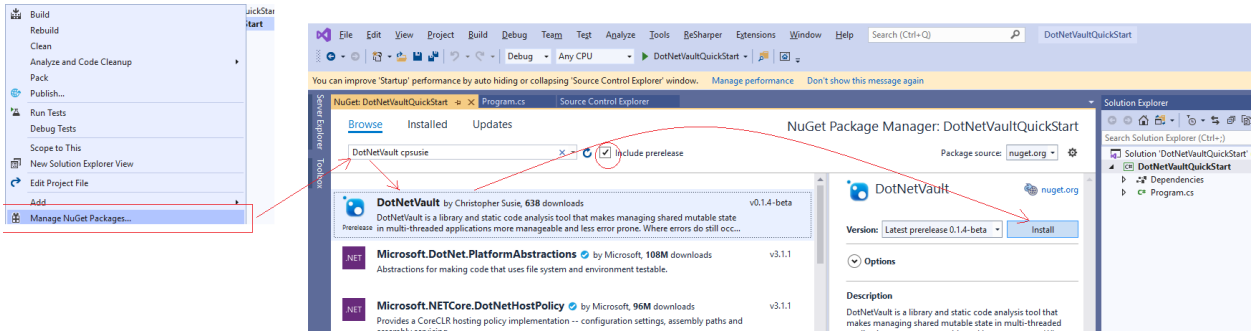
DotNetVault Quick Start Guide – Visual Studio 2019

1- Open visual studio and Create a new .NET Core 3.0+ or .NET Framework 4.8 Console Application.

2- **If you chose .NET Framework 4.8**, open the .csproj file and add the following Line `<LangVersion>8.0</LangVersion>` under each “Platform/Config” PropertyGroup as shown in highlight below. This is unnecessary if you chose a .NET Core 3.0+ based Console Application.

