Simio API Note: Entity Data Handling

March 2018 (Dhouck)

Contents

Simio API Note: Entity Data Handling 1
Overview 2
Some Background Information on the Simio Engine 3
Extending Entities with the API 4
Data Share Process Step Code 7
UserElement 7
UserStep 7
Asynchronous vs Synchronous Writing 9
The Singleton Pattern 10
Running the Model 11
Notes on Use 12
The Experiment (multi-thread) Problem. 12
Adding Logic 12

Overview

This API Note describes a way to interact with Entities/Tokens using the API. It was written to address situations where logic or data requirements for Entities might be difficult to implement entirely within the Simio engine. Observe that this Note describes techniques that might be employed with the API. However, it is generally preferred (and often simpler) to stay entirely within the Simio engine.

The techniques discussed here relate to two issues:

1. Provide the ability for an entity to quickly access and/or modify large amounts of data during its lifetime, and/or implement complex logic with .NET code.
2. Demonstrate how to efficiently report Entity results using asynchronous operations.

Again, these techniques are presented as possible options for unique circumstances which involve very complex and real-time projects being developed by clients.

That being said, it also demonstrates the combined power of the Simio Engine and the Simio API to flexibly craft solutions to user problems.

This Note describes some complex programming topics. It assumes that the reader is familiar with C# and .NET technologies such as threading, asynchronous operations, and the Singleton pattern of programming.

Simio
Forward Thinking

Some Background Information on the Simio Engine

The Simio engine (which we'll just call the Engine) is the logic that implements the simulation and planning logic of Simio. When used as a desktop application, it usually includes a UI.

Generally speaking, a simulation run within the Engine is single threaded, meaning that any delays incurred by an Entity affect the whole run. For example, if an Entity decided to do a long-running operation (such as a large synchronous read or write to an external device), the Simio Engine (including the UI) will "hang" for the duration of that operation.

Note: We are here talking about a single "run". When Simio is doing things such as Experiments, it scales by using a separate thread for each Experiment.

Simio provides the run-time API which allows users to create custom Steps (User Extensions). These are implemented as calls to .NET methods that the user can write and are implement by conforming to the published Interfaces. As such, anything that can be done in .NET can be done in these steps. When a Step is executed it is done by a Token, which is a proxy for an Entity (i.e. the green triangles that are seen flowing through the Simio model). These Entities are born, they travel through the simulation, and they are destroyed.

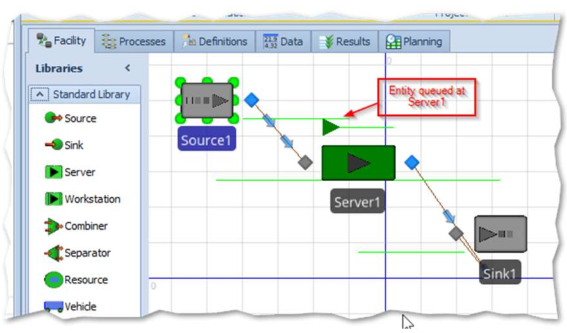


Figure 1 - The Source-Server-Sink Model

Entities are instances of Simio Intelligent Objects, and can share data using the normal Simio mechanism of Properties, State variables, etc. They can also execute logic as the travel through the system using Simio expressions.

![Simio Forward Thinking]

Extending Entities with the API

What if you required more of an Entity than was easy to accomplish with the Engine? This API Note describes an API technique where you can associate a .NET data object of any complexity to an Entity, and also create any native .NET logic to be executed at any given Step.

The diagram below illustrates this technique:

