Introduction

Whether you are creating a traditional RPG, a roguelike, an MMO, or even a game with unique mechanics, Simple RPG Core can adapt to your needs. By providing a robust framework for managing attributes, statistics, levels, classes, and more, it allows you to focus on the creative aspects of game development, simplifying the implementation of the more monotonous parts with a scalable and maintainable 100% inspector-driven experience.

The final outcome should look like this: (TODO) add image

Vocabulary of Simple RPG Core

The package is developed around the concept of *entity*, so let's clarify what we mean by this term in the context of Simple RPG Core. In its most minimal version, an entity is a `GameObject` that has a set of statistics. Optionally, an entity can have attributes, can level up, and can have a class. Let's clarify what we mean by each mentioned term.

Statistics (Stat)

A statistic is a value that quantifies an aspect of the entity. The meaning of this aspect is solely due to the concept it refers to.

Examples

In an RPG, a statistic can be `physical damage`. The concept of physical damage refers the player to the amount of damage inflicted by physical attacks, whether with weapons or without. Other statistics can be `ability power`, `defense`, `speed`, `armor penetration`, `range`, etc.

Attributes

An attribute is a value that can influence the value of one or more statistics. The weight of its influence on the statistics can be variable.

Examples

In an RPG, attributes can be: `strength`, `dexterity`, `intelligence`, `constitution`, etc. Considering the previous example of statistics, `strength` could influence `physical damage`, `dexterity` would increase `speed`, `intelligence` would increase `ability power`, and `constitution` would increase `defense`.

Experience and Level

The entity can gain experience and level up. This functionality is used by the class to express how attributes and statistics grow with levels, for that particular class.

Class

The class is associated with a set of statistics and optionally a set of attributes. The class describes how statistics and attributes vary with levels.

Examples

In RPGs most common classes are: `warrior`, `rogue`, `mage`, `paladin`, and so on. These classes have different attribute values. For example, a warrior will have more `strength` and `constitution` than a mage. The `rogue` might have the highest `dexterity`, etc.

How is Simple RPG Core organized and how does it work?

Entity

Script icon: 

A `GameObject` becomes an entity once the `EntityCore` and `EntityStats` `MonoBehaviour`s (Mono) are added to it. `EntityCore` comes with a built-in `EntityLevel` (plain C# `class`) that manages the experience and the level of the entity.

Stat

Script icon: 

A `Stat` is a class that derives from `ScriptableObject` (SO) and represents a statistic in the game. Each statistic has a name (the name given to the SO instance of the created `Stat`), and we can choose whether to provide it with a maximum and/or minimum value. Additionally, we can define how that statistic grows or is reduced in function of certain `Attributes`.

StatSet

Script icon: 

A `StatSet` is a class that derives from SO and defines a set of `Stat`s.

EntityStats

Script icon: 

`EntityStats` allows us to configure:

- the base statistics
- the flat modifiers
- the *StatToStat* modifiers
- the percentage modifiers

We will see what these modifiers are in the section (TODO).

The base statistics can be *fixed*, or instead derive from a class if the entity has one assigned. If we use the fixed ones, we must also provide a `StatSet`, while if we use those of a class, the class's `StatSet` will be used. If the entity levels up and we want its statistics to grow with levels, we are forced to use a class, as the *fixed* statistics are immutable.

Class

Script icon: 

`Class` derives from SO and represents a game class. Each class has a name, a `GrowthFormula` that defines how the base Max HP grows with levels, a `StatSet`, optionally an `AttributeSet`, and associates each `Stat` of the provided StatSet with a `GrowthFormula` that describes how the statistic varies with levels. Similarly, if an `AttributeSet` is provided, it will be possible to associate a `GrowthFormula` for each `Attribute` present in the set, to describe how the attributes vary with levels.

EntityClass

Script icon: 

`EntityClass` derives from Mono and allows us to assign a `Class` to our entity.

Attribute

Script icon: 

An `Attribute` is a class that derives from SO and represents an attribute in the game. Each attribute has a name and, like statistics, can have a maximum and minimum value.

