Coding Test Summary

Rusty Hann - Zero Allocation Industries

Tooling

I used the following tools. They're my preferred set of tools, and what I purchase through my company, but I can work with any industry standard tool set.

- Visual Studio 2022
    - JetBrains ReSharper
    - CodeMaid
- JetBrains WebStorm
- Visual Studio Code
- PowerShell Core
- Microsoft Edge
    - React Tools Add-On

All Solution Items

I did the following for every item in the solution:

- Upgraded all packages
- Eliminated any warnings and information messages
    - This includes ReSharper refactor recommendations
- I refactored things until there were no warnings
    - A good example of this was changing a lot of the C# classes to static
    - They weren't accessing stateful data, so static is appropriate
    - If they need to maintain state, it's easy to revert them
    - I also changed any test implementations to match the class refactoring
- On the front end side I made sure to add browser specific styles
- I also suppressed the Visual Studio CSS3 definitions / warnings because they're not as up-to-date as the JetBrains definitions

Most of this was done via the tools mentioned above and the .editorconfig I added to the solution. That .editorconfig is not 100% dialed in, but I like to crank warnings, informational messages, and formatting discrepancies (basically everything) up to build errors and then fix all the issues. This is the best way I've found (to date) to write reliable and performant software.

On that note, I usually don't include my .editorconfig in the solution because I it doesn't integrate well with most projects. Part of onboarding anywhere new for me is finding the balance between cranking everything up to 11 and integrating with the team I'm working with.

Algorithms

Format Separators

- I started with a simple solution to get things moving. I turned the array into a list and then manipulated it as needed. The problem is this is slow.
- A marginally faster solution was to reallocate the array at +1 size, bump the previous last item back to last, place and 'and' in the second to last position, and then join each item on a comma. This was ok, but it didn't handle the '*space* and *space* final letter' exceptionally well.
- That's when I decided to address this with spans and ZString. Spans are the hotness in go-fast C# and ZString is a zero allocation string library which is also blazing fast.Together they turn this into three line solution. If you set a span to cover every letter except the last two, you can join the result on a comma using a ZString string builder. You can then append the final letter, space, 'and', final space, and the final letter in two concise statements.
- I really want to run the approaches above through Benchmark.Net to determine exactly how much faster the final solution is.

Factorials

- The first solution I did was the standard recursion algorithm. This is cool, it works, and it's fast. But, it will throw stack overflow errors with sufficiently large values for *n*.
- With large values of n you can use a BigInteger with a for loop or a while loop, which is cool. What would be cooler would be tail recursion. I recently learned about tail recursion while working on an F# project and wanted to see if it could be done in C#. I did a little Google searching and found the links mentioned in the comments. I straight up copied their code, but it worked well, so I cleaned it up and now we have a faster for small integers tha can also handle big integers.
- On that note, the Trampoline / Bounce solution presented is something I will need to sit down and study before I have a significant grasp on what it's doing and why it works. The presented solution is at the edge of my current understanding of C#.

Class Refactoring

- I'm guessing the point was to change 'Swallow' to 'Sparrow'. If there is more to this than I failed it miserably 😀

Container

- I have never implemented my own DI solution, so this was new ground for me. I did the required reading via Google, found a great tutorial, and ran with it. It worked out well.

Frontend

- It's been a while (11 months) since I flexed my React muscles. It was good to get back in the game because if you don't use it you lose it. Also, frontend is where I need to up my game.
- I went ahead and wired everything up so it works, then made the filter work, then did some minimal CSS to make it look ok. It's not a work of art, but it's not a hot mess either. I also added the current date time to the login data because it didn't feel right putting passwords on screen. It's by no means a secure solution, but hitting the basics matters.

Syncing (*There was a lot of depth to these*)

InitializeList

- The InitializeList issue looked like a parallel vs async problem where the parallel and async operations were mixed together, even though they address concurrency in different ways. I chose to split the function up into two. The first fills the concurrent bag using `Parallel.ForEach()` and the second uses Tasks.

InitializeDictionary

- The Dictionary problem was the coolest of the set. I set it up to run with any number of logical processors and then used a span to reach into the initial list. Now that I see it on paper it doesn't look nearly as cool as I thought it looked while coding it, lol.