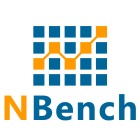
**NBench**

[](https://github.com/petabridge/NBench/blob/dev/images/NBench_logo_square_140.png)

 Cross-platform performance benchmarking and testing framework for .NET applications.

NBench is designed for .NET developers who need to care about performance and want the ability to "unit test" their application's performance just like [XUnit](https://github.com/xunit/xunit) or [NUnit](http://nunit.org/) tests their application code.

**Quickstart**

To use NBench, install the NBench package from NuGet:

PS> Install-Package NBench

And then create a POCO class with a default constructor and some methods, like this:

using NBench.Util;

using NBench;

/// <summary>

/// Test to see if we can achieve max throughput on a <see cref="AtomicCounter"/>

/// </summary>

public class CounterPerfSpecs

{

private Counter \_counter;

[PerfSetup]

public void Setup(BenchmarkContext context)

{

\_counter = context.GetCounter("TestCounter");

}

[PerfBenchmark(Description = "Test to ensure that a minimal throughput test can be rapidly executed.",

NumberOfIterations = 3, RunMode = RunMode.Throughput,

RunTimeMilliseconds = 1000, TestMode = TestMode.Test)]

[CounterThroughputAssertion("TestCounter", MustBe.GreaterThan, 10000000.0d)]

[MemoryAssertion(MemoryMetric.TotalBytesAllocated, MustBe.LessThanOrEqualTo, ByteConstants.ThirtyTwoKb)]

[GcTotalAssertion(GcMetric.TotalCollections, GcGeneration.Gen2, MustBe.ExactlyEqualTo, 0.0d)]

public void Benchmark()

{

\_counter.Increment();

}

[PerfCleanup]

public void Cleanup(){

// does nothing

}

}

After defining some NBench PerfBenchmark methods and declaring some measurements, you can run your benchmark by downloading the NBench.Runner.exe via NuGet.

**Running NBench Tests with dotnet nbench**

The easiest way to run NBench in any runtime, .NET Core or .NET Framework is to install the dotnet-nbench NuGet package and add it as a DotNetCliToolReference to your NBench test projects.

<PackageReference Include="NBench" Version="1.2.1" />

<DotNetCliToolReference Include="dotnet-nbench" Version="1.2.1" />

This will allow you to run NBench specifications for all frameworks your test projects target, including multi-targeted projects, by simply running the following command in the working directory of your test project:

dotnet nbench [--output {dir-path}] [--configuration {file-path}] [--include MyTest\*.Perf\*,Other\*Spec] [--exclude *Long*] [--concurrent {true|false}] [--trace {true|false}] [--teamcity] [--diagnostic]

This will build and compile your project, as well as run the appropriate NBench.Runner executable to run your tests and produce an output report.

N.B. If you don't specify an --output setting while running in this mode, dotnet-nbench will automatically record all output to the [currentDir]/PerfResults/[framework].

**Stand-alone Runner**

Below are the instructions for the standalone NBench.Runner package, which is substantially more manual than the dotnet nbench runner described above.

**.NET 4.5.2 Runner**

The NBench Runner NuGet package now contains executables that support .NET 4.5.2, .NET Core 1.1/win7-x64, and .NET Core 1.1/debian8-x64 benchmark assembly targets. Before v1.0.3, you would expect the NBench.Runner NuGet package to contain a single executable that works with .NET 4.5.2:

lib/

net45/

NBench.Runner.exe

With the additional .NET Core executables, you will now get two additional subfolders in your downloaded package:

lib/

net452/

NBench.Runner.exe

netcoreapp1.0/

win7-x64/

NBench.Runner.dll

netcoreapp2.0/

win7-x64/

NBench.Runner.dll

netcoreapp2.1/

win7-x64/

NBench.Runner.dll

You may choose the appropriate executable for your benchmark assembly/architecture combination

To install and execute the NBench Runner against a .NET 4.5.2 benchmark assembly, you would execute the following:

PS> Install-Package NBench.Runner

PS> <--packageLocation-->\NBench.Runner\lib\net452\NBench.Runner.exe <--benchmarkProjectLocation-->\bin\Debug\net452\MyPerfTests.dll output-directory="C:\Perf"

And this command will run your PerfBenchmark and write output [that looks like this](https://gist.github.com/Aaronontheweb/8e0bfa2cccc63f5bd8bf) to a markdown file in the output-directory.

