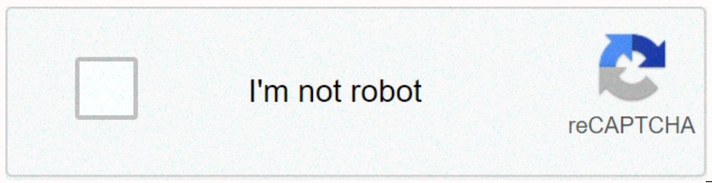


Continue

# Ienumerable vs icollection performance

Today I will write about the collection in C# and eventually in my next blog I will write about some new cool features in .NET Core 3.0.But first focus on the basics of the collection in C#. What is a collection? This is a container that can hold some elements. Collection typesIn case you don't know, there are several types of collections in C#: IEnumerableIQueryableICollectionIListBy they are essentially the same, each of which has specific characteristics and is used in some scenarios. Collections in C#As you can see in the photo above, IEnumerable is the initial class. All other classes come from it. You may ask yourself why IQueryable is moved to the left in the photo. Why isn't it, like the rest of the collection, directly under IEnumerable!? Mainly because doing so makes it easier to show that IQueryable is in a different namespace. While IEnumerable, ICollection, and IList are located in System.Collections, IQueryable is located inside the System.Linq namespace. I'll explain later why! System.CollectionsIEnumerable So as defined above, IEnumerable is the base class that represents the container. It is located in the System.Collections namespace. You cannot add, edit, delete, or count items when items are initially loaded with this collection type. In this way, we prevent any modification of our items. We use this container only to contain a list of items and nothing else. To count our items inside this type of collection, we need to implement IEnumerator.i.e.:p ubowe Class BankAccount: IEnumerator { Public Object Current { get { throw new NotImplementedException(); } } public bo MoveNext() { throw new NotImplementedException(); } public void Reset() { throw new NotImplementedException(} ICollectionThis is an IEnumerable derived class with slightly more functions added in it. With this class, we can add, edit, delete, and count items in a container. You'll see mostly this kind of collection in your dealings with the entity framework. Because there's not so much overhead. The IListThis collection type comes from ICollection. That said, this means that it has all the specifics of IEnumerable and ICollection and some additional features, such as inserting or deleting an item in the middle of the list. This is the easiest collection and probably the most commonly used collection type. But it's overhead that makes you not the king of performance. IQueryableI finally have this kind of collection. And the main difference between this collection and others is that when accessing a database, it generates LINQ to the SQL expression that is executed through the database layer. Final verdict: IEnumerable vs. IQueryable There are some key differences between the two types of collections. I will summarize them in the table below. But one of the biggest differences that are necessary to understand is the way in which each of the two data queries. Although IEnumrable to query in-memory collections such as arrays, lists, etc., IQueryable is to query out-memory colelctions, such as some services, remote databases, etc. Why? By using IEnumerable to query the database, we perform a select server-side query, and then load that data into client-side memory and only then filter the data. On the other hand, using IQueryable, we make a select query with all server-side filters and return the already filtered data. As you can see it's a lot of performance wise to use IQueryable when retrieving filtered data from a database. IEnumerableIQueryableNamespace: System.CollectionsNamespace: System.LinqQueries: IN-MEMORYQueries: OUT-MEMORYDB: Select query on the server, filters on client pageDB: Select query and server-side filtersBeneficial: LINQ to object or LINQ to XMLBeneficial: LINQ to SQL(Visited 19 times, 1 visit today) When people come to speed in C#, collection types present options. And sometimes these options are wrong. Should I use C# IEnumerable, IList, or ICollection? And what is IQueryable, yes? This kind of puzzling usually brings to the official documentation for one of the types. For example, look at the <T> declaration of the IList interface. Public interface IList:<T> ICollection<T>, IEnumerable<T>, IEnumerable IList<T> implements IEnumerable<T>, so IEnumerable must be a kind of subset of IList. Right? And, for performance reasons, we should probably only use what we need. A good rule of law would be then to see if you can easily use IEnumerable and then switch to IList when you need to do something IEnumerable doesn't do. It seems reasonable. Well, it seems reasonable until you really, really understand both types. But I'll go back to that. A Tale of Performance Bioe I think most people in the industry can relate to differences between behavior in development or test and in production. In fact, we immortalized this sentiment with a ubiquitous slogan. It works on my machine! When we think about this sentence, we usually think in terms of behavior. Everything worked when he ran it locally, but then he blew himself up while working on someone else's machine. Oops. You found that you are referring to an environment variable set on your computer, but not elsewhere. But it can also refer to performance issues. You encode something, run it in a local database, and everything seems fine. So you send it to pre-prod or even production. Only then can you discover some terrifyingly different behavior that can't be explained using a simple database scale? After some pain and digging, you'll find yourself learning about something you've never heard of before called an N+1 problem. It seems that your elegant data access code is not as elegant as it initially seemed. Digging into C# IEnumerable, Whiteboards and Lists Let's now get to the topic of true understanding that I just mentioned. Let me start with certain types of collections that you may be more familiar with. Consider, to begin with, a humble board. Boards go about as far as programming, i</T> </T> </T> </T> </T> </T> </T> represent grouped values. They have a declared and fixed capacity, so if you make a 4-element array, it will always store 4 elements. You can set these elements to different values, and you can iteate the backhaf in the array, if you choose to do so by performing an operation on each element. As I mentioned, the boards are old. People used them forever and at that time they came up with their limitations. The property of having a fixed length causes annoyance, and people find it convenient to perform quick operations such as sorting and filtering duplicates. So you end up with heavier weight, more sophisticated types such as List<T>, which implements the IList interface.<T> This interface requires that its implementations support operations such as addition, removal, and cleaning. Many IList implementations will include arrays in their core, decorating them with a convenience feature. So you can think of them as a comfortable, heavyweight matrix. But if the lists are heavyweight boards, does it make IEnumerable<T>, with one GetEnumerator method, sort of a lighter weight array? It turns out not. Not at all. The promise of IEnumerable When using the C# IEnumerable design, things get conceptually strange in no time. At least they do if you're not used to how it works. In a sense, boards and lists are tangible. You have a few strings or integers sitting there in your memory in a row, waiting for you to do things to them. Easy to work with and easy to justify. When it comes to IEnumerable, however, you have nothing tangible. Instead, you have a promise that you'll have things sitting there in a row later when you really need them. Perhaps you've heard the term lazy charging before (don't charge until you need to)? Well, IEnumerable<T> implements a related deferred execution concept that prohibits calculations until someone uses the result. Let's make it more tangible with a simple allegorization. Let's say I'm a method, and my job is to return the fruit. When I return a list of fruits to you, you ask me for fruit and I give it to you in an orderly manner. Here's an apple, here's an orange, etc. But when I return the fruit to you, I give you something completely different. I give you a note that says: This note entitles you to fruit - call me when you want to eat the first fruit, and I will produce it at that time. Back in the world of programming, you can think of it as a commitment or a promise of sorts. In fact, all ienumerable promises are the basic strategy of delivering the next element — the state machine, if you want to get technical information about it. Back to the world of databases and performance Let's go back to narration now. You sit at your desk implementing the MVC application in the database and

