If *tgconfig.json* is not present, the default configuration will be used.
-Get-Cwd	A utility option. If present, the current working directory of the CLI will be outputted. No file generation will occur when this option is present.
-h or -Help	Shows the help.
-v or -Verbose	The CLI will run in verbose mode. More information will be outputted.

5.3 Configuration file

TypeGen CLI uses a JSON configuration file to read the file generation options. By default, the configuration file is read from the *tgconfig.json* file present in the specified project folder. If *tgconfig.json* does not exist, default options are used. Configuration file path can also be specified as a CLI argument (see syntax above). For configuration parameters not present in the configuration file, their default values are used upon file generation.

The table below shows all available config parameters, with their default values and a description:

Parameter	Type	Default value	Description
addFilesToProject	boolean	false	**Only for .NET Framework apps (not .NET Core)**. Whether to add the generated TypeScript files to the project file (*.csproj)
assemblies	string[]		An array of paths to assembly files to generate TypeScript sources from. If null or empty, the default strategy for finding an assembly will be used. **Note:** the order of assemblies also determines the order of reading any custom converters.
(DEPR.) assemblyPath	string	null	The path to the assembly file with C# types to generate TypeScript sources for. If null or empty, the default strategy for finding an assembly will be used.
buildProject	boolean	false	Whether to build the project before performing file generation
clearOutputDirectory	boolean	false	Whether to clear the output directory before generating new files (removing all files and recursively removing all subdirectories in the output directory)
(DEPR.) createIndexFile	boolean	false	Whether to generate an index (barrel) file in the root TypeScript output directory. The generated barrel file exports everything from all generated TypeScript files. This option should be avoided in favor of generating barrels from generation specs.
csNullableTranslation	string	""	Determines which strict-null-checking type unions will be added to C# nullable property types by default. Possible values: "null", "undefined", "nullundefined" or "".
customTypeMappings	Object	{}	Object containing a map of custom [C# to TypeScript] type mappings (example below)
defaultValuesForTypes	Object	{}	Object containing a map of default values for the specified TypeScript types (example below)
enumStringInitializers	boolean	false	Whether to use TypeScript enum string initializers by default
enumStringInitializersConverters (*)	string[]		Converter chain used for converting C# enum value names to TypeScript enum string initializers. See the (*) explanation below regarding ways in which class names can be specified.
enumValueNameConverters (*)	string[]		Converter chain used for converting C# enum value names to TypeScript enum value names. See the (*) explanation below regarding ways in which class names can be specified.
explicitPublicAccessor	boolean	false	Whether to use explicit *public* accessor in the generated TypeScript class files
externalAssemblyPaths	string[]		An array of paths to external assemblies. These paths are searched (recursively) for any assembly references that cannot be automatically resolved. NuGet package folders (global + machine-wide and project fallback) are searched by default.
fileHeading	string	null	TypeScript file heading text (default is "(...) This is a TypeGen auto-generated file. (...)")
fileNameConverters (*)	string[]	["PascalCaseToKebabCaseConverter"]	Converter chain used for converting C# type names to TypeScript file names. See the (*) explanation below regarding ways in which class names can be specified.
generateFromAssemblies	boolean	null	Whether to generate files from assemblies specified in *assemblies* parameter. If null, files are generated from assemblies only if no generation specs are specified.

(*) The rules for specifying class names are as follows:

- Class names can be specified as a name or a fully qualified name.
- If only the name of a class is specified, the class will first be searched in the assemblies specified in *assemblies* (or the project's assembly if no assemblies are specified) and then (if not found) in *TypeGen.Core*.
- To read a class from a specific assembly, path can be defined in the following format: *assembly/path/assembly.dll:ClassName*, where assembly path is relative to the project's folder.