AttributeSet

Script icon: 

An `AttributeSet` is a class that derives from SO and defines a set of `Attribute`s.

EntityAttributes

Script icon: 

Optionally, we can add the Mono `EntityAttributes` to our entity if we want to give it attributes. `EntityAttributes` allows us to specify how many attribute points to provide at each new level. These points can be spent on various attributes to increase their value. For `EntityAttributes` we can configure:

- the base attributes

- the flat modifiers
- the percentage modifiers Similarly to `EntityStats`, we can decide whether the base attributes are *fixed* or if they instead derive from the class associated with `EntityClass`.

Growth Formula

Script icon: 

To express how `Stats`, `Attributes`, Max HP, and the experience required to level up vary at each level, we can use instances of `GrowthFormula`. This is a class that derives from SO and allows us to define a mathematical function, or a system of functions, that describe how a value changes as levels increase. We will see in more detail how to define a `GrowthFormula` in (TODO).

Scaling Formula

Script icon: 

Although we haven't mentioned `ScalingFormula` until now, we briefly introduce it here before discussing it in detail in (TODO).
`ScalingFormula` is a class that derives from SO and allows us to define how a value changes based on other values. In the most common case, the scaling formula is defined in terms of statistics and/or attributes. Each `ScalingFormula` consists of a base value, fixed or defined through a `GrowthFormula`, and a series of `ScalingComponent`s. The `ScalingComponent` define the scaling for a certain type of values. The package provides `StatScalingComponent` and `AttributeScalingComponent`.

`ScalingFormula`s are highly flexible components that can be used in various contexts, such as the damage inflicted by abilities. For example, suppose our character has an ability called `Mace Slam`, which deals `100 + (physical damage * 1.5)` damage. The `ScalingFormula` of `Mace Slam` will have a base damage of 100 and a `StatScalingComponent` that associates the `physical damage` statistic with a 1.5x scaling.

The `ScalingFormula` allows us to insert the various `ScalingComponent`s into two collections: one that refers to the user of the value and one for the potential target. In the previous example, the `StatScalingComponent` referred to the user's collection (of the ability): the higher the `physical damage` of our character, the greater the damage inflicted. Nothing prevents us from adding any `ScalingComponent` based on certain values possessed by the target. For example, we can add a `StatScalingComponent` to the "target" collection that calculates `defense * 0.5` as an additional damage value. Therefore, the higher the defense of the target of our ability, the greater the damage inflicted on it by `Mace Slam`.

Scaling Component

As mentioned in Scaling Formula, it can constitute a part of the `ScalingFormula` to define how the final value scales with one or more values that belong to the same categories. We have seen the

`StatScalingComponent` in the example previously.
It is worth mentioning that the scaling of a `Stat` in function of the `Attributes`, mentioned in the Stat paragraph, is defined through an `AttributeScalingComponent`.

How is Simple RPG Core implemented?

The package is developed following the principles of SOAP (Scriptable Object Architecture Pattern), and has been inspired by the GDC talk of Ryan Hipple. In a nutshell, the main benefits provided by this architecture are:

- **encapsulation**: separation of game logic from data. Game logic code shouldn't mix with data. All data is nicely wrapped withing SO instances
- **game designers friendly**: game designers can make changes and balancements from the inspector without touching the code
- **greater reusability**: Each object is a `ScriptableObject` that can be reused by many components
- **greater testability**: being data separated from code, is easier to isolate and fix bugs. Moreover, SO events can be raised with ease at the press of a button from the inspector interface, easing and speeding up debugging even further.

