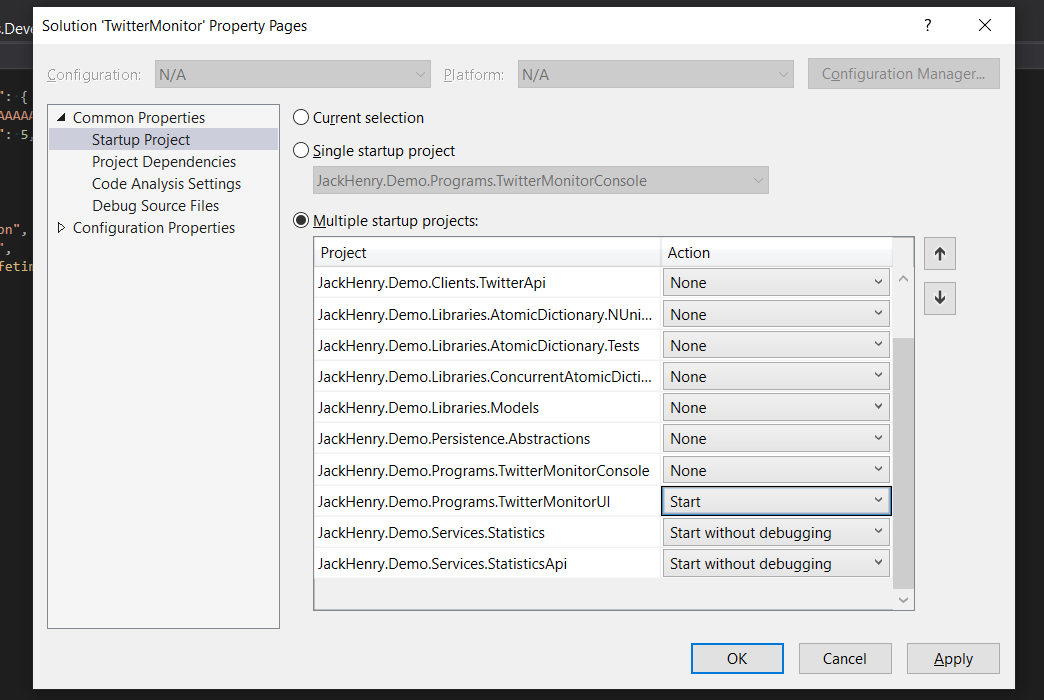
To run full demo:

* Restore and build all projects
* Right-click the solution and select “Set start up projects”
* Choose “Multiple startup projects”
* Set the three main projects to run as follows:



My primary techniques to keep the code responsive and achieve scaling were:

* Multi-threading
* Use of concurrent collections to allow multiple threads to operate in a thread-safe way
* Use of separate processes to for collecting, storing, and displaying the data to allow the individual components to be scaled vertically independent of one another
* (Configurable) Batching of collected tweets to minimize thread-switching and inter-process communication
* Summarized data - individual tweet objects are counted but not retained to minimize memory pressure
* Balanced processing - the data collection code does some processing but the UI also derives the stats it needs to display from the summarized data to spread the CPU burden between processes
* .NET Core async processing where appropriate

Main projects:

* **JackHenry.Demo.Services.Statistics**: this is mainly an external cache application. I built using the gRPC library in .Net 5, because of its advertised performance. It is exceedingly fast. When run without the debugger attached calls to add or update the in-memory stats from the gRPC client to the server typically return in around a millisecond. It was first time using these tools and while it was somewhat clunky, it works well. I played around with two ways of maintaining the state, a simple wrapper around a System.Collections.Concurrent.ConcurrentDictionary (called "ConcurrentAtomicDictionary") and one that maintained two separate ConcurrentDictionary instances one to read from without any blocking at all, and a second instance that gets written to which does block while being written to. periodically -on a configurable interval - the write dictionary is swapped to become the read dictionary, while the read dictionary gets back-filled with all of the data from the old write dictionary and becomes the new write dictionary. So the net effect is that there is never any read blocking and the reads are fast but the read data is behind by whatever increment of swap-time is defined. This was an experiment and it did improve client/server communication time, but the extra complexity is probably not worth it, especially given the lag time. In a real-world application this could all be backed by a redis cache, this was simply a quick replacement for redis so avoid having to set it up as you would on a server or in the cloud.

* **JackHenry.Demo.Services.StatisticsApi**: this is a .Net 5 web app that acts as a service host to my custom 'TwitterMonitorService'. This has the net effect of emulating a windows service, without needing to be installed. This project consumes other .Net 5 library projects to pull sampled data from the twitter API and then push them to the stat collection service. My thought was that if the situation arose, a future version could be rewritten so that the UI got its data from the API rather than the service directly. For performance reasons I currently have the UI accessing the data from the statistic service (which is essentially acting like a redis cache) because it saves a server to server invocation. If business logic were required clients would likely need to be routed through API controllers rather than pulling the data themselves. Projects consumed(/dependencies):
  + JackHenry.Demo.Clients.TwitterApi: this is my wrapper library for connecting to the twitter stream API and pushing out the tweets it receives in batches. The number per batch is configurable. Batching here allows a thread from the thread pool to be used each time the batch size is reached in lieu of running in a different thread for every tweet received. It allows for a configurable, appropriate level of multi-threading so that a balance can be achieved between using the cores available on the serve-class machine and context-switching. It also handles a simple, configurable number of re-try attempts when responses are not received from the Twitter API.
  + JackHenry.Demo.Clients.Statistics: this is a convenience package that facilitates communications over gRPC to the stats service. This allows consuming projects to interact with the stats service through POCO's, without needing to do the translation back and forth into and out of gRPC classes (which are somewhat funky).

* **JackHenry.Demo.Programs.TwitterMonitorUI**: This is a .Net 5 WPF client application that also consumes the JackHenry.Demo.Clients.Statistics library to pull the latest stats from the gRPC service. On a timed basis It pulls the summarized stats from the service using the client, performs some further calculations on them to derive what it will show in the UI, and then the XAML content is data-bound against the latest transformed data. Again, by storing the correctly formatted summary information and pulling that in its raw form from the service the UI layer takes on some of the computational load, distributing it across multiple processed (which of course here are all running on the single host machine)

* **Other projects**
  + JackHenry.Demo.Persistence.Abstractions: this is where a key interface, 'IAtomicDictionary' is defined. I wanted the updates of the component summary data all to be written and read together, which is not the default nature of a hashtable, so I wrapped access to the HT in this interface allowing a set of key/value pairs to be modified as a group before they can be read. I have done two implementations of the interface in the "ConcurrentAtomicDictionary" project and run the project with both. The slight performance gain from the 'swapping' dictionary class does not appear to be significant enough to warrant its associated lag time.
  + The "TwitterMonitorConsole" program was a way to run other aspects of the code easily at the start of the project before setting up the API project.
  + the "…models" class simply contains some POCOs share by multiple projects
  + There are a smattering of unit tests, but with additional time I'd have added many more, particularly around the stat summary integration code in the API project

* **Other notes**
  + I did not code the emojis aspect. I was focused on the separation of concerns, the new gRPC technology, and performance, and can circle back to add that but felt it was more important to move on
  + In any type of production-like environment each of these would be containerized, I did not spend the additional time needed to do that
  + Note that realistically I would never include my twitter client credentials in any source code, however the idea of setting environment variables in a reliable way required too much explanation and overhead, so they are checked in to this repo