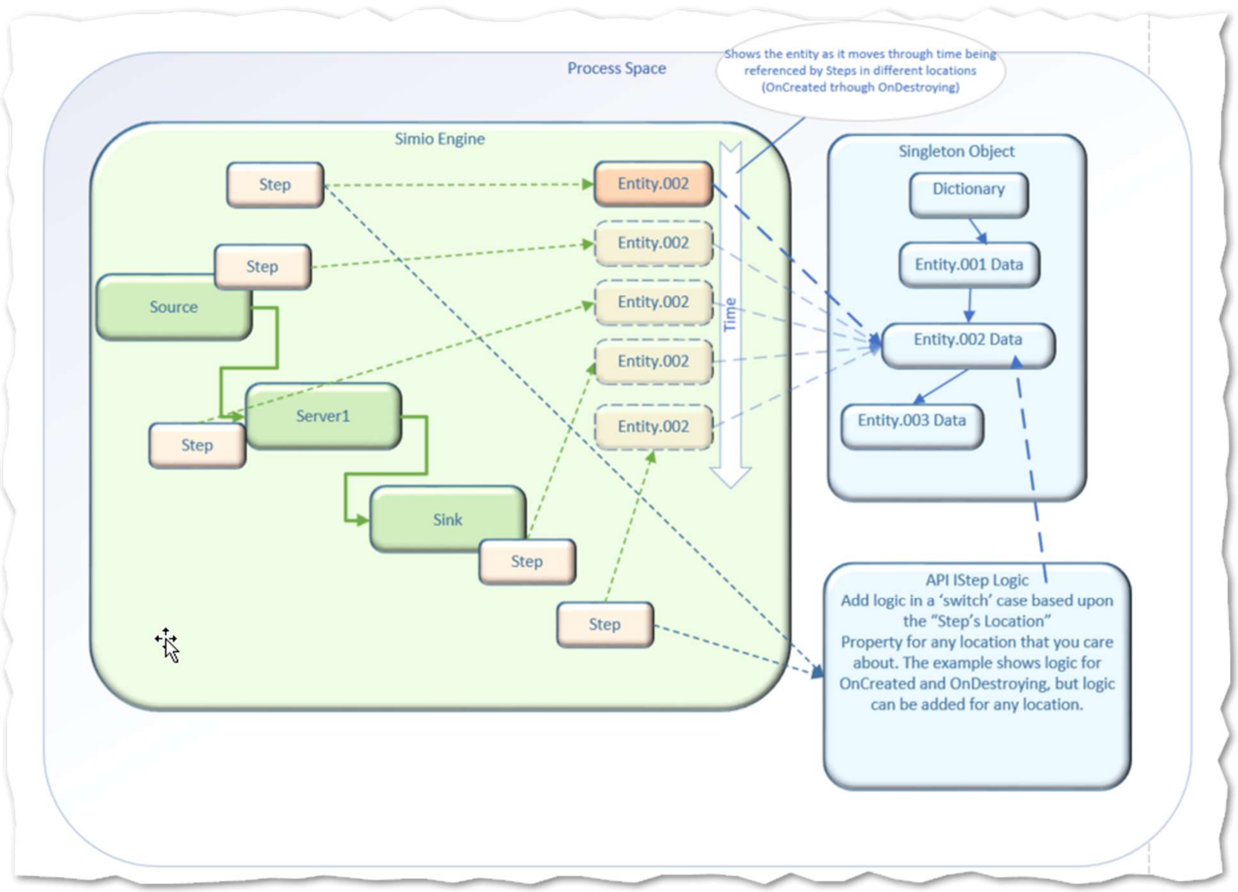


Figure 2 - Attempted Visualization of the Logic

A very basic SourceServerSink (SSS) model has been created. We have added a State variable in the ModelEntity class called WriteOptions so that we can switch from Sync to Async write mode to illustrate the effect of long-running operations.

The custom step (named Data Share Step) will add its own Property called "Step's Location", and will be placed into the model at various points (Processes) within both the ModelEntity and Model.

Here are the (rather arbitrary) locations that we've decided to place this step.

1. OnCreated (ModelEntity)
2. OnDestroying (ModelEntity)
3. Source1_Exited (Model)
4. Server1_BeforeProcessing (Model)
5. Server1_AfterProcessing (Model)