Flexibility of Simple RPG Core

Although the package is specifically designed for RPG games or games with progression systems, its flexibility allows it to be used in almost any game. As it allows creating attributes like `strength`, `dexterity`, `agility`, etc., and statistics such as `physical damage`, `magic power`, `defense`, etc., in RPG, Roguelike, MMO games, etc., nothing prevents it from being used, for example, to implement a firearm. The attributes could be `weight`, `size`, `ergonomics`, etc., and the statistics `recoil`, `handling`, `stability`, `intimidation`, etc. Attributes can influence statistics. A heavier weapon could reduce `handling` but increase `stability`. A larger weapon could reduce `handling` but increase `intimidation`. A more ergonomic weapon could reduce `recoil` and increase `handling`. And so on... The weapon's levels, if present, influence the attributes and statistics, progressively improving them. Classes could represent weapon types (assault rifles, snipers, shotguns, etc.), and each class could have its own set of dedicated attributes and statistics. For example, shotguns could have, in addition to the aforementioned ones, the `barrel length` attribute that influences the `pellet spread` statistic.

Workflows

Some utilities

Almost every class provided by this package uses events or variables in the form of `ScriptableObject`. Therefore, let's quickly introduce these concepts so that we are clear about what we are talking about when we encounter them in the following paragraphs.

Game events as `ScriptableObjects`

The SOAP architecture allows us to implement the Observer pattern through scriptable objects. In the simplest case, with events without context, we can define various game events as `GameEvent` instances: a class that derives from `ScriptableObject`. For example, we can create an instance called `PlayerJumped` that represents the event "The player has jumped". This event will notify all listening systems when it occurs. Systems subscribe to this event using the `MonoBehaviour GameEventListener`. We can assign a `GameEvent` to this component, and it will handle the subscription and invoke a callback when the event is triggered. The callback is a UnityEvent, so we can select a callback to invoke in response to our event directly from the inspector.

For more details, see the Game Events section.

Int and Long Vars

Another common use of `ScriptableObject` in the SOAP architecture is to define variables. The main advantage of these variables in the form of SO is that they can be easily shared between various objects that may decide to share the same value. A common example is the player's game score. There could be a game manager that adds or removes points from this variable, while the UI HUD uses it to display its value on the screen. This way, we can keep the game manager and UI completely decoupled, passing shared values (like variables) through the inspector.

Int and Long Refs

`IntRef` and `LongRef` allow choosing whether to use a native value (`int` or `long`) or an `IntVar`/`LongVar`. As mentioned in the previous paragraph, `IntVar` and `LongVar` have the advantage of being shareable between different components/game objects, while native values are more immediate to use and require less setup (no need to instantiate an `IntVar`/`LongVar` and assign it in the inspector).

Make a `GameObject` an entity

To make a `GameObject` an entity, we need to add the `MonoBehaviour EntityCore` to it. Select your object from the hierarchy and click, in the inspector, on "Add component". Then search for and select `EntityCore`.

(TODO) Add image of the entity core

From the inspector, we can configure a series of values. Let's analyze them one by one.

`Level`: defines the level of the entity. By changing its value, we can assign a different level to the entity directly from the inspector. This can be useful for testing purposes. You will notice the `Use Constant` checkbox. If checked, you can pass an `IntVar` instead of using a constant. `Current Total Experience`: Represents the total experience possessed by the entity. This value cannot be modified.

Growth Formulas

As already mentioned in Introduction, `GrowthFormula` allows defining how a certain value varies as levels increase. A `GrowthFormula` can be instantiated through the hierarchy context menu by going to `Simple RPG Core -> Growth Formula`. The package provides a custom property drawer for `GrowthFormula`.

For more details, see the Growth Formulas section.

Make a `GameObject` an entity

To make a `GameObject` an entity, we need to add the `MonoBehaviour EntityCore` to it. Select your object from the hierarchy and click, in the inspector, on "Add component". Then search for and select `EntityCore`.

(TODO) Add image of the entity core

From the inspector, we can configure a series of values. Let's analyze them one by one.

`Level`: defines the level of the entity. By changing its value, we can assign a different level to the entity directly from the inspector. This can be useful for testing purposes. You will notice the `Use Constant` checkbox. If checked, you can pass an `IntVar` instead of using a constant.
`Current Total Experience`: Represents the total experience possessed by the entity. Being this a `LongRef`, you can choose whether to use a const value (a native `long`), or a `LongVar` instead.

