When working with applications, you will often want to know the memory allocation, garbage collection (GC) overhead, and throughput of the code. Your application might be slow, or it might be consuming a lot of resources, and you want to find out what’s wrong.

Although you can detect functional problems and code defects using unit tests and code reviews, you might still need a way to isolate performance issues. Here’s where NBench comes in handy. Here we discuss of NBench and how we can use it to write performance tests for .NET applications.

**What is NBench? Why should I use it?**

NBench is a popular performance testing framework that can be used to profile the performance of methods in our application. NBench can measure the throughput of your application’s code, the memory allocation, and the GC overhead involved in reclaiming memory by cleaning up unwanted objects.

Or

[**NBench** - a .NET performance-testing, stress-testing, and benchmarking framework for .NET applications that works and feels a lot like a unit test](https://github.com/petabridge/NBench).

 NBench is that it can be integrated into your build pipeline. And even though NBench has its own runner, you can still run NBench using NUnit or Resharper. It feels just like running your unit tests.

NBench is distributed as a NuGet package. Assuming that Visual Studio is already installed on your system, you can install NBench via the NuGet package manager or by using the following command in the package manager Console.

**Install**-**Package** **NBench**

## Writing performance tests using NBench

Let’s explore how we can write and execute performance tests using NBench. Create a new class library project, and save it with a helpful name. Next, add the NBench and NBench.Runner packages I mentioned above. Here is the start of our NBench performance test method.

You should also install the NBench.Runner package, which is used to run your benchmark. You can do that via NuGet also, or execute the following command from the package manager console.

**Install**-**Package** **NBench**.**Runner**

[**PerfBenchmark**(**NumberOfIterations** = 1, **RunMode** = **RunMode**.**Throughput**,  
**TestMode** = **TestMode**.**Test**, **SkipWarmups** = **true**)]  
[**ElapsedTimeAssertion**(**MaxTimeMilliseconds** = 5000)]  
**public** **void** **Benchmark\_Performance\_ElaspedTime**()  
{  
    *//Write your code to be benchmarked here*  
}

Note that because we are benchmarking performance, we need to mark our method using the PerfBenchmark attribute. This attribute tells the runner what to do with this method. We also need to include one or more measurement attributes. Since we are testing for speed of execution, we use the ElapsedTimeAssertion attribute to specify the time within which the method should complete. There are many other assertion attributes that you can take advantage of. The supported assertions in NBench include the following

**NBench Assertions**

* MemoryAssertionAttribute
* This MemoryAssertion attribute specifies that we want to measure the total number of bytes allocated on each iteration of this benchmark, and if that value ever matches or exceeds 32kb then this performance test is considered to be a failure.
* GcTotalAssertionAttribute
* ElapsedTimeAssertionAttribute
  + attribute to specify the time within which the method should complete.
* CounterTotalAssertionAttribute
* GcThroughputAssertionAttribute
* CounterThroughputAssertionAttribute
* PerformanceCounterTotalAssertionAttribute
* PerformanceCounterTotalAssertionAttribute

If you want to benchmark performance based on memory consumption, here is a test method you can use.

* [**PerfBenchmark**(**Description** ="You can write your description here.",  
  **NumberOfIterations** = 5, **RunMode** = **RunMode**.**Throughput**, **RunTimeMilliseconds** = 2500, **TestMode** = **TestMode**.**Test**)]  
  [**MemoryAssertion**(**MemoryMetric**.**TotalBytesAllocated**, **MustBe**.**LessThanOrEqualTo**, **ByteConstants**.**SixtyFourKb**)]  
  **public** **void** **Benchmark\_Performance\_Memory**()  
  {  
      *//Write your code to be benchmarked here*  
  }

The following method illustrates how we can benchmark the performance of the garbage collector. The Benchmark\_Performance\_GC method gives us the max, min, average, and standard deviation of collections that occur for each of the three GC generations (generation 0, 1, and 2).

[**PerfBenchmark**(**RunMode** = **RunMode**.**Iterations**, **TestMode** = **TestMode**.**Measurement**)]  
[**GcMeasurement**(**GcMetric**.**TotalCollections**, **GcGeneration**.**AllGc**)]  
**public** **void** **Benchmark\_Performance\_GC**()  
{  
    *//Write your code to be benchmarked here*  
}

If you want to benchmark performance based on memory consumption, here is a test method you can use.

[**PerfBenchmark**(**Description** ="You can write your description here.",  
**NumberOfIterations** = 5, **RunMode** = **RunMode**.**Throughput**, **RunTimeMilliseconds** = 2500, **TestMode** = **TestMode**.**Test**)]  
[**MemoryAssertion**(**MemoryMetric**.**TotalBytesAllocated**, **MustBe**.**LessThanOrEqualTo**, **ByteConstants**.**SixtyFourKb**)]  
**public** **void** **Benchmark\_Performance\_Memory**()  
{  
    *//Write your code to be benchmarked here*  
}

The MemoryAssertion attribute can be used to specify that you want to restrict the method under test to consume not more than the specified amount of memory in each run of the benchmark. As an example, if the method shown above consumes more that 64KB of memory, the test is considered to have failed.

### 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.