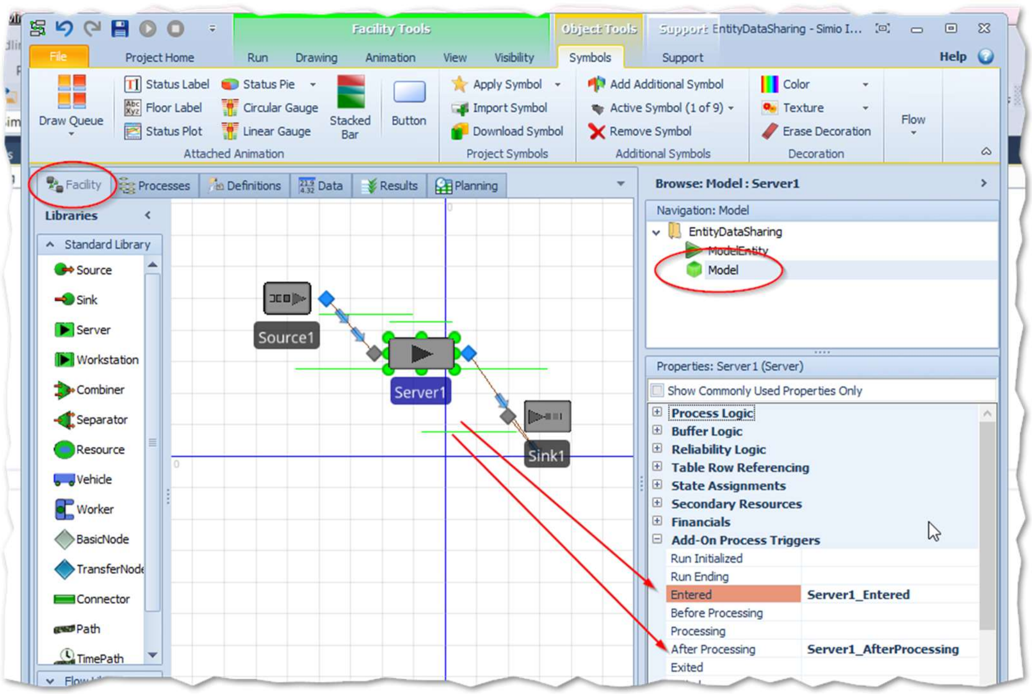


Figure 3 - Adding the Process Triggers to Server1

Each custom Step has a Property set to indicate the location to the called method. By convention, we use the event names above.

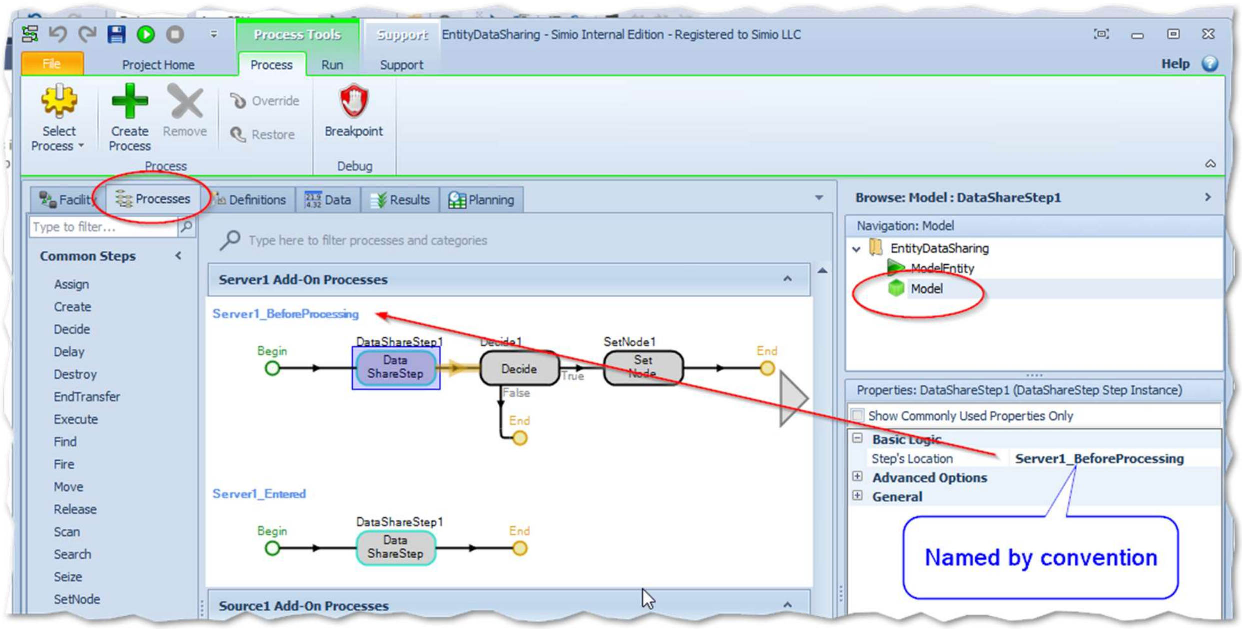


Figure 4 - Adding the location to the Property "Step's Location"

Finally, each Entity as it is being destroyed will issue a long-running “Write” operation, with the choices being Sync, Async, or None.