Game events

The package also supports game events with up to 4 context parameters. They are generics, but in Unity, it is not possible to instantiate classes that derive from ScriptableObject if they are generics with unspecified type parameters. To use them, we must explicitly declare classes that derive from the generic GameEvent and fix the type parameters with concrete types. To simplify the definition of new event types, with specific types as context parameters, the package provides `GameEventGenerator`. These generators, which derive from SO, allow generating the concrete classes of `GameEvent`. We will see these generators in more detail in the section (TODO). Some game events are already defined and made available by the package (see the Samples page).

Int and Long Vars

Int and Long Refs

`IntRef` and `LongRef` allow choosing whether to use a native value (`int` or `long`) or an `IntVar`/`LongVar`. As mentioned in the previous paragraph, `IntVar` and `LongVar` have the advantage of being shareable between different components/game objects, while native values are more immediate to use and require less setup (no need to instantiate an `IntVar`/`LongVar` and assign it in the inspector).

Thanks to a custom property drawer, it will be possible, from the inspector, to check a checkbox named `Use constant` to use a native value instead of a `Ref`, and vice versa.

`IntRef` and `LongRef` are widely used in the package's `MonoBehaviour`.

Growth Formulas

As already mentioned in Introduction, `GrowthFormula` allows defining how a certain value varies as levels increase. A `GrowthFormula` can be instantiated through the hierarchy context menu by going to `Simple RPG Core -> Growth Formula`. The package provides a custom property drawer for `GrowthFormula`.

Max level for the values

In the inspector of a `GrowthFormula`, we can pass an `IntVar` to define up to which level to grow the values.

Use constant at level one

If the checkbox named `Use constant value at level 1` is checked, the respective constant value will be used.

Growth equations

The various values of the `GrowthFormula` are defined by a function where values, the y-axis, are expressed in function of the levels, the x-axis. Such function is defined as a system of equations. Each equation is a string that associates a math expression to a range of levels. The string can be defined by using the Unity ExpressionEvaluator syntax. On top of it, the following terms can be used:

- `LVL`: the level at each iteration
- `PRV`: the previous value of the `GrowthFormula` (value evaluated at the previous level)
- `SPRV`: the second previous value of the `GrowthFormula` (value evaluated 2 levels ago)
- `SUM`: the sum of the values of the `GrowthFormula` from level 1 up to the previous level

Make a `GameObject` an entity

To make a `GameObject` an entity, we need to add the `MonoBehaviour EntityCore` to it. Select your object from the hierarchy and click, in the inspector, on "Add component". Then search for and select `EntityCore`.