**.NET Core Runner**

If you have compiled a benchmark assembly that targets .NET Core 1.1, you may use the appropriate executable from the lib/netcoreap1.1 folder of the downloaded NBench.Runner NuGet package;

PS> Install-Package NBench.Runner

PS> <--packageLocation-->\NBench.Runner\lib\netcoreapp1.0\win7-x64\NBench.Runner.exe <--benchmarkProjectLocation-->\bin\Debug\netcoreapp1.1\MyPerfTests.dll output-directory="C:\Perf"

**It's probably easier to just use dotnet-nbench**.

**Command Line Parameters**

NBench.Runner.exe [assembly names] [--output {dir-path}] [--configuration {file-path}] [--include MyTest\*.Perf\*,Other\*Spec] [--exclude *Long*] [--concurrent {true|false}] [--trace {true|false}] [--teamcity] [--diagnostic]

* **assembly names** - list of assemblies to load and test. Space delimited. Requires .dll or .exe at the end of each assembly name
* **--output-directory path** - folder where a Markdown report will be exported. Report will [look like this](https://gist.github.com/Aaronontheweb/8e0bfa2cccc63f5bd8bf)
* **--configuration path** - folder with a config file to be used when loading the assembly names
* **--include name test pattern** - a ","(comma) separted list of wildcard pattern to be mached and included in the tests. Default value is \* (all) The test is executed on the complete name of the benchmark Namespace.Class+MethodName
* **--exclude name test pattern** - a ","(comma) separted list of wildcard pattern to be mached and excluded in the tests. Default value is `` (none) The test is executed on the complete name of the benchmark Namespace.Class+MethodName
* **-- concurrent true|false** - disables thread priority and processor affinity operations for all benchmarks. Used only when running multi-threaded benchmarks. Set to false (single-threaded) by default.
* **--trace true|false** - turns on trace capture inside the NBench runner and will save any captured messages to all available output targets, including Markdown reports. Set to false by default.
* **--diagnostic** - turns on diagnostic logging inside the NBench.Runner and dotnet-nbench executables.
* **--teamcity** - turns on TeamCity message formatting.

Supported wildcard patterns are \* any string and ? any char. In order to include a class with all its tests in the benchmark you need to specify a pattern finishing in \*. E.g. include=\*.MyBenchmarkClass.\*.

Example patterns:

--include "\*MyBenchmarkClass\*" (include all benchmarks in MyBenchmarkClass)

--include "\*MyBenchmarkClass+MyBenchmark" (include MyBenchmark in MyBenchmarkClass)

--include "\*MyBenchmarkClass\*,\*MyOtherBenchmarkClass\*" (include all benchmarks in MyBenchmarkClass and MyOtherBenchmarkClass)

--exclude "\*MyBenchmarkClass\* "(exclude all benchmarks in MyBenchmarkClass)

--exclude "\*MyBenchmarkClass+MyBenchmark" (exclude MyBenchmark in MyBenchmarkClass)

--exclude "\*MyBenchmarkClass\*,\*MyOtherBenchmarkClass\*" (exclude all benchmarks in MyBenchmarkClass and MyOtherBenchmarkClass)

**API**

Every NBench performance test is created by decorating a method on a POCO class with a PerfBenchmark attribute and at least one type of "measurement" attribute.

**Creating a Benchmark**

Here are the different options for creating a PerfBenchmark:

* Description - optional. Used to describe the purpose of a particular benchmark.
* RunMode - sets the run mode for this. Possible options are RunMode.ThroughPut and RunMode.Iteration.
* TestMode - sets the test mode for this benchmark. Possible options are TestMode.Measurement and TestMode.Test. More on what those options mean in a moment.
* NumberOfIterations - determines how many times this benchmark will be run. All final benchmark statistics are reported as an aggregate across all iterations.
* RunTimeMilliseconds - for RunMode.ThroughPut, this indicates how long we'll attempt to run a test for in order to measure the metric per second values.
* SkipWarmups - disables "warmup" iterations that are used to perform cache warming on the CPU. Disabling warmups is often used in long-running iteration tests.

You can declare a PerfBenchmark attribute on multiple methods within a single POCO class and each one will be run as its own independent benchmark.

A PerfBenchmark, PerfSetup, or PerfCleanup method can either take no arguments, or it can take an NBench.BenchmarkContext object.