5.3.1 Example

An example of a configuration file (*tgconfig.json*) is presented below:

```
{
    "assemblies": ["my/app/MyApp.Web.dll", "my/app/MyApp.Models.dll"],
    "fileNameConverters": ["converters/MyApp.Converters.dll:StripDto",
↪"PascalCaseToKebabCase"],
    "typeNameConverters": ["converters/MyApp.Converters.dll:Fqcn.Converters.StripDto
↪"],
    "propertyNameConverters": [],
    "enumValueNameConverters": ["UnderscoreCaseToPascalCase"],
    "typeScriptFileExtension": "ts",
    "tabLength": 2,
    "explicitPublicAccessor": true,
    "defaultValuesForTypes": {
        "number": "-1",
        "Date | null": "null",
        "string": "\"\""
    },
    "customTypeMappings": {
        "System.DateTime": "string",
        "Some.Custom.Type": "number"
    },
    "typeUnionsForTypes": {
        "string": ["null", "undefined"],
        "Date": ["string"]
    }
}
```

CHAPTER 6

Programmatical API

If the *Command Line Interface* doesn't meet your requirements, or you just want to generate TypeScript files from the code level, you can use the TypeGen file generator class, located in TypeGen.Core assembly.

6.1 The Generator class

After installing TypeGen as a NuGet package, you'll have access to the *Generator* (*TypeGen.Core.Generator*) class, which is responsible for generating TypeScript files. This class is also used by *TypeGen CLI*.

6.1.1 Generator.Generate()

To generate TypeScript files, the *Generate()* method is used. All its overloads except for *Generate(GenerationSpec)* generate TypeScript files based on the used attributes (from the *TypeGen.Core.TypeAnnotations* namespace). The *Generate(GenerationSpec)* takes all information from the generation spec and ignores any attributes by default.

6.1.2 Events

There is currently one event in the *Generator* class you can subscribe to - the *Generator.FileContentGenerated* event. This event fires whenever a TypeScript file content is generated. The event handler for this event takes an argument of type *TypeGen.Core.FileContentGeneratedArgs*, which contains the following properties (event data):

- *Type : Type* - the type for which the file was generated (can be null if the file has not been generated based on a type)
- *FilePath : string* - the generated file's path
- *FileContent : string* - the generated file content

The default handler for this event saves the generated file content to the file system. There is a possibility to unsubscribe the default event handler (by using the *Generator.UnsubscribeDefaultFileContentGeneratedHandler()* method), which will cause the generated file content to not be saved in the file system. You can also re-subscribe the default handler by using the *Generator.SubscribeDefaultFileContentGeneratedHandler()* method.

6.1.3 Logging

The *Generator* class logs messages using the *Generator.Logger* instance. You can either provide your custom implementation of the *TypeGen.Core.Business.ILogger* interface, or you can use the built-in *TypeGen.Core.Business.ConsoleLogger* for logging purposes. *Generator.Logger* is null by default, or you can also set it to null - in this case, no messages will be logged.

6.2 Generator options

The *Generator* class also has a public property named *Options*, which holds the file generation options. The options can be assigned by setting the *Generator.Options* property to an instance of *GeneratorOptions* class, or by modifying the existing *Generator.Options* instance.

The current list of options available in *GeneratorOptions* is provided in API Reference | GeneratorOptions.

6.3 Example

An example of programmatical usage is shown below:

```
var options = new GeneratorOptions { BaseOutputDirectory = @"C:\my\output" }; //␣
↪create the options object
var generator = new Generator { Options = options }; // create the generator instance
var assembly = Assembly.GetCallingAssembly(); // get the assembly to generate files␣
↪for
generator.Generate(assembly); // generates the files
```

CHAPTER 7

Converters

Converters allow for converting C# names to TypeScript names, by defining conversion rules between naming conventions.

A Converter is a class that defines logic for switching from one naming convention to another. There are 2 types of converters in TypeGen: *member name converters* and *type name converters*. *Type name converters* can convert names depending on the currently generated C# type (Type instance) and *member name converters* can convert names depending on the currently generated C# member (MemberInfo instance). All *member name converters* implement the *IMemberNameConverter* interface, and all *type name converters* implement *ITypeNameConverter*.

