Announcing .NET 5.0 RC 1

Richard

September 13th, 2020

Today, we are shipping .NET 5.0 Release Candidate 1 (RC1). It is a near-final release of .NET 5.0, and the first of two RCs before the official release in November. RC1 is a "go live" release; you are supported using it in production. At this point, we're looking for reports of any remaining critical bugs that should be fixed before the final release. We need your feedback to get .NET 5.0 across the finish line.

We also released RC1 versions of ASP.NET Core and EF Core today.

You can download .NET 5.0, for Windows, macOS, and Linux:

- Installers and binaries
- Container images
- Snap installer
- Release notes
- Known issues
- GitHub issue tracker

You need the latest preview version of Visual Studio (including Visual Studio for Mac) to use .NET 5.0.

.NET 5.0 includes many improvements, notably single file applications, smaller container images, more capable JsonSerializer APIs, a complete set of nullable reference type annotations, new target framework names, and support for Windows ARM64. Performance has been greatly improved, in the NET libraries, in the GC, and the JIT. ARM64 was a key focus for performance investment, resulting in much better throughput and smaller binaries. .NET 5.0 includes new language versions, C# 9 and F# 5.0.

We recently published a few deep-dive posts about new capabilities in 5.0 that you may want to check out:

- F# 5 update for August
- ARM64 Performance in .NET 5
- Improvements in native code interop in .NET 5.0
- Introducing the Half type!
- App Trimming in .NET 5
- Customizing Trimming in .NET 5
- Automatically find latent bugs in your code with .NET 5

Just like I did for .NET 5.0 Preview 8 I've chosen a selection of features to look at in more depth and to give you a sense of how you'll use them in real-world usage. This post is dedicated to records in C# 9 and `System.Text.Json.JsonSerializer`. They are separate features, but also a nice pairing, particularly if you spend a lot of time crafting POCO types for deserialized JSON objects.

C# 9 — Records

Records are perhaps the most important new feature in C# 9. They offer a broad feature set (for a language type kind), some of which requires RC1 or later (like `record.ToString()`).

The easiest way to think of records is as immutable classes. Feature-wise, they are closest to tuples. One can think of them as custom tuples with properties and immutability. There are likely many cases where tuples are used today that would be better served by records.

If you are using C#, you will get the best experience if you are using named types (as opposed to a feature like tuples). Static typing is the primary design point of the language. Records make it easier to use small types, and take advantage of type safety throughout your app.

Records are immutable data types

Records enable you to create immutable data types. This is great for defining types that store small amounts of data.

The following is an example of a record. It stores user information from a login screen.

```
public record LoginResource(string Username, string Password, bool
RememberMe);
```

It is semantically similar (almost identical) to the following class. I'll cover the differences shortly.

```
public class LoginResource
{
    public LoginResource(string username, string password, bool rememberMe)
    {
        Username = username;
        Password = password;
        RememberMe = rememberMe;
    }

    public string Username { get; init; }
    public string Password { get; init; }
    public bool RememberMe { get; init; }
}
```

`init` is a new keyword that is an alternative to `set`. `set` allows you to assign to a property at any time. `init` allows you to assign to a property only during object construction. It's the building block that records rely on for immutability. Any type can use `init`. It isn't specific to records, as you can see in the previous class definition.

`private set` might seem similar to `init`; `private set` prevents other code (outside the type) from mutating data. `init` will generate compiler errors when a type mutates a property accidentally (after construction). `private set` isn't intended to model immutable data, so doesn't generate any compiler errors or warnings when the type mutates a property value after construction.

Records are specialized classes

As I just covered, the record and the class variants of `LoginResource` are almost identical. The class definition is a semantically identical *subset* of the record. The record provides more, specialized, behavior.

Just so we're on the same page, the following comparison is between a `record`, and a `class` that uses `init` instead of `set` for properties, as demonstrated earlier.

What's the same?

- Construction
- Immutability
- Copy semantics (records are classes under the hood)

What's different?

- Record equality is based on content. Class equality based on object identity.
- Records provide a `GetHashCode()` implementation that is based on record content.
- Records provide an `IEquatable<T>` implementation. It uses the unique `GetHashCode()` behavior as the mechanism to provide the content-based equality semantic for records.
- Record ToString() is overridden to print record content.

The differences between a record and a class (using `init`) can be seen in the disassembly for LoginResource as a record and LoginResource as a class.

I'll show you some code that demonstrates these differences.

```
using System;
using System.Linq;
using static System.Console;

var user = "Lion-O";
var password = "jaga";
var rememberMe = true;
LoginResourceRecord lrr1 = new(user, password, rememberMe);
var lrr2 = new LoginResourceRecord(user, password, rememberMe);
var lrc1 = new LoginResourceClass(user, password, rememberMe);
var lrc2 = new LoginResourceClass(user, password, rememberMe);

WriteLine($"Test record equality -- lrr1 == lrr2 : {lrr1 == lrr2}");
WriteLine($"Test class equality  -- lrc1 == lrc2 : {lrc1 == lrc2}");
WriteLine($"Print lrr1 hash code -- lrr1.GetHashCode(): {lrr1.GetHashCode()}");
WriteLine($"Print lrr2 hash code -- lrr2.GetHashCode(): {lrr2.GetHashCode()}");
WriteLine($"Print lrc1 hash code -- lrc1.GetHashCode(): {lrc1.GetHashCode()}");
WriteLine($"Print lrc2 hash code -- lrc2.GetHashCode(): {lrc2.GetHashCode()}");
WriteLine($"{nameof(LoginResourceRecord)} implements IEquatable<T>: {lrr1 is IEquatable<Login
WriteLine($"{nameof(LoginResourceClass)}  implements IEquatable<T>: {lrr1 is IEquatable<Login
WriteLine($"Print {nameof(LoginResourceRecord)}.ToString -- lrr1.ToString(): {lrr1.ToString()
WriteLine($"Print {nameof(LoginResourceClass)}.ToString  -- lrc1.ToString(): {lrc1.ToString()

public record LoginResourceRecord(string Username, string Password, bool RememberMe);

public class LoginResourceClass
{
    public LoginResourceClass(string username, string password, bool rememberMe)
    {
        Username = username;
        Password = password;
        RememberMe = rememberMe;
    }

    public string Username { get; init; }
    public string Password { get; init; }
    public bool RememberMe { get; init; }
}
```

Program.cs hosted with ♥ by GitHub view raw

Note: You will notice that the `LoginResource` types end in `Record` and `Class`. That pattern is not the indication of a new naming pattern. They are only named that way so that there can be a record and class variant of the same type in the sample. Please don't name your types that way.