everything seems fine. Observe the thumb rule I mentioned earlier, using IEnumerable<T> (or IQueryable),<T>when they have sufficient functionality. idzie wzdłuż pływanie jak</T> </T> </T> </T> </T> </T> </T> </T> development and conduct of tests. When you see some performance issues, they don't appear during a database operation. Instead, they appear when you move between records and create pages that you want to return over the network. So whatever slowness you notice must come from some part of the web structure itself is slow. Your database queries are coming back at lightning speed, so everything is good! Except, now you know it's not true. As you step-by-step through the call stack down toward database calls and then back up again, you can understand what you're really getting. This call that seems to trigger SELECT* and the result population really only returns the confidence that it will be a call when the time comes. This method returned iEnumerable says: Yes, please, one note promises that we will make the database later when you need the data. Keeping performance intact With this knowledge in mind, you can have better antennas for types of performance issues that seem small in dev but appear huge in production. Deferred execution (and load delay) offer a good way to postpone potential performance hits to the absolute last minute. But in this way, they force a compromise on you, in which they make execution more difficult to reason. When you widely use these patterns, you lose track where bottlenecks really lie. So you have to be vigilant. Also, note that not all IEnumerable implementations use deferred execution. After all, IList implements IEnumerable and does not defer anything. Deferred execution is only a possibility, depending on the implementation that you end up with. So keep an eye on it. And, more broadly, make sure you understand exactly what IEnumerable is and what you sign up for when you use it. It's not just a lighter weight list. This understanding will help you keep both performant and correct code. Correct.