**Benchmark modes**

There are two important modes that you can use in the design of your benchmarks - the RunMode and the TestMode.

RunMode indicates how the benchmark will be run. RunMode.Throughput is designed for *very* small benchmarks, like single-line methods, and is meant for scenarios where you really need to measure the throughput of a given operation. During a Throughput benchmark the `Benchmark

Of course you can also measure things like Garbage Collection and memory allocation too, but those are typically more interesting inside RunMode.Iteration tests. RunMode.Iteration is designed for running more complex blocks of code where you want to profile things like memory allocation or garbage collection and are less concerned with measuring the throughput of a particular block of code.

TestMode indicates how NBench will evaluate the data at the end of the benchmark. If set to TestMode.Measure, NBench will simply report the measurements of all collected values and not perform any unit test-style assertions on any of them; this includes metrics that are declared with "Assertion" attributes such as CounterThroughputAssertion and others.

When TestMode.Test is enabled, all available assertions will be checked against the collected data. If any such assertions fail, NBench.Runner.exe will return with a failure exit code.

**Benchmark lifecycle**

During the course of each iteration of a benchmark, NBench follows the following object creation lifecycle:

Initialize --> PreWarmup --> Warmup --> Benchmark --> Report

The purpose of the *Prewarmup* and *Warmup* phases is a little different depending on your RunMode, but the general idea is to pre-JIT all of the code that's about to be benchmarked and to estimate how long the benchmark will take to run.

The *Prewarmup*, *Warmup*, and *Benchmark* phases will all perform the following calls against your POCO class:

For each iteration:

Create Object --> PerfSetup --> PerfBenchmark --> PerfCleanup --> Destroy Object

During a RunMode.Throughput benchmark, the PerfBenchmark method will be invoked continuously until the estimated time (determined by estimated time gathered during the *Warmup* phase) has elapsed. PerfSetup and PerfCleanup methods will not be called again until after the throughput test is complete, in other words.

**Measuring and Asserting Benchmark Data**

As of the most recent stable release, NBench allows you to gather the following types of data:

1. Memory allocation - how much memory has the code in your benchmark demanded from the CLR?
2. Garbage collection (GC) - how many collections for each GC generation has the code in your benchmark used?
3. Counters - how many calls were made against a specific counter by your code?

**Memory**

There are two types of attributes you can use to instrument a benchmark for memory measurement:

* MemoryMeasurement - just measures a pre-defined memory metric without specifying any assertions against the data.
* MemoryAssertion - collects data and performs an assertion.

Inside the constructor of each of these attributes you can specify one of a possible number of memory metrics. Right now here is what NBench supports:

* MemoryMetric.TotalBytesAllocated - Measure the total bytes allocated during a benchmark.

To collect data for this metric, declare an attribute on your PerfBenchmark method like so:

class MyMemoryBenchmark{

[PerfBenchmark(NumberOfIterations = 3, RunMode = RunMode.Iteration,

RunTimeMilliseconds = 1000, TestMode = TestMode.Test)]

[MemoryAssertion(MemoryMetric.TotalBytesAllocated, MustBe.LessThanOrEqualTo,

ByteConstants.ThirtyTwoKb)]

public void SomeBenchmarkMethod()

{

// some code

}

}

In the example above, if the block of code inside SomeBenchmarkMethod causes the CLR allocate more than 32kb of memory on average across 3 iterations of this code then the MemoryAssertion will write a failure into the report produced by NBench.

NOTE: The way MemoryMetric.TotalBytesAllocated is measured is by taking a *before* and *after* snapshot of total memory as reported by the CLR Garbage Collector.

Memory is typically allocated in pages, i.e. the OS might allocate 8kb when all you need is 12 bytes, that way it doesn't have to constantly allocate lots of small segments of memory to the process, so **to get best results it's recommended that you write your memory benchmarks such that you allocate a large number of the objects you want to benchmark**. That helps average out the noise produced by this allocation strategy on the part of the OS.

**Garbage Collection**

The goal of checking GC metrics is to help eliminate the garbage collector as a source of pauses, slowdowns, and negative impact on throughput in your application. Therefore NBench's GC measurement capabilities are designed to help you track the number of GC collections that occur at each generation as part of your benchmark.