This code produces the following output.

```
rich@thundera records % dotnet run
Test record equality -- lrr1 == lrr2 : True
Test class equality  -- lrc1 == lrc2 : False
Print lrr1 hash code -- lrr1.GetHashCode(): -542976961
Print lrr2 hash code -- lrr2.GetHashCode(): -542976961
Print lrc1 hash code -- lrc1.GetHashCode(): 54267293
Print lrc2 hash code -- lrc2.GetHashCode(): 18643596
LoginResourceRecord implements IEquatable<T>: True
LoginResourceClass  implements IEquatable<T>: False
Print LoginResourceRecord.ToString -- lrr1.ToString(): LoginResourceRecord {
Username = Lion-O, Password = jaga, RememberMe = True }
Print LoginResourceClass.ToString -- lrc1.ToString(): LoginResourceClass
```

Record syntax

There are multiple patterns for declaring records that cater to different use cases. After playing with each one, you start to get a feel for the benefits of each pattern. You'll also see that they are not distinct syntax but a continuum of options.

The first pattern is the simplest one — a one liner — but offers the least flexibility. It's good for records with a small number of required properties.

Here is the LoginResource record, shown earlier, as an example of this pattern. That's it. That one line is the entire definition.

```
public record LoginResource(string Username, string Password, bool
RememberMe);
```

Construction follows the requirements of a constructor with parameters (including the allowance for optional parameters).

```
var login = new LoginResource("Lion-O", "jaga", true);
```

You can also use target typing if you prefer.

```
LoginResource login = new("Lion-O", "jaga", true);
```

The next syntax makes all the properties optional. There is an implicit parameterless constructor provided for the record.

```
public record LoginResource
{
    public string Username {get; init;}
    public string Password {get; init;}
    public bool RememberMe {get; init;}
}
```

Construction uses object initializers and could look like the following:

```
LoginResource login = new()
{
    Username = "Lion-O",
    TemperatureC = "jaga"
};
```

Maybe you want to make those two properties required, with the other one optional. This last pattern would look like the following.

```
public record LoginResource(string Username, string Password)
{
    public bool RememberMe {get; init;}
}
```

Construction could look like the following, with `RememberMe` unspecified.

```
LoginResource login = new("Lion-O", "jaga");
```

And with `RememberMe` specified.

```
LoginResource login = new("Lion-O", "jaga")
{
    RememberMe = true
};
```

I want to make sure that you don't think that records are exclusively for immutable data. You can opt into exposing mutable properties, as you can see in the following example that reports information about batteries. `Model` and `TotalCapacityAmpHours` properties are immutable and `RemainingCapacityPercentange` is mutable.

```
using System;

Battery battery = new Battery("CR2032", 0.235)
{
    RemainingCapacityPercentage = 100
};

Console.WriteLine (battery);

for (int i = battery.RemainingCapacityPercentage; i >= 0; i--)
{
    battery.RemainingCapacityPercentage = i;
}

Console.WriteLine (battery);

public record Battery(string Model, double TotalCapacityAmpHours)
{
    public int RemainingCapacityPercentage {get;set;}
}
```

Program.cs hosted with ♥ by GitHub view raw

It produces the following output.

Produces the following output:

```
rich@thundera recordmutable % dotnet run
Battery { Model = CR2032, TotalCapacityAmpHours = 0.235,
RemainingCapacityPercentage = 100 }
Battery { Model = CR2032, TotalCapacityAmpHours = 0.235,
RemainingCapacityPercentage = 0 }
```

Non-destructive record mutation

Immutability provides significant benefits, but you will quickly find a case where you need to mutate a record. How can you do that without giving up on immutability? The `with` expression satisfies this need. It enables creating a new record in terms of an existing record of the same type. You can specify the new values that you want to be different, and all other properties are copied from the existing record.

Let's transform the username to lower-case. That's how usernames are stored in our pretend user database. However, the original username casing is required for diagnostic purposes. It could look like the following, assuming the code from the previous example:

```
LoginResource login = new("Lion-O", "jaga", true);
LoginResource loginLowercased = login with {Username =
login.Username.ToLowerInvariant()};
```

The `login` record hasn't been changed. In fact, that's impossible. The transformation has only affected `loginLowercased`. Other than the lowercase transformation to `loginLowercased`, it's identical to `login`.

We can check that `with` has done what we expect using the built-in `ToString()` override.

```
Console.WriteLine(login);
Console.WriteLine(loginLowercased);
```

This code produces the following output.

```
LoginResource { Username = Lion-O, Password = jaga, RememberMe = True }
LoginResource { Username = lion-o, Password = jaga, RememberMe = True }
```

We can go one step further with understanding how `with` works. It copies all values from one record to the other. This isn't a delegation model where one record depends on another. In fact, after the `with` operation completes, there is no relationship between the two records. `with` only has meaning for record construction. That means for reference types, the copy is just a copy of the reference. For value types, the value is copied.

You can see that semantic at play with the following code.

```
Console.WriteLine($"Record equality: {login == loginLowercased}");
Console.WriteLine($"Property equality: Username == {login.Username ==
loginLowercased.Username}; Password == {login.Password ==
loginLowercased.Password}; RememberMe == {login.RememberMe ==
loginLowercased.RememberMe}");
```

It produces the following output.

```
Record equality: False
Property equality: Username == False; Password == True; RememberMe == True
```

Record inheritance

It's easy to extend a record. Let's assume a new `LastLoggedIn` property. It could be added directly to `LoginResource`. That's a fine idea. Records are not brittle like interfaces traditionally have been, unless you want to make new properties required constructor parameters.

In this case, I want to make `LastLogin` required. Imagine the codebase is large, and it would be expensive to sprinkle knowledge of the `LastLoggedIn` property in all the places where a `LoginResource` is created. Instead, we're going to create a new record that extends `LoginResource` with this new property. Existing code will work in terms of `LoginResource` and new code will work in terms of a new record that can then assume that the `LastLoggedIn` property has been populated. Code that accepts a `LoginResource` will happily accept the new record, by virtue of regular inheritance rules.

This new record could be based on any of the `LoginResource` variants demonstrated earlier. It will be based on the following one.

```
public record LoginResource(string Username, string Password)
{
    public bool RememberMe {get; init;}
}
```

The new record could look like the following.

```
public record LoginWithUserDataResource(string Username, string Password,
DateTime LastLoggedIn) : LoginResource(Username, Password)
{
    public int DiscountTier {get; init};
    public bool FreeShipping {get; init};
}
```

I've made `LastLoggedIn` a required property, and taken the opportunity to add additional, optional, properties that may or may not be set. The optional `RememberMe` property is also defined, by virtue of extending the `LoginResource` record.

Modeling record construction helpers

One of the patterns that isn't necessarily intuitive is modeling helpers that you want to use as part of record construction. Let's switch examples, to weight measurements. Weight measurements come from an internet-connected scale. The weight is specified in Kilograms, however, there are some cases where the weight needs to be provided in pounds.