This option is placed in a WriteOption property:

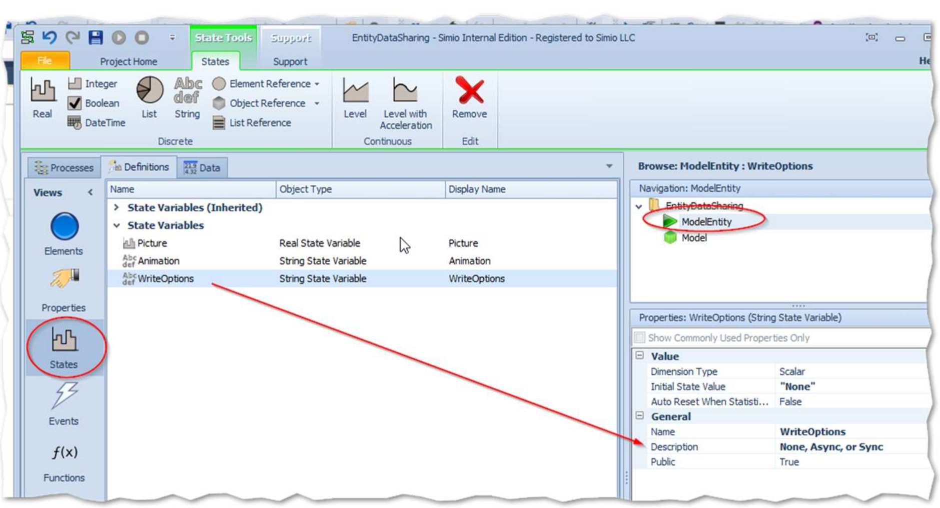


Figure 5 - Entering an option to the ModelEntity WriteOption (Note: the "None" should not be entered with quotes)

Data Share Process Step Code

The code for the Process Step is in the project EntityDataHandling.

UserElement

Since our custom step has a string property called “Step’s Location”, the code knows from whence It is called. Each time we add this Step we will fill in this location property and by convention name it the same as the Process Event (e.g. Server1_Entered). Later in this document we’ll show how we can use this property to build a C# ‘switch’ statement so we can add logic specific to the Step’s location.

```
/// <summary>
/// Method called that defines the property schema for the step.
/// </summary>
public void DefineSchema(IPropertyDefinitions schema)
{
    // Example of how to add a property definition to the step.
    IPropertyDefinition pd;
    pd = schema.AddStringProperty("StepLocation", "??");
    pd.DisplayName = "Step's Location";
    pd.Description = "The Step's location, e.g. On entering.";
    pd.Required = true;

}
```

UserStep

Nearly all the Step logic is in the Execute, which:

1. Uses the “Step’s Location” State variable to determine where we are.
2. Gets a reference to our Entity (Associated Object) and to Entity data which is in a Singleton.
3. Uses our entity name to find our data cache by using a Dictionary object within the Singleton object.
4. Executes “switch” logic based on the location.

If the Entity is being created, then set up our data.

If we are at the “being destroyed”, then demonstrate the efficiency of using an async (asynchronous) technique to ‘write’ the data to an imaginary/emulated device. There is also the choice of doing the same ‘write’ synchronously, which demonstrates how this “locks-up” the Simio engine while the write is going on.

Simio
Forward Thinking

```csharp
            // We're only going to do processing for two locations here, but you
            // can see that any of the identified steps can do processing.
            switch (myLocation)
            {
                case "OnCreated": // Create our data structures
                    {
                        // A one-time build of data
                        if (entityData.DummyList.Count == 0)
                        {
                            entityData.DummyList = new List<DummyDataClass>();

                            // Let's store a bunch of objects for this entity
                            for (int kk = 0; kk < 100000; kk++)
                            {
                                DummyDataClass ddc = new DummyDataClass(kk, (double)kk * kk);
                                entityData.DummyList.Add(ddc);
                            }
                        }

                    };
                    break;
                case "OnDestroying":
                    {
```