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  <PlatformTarget>AnyCPU</PlatformTarget>
  <DebugSymbols>true</DebugSymbols>
  <DebugType>full</DebugType>
  <Optimize>false</Optimize>
  <OutputPath>bin\Debug\</OutputPath>
  <DefineConstants>DEBUG;TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <LangVersion>8.0</LangVersion>
  <WarningLevel>4</WarningLevel>
</PropertyGroup>
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
  <PlatformTarget>AnyCPU</PlatformTarget>
  <DebugType>pdbonly</DebugType>
  <Optimize>true</Optimize>
  <OutputPath>bin\Release\</OutputPath>
  <DefineConstants>TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <LangVersion>8.0</LangVersion>
  <WarningLevel>4</WarningLevel>
</PropertyGroup>
```

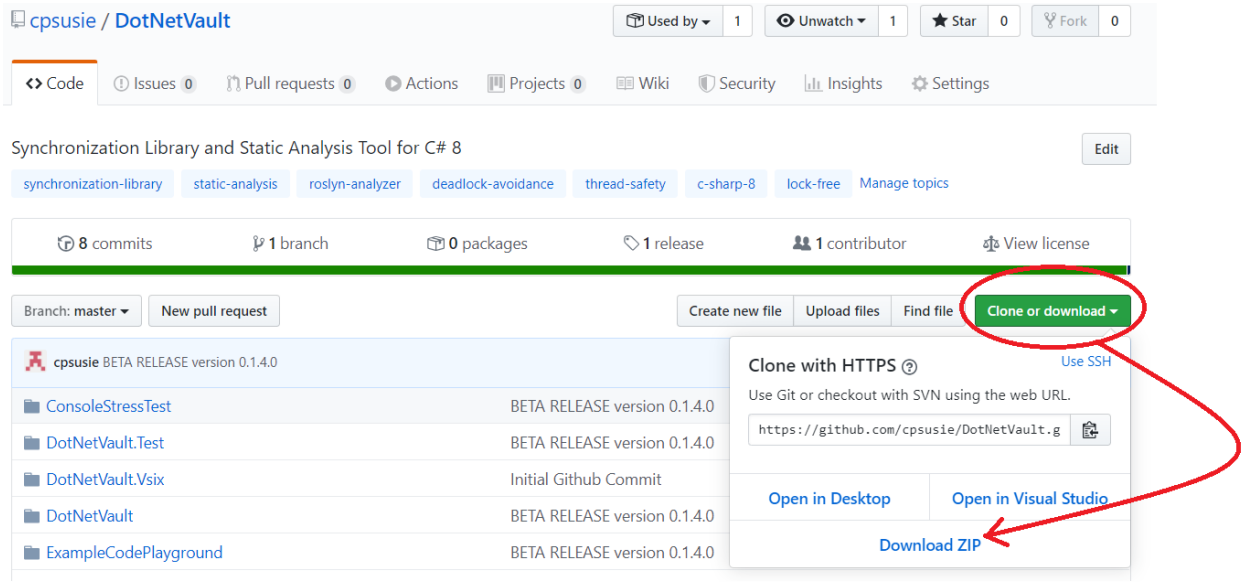
3- Right click on your project and chose “Manage NuGet packages”. Click on the Browse option, select “Include Prerelease”, enter “cpsusie dotnetvault” into the search block, select DotNetVault, then click “Install”



4- You are now set up to use DotNetVault in your project.

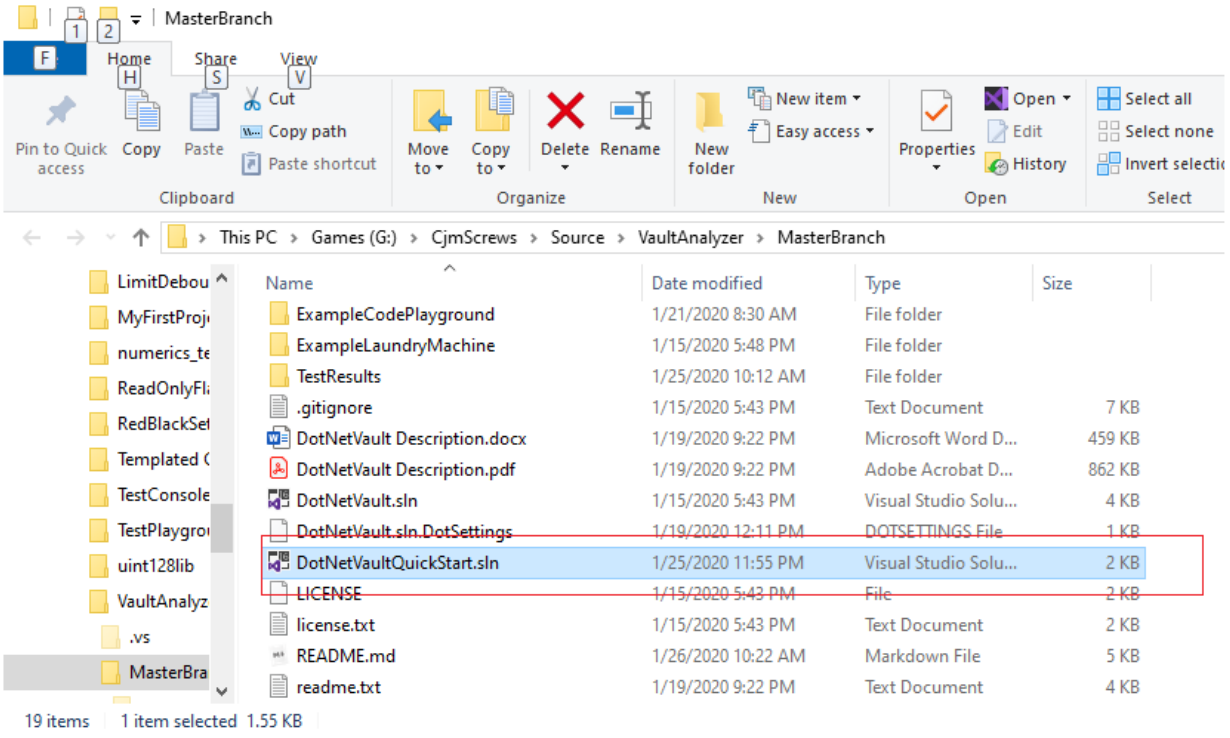
5- To see what can be done:

a. download the source code for DotNetVault from GitHub.

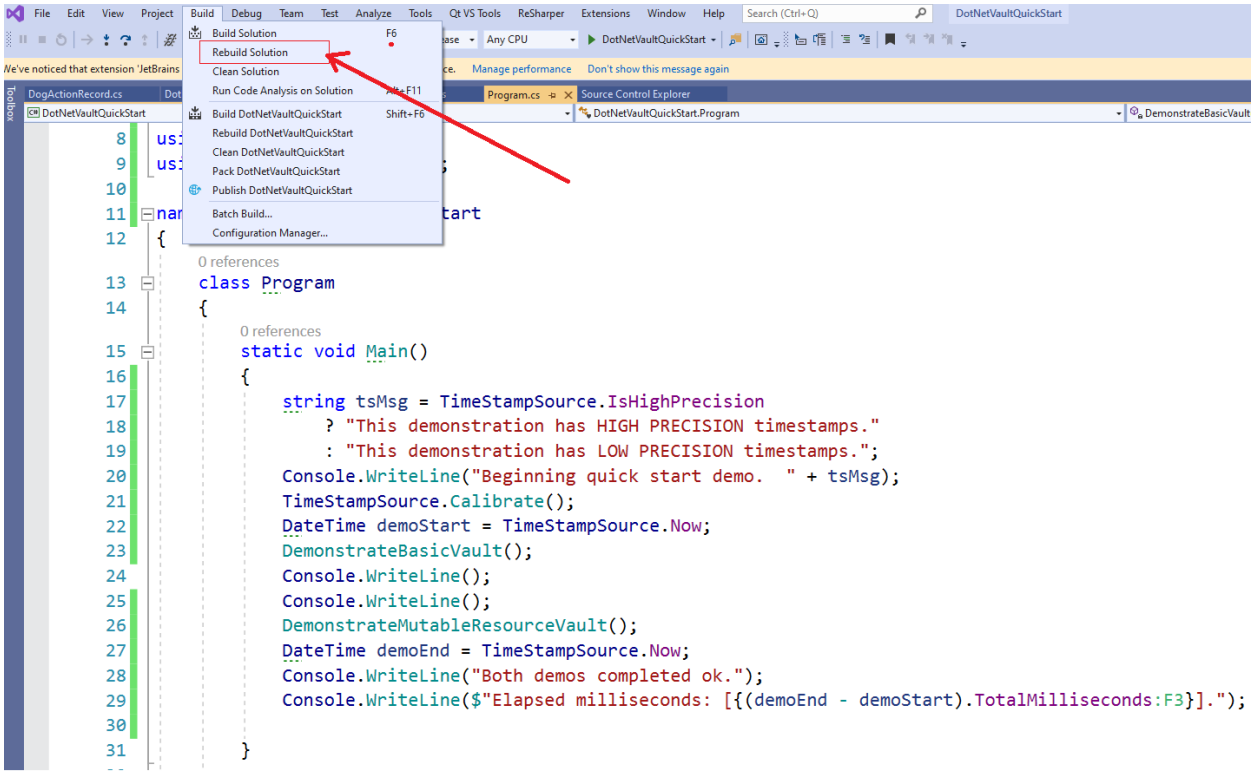


b. Extract the zip file contents to a folder in a convenient place

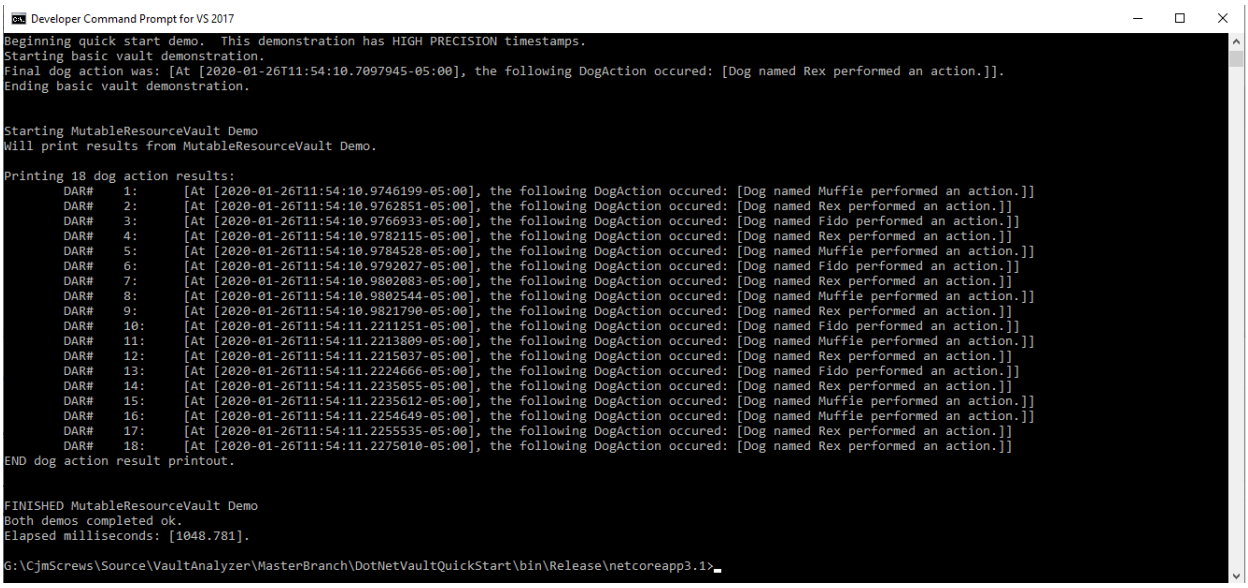
c. Go into the folder structure and open the file “DotNetVaultQuickStart.sln” using Visual Studio 2019



d. Build the solution.



e. Run the solution and check output, which should be something like this:



f. As you can see, from Main(), the demo demonstrates the use of the BasicVault, which stores VaultSafe objects and the MutableResourceVault which stores objects that are not VaultSafe.

i. VaultSafe objects are ones that do not require effort to keep isolated. They include