The following record declaration could be used.

```
public record WeightMeasurement(DateTime Date, double Kilograms)
{
    public double Pounds {get; init;}
    public static double GetPounds(double kilograms) => kilograms * 2.20462262
;
}
```

This is what construction would look like.

```
var weight = 200;
WeightMeasurement measurement = new(DateTime.Now, weight)
{
    Pounds = WeightMeasurement.GetPounds(weight)
};
```

In this example, it is necessary to specify the weight as a local. It isn't possible to access the `Kilograms` property within an object initializer. It is also necessary to define `GetPounds` as a static method. It isn't possible to call instance methods (for the type being constructed) within an object initializer.

Records and Nullability

You get nullability for free with records, right? Everything is immutable, so where would the nulls come from? Not quite. An immutable property can be null and will always be null in that case.

Let's look at another program without nullability enabled.

```
using System;
using System.Collections.Generic;

Author author = new(null, null);

Console.WriteLine(author.Name.ToString());

public record Author(string Name, List<Book> Books)
{
    public string Website {get; init;}
    public string Genre {get; init;}
    public List<Author> RelatedAuthors {get; init;}
}

public record Book(string name, int Published, Author author);
```

This program compiles and will throw a `NullReference` exception, due to dereferencing `author.Name`, which is `null`.

To further drive home this point, the following will not compile. `author.Name` is initialized as `null` and then cannot be changed, since the property is immutable.

```
Author author = new(null, null);
author.Name = "Colin Meloy";
```

I'm going to update my project file to enable nullability.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <LangVersion>preview</LangVersion>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

I'm now seeing a bunch of warnings like the following.

```
/Users/rich/recordsnullability/Program.cs(8,21): warning CS8618: Non-nullable
property 'Website' must contain a non-null value when exiting constructor.
Consider declaring the property as nullable.
[/Users/rich/recordsnullability/recordsnullability.csproj]
```

I updated the `Author` record with null annotations that describe my intended use of the record.

```
public record Author(string Name, List<Book> Books)
{
    public string? Website {get; init;}
    public string? Genre {get; init;}
    public List<Author>? RelatedAuthors {get; init;}
}
```

I'm still getting warnings for the `null, null` construction of `Author` seen earlier.

```
/Users/rich/recordsnullability/Program.cs(5,21): warning CS8625: Cannot
convert null literal to non-nullable reference type.
[/Users/rich/recordsnullability/recordsnullability.csproj]
```

That's good, since that's a scenario I want to protect against. I'll now show you an updated variant of the program that plays nicely with and enjoys the benefits of nullability.

```
using System;
using System.Collections.Generic;
using System.Diagnostics.CodeAnalysis;



Author lord = new Author("Karen Lord")
{
    Website = "https://karenlord.wordpress.com/",
    RelatedAuthors = new()
};

lord.Books.AddRange(
    new Book[]
    {
        new Book("The Best of All Possible Worlds", 2013, lord),
```

```
        new Book("The Galaxy Game", 2015, lord)
    }
);

lord.RelatedAuthors.AddRange(
    new Author[]
    {
        new ("Nalo Hopkinson"),
        new ("Ursula K. Le Guin"),
        new ("Orson Scott Card"),
        new ("Patrick Rothfuss")
    }
);

Console.WriteLine($"Author: {lord.Name}");
Console.WriteLine($"Books: {lord.Books.Count}");
Console.WriteLine($"Related authors: {lord.RelatedAuthors.Count}");


public record Author(string Name)
{
    private List<Book> _books = new();

    public List<Book> Books => _books;

    public string? Website {get; init;}
    public string? Genre {get; init;}
    public List<Author>? RelatedAuthors {get; init;}
}

public record Book(string name, int Published, Author author);
```

Program.cs hosted with ❤ by GitHub view raw

This program compiles without nullable warnings.

You might be wondering about the following line:

```
lord.RelatedAuthors.AddRange(
```

`Author.RelatedAuthors` can be null. The compiler can see that the `RelatedAuthors` property is set just a few lines earlier, so it knows that `RelatedAuthors` reference will be non-null.

However, imagine the program instead looked like the following.

```
Author GetAuthor()
{
    return new Author("Karen Lord")
    {
        Website = "https://karenlord.wordpress.com/",
        RelatedAuthors = new()
    };
}

Author lord = GetAuthor();
```

The compiler doesn't have the flow analysis smarts to know that `RelatedAuthors` will be non-null when type construction is within a separate method. In that case, one of two following patterns would be needed.

```
lord.RelatedAuthors!.AddRange(
```

or

```
if (lord.RelatedAuthors is object)
{
    lord.RelatedAuthors.AddRange( ...
}
```

This is a long demonstration of records nullability just to say that it doesn't change anything about the experience of using nullable reference types.

Separately, you may have noticed that I moved the `Books` property on the `Author` record to be an initialized get-only property, instead of being a required parameter in the record constructor. This was driven by there being a circular relationship between `Author` and `Books`. Immutability and circular references can cause headaches. It is OK in this case, and just means that all `Author` objects need to be created before `Book` objects. As a result, it isn't possible to provide a fully initialized set of `Book` objects as part of `Author` construction. The best we could ever expect as part of `Author` construction is an empty `List<Book>`. As a result, initializing an empty `List<Book>` as part of `Author` construction seem like the best choice. There is no rule that all of these properties need to be `init` style. I've chosen to do that to demonstrate the behavior when you do.

We're about to transition to talk about JSON serialization. This example, with circular references, relates to the **Preserving references in JSON object graphs** section coming shortly. `JsonSerializer` supports object graphs with circular references, but not with types with parameterized constructors. You can serialize the `Author` object to JSON, but not back to an `Author` object as it is currently defined. If `Author` wasn't a `record` or didn't have circular references, then both serialization and deserialization would work with `JsonSerializer`.

System.Text.Json

`System.Text.Json` has been significantly improved in .NET 5.0 to improve performance, reliability, and to make it easier for people to adopt that are familiar with Newtonsoft.Json. It also includes support for deserializing JSON objects to records, the new C# feature covered earlier in this post.

If you are looking at using `System.Text.Json` as an alternative to `Newtonsoft.Json`, you should check out the migration guide. The guide clarifies the relationship between these two APIs. `System.Text.Json` is intended to cover many of the same scenarios as `Newtonsoft.Json`, but it's not intended to be a drop-in replacement for or achieve feature parity with the popular JSON library. We try to maintain a balance between performance and usability, and bias to performance in our design choices.

`HttpClient` extension methods

`JsonSerializer` extension methods are now exposed on `HttpClient` and greatly simplify using these two APIs together. These extension methods remove complexity and take care of a variety of scenarios for you, including handling the content stream and validating the content media type. Steve Gordon does a great job of explaining the benefits in Sending and receiving JSON using HttpClient with System.Net.Http.Json.