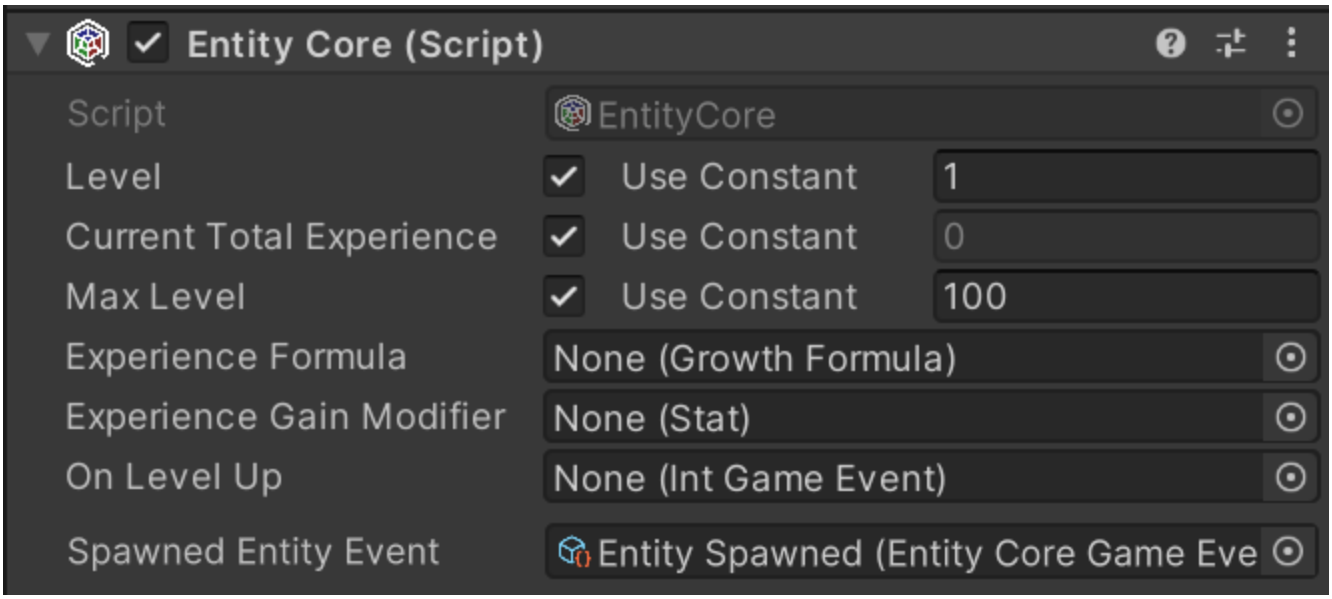


Image - Entity Core Custom Editor

From the inspector, we can configure several values. Let's analyze them one by one.

`Level`: defines the entity's level. By changing its value, we can assign a different level to the entity directly from the inspector. This can be useful for testing purposes. You'll notice the `Use Constant` checkbox. If checked, you can pass an `IntVar` instead of using a constant.

`Current Total Experience`: Represents the total experience possessed by the entity.

> **WARNING**
>
> If you've passed a `LongRef` for the current total experience, the value contained in this variable should not be modified manually. If `Use constant` is checked instead, the value is readonly.

`Max Level`: The maximum level the entity can reach

`Experience Formula`: `GrowthFormula` that describes how the total experience required to reach the next level grows at each level.

`On Level Up`: `IntGameEvent` that should be raised when the entity levels up.

`Spawned Entity Event`: `EntityCoreGameEvent` that should be raised when this entity's `Start()` method is executed.

You may notice that a game event is already assigned to `Spawned Entity Event`. This is because an instance of that game event has been explicitly assigned directly in the inspector of the `EntityCore` script. This choice was made since in most cases the same event instance will always be used for entity spawning. This means you don't have to reassign this event every time you create a new entity in Unity. As we'll see later, this default assignment mechanism has been used for other components as well.

Creating Simple RPG Core assets

All the instances of the various assets that derive from `ScriptableObject`s can be created in the following ways:

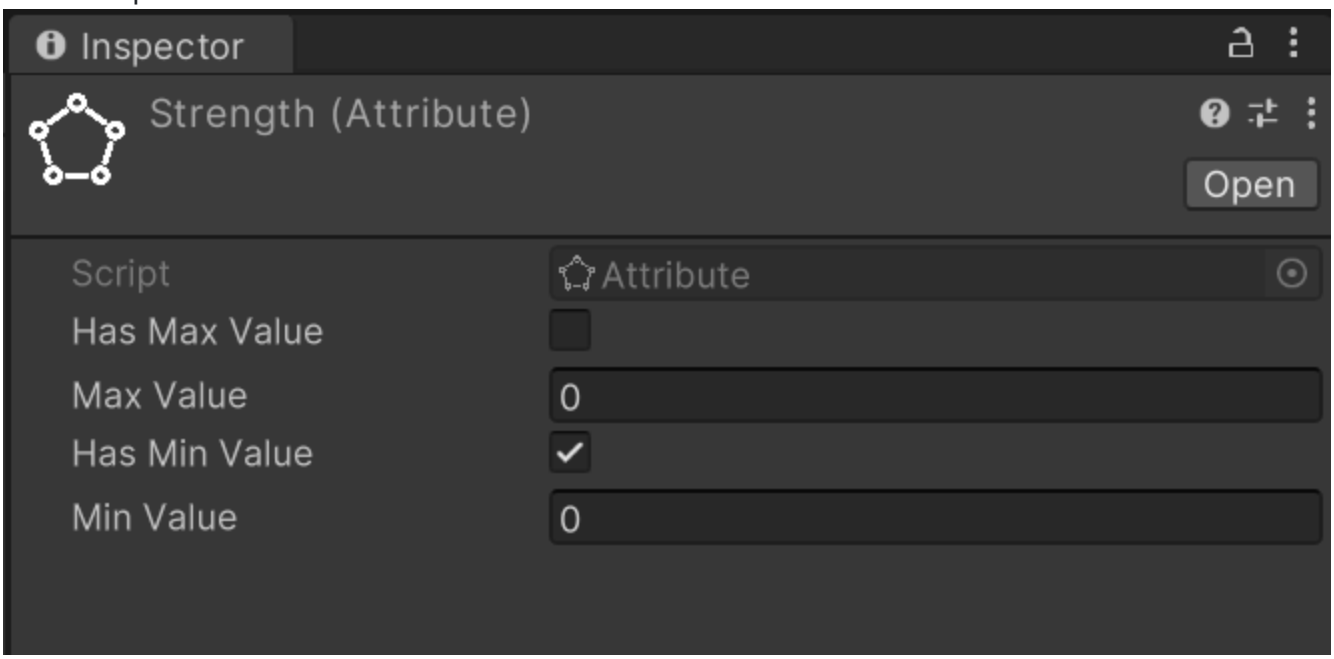
- Context menu: `Right click on the hierarchy > Create > Simple RPG Core`
- Top bar: `Assets > Create > Simple RPG Core`
- Hotkeys: By pressing the respective keyboard shortcut while a folder or an element of the hierarchy is currently selected

> **NOTE**
>
> For Mac users the `Ctrl` key corresponds to the `Cmd` key.

Create attributes

Keyboard shortcut: `A`
Relative path: `Attribute`

Once created a new attribute you can name it as you wish and you'll be able tweak some settings in the inspector. For example lets create a `Strength` attribute. Create an `Attributes` folder in your hierarchy, then press `A` and name the newly created attribute `Strength`.
In the inspector it should look like:



By checking `Has Max Value`, we will set a maximum value for the attribute. By default, there is no maximum value.

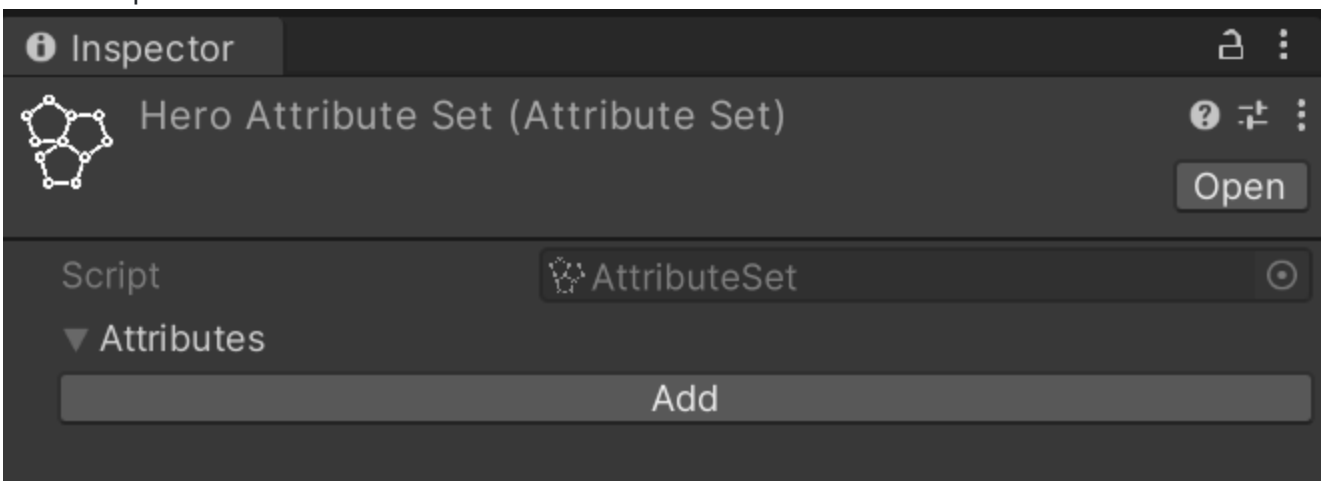
By checking `Has Min Value`, we will set a minimum value for the attribute. By default, the minimum value is zero.

