Table of Contents

WELCOME TO STATINATOR

Thanks for purchasing STATINATOR by Cleverous!

This is a basic Getting Started document. The API is pretty straightforward, but there is commentary and explanations in here which you may find useful to understand the reasons behind the implementation and how you can use it to your benefit.

CONTACT INFORMATION AND WEBSITE LINKS

Official Website: **http://www.cleverous.com/**
Support Email: **cleverousdev@gmail.com**

STRUCTURE

Basic stuff about the tool.

STATINATOR provides a back-end for populating and maintaining character stats on entities. The API offers access to "Root" stat values, "Modifier" total values and "True" values for every Stat.

THE STAT AND INTEGRATION

An IUseStats interface is provided and integrates the stats into characters by way of a Stat[] array. These are populated at runtime during initialization based on your target StatPreset asset.

After being created you must call Update on each stat either every frame or when you are certain that there are changes. The example does this every frame. Update will add the Modifier values to the Root values and clamp them to min/max values based on level and growth affinity for that stat.
It is assumed that a Level stat exists. I don't know why you would ever not have one, but don't delete it unless you want to change the calculations.

STAT ENUM CONCEPT AND BASIC CALLS

In concept, there are numerous ways to do stat lookups. You could lookup by string, but it is expensive and clunky. You could hardcode all the variables, but it is restrictive. STATINATOR chooses to lookup by Enum. This is inexpensive and is a good mix of flexibility and maintainability. Long term you may choose to hard code stat arrays when your stats are final, but it would only be necessary if you were seeing significant perf hits due to an excessively abnormal number of Stat users.

A Stat consists of a bunch of StatProperty's. When you want a stat value you will do something like this, assuming you are using the standard implementation:

```
public IUseStats Target;
private float myLevelValue = Target.GetStatValue(StatType.Level);
```

… Where the IUseStats implementation is something like this:

```
public virtual float GetStatValue(StatType stat)
{
   return Stats[(int)stat].Get(StatProperty.Value);
}
```

You'll notice how Stats[] is an ordered array which uses the Enum StatType to cast to an int for the index and call .Get(StatProperty property) on that index. This can be a bit ugly on first glance, but results in a lot of flexibility and is very fast to call.

STATPROPERTY

You can .Get() any StatProperty. The default ones are…
public enum StatProperty { Value, Base, Min, Max, Affinity, MaxAffinity }

Value is the most common one, but you will want the others too. Internally, these are used to handle calculating and clamping Stats correctly.

- **StatProperty.Value**
  - Main value of this Stat
- **StatProperty.Base**
  - Base or Root value. Consider this a pivot origin for calculations. The Value can change dynamically but the Base is more of an anchored, usually safe value.
- **StatProperty.Min**
  - This is the minimum value the Stat is clamped to.
- **StatProperty.Max**
  - This is the maximum value the Stat is clamped to.
- **StatProperty.Affinity**
  - The Value adds this number multiplied by the Level of the Host. Basically, this is added for every level.
- **StatProperty.MaxAffinity**
  - The same as Affinity, but for the Max attribute. You can grow the Max as you level up, for instance increasing Health.Max per level.

STATMODIFIERS

You can attach modifiers to each Stat and it will compute their effects and clean up as instructed. The example shows how this is done with the Buff! button.

StatModifier's are ScriptableObject assets that you can configure to target a specific Stat, Property and infer that it is timed (active) or neverending (direct). You can always strip Modifiers from stats at any time. If you need more complex modifiers, like something that gets weaker over time then you can extend the class and create your own Modifiers which do something different in the GetEffectDelta() method.

Modifiers are attached to and processed on each Stat, not the Host. The host basically just calls Update on Stats and they take care of processing their values, modifiers, etc.

KNOWN ISSUES AND LIMITATIONS

Things you can't do, because reasons.

MODIFYING GLOBAL STAT TYPES

You can add more Stats easily, but since StatPreset uses an array of Stat[] and does not autodetect updates you will have to do this process for adding StatType's:

- Add your stat(s) to StatType enum.
- Change StatUtility.BaseCharacterStats() to include default values for your new stat(s).
- Manually reset every StatPreset, or Stat[] array
- If you miss any, you could get OutOfRange exceptions when stuff queries the Stat[] of an IUseStats.

A solution to add stats without corrupting existing presets is being looked into and there are viable ideas, but it's not solved in version 1.

So basically, you can add stats easily but make sure you do it early in the project, add new stats to the end of the enum (ideally) and record values before you go updating your presets. This is the only real tricky part of the tool, so don't screw it up.