The following example deserializes weather forecast JSON data into a `Forecast` record, using the new `GetFromJsonAsync<T>()` extension method.

```
using System;
using System.Net.Http;
using System.Net.Http.Json;

string serviceURL = "https://localhost:5001/WeatherForecast";
HttpClient client = new();
Forecast[] forecasts = await client.GetFromJsonAsync<Forecast[]>(serviceURL);

foreach(Forecast forecast in forecasts)
{
    Console.WriteLine($"{forecast.Date}; {forecast.TemperatureC}C; {forecast.Summary}");
}
```

```
13
14  // {"date":"2020-09-06T11:31:01.923395-07:00","temperatureC":-1,"temperatureF":31,"summary":"
15  public record Forecast(DateTime Date, int TemperatureC, int TemperatureF, string Summary);
```

Program.cs hosted with ♥ by GitHub view raw

This code is compact! It is relying on top-level programs and records from C# 9 and the new `GetFromJsonAsync<T>()` extension method. The use of `foreach` and `await` in such close proximity might be make you wonder if we're going to add support for streaming JSON objects. Yes, in a future release.

You can try this on your own machine. The following .NET SDK commands will create a weather forecast service using the WebAPI template. It will expose the service at the following URL by default: `https://localhost:5001/WeatherForecast`. This is the same URL used in the sample.

```
rich@thundera ~ % dotnet new webapi -o webapi
rich@thundera ~ % cd webapi
rich@thundera webapi % dotnet run
```

Make sure you've run `dotnet dev-certs https --trust` first or the handshake between client and server won't work. If you're having trouble, see Trust the ASP.NET Core HTTPS development certificate.

You can then run the previous sample.

```
rich@thundera ~ % git clone
https://gist.github.com/3b41d7496f2d8533b2d88896bd31e764.git weather-forecast
rich@thundera ~ % cd weather-forecast
rich@thundera weather-forecast % dotnet run
9/9/2020 12:09:19 PM; 24C; Chilly
9/10/2020 12:09:19 PM; 54C; Mild
9/11/2020 12:09:19 PM; -2C; Hot
9/12/2020 12:09:19 PM; 24C; Cool
9/13/2020 12:09:19 PM; 45C; Balmy
```

Improved support for immutable types

There are multiple patterns for defining immutable types. Records are just the newest one. `JsonSerializer` now has support for immutable types.

In this example, you'll see the serialization with an immutable struct.

```
1  using System;
2  using System.Text.Json;
3  using System.Text.Json.Serialization;
4
5  var json = "{\"date\":\"2020-09-06T11:31:01.923395-07:00\",\"temperatureC\":-1,\"temperatureF
6  var options = new JsonSerializerOptions()
7  {
8      PropertyNameCaseInsensitive = true,
9      IncludeFields = true,
10     PropertyNamingPolicy = JsonNamingPolicy.CamelCase
11 };
12 var forecast = JsonSerializer.Deserialize<Forecast>(json, options);
13
14 Console.WriteLine(forecast.Date);
15 Console.WriteLine(forecast.TemperatureC);
16 Console.WriteLine(forecast.TemperatureF);
17 Console.WriteLine(forecast.Summary);
18
19 var roundTrippedJson = JsonSerializer.Serialize<Forecast>(forecast, options);
20
21 Console.WriteLine(roundTrippedJson);
22
```

```
public struct Forecast{
    public DateTime Date {get;}
    public int TemperatureC {get;}
    public int TemperatureF {get;}
    public string Summary {get;}
    [JsonConstructor]
    public Forecast(DateTime date, int temperatureC, int temperatureF, string summary) => (Da
}
```

Program.cs hosted with ❤ by GitHub view raw

Note: The `JsonConstructor` attribute is required to specify the constructor to use with structs. With classes, if there is only a single constructor, then the attribute is not required. Same with records.

It produces the following output:

```
rich@thundera jsonserializerimmutabletypes % dotnet run
9/6/2020 11:31:01 AM
-1
31
Scorching
{"date":"2020-09-06T11:31:01.923395-07:00","temperatureC":-1,"temperatureF":31,"summary":"Scorching"}
```

Support for records

`JsonSerializer` support for records is almost the same as what I just showed you for immutable types. The difference I want to show here is deserializing a JSON object to a record that exposes a parameterized constructor and an optional init property.

Here's the program, including the record definition:

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

Forecast forecast = new(DateTime.Now, 40)
{
    Summary = "Hot!"
};

string forecastJson = JsonSerializer.Serialize<Forecast>(forecast);
Console.WriteLine(forecastJson);
Forecast? forecastObj = JsonSerializer.Deserialize<Forecast>(forecastJson);
Console.Write(forecastObj);

public record Forecast (DateTime Date, int TemperatureC)
{
    public string? Summary {get; init;}
};
```

Program.cs hosted with ❤ by GitHub view raw

It produces the following output:

```
rich@thundera jsonserializerrecords % dotnet run
{"Date":"2020-09-12T18:24:47.053821-07:00","TemperatureC":40,"Summary":"Hot!"}
Forecast { Date = 9/12/2020 6:24:47 PM, TemperatureC = 40, Summary = Hot! }
```

Improved `Dictionary<K,V>` support

`JsonSerializer` now supports dictionaries with non-string keys. You can see what this looks like in the following sample. With .NET Core 3.0, this code compiles but throws a `NotSupportedException`.

```cs
using System;
using System.Collections.Generic;
using System.Text.Json;

Dictionary<int, string> numbers = new ()
{
    {0, "zero"},
    {1, "one"},
    {2, "two"},
    {3, "three"},
    {5, "five"},
    {8, "eight"},
    {13, "thirteen"},
    {21, "twenty one"},
    {34, "thirty four"},
    {55, "fifty five"},
};

var json = JsonSerializer.Serialize<Dictionary<int, string>>(numbers);

Console.WriteLine(json);

var dictionary = JsonSerializer.Deserialize<Dictionary<int, string>>(json);

Console.WriteLine(dictionary[55]);
```

Program.cs hosted with ❤ by GitHub view raw

It produces the following output.

```
rich@thundera jsondictionarykeys % dotnet run
{"0":"zero","1":"one","2":"two","3":"three","5":"five","8":"eight","13":"thirt
een","21":"twenty one","34":"thirty four","55":"fifty five"}
fifty five
```

Support for fields

`JsonSerializer` now supports fields. This change was contributed by @YohDeadfall. Thanks!

You can see what this looks like in the following sample. With .NET Core 3.0, `JsonSerializer` fails to serialize or deserialize with types that use fields. This is a problem for existing types that have fields and cannot be changed. With this change, that's no longer an issue.