Repeat the process for also the `Constitution`, `Intelligence`, and `Dexterity` attributes.

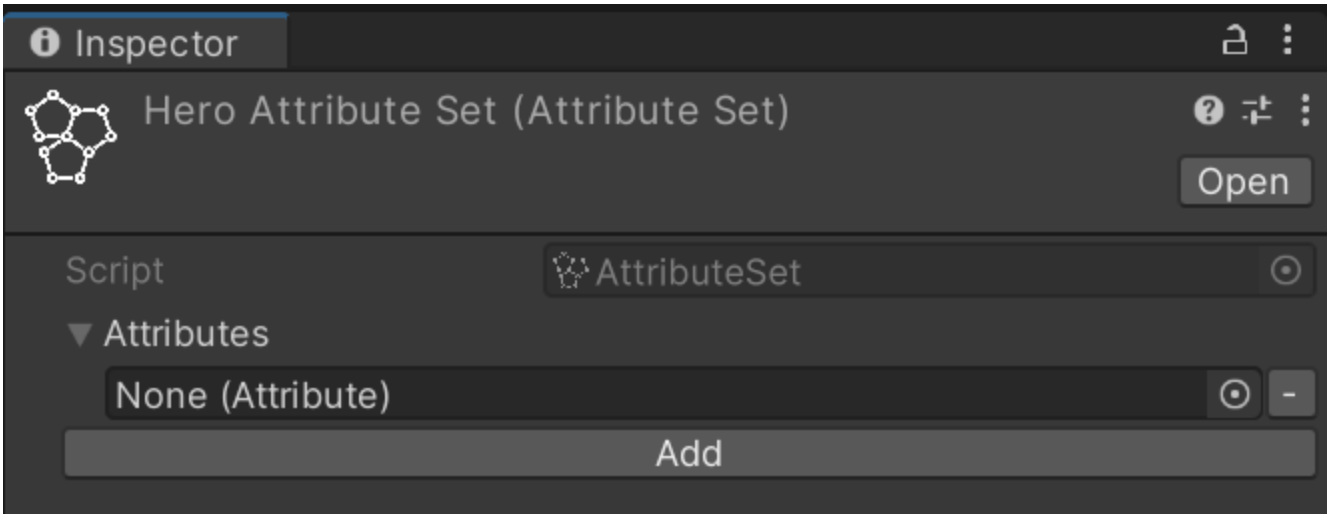
Create an attribute set

Keyboard shortcut: `Alt + A`
Relative path: `Attribute Set`

Now that we have some attributes let's create an `AttributeSet` named, for example, `Hero Attribute Set`. In the inspector it should look like this:



An attribute set without attributes isn't very useful, so let's add the previously created ones, one at a time. To do this, click on the `Add` button. Notice that an entry with `None (Attribute)` appears:



To assign an attribute to the entry, we can either drag&drop from the hierarchy or click on the small circle button on the right of the newly appeared entry. This mechanism is the same used for public variables or, more generally, for fields annotated with `SerializeField`, so it will be familiar to you.
Let's add `Strength` using whichever method you prefer. Repeat the process of adding an attribute to the set for `Constitution`, `Intelligence`, and `Dexterity` as well.

If you want to remove an attribute from the set, you can click on the small - button on the right of the attribute you want to remove.