Homework

Task 1. `memoryleak` app (preparation)

We will use the same app as in the previous module, so here is the setup again:

- clone **`git clone https://github.com/sebastienros/memoryleak.git`** (great testing app by Sébastien Ros)
- run it (in **`.\src\MemoryLeak\MemoryLeak`**):

```
dotnet run -c Release
```

- **`https://localhost:5001/`** should present a nice introspective graph about memory usage (Working Set), allocations, CPU and current Request Per Second (RPS)
- MemoryLeak exposes some REST endpoints for testing various memory-related scenarios, we will use **`\bigstring`** which just allocates and returns 10KB string. You can test it at **`https://localhost:5001/api/bigstring`**
- you can hit it by F5 many times to observe some memory usage change
- we will make a simple load test agains **`bistring`** endpoint using **`https://github.com/aliostad/SuperBenchmarker`** command-line tool (just download single EXE file from the repository). Run the following command to confirm it is working correctly:

```
.\sb.exe -n 10 -c 1 -u http://localhost:5000/api/bigstring
```

  As you see we use **`http`** endpoint to avoid unnecessary https handshake overhead.

Task 1. `memoryleak` app - counters

- let's make a load test like in the previous module but now we will measure the app with **counters** and **dumps**
- start the **`memoryleak`** app
- look what apps you can measure with the **`dotnet-counters ps`**
- start the load test:

```
❯ .\sb.exe -y 100 -n 10000000 -c 16 -u http://localhost:5000/api/bigstring
```

- (optional) take a sneak peak at the current counters values with the command:

```
❯ dotnet-counters monitor -n MemoryLeak
```

- start collecting counters into CSV format by command (while load test is running):

```
❯ dotnet-counters collect -n MemoryLeak -o task1.csv
```

- wait few minutes and stop the session
- go to the https://www.csvplot.com/ and upload the **`task1.csv`** file
- change to the *Line Chart* in the upper, right corner

Task 1. `memoryleak` app - counters

- how does *"% Time in GC since lat GC (%)"* look?
- how does *"Allocation Rate (B / 1 sec)"* look? Is it stable?
- how does *"ThreadPool Thread Count"* look? Is the number stable?
- how does *"ThreadPool Queue Length"* look? Is there any excessive queuing?
- Let's do some additional sanity checks:
  - check also "type loader"-related counters like *"IL Bytes Jitted (B)"*, *"Number of Assemblies Loaded"* and *"Number of Methods Jitted"* - they should be stable to confirm there is no "dynamic type generation" leak
  - check *"CPU Usage (%)"* to make sure the traffic is not too big and the app is able to handle it
  - also *"ThreadPool Completed Work Item Count (Count / 1 sec)"* should be pretty stable, showing that the application is constantly in a healthy state
  - the last but not the least, check *"Exception Count (Count / 1 sec)"* to make sure there is no excessive exception handling in the app
- select *"GC Heap Size (MB)"* and *"Working Set (MB)"* counters. What's the behavior? Is it aligned to what we've seen under **`https://localhost:5001/`** graph?
- A **bonus**. Select *"Gen 0 Size (B)"*, *"Gen 1 Size (B)"*, *"Gen 2 Size (B)"*, *"LOH Size (B)"* and *"POH Size (B)"* counters. Although we haven't touched the topic of generations, yet - just look what are the reported sizes? What's the maximum?

Task 2. `memoryleak` app - dumps

Now, let's observe with counters an app while taking **`gcdump`**! Restart the app (just to have clear state), rerun the load test and start counters session again:

```
❯ dotnet-counters collect -n MemoryLeak -o task2.csv
```

Now, in another console issue a command to take a heap dump:

```
❯ dotnet-gcdump collect -n MemoryLeak
```

Open recorded **`task2.csv`** in a tool like https://www.csvplot.com/. The moment of taking a dump should be clearly visible? How?

As a result from the **`dotnet-gcdump`** command we should have a file like **`20211126_173518_19000.gcdump`**. Open it in PerfView and open (the only one) *Heap Stacks* view. There, look around in the *RefTree* tab that allows to top-down analysis of the memory usage. What's taking up the most memory?

Is there something "not reachable" in the Managed Heap? Clear **`[not reachable from roots]`** text from **`ExcPats`** input at the top of the *Heap Stacks* dialog.

(Optional) Open **`.gcdump`** file in Visual Studio to see how it is presented there.

(optional) Task 3. `memoryleak` app in Docker + `dotnet-monitor`

This module require you have Docker installed on your machine. Use the module's repository to get **`memoryleak`** version with Docker file prepared and run it:

```
❯ git clone https://github.com/sebastienros/memoryleak.git
❯ docker build --pull -t memoryleak-net50 -f Dockerfile.net50 .
```

Create shared volume to represent **`/tmp`** folder (used by the IPC communication of diagnostic protocol):

```
❯ docker volume create dotnet-tmp
```

Now, run the container with some additional limits (**`cpus`** and **`memory`**), also mounting just created shared volume:

```
❯ docker run -it --rm -p 5000:80 --cpus=4 --memory=1GB --mount "source=dotnet-tmp,target=/tmp" memoryleak-net50
```

Go to **`http://localhost:5000/`** to confirm it is working as expected.

(optional) Task 3. `memoryleak` app in Docker + `dotnet-monitor`

Now, in separate console download and run **`dotnet-monitor`** container:

```
❯ docker run -it --rm -p 52323:52323 --mount "source=dotnet-tmp,target=/tmp" \
            mcr.microsoft.com/dotnet/monitor --urls http://*:52323 --no-auth
```

Visit **`http://localhost:52323/processes`** to confirm there is only a single process listed (with **`isDefautl:true`**) - this is the process from the application container, thanks to the sharing of **`/tmp`** folder. You can also visit **`http://localhost:52323/info`** to see information about the tool itself - **`diagnosticPortMode`** should be **`Connect`**.

Play with the endpoints listed in the Announcing dotnet monitor in .NET 6 article.

You can also use the environment variables, for example to change GC:

```
❯ docker run -it --rm -p 5000:80 --cpus=4 --memory=1GB -e COMPlus_gcServer=0 memoryleak-net50-image
```