```cs
using System;
using System.Text.Json;

var json = "{\"date\":\"2020-09-06T11:31:01.923395-07:00\",\"temperatureC\":-1,\"temperatureF
var options = new JsonSerializerOptions()
{
    PropertyNameCaseInsensitive = true,
    IncludeFields = true,
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
};
var forecast = JsonSerializer.Deserialize<Forecast>(json, options);

Console.WriteLine(forecast.Date);
Console.WriteLine(forecast.TemperatureC);
Console.WriteLine(forecast.TemperatureF);
Console.WriteLine(forecast.Summary);

var roundTrippedJson = JsonSerializer.Serialize<Forecast>(forecast, options);

Console.WriteLine(roundTrippedJson);
```

```
21
22  public class Forecast{
23      public DateTime Date;
24      public int TemperatureC;
25      public int TemperatureF;
26      public string Summary;
27  }
```

Program.cs hosted with ❤ by GitHub view raw

Produces the following output:

```
rich@thundera jsonserializerfields % dotnet run
9/6/2020 11:31:01 AM
-1
31
Scorching
{"date":"2020-09-06T11:31:01.923395-
07:00","temperatureC":-1,"temperatureF":31,"summary":"Scorching"}
```

Preserving references in JSON object graphs

`JsonSerializer` has added support for preserving (circular) references within JSON object graphs. It does this by storing IDs that can be reconstituted when a JSON string is deserialized back to objects.

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text.Json;
4   using System.Text.Json.Serialization;
5
6   Employee janeEmployee = new()
7   {
8       Name = "Jane Doe",
9       YearsEmployed = 10
10  };
11
12  Employee johnEmployee = new()
13  {
14      Name = "John Smith"
15  };
16
17  janeEmployee.Reports = new List<Employee> { johnEmployee };
18  johnEmployee.Manager = janeEmployee;
19
20  JsonSerializerOptions options = new()
21  {
22      // NEW: globally ignore default values when writing null or default
23      DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault,
24      // NEW: globally allow reading and writing numbers as JSON strings
25      NumberHandling = JsonNumberHandling.AllowReadingFromString | JsonNumberHandling.WriteAsSt
26      // NEW: globally support preserving object references when (de)serializing
27      ReferenceHandler = ReferenceHandler.Preserve,
28      IncludeFields = true, // NEW: globally include fields for (de)serialization
29      WriteIndented = true,};
30
31  string serialized = JsonSerializer.Serialize(janeEmployee, options);
32  Console.WriteLine($"Jane serialized: {serialized}");
33
34  Employee janeDeserialized = JsonSerializer.Deserialize<Employee>(serialized, options);
35  Console.Write("Whether Jane's first report's manager is Jane: ");
36  Console.WriteLine(janeDeserialized.Reports[0].Manager == janeDeserialized);
37
38  public class Employee
39  {
```

```
40      // NEW: Allows use of non-public property accessor.
41      // Can also be used to include fields "per-field", rather than globally with JsonSerializ
42      [JsonInclude]
43      public string Name { get; internal set; }
44
45      public Employee Manager { get; set; }
46
47      public List<Employee> Reports;
48
49      public int YearsEmployed { get; set; }
50
51      // NEW: Always include when (de)serializing regardless of global options
52      [JsonIgnore(Condition = JsonIgnoreCondition.Never)]
53      public bool IsManager => Reports?.Count > 0;
54  }
```

Program.cs hosted with ♥ by GitHub view raw

Performance

`JsonSerializer` performance is significantly improved in .NET 5.0. Stephen Toub covered some `JsonSerializer` improvements in his Performance Improvements in .NET 5 post. I'll cover a few more here.

Collections (de)serialization

We made significant improvements for large collections (~1.15x-1.5x on deserialize, ~1.5x-2.4x+ on serialize). You can see these improvements characterized in much more detail dotnet/runtime #2259.

The improvements to `List<int>` (de)serialization is particularly impressive, comparing .NET 5.0 to .NET Core 3.1. Those changes are going to be show up as meaningful with high-performance apps.

Method	Mean	Error	StdDev	Median
Deserialize before	76.40 us	0.392 us	0.366 us	
After ~1.5x faster	50.05 us	0.251 us	0.235 us	
Serialize before	29.04 us	0.213 us	0.189 us	
After ~2.4x faster	12.17 us	0.205 us	0.191 us	

Property lookups — naming convention

One of the most common issues with using JSON is a mismatch of naming conventions with .NET design guidelines. JSON properties are often camelCase and .NET properties and fields are typically PascalCase. The json serializer you use is responsible for bridging between naming conventions. That doesn't come for free, at least not with .NET Core 3.1. That cost is now negligible with .NET 5.0.

The code that allows for missing properties and case insensitivity has been greatly improved in .NET 5.0. It is ~1.75x faster in some cases.

The following benchmarks for a simple 4-property test class that has property names > 7 bytes.

3.1 performance

Method	Mean	Error	StdDev	Median	Min	Max	Gen 0	Gen 1	Gen 2	Allocated
CaseSensitive_Matching	844.2 ns	4.25 ns	3.55 ns	844.2 ns	838.6 ns	850.6 ns	0.0342	-	-	224 B
CaseInsensitive_Matching	833.3 ns	3.84 ns	3.40 ns	832.6 ns	829.4 ns	841.1 ns	0.0504	-	-	328 B
CaseSensitive_NotMatching(Missing)	1,007.7 ns	9.40 ns	8.79 ns	1,005.1 ns	997.3 ns	1,023.3 ns	0.0722	-	-	464 B
CaseInsensitive_NotMatching	1,405.6 ns	8.35 ns	7.40 ns	1,405.1 ns	1,397.1 ns	1,423.6 ns	0.0626	-	-	408 B

5.0 performance

Method	Mean	Error	StdDev	Median	Min	Max	Gen 0	Gen 1	Gen 2	Allocated
CaseSensitive_Matching	799.2 ns	4.59 ns	4.29 ns	801.0 ns	790.5 ns	803.9 ns	0.0985	-	-	632 B
CaseInsensitive_Matching	789.2 ns	6.62 ns	5.53 ns	790.3 ns	776.0 ns	794.4 ns	0.1004	-	-	632 B
CaseSensitive_NotMatching(Missing)	479.9 ns	0.75 ns	0.59 ns	479.8 ns	479.1 ns	481.0 ns	0.0059	-	-	40 B
CaseInsensitive_NotMatching	783.5 ns	3.26 ns	2.89 ns	783.5 ns	779.0 ns	789.2 ns	0.1004	-	-	632 B

TechEmpower improvement

We've spent significant effort improving .NET performance on the TechEmpower benchmark. It made sense to validate these `JsonSerializer` improvements with the TechEmpower JSON benchmark. Performance is now ~ 19% better, which should improve the placement of .NET on that benchmark once we update our entries to .NET 5.0. Our goal for the release was to be more competitive with `netty`, which is a common Java webserver.

