Name: Shant Ahuja Date: 5.30.20

Lab 7 – Asynchronous Programming

Objectives

Part 1 – Downloading a Large File

Part 2 – Compute the Hashes

Part 3 – Double Extra Credit: Verifying the Signature

Background/Scenario

No matter how amazing or feature-packed the application is, if the user interface stops responding due to a long running calculation or IO operation, the user will not have a good experience and will think the application crashed. If the application's UI stops responding for long enough, Windows might even prompt the user to end the application.

Asynchronous programming is important so you can have a responsive user interface while simultaneously being able to do long running operations. There are many ways to program asynchronously in .NET, but in this lab, we will be focusing on the Task class.

The Task class provides a simple means to have the .NET Runtime do asynchronous work without the need for you to worry about threads and resources. Combining the Task class with the async and await keywords makes programming asynchronously is very simple.

In this lab, you are going to download a large file from the internet and compute its hash checksum to verify the download has not been tampered with. Long web and disk operations are good examples of when you will almost always need to process data asynchronously so as to not bog down the UI thread.

Required Resources

- Visual Studio
- Internet Connection

Part 1: Downloading a Large File

When working with Linux, it is quite common to always need to download large disk images whenever you need to update or reinstall a new distro. Because Linux is an operating system, you want to make sure the disk image you download is not corrupted or has been tampered with. That's why a lot of distros release hash checksums with the download so you can be sure that the files you download are exactly what they built.

Step 1: Access the repository

For this lab, you will be working with the Ubuntu flavor of Linux. The repository URL, for the latest version as of this lab, is here:

```
http://releases.ubuntu.com/focal/
```

Focal is the current version of Ubuntu. If you remove Focal from the repository URL, you can see repositories for prior versions of Ubuntu. Choosing Focal for this lab was arbitrary since all the other versions have the same hash files we will need for this lab.

If you view the repository in a web browser, you will see several files that you can download. The files that we will be interested in are the ISO image, the MD5 checksum, and the SHA checksums. At the time of the posting of the lab, the filenames are:

File to Download	URL
ubuntu-20.04-desktop-amd64.iso	http://releases.ubuntu.com/focal/ubuntu-20.04-desktop-amd64.iso
MD5SUMS	http://releases.ubuntu.com/focal/MD5SUMS
SHA1SUMS	http://releases.ubuntu.com/focal/SHA1SUMS
SHA256SUMS	http://releases.ubuntu.com/focal/SHA256SUMS

Note: If you open any of the hash checksum files, you will see that they provide a checksum for both the desktop and server iso files. For this lab, we only care about the desktop checksum, not the server. Be sure to parse out the correct hash checksum.

Step 2: Download the files.

In the prior labs, you have downloaded JSON data and images from various web services. In most cases, the download happened very quickly and did not really tie up the user interface if you did the downloading in the UI thread. Now that you will be downloading a large file, you will quickly notice that your UI thread will be blocked when you download the ISO. Use the asynchronous functions on the WebClient class to create a Task to download the files listed above.

Provide the user the option to specify where the download files will be saved to.

Step 3: Display a progress notification in the UI.

Display some form of progress notification in the user interface that indicates that a download is happening. Once the download completes, update the UI and let the user know the download finished. Ensure the UI is not blocked at all during the download process.

Part 2: Compute the hashes for the ISO.

Downloading large files can sometimes get interrupted or corrupted and you don't always know you have the file accurate to what the server was distributing. This is the reason why hash checksums are typically distributed along with the files.

Note: Because these ISO files are very large and will take a significant amount of time to download, you should provide a means of selecting an existing file (using an OpenFileDialog box) to compute the hash checksum for rather than downloading the files every time you want to debug the program skipping the download part.

Step 1: Display the hash checksums.

From the previous part, you have downloaded the Linux ISO and three hash checksum files. Parse the hash checksum files and display each checksum in the user interface

Step 2: Calculate the hash checksum.

Calculating the hash checksum is a very computationally expensive process involving a lot of disk activity. You will need to, again, use the Task class to do the hash computation so as to not block the user interface.

> **Warning**: Do NOT read the file's entire contents into memory and store it as a byte[]! This may be ok to do for small files, but when working with large files, you will very quickly run out of memory or worse, get an out of memory exception!

Step 3: Extra Credit: Optimize the hash checksum calculation.

Calculate all three hash checksums simultaneously while only accessing the downloaded file once. If you simply calculate each hash checksum one after another, you will essentially read the file from the disk three times. Keep in mind that if you create three Tasks and run them in parallel, you are also essentially reading the file from disk three times as well. Your goal is to read the file from disk only once.

Step 4: Display a progress notification in the UI.

Display some form of progress notification in the user interface that indicates that the hash checksum is being performed. Once the checksum completes, update the UI and let the user know the calculation has finished. Ensure the UI is not blocked at all during the calculation.

Step 5: Compare the hash checksums.

Compare all three hash checksums and ensure you got the same result. Display the result to the user.

Part 3: Double Extra Credit: Verify the signature

If you have noticed, there are .gpg files for each hash checksum. To ensure that the hashes come from the source, Ubuntu has posted their public key and signed the hash checksums. Download the public key from the Ubuntu website and verify the signature of the three hashes.

> Note: The signatures are generated with GPG and not RSA so you won't be able to use the RSA classes to verify this signature. (That's why it's double extra credit!)

(Reflection continued on the next page)

Reflection

1. Hash checksums are a good way to ensure file integrity, with Linux distros being just one example. Search the internet for any other places where the web services provides hash checksums. What are some other services that provide hash checksums to verify integrity?

   Windows Checksum Utility, online checksum calculator

2. Did you make use of the await keyword in the lab? If so, where did you use it? If not, why did you decide not to?

   yes, I used it while check sums.

3. Was this your first time using asynchronous programming to perform long running operations that do not block the UI thread? If not, what other asynchronous methods (e.g. multithreading) have you used before and how does it compare? If this was your first time, how did working with tasks compare to programming synchronously (procedurally)?

   yes it was. It works well, I think it's cool that you can do multiple things at once. can come in handy

Appendix A – Sample UI

Sample UI

Like previous labs, a here is a sample user interface to give ideas on how to create your UI. While you can copy this UI exactly, feel free to experiment and have fun with designing your own UI that makes use of some new WPF features that you have never used before.