Figure 6 - Switch logic for Creating the Entity

During the OnDestroying case there is a test for the Write option. For this example, we don't have logic for the other Locations that this Entity visits.

```csharp
                    break;
                case "OnDestroying":
                    {
                        TimeSpan delta = DateTimeOffset.Now.Subtract(entityData.CreationTime);
                        StringBuilder sb = new StringBuilder($" It took me {delta.TotalMilliseconds} ms to visit: ");
                        var writeOptions = ((IStringState)simioEntity.States["WriteOptions"]).Value;

                        foreach (string ss in entityData.TheStepsIHaveSeen)
                            sb.Append($"{ss} ");

                        // Example of how to display a trace line for the step.
                        context.ExecutionInformation.TraceInformation($"{sb}");

                        // Emulate the writing of the data using a very slow protocol
                        if (writeOptions == "Async")
                        {
                            Task task = SendData(dss, entityData);
                        }
                        else if (writeOptions == "Sync")
                        {
                            SendDataSync(dss, entityData);
                        }
                        else // If neither... don't write, but we better release our memory.
                        {
                            ReleaseMemory(dss, entityData);
                        }
                    };
                    break;
                case "Server1_BeforeProcessing":
                    break;
                case "Server1_AfterProcessing":
                    break;
                case "Source_OnCreating":
                    break;
```

Options

Asynchronous vs Synchronous Writing

The code for our emulated long-running "write" is shown here. To make this long-running we traverse through our 100,000 objects and stop for 10 milliseconds after each 700 objects (clearly, I tinkered with this to get just enough pausing to demonstrate the desired effect without being too annoying).

The topic of programming for asynchronous operations is very complicated (search on ".NET asynchronous" on Stack Overflow to see what I mean). I would recommend Jon Skeet's "C# In Depth" if you wish to delve.

This example shows just one way to launch a write thread when we don't really care about the results. The code within the Task.Run() block is writing asynchronously on a different thread.

```
/// <summary>
/// Emulates a very long-running send operation, followed by a memory release.
/// Note that we don't care about the status of the 'send', but if we
/// did, it should likely be logged (since we model has 'moved on' anyway).
/// </summary>
/// <param name="myData">Entity's data</param>
/// <returns></returns>
private async Task SendData(DataShareSingleton dss, MyEntityData myData)
{
    await Task.Run(() =>
        {
            int nn = 0;
            foreach (DummyDataClass ddc in myData.DummyList)
            {
                if ((nn % 700) == 0)
                    System.Threading.Thread.Sleep(10);

                nn++;
            }

            ReleaseMemory(dss, myData);
        });
}

// Synchronous case for comparison
private void SendDataSync(DataShareSingleton dss, MyEntityData myData)
{
    int nn = 0;
    foreach (DummyDataClass ddc in myData.DummyList)
    {
        if ((nn % 700) == 0)
            System.Threading.Thread.Sleep(10);

        nn++;
    }

    ReleaseMemory(dss, myData);
}
```

Simio
Forward Thinking

The Singleton Pattern

The Singleton pattern references a single object in memory. If it is there, then we use it, and if it is not there (generally this happens only once during startup) we create it.

The particular pattern chosen uses a thread-safe Concurrent Dictionary.

```
/// <summary>
/// A singleton class to hold our data
/// </summary>
9 references
public class DataShareSingleton
{
    private static DataShareSingleton _instance;

    /// <summary>
    /// A place to store information at runtime
    /// </summary>
    6 references
    public ConcurrentDictionary<string,object> RuntimeInfoDict { get; set; }

    /// <summary>
    /// The singleton constructor
    /// </summary>
    1 reference
    private DataShareSingleton() { RuntimeInfoDict = new ConcurrentDictionary<string, object>(); }

    /// <summary>
    /// The singleton pattern implemented.
    /// </summary>
    1 reference
    public static DataShareSingleton Instance
    {
        get
        {
            if ( _instance == null )
            {
                _instance = new DataShareSingleton();
            }
            return _instance;
        }
    }
```

Our Entity Data

if not created, then create it.

Simio
Forward Thinking

Running the Model

The model is the standard (created with the Design Add-In) Source-Server-Sink.