These changes, and the performance measurements are covered in detail at dotnet/runtime #37976. There are two sets of benchmarks there. The first is validating performance with the `JsonSerializer` performance benchmarks that the team maintains. There is an ~8% improvement observed. The next section is for TechEmpower. It measures three different approaches for satisfying the requirements of the TechEmpower JSON benchmark. `SerializeWithCachedBufferAndWriter` is the one we use in the official benchmark.

Method	Mean	Error	StdDev	Median
SerializeWithCachedBufferAndWriter (before)	155.3 ns	1.19 ns	1.11 ns	
SerializeWithCachedBufferAndWriter (after)	130.8 ns	1.50 ns	1.40 ns	

If we look at `Min` column, we can do some simple math to calculate the improvement: `153.3/128.6 = ~1.19`. That's a 19% improvement.

Closing

I hope you've enjoyed this deeper dive into records and `JsonSerializer`. They are just two of the many improvement in .NET 5.0. The Preview 8 post covers a larger set of features, that provides a broader view of the value that's coming in 5.0.

As you know, we're not adding any new features in .NET 5.0 at this point. I'm using these late preview and RC posts to cover all the features we've built. Which ones would you like to see me cover in the RC2 release blog post? I'd like to know what I should focus on.

Please share your experience using RC1 in the comments. Thanks to everyone that has installed .NET 5.0. We appreciate all the engagement and feedback we've received so far.

Richard Lander

Program Manager, .NET Team

Follow

Posted in .NET

Read next

Announcing Entity Framework Core (EFCore) 5.0 RC1

Announcing Entity Framework EFCore 5.0 RC1, a "go-live" supported release. This release includes new features like many-to-many, property bags, event counters, required 1:1 dependents and the ability to intercept SaveChanges and listen to save events. It also includes improvements to model-building, migrations, and more.

Jeremy Likness September 14, 2020

17 comments

The future of .NET Standard

Since .NET 5 was announced, many of you have asked what this means for .NET Standard and whether it will still be relevant. In this post, I'm going to explain how .NET 5 improves code sharing and replaces .NET Standard. I'll also cover the cases where you still need .NET Standard.

Immo Landwerth September 15, 2020

94 comments

100 comments

Log in to join the discussion.

Newer Comments →

Eaton September 14, 2020 10:30 am

Can you provide some updates on the WinForms designer please?

Log in to Reply

Alexey Leonovich September 14, 2020 11:24 am

I'm also interested in WinForms designer updates..

Log in to Reply

Nick Ac September 14, 2020 4:52 pm

Just wondering, where's the Java interop for Desktop VMs that was once mentioned when talking about .NET 5 features? Will it make it to newer versions?

Log in to Reply

Tonytins September 14, 2020 8:18 pm

I *think* that got pushed back because of COVID.

Log in to Reply

Paul Faisant September 15, 2020 9:46 am

Also interested to know when it will be fully supported

Log in to Reply

Ismail Demir September 14, 2020 12:06 pm

Any updates on VB.Net WinForms ?

Log in to Reply

Kathleen Dollard September 14, 2020 12:33 pm

We are tying up loose ends:

- The templates will be in RC2, we're still settling on some issue around HighDPI.
- Project Properties updates will be in the next Visual Studio preview.
- I think the fix to the terrible, awful Event Handler designer bug made RC1, if not, it's fixed in RC2 (where double clicking a control gives you a new handler even if one exists)

So, look for RC2! As well as blog post(s) around that time

Log in to Reply

Joseph Musser September 14, 2020 1:16 pm

The designer feature I'm trying to keep an eye out for is support for third party controls and designers. That is the factor that determines when we can begin to move our Windows Forms projects to .NET Core/5. Think it'll be near .NET 5 or more like 2021?

Log in to Reply

Olia Gavrysh September 14, 2020 4:45 pm

We are in touch with many control vendors regrading helping them to support the new WinForms designer. Progress Telerik will release their is the first out with a preview of their controls (including designers) to support 16.8, and we're hoping others follow soon.

On our side, we are working on an extensibility SDK and documentation for control developers. There will be substantial changes in how to create control designers for .NET 5 that we will cover in the documentation. If you are a control developer and want help getting started, please reach out to me directly via twitter @oliagavrysh.

Kirsan September 14, 2020 8:27 pm

Yes, what about datagridview? Visual inheritance?

Log in to Reply

Jon Miller September 24, 2020 11:07 am

No, that's why they spent the whole page writing about performance improvements instead.

Log in to Reply

Mark Pflug September 14, 2020 10:35 am

"smart screen" is blocking the rc1 win-64-sdk installer. Running it anyway, the installer produces in an error "The file or directory is corrupted and unreadable".
There is a log file produced:

[0EC8:5294][2020-09-14T10:32:06]e000: Error 0x80070570: Failed to extract all files from container, erf: 1:4:0
[0EC8:5AC8][2020-09-14T10:32:06]e000: Error 0x80070570: Failed to begin and wait for operation.
[0EC8:5AC8][2020-09-14T10:32:06]e000: Error 0x80070570: Failed to extract payload: a3 from container: WixAttachedContainer
[0EC8:5AC8][2020-09-14T10:32:06]e312: Failed to extract payloads from container: WixAttachedContainer to working path: c:\Users\REDACTED\AppData\Local\Temp{57928144-4098-404F-AA9A-D781A36EB336}\8C09F345E398B4710CE74EE6AAFC4D9F91CB9A11, error: 0x80070570.
[0EC8:5374][2020-09-14T10:32:06]e000: Error 0x80070570: Cache thread exited unexpectedly.

Log in to Reply

Richard Lander September 14, 2020 10:44 am

We are working on fixing this currently. Major apologies on that!

Log in to Reply

Fabio Vides September 14, 2020 1:02 pm

Hi Richard

Do you have an estimated time frame?

Also very cool features. Can't wait to start testing them.

Log in to Reply

Maira Wenzel September 14, 2020 2:43 pm

This is now fixed. Apologies for the inconvenience.

Log in to Reply

Wiesław Šoltés September 14, 2020 10:59 am

The x64 windows installer file is corrupted https://dotnet.microsoft.com/download/dotnet/thank-you/sdk-5.0.100-rc.1-windows-x64-installer

Log in to Reply

Richard Lander September 14, 2020 11:54 am

We are working on this right now. Sorry!

Log in to Reply

Maira Wenzel September 14, 2020 2:46 pm

This issue has been fixed now. Apologies for the inconvenience.

Log in to Reply

Ismail Demir September 14, 2020 12:04 pm

Hello Rich,

[quote]The improvements to List(de)serialization is particularly impressive, comparing .NET 5.0 to .NET Core 3.1. [/quote]

Have you tested on VB.Net too?

Log in to Reply

Richard Lander September 14, 2020 12:07 pm

> Have you tested on VB.Net too?