Here are the types of attributes you can use for measuring GC inside your benchmarks:

* GcMeasurement - measure the number of collections that occur for the specified GC generation without performing any assertions.
* GcTotalAssertion - measure and assert against the total number of collections that occurred at the specific GC generation.
* GcThroughputAssertion - measure and assert against the total number of collections **per second** that occurred at the specific GC generation.

Inside the constructor of each of these attributes you can specify the type of GC metric you want to collect. Here's what is currently supported inside NBench:

* GcMetric.TotalCollections - the total number of GC collections at the specified generation.

You must also specify one of the following GC generations in the constructor of your attribute, as it indicates for which GC generation you want to apply this measurement:

* GcGeneration.Gen0 - Gen 0.
* GcGeneration.Gen1 - Gen 1.
* GcGeneration.Gen2 - Gen 2. This is usually the one you want to pay close attention to.
* GcGeneration.AllGc - all supported generations.

To collection GC data, declare one of the GC measurement attributes on your PerfBenchmark method like this:

public class CounterPerfSpecs

{

private Counter \_counter;

[PerfSetup]

public void Setup(BenchmarkContext context)

{

\_counter = context.GetCounter("TestCounter");

}

[PerfBenchmark(Description = "Test to ensure that a minimal throughput test can be rapidly executed.",

NumberOfIterations = 3, RunMode = RunMode.Throughput,

RunTimeMilliseconds = 1000, TestMode = TestMode.Test)]

[GcTotalAssertion(GcMetric.TotalCollections, GcGeneration.Gen2, MustBe.ExactlyEqualTo, 0.0d)]

public void Benchmark()

{

\_counter.Increment();

}

[PerfCleanup]

public void Cleanup(){

// does nothing

}

}

In the example above, this spec would fail if any Gen2 garbage collection occurred throughout the 1000ms duration of this benchmark (which, based on the reading of this code, none should.)

NOTE: NBench will force the .NET garbage collector to perform a full cleanup before the benchmark runs and AFTER it completes. It does not force the garbage collector to run *before* we measure GC collection attempts. So to get the best possible measurement, design your GC benchmarks such that they offer opportunities for the GC to run. Long-running iteration benchmarks that repeat the same code deterministically (i.e. a for loop) are one option.

**BenchmarkContext, Counters, and Throughput**

The role of BenchmarkContext is solely for accessing user-defined counters that are declared using any one of the following three attributes:

* CounterThroughputAssertion - used to perform an assertion against the number of operations per second measured by this named counter.
* CounterTotalAssertion - used to perform an assertion against the TOTAL number of operations measured by this named counter.
* CounterMeasurement - used to simply measure and report on a counter without any sort of assertions.

You can declare multiple counter attributes that all measure against the same counter, i.e.

public class CounterExample

{

private Counter \_counter;

[PerfSetup]

public void Setup(BenchmarkContext context)

{

\_counter = context.GetCounter("TestCounter");

}

// Perfectly valid counter setup

[PerfBenchmark(NumberOfIterations = 3, RunMode = RunMode.Throughput,

RunTimeMilliseconds = 1000, TestMode = TestMode.Test)]

[CounterThroughputAssertion("TestCounter", MustBe.GreaterThan, 10000000.0d)]

[CounterTotalAssertion("TestCounter", MustBe.GreaterThan, 10000000.0d)]

[CounterMeasurement("TestCounter")]

public void Benchmark()

{

\_counter.Increment();

}

}

To gain access to a Counter for use inside your benchmark, you will need to pass in the BenchmarkContext to either your PerfSetup method or your PerfBenchmark method, as shown in the example above. From there you can call context.GetCounter("{YOUR COUNTER NAME}") and retrieve access to the counter instance being tracked by NBench.

NOTE: If you call BenchmarkContext.GetCounter(string counterName) and counterName doesn't match the name of any of your counters declared in your CounterThroughputAssertion, CounterTotalAssertion, and CounterMeasurement attributes then you will receive an NBenchException.

**Best Practices for Working with Counters**

Here are a few best practices to bear in mind when working with counters:

1. It's always best to store references to your Counter instances as fields inside your POCO class and to get those references during a PerfSetup call, rather than get references to them on-the-fly inside your benchmark.
2. Adding the BenchmarkContext parameter to your PerfBenchmark methods will improve throughput slightly, due to a design of NBench's ReflectionInvoker.