All converters available out-of-the-box in TypeGen are both *member name converters* and *type name converters* (implement both interfaces). The natively available converters are:

- *PascalCaseToCamelCaseConverter* - converts PascalCase names to camelCase names
- *PascalCaseToKebabCaseConverter* - converts PascalCase names to kebab-case names
- *UnderscoreCaseToCamelCaseConverter* - converts underscore_case (or UNDERSCORE_CASE) names to camelCase names
- *UnderscoreCaseToPascalCaseConverter* - converts underscore_case (or UNDERSCORE_CASE) names to PascalCase names

7.1 Converter collections

Converter collections (or chains) are used in TypeGen to perform a conversion between naming conventions. There are 2 types of converter collections: *member name converter collection* (for *member name converters*) and *type name converter collection* (for *type name converters*). A converter collection defines a chain of converters that will be used to convert names.

For example, a converter collection containing two converters (in this order): **UnderscoreCaseToPascalCaseConverter** and **StripDtoConverter** (which removes the "Dto" suffix), will convert the name **MY_CLASS_DTO** in the following way:

1. First, **UnderscoreCaseToPascalCaseConverter** will convert **MY_CLASS_DTO** to **MyClassDto**

2. Then, the second converter (**StripDtoConverter**) will be used to convert **MyClassDto** to **MyClass**

Therefore, this particular converter collection will convert a name **MY_CLASS_DTO** to **MyClass**.

7.2 Writing your own converter - example

7.2.1 Type name and member name converter

Let's say you want the generated TypeScript files' names to be *kebab-case*, but without the trailing *DTO*, which is present in your C# class names (let's say you have e.g. a C# class called *ProductDTO*).

One way to do this is to combine the natively available *PascalCaseToKebabCase* converter with some custom converter that strips the *DTO* suffix from the C# class name (if the class name ends with *DTO*). Such converter could be implemented as both *member name* converter and *type name* converter, since it doesn't depend on the generated type (it could be reused for property name conversion).

Here's one implementation of such converter:

```
public class StripDtoConverter : IMemberNameConverter, ITypeNameConverter
{
    public string Convert(string name, MemberInfo memberInfo)
    {
        return ConvertName(name);
    }

    public string Convert(string name, Type type)
    {
        return ConvertName(name);
    }

    public string ConvertName(string name)
    {
        return name.ToUpperInvariant().EndsWith("DTO") ? name.Substring(0, name.
↪Length - 3) : name;
    }
}
```

Now you can create a converter chain that first strips the *DTO* from the C# class name, and later converts the DTO-less name to kebab-case:

- in CLI: specify *["StripDto", "PascalCaseToKebabCase"]* for the *fileNameConverters* parameter in your *tgconfig.json*
- programmatically: create a converter chain: *new TypeNameConverterCollection(new StripDtoConverter(), new PascalCaseToKebabCaseConverter())* and pass it in the generator options (*FileNameConverters* property)

7.2.2 Type name converter

Sometimes you may want to make TypeScript file (or type) names dependent on the actual C# type being generated. For example, you may want the enum file names to end with *.enum.ts* or the interface file names to end with *.interface.ts*.

Let's say you're generating all C# classes as TypeScript interfaces (by annotating them with the *ExportTsInterface* attribute). In this case, to achieve the abovementioned conversion, you may write the following type name converter:

```
public class TypeSuffixConverter: ITypeNameConverter
{
    public string Convert(string name, Type type)
    {
        if (type.IsClass)
        {
            return name + ".interface";
        }

        if (type.IsEnum)
        {
            return name + ".enum";
        }

        return name;
    }
}
```

You could then combine it with the *PascalCaseToKebabCase* converter in the following way:

- in CLI: specify *["PascalCaseToKebabCase", "TypeSuffix"]* for the *fileNameConverters* parameter in your *tgconfig.json*
- programmatically: create a converter chain: *new TypeNameConverterCollection(new PascalCaseToKebabCaseConverter(), new TypeSuffixConverter())* and pass it in the generator options (*FileNameConverters* property)

7.3 Using converters

To use converters when generating your TS sources, specify a converter chain (i.e. a chain of converters identified by their class names) either in *tgconfig.json* (for *TypeGen CLI*) or in the generator options (for *programmatical usage*).