No, but the language used shouldn't matter.

Log in to Reply

Ismail Demir September 14, 2020 12:23 pm

Ok, Thank you!

AFAIK there was small differences between C# and VB on nested arrays.

Hence the question

Log in to Reply

Layomi Akinrinade September 14, 2020 3:39 pm

Thanks for the feedback. We'll take a look at this scenario. If you have a repro for an issue, please feel free to share it here or on GitHub (https://github.com/dotnet/runtime/issues/new?template=01_bug_report.md).

Log in to Reply

Marcel Kwiatkowski September 14, 2020 12:07 pm

How about UWP? Is it ever going to support .NET 5

Log in to Reply

Richard Lander September 14, 2020 12:10 pm

The Windows team is working on that. See https://github.com/microsoft/ProjectReunion

Log in to Reply

HaloFour September 14, 2020 12:27 pm

I think that the description of records is very misleading. Records aren't immutable versions of classes. There's nothing about records that enforces or even encourages immutability. You are free to declare mutable fields and properties on a record just as you can on a class today. What records offer is value equality, "withers", deconstruction and little else. I think a *lot* of developers are going to end up confused (and probably angry) when they find out that records are mutable by design.

The closest records come to encouraging immutability is only if you declare a positional record with a primary constructor. In that case the parameters of the primary constructor are promoted, by default, to init-only properties. But that's it. You can declare additional mutable fields or properties. You can even override the default property definitions to be mutable.

Log in to Reply

Richard Lander September 14, 2020 12:55 pm

I showed both the immutable and mutable patterns in the post. I think of records as being biased to immutability giving that (as you say) properties declared via the constructor are immutable (`init` style). A big design point of records and `init` is that you are not locked in. Classes can use `init` and records can use `set`.

FWIW, C# design team members reviewed my wording.

Log in to Reply

Joseph Musser September 14, 2020 1:00 pm

"Records are immutable data types" sounds less about bias and more about guarantee.

Log in to Reply

Stilgar Naib September 14, 2020 6:40 pm

Why are the properties in the primary constructor style init rather than get-only? If I am going to provide the argument to the constructor anyway what use do I have for the init?

Log in to Reply

Andy Gocke September 14, 2020 10:17 pm

`with` expression requires it

Log in to Reply

Richard Lander September 14, 2020 10:54 pm

For folks not quite following, when you `with` a record, a new record instance is cloned from the given instance and then an object initializer is used (if that's the right terminology) to create a new record per the `with` specification.

Here's an example: https://gist.github.com/richlander/bcbbcc9e0b541a06eb805d663ebf6334

Stilgar Naib September 15, 2020 2:04 am

Sounds like they could have made the with syntax work with constructors but I guess that's also good enough

Nicolas Musset September 15, 2020 4:23 am

Well that seems like a potential problem. Records shouldn't allow any additional properties that are not `readonly` (or `init` style).

A quick attempt shows that the code generated by the compiler is broken in such case: the `GetHashCode()` uses the mutable fields in its computation which is incorrect (would break any hashing collection when used as a key). And the additional property is not part of the deconstruction which is also unexpected.

https://sharplab.io/#v2:EYLgtghgzgLgpgJwD4AEBMBGAsAKBQZgAIE4BjAewQBNCAZcgcwEsA7AJTinIFcFS4AFCgwAGQgFUoiFhDBwANIWFiACtCgB3SlUXBy5ADaEOcsMEQBZOAEpcAb1yEnSossIXuMCMANxCdwgY4GABuQilQwgBfXBicXBk5KAAHCH4lDAA6Nm4WGCY5TIBhcjBkpl8EAGVEADcmfih7RyU0QgBJKABRAA94BBkDdpYmGH9o2KA===

A more obvious example of why this is broken:

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        MyRecord record = new MyRecord(5);
        MyRecord clone = record with {};

        // check that we do have a clone
        Console.WriteLine(Equals(record, clone)); // true
        Console.WriteLine(ReferenceEquals(record, clone)); // false

        var hash = new HashSet();
        hash.Add(record);

        Console.WriteLine(hash.Contains(record)); // true
        Console.WriteLine(hash.Contains(clone)); // true

        record.Danger = true; // uh oh

        Console.WriteLine(hash.Contains(record)); // false
        Console.WriteLine(hash.Contains(clone)); // false
    }
}


public record MyRecord(int Value)
{
    public bool Danger { get; set; }
}

namespace System.Runtime.CompilerServices
{
    internal class IsExternalInit { }
}
```

Log in to Reply

Jared Parsons September 15, 2020 9:57 am

> Records shouldn't allow any additional properties that are not readonly (or init style).

This is essentially a call to force all `record` types to be immutable, essentially offering no method for creating mutable style records. Doing so artificially limits the use cases of the `record` feature to the point they wouldn't be useful

to a large set of our customers.

Yes indeed there are cases where you should strongly prefer an immutable style record. The easiest example is when the values will be used as a key in a `Hashset<T>` or `Dictionary<TKey, TValue>`.

However there are many completely valid uses cases for `record` that involve mutable data. Hence the design explicitly allows for developers to create and use such types.

Log in to Reply

Nicolas Musset September 29, 2020 2:50 am

My remark is more, why does `GetHashCode()` also take the mutable properties into account?
Couldn't it just skip them? It would still be correct from the definition of what it is supposed to do (there would just be collisions with instances having the same immutable values, but different mutable values), but that's already expected to happen with `GetHashCode()` anyway.

Skipping it would make it always correct to use in a hashing collection. That makes the record feature many times more useful: you can have "attached" or "debug" properties that are mutable but don't affect the rest of the record.

In other word: why ship a broken feature when it could easily not be?

Mads Torgersen September 15, 2020 9:59 am

Records are value-based. Their equality and hashcode are based on their fields. If records are supposed to e.g. be used in dictionaries, then it's a good idea to make them immutable, and I suspect most people will. That is also the default for properties generated from the records's primary constructor.

But we do not enforce immutability. We don't want to preclude scenarios like caching or lazily computed properties. You can provide your own equality if you want to exclude certain members, or compare them in non-standard ways. Also, when we get to struct records in C# 10, they do not have the same dangers around mutability, because they are not stored by reference.

I do get that for the common case folks would like to not be mutable by accident. We should probably look at warnings/diagnostics or similar when you declare mutable fields in a record.

Log in to Reply

Jeroen Haegebaert September 16, 2020 8:19 am

If you provide your own equality and have mutable properties, where is the value in using records over classes?

Joseph Musser September 14, 2020 12:59 pm

Yes, thinking "if it's a record, then it's immutable" has been steadily confusing people. We hear this on Twitter, Discord, and Gitter, and the csharplang GitHub repo. Here are some mutable records:

```
record Mutable(int A)
{
    public int A { get; set; } = A;
}

record AlsoMutable(int A)
{
    public int B { get; set; }
}

record MutableAgain
{
    public int A { get; set; }
}
```

Log in to Reply

Richard Lander September 14, 2020 1:19 pm

Fair. Maybe “records enable creating immutable data types” would be better phrasing.

Log in to Reply

Joseph Musser September 14, 2020 1:56 pm

Thanks for listening! Excellent blog post as usual. I always look forward to these.

Log in to Reply

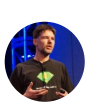
Richard Lander September 14, 2020 3:08 pm

NP. Thanks!

Stilgar Naib September 14, 2020 6:34 pm

I am already angry records are not immutable. I hope there is editor.config setting that lets me ban the property declaration syntax for records in my projects.

Log in to Reply



Mads Torgersen September 15, 2020 10:04 am

We should consider an analyzer that tells you if you declare mutable state in a record. Property declarations in and of themselves are not a problem WRT mutability, and declaring your (immutable) state with init properties instead of constructor parameters can lead to less brittle code especially when your have inheritance hierarchies of records. Also for bigger record types, constructors can get long and unwieldy.

It’s your choice of course, but it was important for the design of the feature not to create a strong coupling between immutability/value-based semantics and a positional style of declaration and creation. We want you to have the same choice as you do for mutable types, and for inheritance to work as well.

Log in to Reply

Andy Gocke September 14, 2020 10:14 pm

I disagree. I think immutable-by-default is an appropriate way to phrase it, partially because the defaults in a language are purely what the community encourages and enforces. Sure, there's nothing stopping you from declaring mutable properties or fields, but if we describe records as being purpose-built for immutable data, then usage should follow.

Moreover, because of the "dictionary mutation" problem, there is a particularly good reason to be very careful about mutable data in record.

Log in to Reply

Adir Hudayfi September 14, 2020 1:39 pm

Thanks for the great article Richard, looks amazing!
One question – do you expect to ship .NET 5 with WinRT APIs (net5-windows10.0.xxxxx and CSWinRT nuget) fully working like the .NET Core 3.1 solution (Microsoft.Windows.SDK.Contracts)? Because a lot of the APIs are broken (InvalidCastExceptions and such) and that's the only thing holding me back from completetly upgrading

Thanks!

Log in to Reply

Scott Jones September 14, 2020 3:11 pm

Adir, the CsWinRT team is working through those issues now (InvalidCastExceptions, etc). Our goal is to have them all addressed by .NET 5 RTM. Any remaining issues will be addressed as soon as possible after that and fixes can be consumed with an explicit nuget reference to the Microsoft.Windows.CsWinRT package.

Log in to Reply

Adir Hudayfi September 15, 2020 3:46 am

Thanks for the reply Scott, looking forward to it.

Log in to Reply

Joseph Musser September 14, 2020 1:45 pm

```
if (lord.RelatedAuthors is object)
```

If you like, there's a C# 9 feature that lets you use `is not null` instead of `is object`.

Log in to Reply

Richard Lander September 14, 2020 3:06 pm

Ah, I forgot about that. You are right. I'll leave it as is but will consider that for another post.

Log in to Reply

Damian Wyka September 14, 2020 1:47 pm

A lot of nice improvements!

However i have a question about json serializer: do you plan adding in future support for private members including fields?

Having to expose public member just to serialize private data is huge smell and potentially PITA when you would like to prevent direct access to private data (eg. In library code or if you want to minimize risk of breaking changes when modifying internals) but being still fine with exposing it in string since it not breaking security. Eg detecting changes in unknown object state

Log in to Reply

Layomi Akinrinade September 14, 2020 3:50 pm

Thanks for the feedback. Yes we plan to enable (de)serialization of private fields/properties in an opt-in manner. We held off on enabling this support until the implications to an ongoing JSON code-gen effort (https://github.com/dotnet/runtime/issues/1568) are resolved. The earliest it can come is in .NET 6 (https://github.com/dotnet/runtime/issues/31511). The workaround today is to add a custom converter for the declaring types of each non-public member to be (de)serialized.

Note that public field support has been added in .NET 5. It is now also possible to use non-public accessors on public properties using JsonIncludeAttribute. This also needs support in the codegen effort.

Log in to Reply

Damian Wyka September 14, 2020 6:52 pm

I hope even if you dont manage to support private in code-gen (although if you ditch source generators or mix them with traditional ones or allow source generators to spit IL code side by side with c#, private access should be possible) then let us decide between sticking to emit with private support or code-gen without private, otherwise system.text.json will always feel lacking

Log in to Reply

Layomi Akinrinade September 15, 2020 2:50 pm

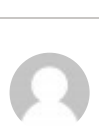
Yes if non-public members are not supported in the code-gen usage pattern, or if the code-gen option is not enabled by the user, you should still be able to use the dynamic serializer for all features including non-public support.

Log in to Reply

Michael Quinlan September 14, 2020 3:47 pm

Is ToString() implemented for records? I am using Visual Studio. for Mac 8,8 Preview and .NET 5.0.100-rc.1.20452.10 and when I use ToString() I get the record name and not the contents.

Log in to Reply



Richard Lander September 14, 2020 4:45 pm

Can you try this sample: https://gist.github.com/richlander/83d2ccde88218e56e6b93a55db3f986b#file-program-cs

This is what I see: https://gist.github.com/richlander/72091c7fb77c6b82ddd93da342326866

Log in to Reply

Michael Quinlan September 15, 2020 9:50 am

Running on Visual Studio for Mac I get

```
Battery
Battery
```

I tried this to test ToString and Equals

```
var r = new Record(0, "Value");
System.Console.WriteLine($"r={r} Equals={r == new Record(0, "Value")}");
public partial record Record(int Key, string Value = null);
```

And it produces

```
r=Record Equals=True
```

Which suggests that (for me) Equals is working but ToString is not.

Again, this is with Visual Studio. for Mac 8,8 Preview and .NET 5.0.100-rc.1.20452.10

Log in to Reply

Michael Quinlan September 21, 2020 12:56 am

I've tested and the code prints the correct value on Visual Studio 2019 Preview for Windows. It is only on the Mac that it prints the class name and not the values.

Log in to Reply

Newer Comments →

Relevant Links

.NET Download

.NET Hello World

.NET Meetup Events

.NET Documentation

.NET API Browser

.NET SDKs

.NET Application Architecture Guides

Web apps with ASP.NET Core

Mobile apps with Xamarin.Forms

Microservices with Docker Containers

Modernizing existing .NET apps to the cloud

Archive

October 2020

September 2020

August 2020

July 2020

June 2020

May 2020

April 2020

March 2020

February 2020

January 2020

December 2019

Topics

Dot.Net

.NET

.NET Core

.NET Framework

Entity Framework

C#

ML.NET

Visual Studio

F#

WPF

Stay informed