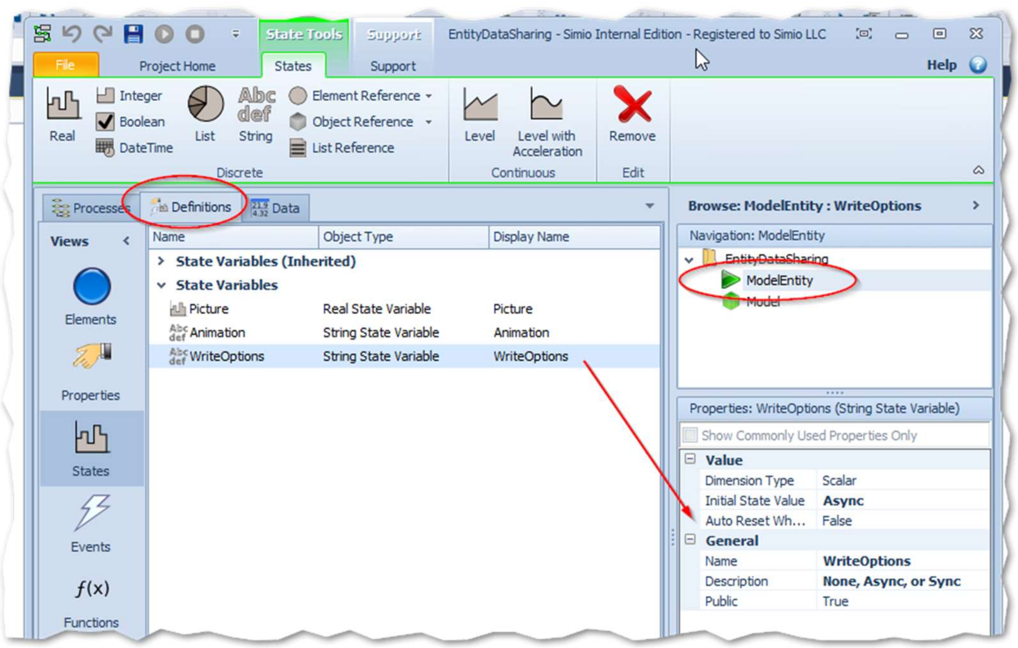
.

As the Entity enters each of our custom steps, we keep a list of where it has been (just for fun). We only have logic for OnCreated and OnDestroying, but you can see any logic could be implemented within the switch statement.

OnCreated Location: As each Entity is created, it is given 100,000 dummy data objects

OnDestroying: As each Entity is about to be destroyed, it does an emulated “write” of those data objects, and then destroys the objects (employing the .NET garbage collector). Note that more efficient techniques for memory management (such as re-using objects) could be employed.

Prior to running the model, you can go to the ModelEntity and select the WriteOptions State.



Run the model with WriteOptions set the Async. Then stop it and try Sync and observe the difference, which is the “hanging” effect that you get when using the Sync option.

Simio
Forward Thinking

Notes on Use

The Experiment (multi-thread) Problem.

Th example in this API Note was based upon a single run. If you wished to run it within an Experiment, then you would have trouble, since each run (replication) within the experiment runs on a different thread and tries to access the same data. You will get exceptions on things like trying to modify an enumeration. It is a good example of the non thread-safe programming.

The simplest solution would be to keep separate data objects for each replication. You could implement this as a dictionary of Replication objects within the Singleton, or a list of Singletons, each with a different name. The name needs to be unique, so you could build it from the ExecutionInformation as you entered the step. For example:

```
/// <summary>
/// Method called when a process token executes the step.
/// </summary>
0 references
public ExitType Execute(IStepExecutionContext context)
{
    // Examine what our AssociatedObject is
    var simioEntity = context.AssociatedObject;

    string name = simioEntity.HierarchicalDisplayName;

    var info = context.ExecutionInformation;
    string runInfo = $"Model={info.ModelName} Experiment={info.ExperimentName} Scenario={info.ScenarioName} Rep#={inf
    string uniqueKey = $"{info.ModelName}:{info.ExperimentName}:{info.ScenarioName}:{info.ReplicationNumber}";
    
```

Adding Logic

When adding logic, you always have to ask yourself “Do I need the result now?”. If the result in needed before you continue out of the Step, then certainly there is no advantage to being async. But if the result is something that isn’t needed right now – for example a calculation of a value that isn’t needed until the end of the run, then consider making it async.

This brings up another point however; when the Simulation run does end, you need to make sure that any async operations you started during the run have completed. There are many ways to do this and it would depend heavily on your situation.