    1. Unmanaged value types (e.g. long, enums, DateTime, TimeSpan, etc.)
    2. Sealed immutable reference types that are annotated with the VaultSafe attribute (string is automatically considered vault-safe; Immutable collections from System.Collections.Immutable that have only vault-safe type arguments are also considered vault-safe automatically).
    3. Value types that are annotated with the VaultSafe attribute and contain only other types that comply with #1, 2 and 3
    4. Unmanaged value types, strings and qualifying immutable collections are considered VaultSafe without need for annotation with the VaultSafe attribute.

    These objects are easy to isolate because copies of the are either true deep copies (unmanaged types), totally immutable reference types (no danger of a stored reference changing the protected value), or value types that contain only types that are unmanaged value types and immutable reference types. The resource protected in the `DemonstrateBasicVault()` in *Program.cs* is a *DogActionRecord*, found in DogActionRecord.cs.

    DogActionRecord is VaultSafe because:

    1. It is annotated with the VaultSafe attribute
    2. It is a value type with field members that include
        a. An unmanaged value type (DateTime) and
        b. A sealed immutable reference type (string)

g. The `DemonstrateBasicVault()` in *Program.cs* creates a BasicVault<DogActionRecord>, a vault that protects a DogActionRecord and can be used to obtain locks for synchronized access to the value. It then creates a List of Dogs (Dog.cs) then calls their `DoDogActions()` methods, which causes each to spawn a thread that obtains a lock on the vault and overwrites the value stored therein with an action.

The syntax is straight forward:

```
while (numActions-- > 0)
{
    {
        //using the basicvault's locked resource object is easy: get or set.
        //we don't have to worry about copying the value out of the vault -- because it is vault-saf
        //the value is effectively a deep copy (and the string member is immutable and sealed).
        //The copy cannot be used to affect the value stored in the vault after the lock is released
        using var lck = _recordVault.SpinLock();
        lck.Value = new DogActionRecord($"Dog named {Name} performed an action.");
    }
    Thread.Sleep(TimeSpan.FromMilliseconds(1));
}
```

Figure 1

At the end of the demonstration, the main thread obtains the lock and prints out the *DogActionRecord* that happens to be there (because the order is non-deterministic, there will be different results):

```
//SpinLock is a busy wait; Lock sleeps.
//You will not deadlock -- If you cannot obtain the resource within the specified time period,
//an exception of type TimeoutException may be thrown.  The parameterless Lock and SpinLock
//methods use a default timeout period.  You may also specify a positive timespan as your own
//timeout period.  Alternatively, you may supply a cancellationtoken, either alone or in conjunctior
//with the timeout period.  If just the cancellation token is supplied, attempts to obtain the lock
//will continue until it is obtained or cancellation request is propagated to token.  If token and
timespan
//are supplied attempts will continue until the earlier of:
// 1- resource is obtained
// 2- cancel request propagated to token
// 3- timeout period expires
using var lck = actionVault.SpinLock();
Console.WriteLine($"Final dog action was: [{lck.Value.ToString()}].");
```

Figure 2

h. The `DemonstrateMutableResourceVault()` method in Program.cs shows protection of a resource that IS NOT vault-safe. It is a SortedSet of DogActionRecords. The dogs in this demo, instead of overwriting a single DogActionRecord, will add their DogActionRecords to the SortedSet (which maintains them ordered by timestamp, convenient for printing in order at end).

   Resources that are not VaultSafe are protected by a MutableResourceVault, the lock objects of which are more restrictive because they need to make sure that only VaultSafe types are passed into the protected resource or received out from the protected resource; otherwise, mutable state (to which a reference may exist outside) could mingle with the protected resource or mutable state inside the protected resource could leak to the outside and cause unsynchronized access to the protected resource.

The syntax for creating a MutableResourceVault is shown:

```
//You supply the ctor for the protected resource to the mutable resource vault factory,
//that way the resource vault will create it and there is no way to get to it outside the vault
//do not use a lambda that simply provides access to existing object: there is no reasonable
//way for the analyzer to protect you.  Pass the ctor in as a lambda, the vault will create
//and guard the resource.  The provision of the timespan argument is optional.
//If you do not provide one, a default value will be used.
//See comments in the BasicVault demonstration regarding disposing VAULTS
using (var mutableResourceVault =
    MutableResourceVault<SortedSet<DogActionRecord>>.CreateMutableResourceVault(
        () => new SortedSet<DogActionRecord>(), TimeSpan.FromSeconds(1)))
```

Figure 3

Accessing the mutable resource through the lock is mediated by delegates:

```
        while (numActions-- > 0)
        {
            {
                //using the mutable resource vault is a little more tricky.  All inputs to and outputs from the non-vault-safe
                //resource must THEMSELVES be vault-safe even though the protected resource is not.  Thus, access
                //to the protected mutable resource is mediated by delegates with special attributes that are meaningful
                //to the integrated static analyzer.  Using non-vault safe parameters or capturing or referencing non-vault
                //safe values (other than the protected resource itself) is detected by the analyzer, which will refuse
                //to compile the code until you use a vault-safe alternative.  If use of shared mutable state were allowed
                //the analyzer would have no reasonable way to prevent shared mutable state from escaping the vault or
                //shared mutable state accessible from outside the vault from changing values inside of it.
                using var lck = _vault.SpinLock();
                bool addedOk= lck.ExecuteMixedOperation(
                    (ref SortedSet<DogActionRecord> res, in DogActionRecord record) => res.Add(record),
                    new DogActionRecord($"Dog named {Name} performed an action."));
                //See the Project Description Pdf for a full description of the delegates used to update protected mutable resources
                //and for how to use extension methods and/or custom vault objects to make this less cumbersome if used very
                //frequently
                if (!addedOk)
                {
                    Console.Error.WriteLineAsync(
                        $"At {TimeStampSource.Now:O}, a dog action record could not be added to the sorted set.");
                }
            }
            //Help keep the output randomized not dominated by same dog
            Thread.SpinWait(_rgen.Value.Next(1, 2500));
            Thread.Sleep(TimeSpan.FromMilliseconds(1));
        }
```

Figure 4

At the end of the simulation, a lock is acquired that returns the SortedSet to the main thread as an ImmutableSortedSet<DogActionRecord> as shown:

```
    //The resource itself is mutable -- a SortedSet, though the items it contains are VaultSafe.
    //It is not recommended to store COLLECTIONS of items that are not vault safe because there is almost
    //no way to ensure their isolation (and thus freedom from race conditions).  If the resource itself, however,
    //such a string, a DateTime, a DogActionRecord, a long, an enum, etc is VaultSafe itself, it is easy to store
    //them in a mutable collection protected by the a MutableResourceVault.
    ImmutableSortedSet<DogActionRecord> results;
    {
        using var lck = mutableResourceVault.SpinLock();
        results = lck.ExecuteQuery((in SortedSet<DogActionRecord> res) => res.ToImmutableSortedSet());
    } //lock is released here

    finalResults = ProcessResults(results);
}
Console.WriteLine("Will print results from MutableResourceVault Demo.");
Console.WriteLine(Environment.NewLine + finalResults + Environment.NewLine);
Console.WriteLine("FINISHED MutableResourceVault Demo");
```

Figure 5

i. Further resources.
    i. The DotNetVault itself comes with an example of a vault customized to protected StringBuilder resources without resort to the inconvenient delegate syntax. It contains directions for making your own customized vaults and locked resource objects.
    ii. Included in the source repository is the ConsoleStressTest using .NET Core 3.1. It can be run in a Linux or Windows environment and demonstrates the effectiveness of DotNetVault at protecting a mutable resource with a high degree of thread contention.
    iii. The LaundryMachine.sln and the LaundryStressTest project therein requires Windows because it uses WPF. It demonstrates the usage of many vaults and many threads in contention in a highly (unnecessarily so, but good for demonstration purposes) complex multithreaded state machine scenario where LaundryMachines have their own threads and loader and unloader robots (also with their own threads) contend for access to laundry machine. The simulation runs until all soiled articles are cleaned. The loader robots put dirty laundry into one of the machines then start the cycle, the unloader robots take dirty laundry from the machine and put them in the clean bin. The robots constantly contend with each other for access to the machine and since each Laundry Machines state machine thread is independent, the robots also contend with the state machine threads as well as each other to access the machines.
    iv. The ExampleCodePlayground project is used to get a feel for the static analyzer's rules and how they work.
    v. There is a unit test project primarily oriented around ensuring that the static analyzer rules work properly.

j. The DotNetVault Description.pdf provides detailed information on the DotNetVault, the vaults and LockedResourceObjects, its reasoning and static analysis rules as well as examples.