CHAPTER 8

Other features

8.1 Preserving parts of a TypeScript file

Since TypeGen 1.3, there is a possibility of preserving parts of a TypeScript file, so that they won't be overridden when regenerating the file. This can be achieved with <*custom-head*> and <*custom-body*> tags specified in the comments, as shown below:

```
//<custom-head>
import { Foo } from "../bar";
//</custom-head>

export class Product {
    netPrice: number;
    vat: number;

    //<custom-body>
    get price(): number {
        return this.netPrice + this.vat;
    }
    //</custom-body>
}
```

This will keep both the extra import and the *price* property intact upon file regeneration.

Multiple code fragments can be tagged with <*custom-head*> or <*custom-body*> as well (however, multiple blocks are merged into one upon file generation):

```
export class Product {
    netPrice: number;
    vat: number;

    //<custom-body>
    get price(): number {
        return this.netPrice + this.vat;
```

(continues on next page)

(continued from previous page)

```
    }
    //</custom-body>

    //<custom-body>
    get priceSpecified(): boolean {
        return this.netPrice !== undefined;
    }
    //</custom-body>
}
```

Note: `//<custom-xyz>` (followed by a new line) is the only acceptable format of a tag. The following will not work:

- `// <custom-xyz>`
- `//<custom-xyz >`
- `//<custom-xyz> some comments after this`

<*custom-head*> and <*custom-body*> tags are case-insensitive, so e.g. `<CUSTOM-HEAD>` is also acceptable.

Deprecation note:

<*custom-head*> and <*custom-body*> tags are available since TypeGen 1.4.0. Prior to this version, the <*keep-ts*> tag was used for preserving parts of the type's definition. The <*keep-ts*> tag will still work in versions higher than 1.3.0 (unless decided otherwise), however its use is deprecated since version 1.4.0.

8.2 Strict null checking

You can specify how C# nullable property/field types will be translated to TypeScript by default, by using the *csNullableTranslation* parameter (CLI or generator options). Available choices are:

- type | null
- type | undefined
- type | null | undefined
- type (not null and not undefined)

To override the default C# nullable types translation or to specify a type union for non-nullable types, you can use the following attributes on a property or field: *TsNull*, *TsNotNull*, *TsUndefined*, *TsNotUndefined* (you can find more information on these attributes in the *TypeGen attributes* section).

8.3 Adding files to a .NET framework project

This feature is only available in TypeGen CLI (not available in programmatical API, as the programmatical API doesn't operate on a per-project level). When this option is selected, generated TS sources will automatically be added to a .NET Framework project in the chosen project folder. This feature is only available for .NET Framework projects (not .NET Core), because .NET Core projects don't specify included files in the project file.

8.4 Creating TypeScript barrel files

Barrels are created by adding them to a *generation spec* (please visit the generation spec section for the details on how to add barrels). There is also a possibility to add a "global" barrel file by enabling the *createIndexFile* config option, however this option offers very limited functionality; also it's deprecated and will be removed in the future.

8.5 Default values for TS properties based on the type

It is possible to generate default values for properties inside TS classes/interfaces, depending on the property type. The [TS type -> default value] mappings can be specified either in the *defaultValuesForTypes* CLI parameter or in the *GeneratorOptions.DefaultValuesForTypes* property.

8.6 Custom type mappings

TypeGen allows to override its default C# to TS type mappings or create new custom mappings. The way to define mappings is via either the *customTypeMappings* CLI parameter or the *GeneratorOptions.CustomTypeMappings* property.

8.7 Default exports

There is a possibility to use default exports instead of named exports when generating TypeScript types. This feature can be enabled/disabled globally, by setting the *useDefaultExport* CLI parameter or the *GeneratorOptions.UseDefaultExport* property, or a one-time opt-in/out can be used with either the *TsDefaultExportAttribute* or the *DefaultExport(bool)* generation spec method.

CHAPTER 9

Troubleshooting

9.1 TypeGen.Core is not compatible with your .NET Core project

It's most likely a NuGet issue, which can be resolved by clearing the NuGet cache:

```
./nuget.exe locals -clear all
```

CHAPTER 10

Legal

TypeGen is published under the MIT license. To view the full license content, see TypeGen license on GitHub.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search