1. Know your enemy

Constructing NPCs largely always follows the same set of initial steps:

1. Read through any design documentation.
2. Figure out how the behaviour breaks down into distinct states.
3. Identify which parts of the behaviour already exist in predefined components.
4. Identify any parts of the behaviour that are likely to be reused across other similar templates.
5. (Optional) Identify any particularly complex parts of the behaviour that are likely to become generally reusable.

For this tutorial, we're looking at the Goblin Ogre.

Minimum design requirements we were given:

1. A big and slow NPC with a lot of health, found in specific POIs.
2. Can follow a pre-determined path and guard that position.
3. Can sleep and if other goblins annoy the Ogre, there will be a special "sleepy" reaction to that.
4. Can summon rats to eat.
5. Has a melee attack

Note: For the purposes of this tutorial, we'll stick to this design brief, though it may no longer reflect the NPC used in-game.

Step 1: Read the documentation!

This is also a strong suggestion to take a look through the NPC core element documentation and component documentation here, if you haven't already. It provides an overview of all core elements available as sensors/actions/motions/etc. Refer to it anytime you want to know if something is possible and what's needed to achieve it (e.g. picking up items). This will also be updated over time as more features are added!

Step 2: Decide on the states!

The Goblin Ogre isn't too complicated in this regard. We start with the main top-level states:

- An **Idle** state where it mostly remains stationary at a specific point.
    - This can also encompass its general 'observed' flavour actions.
    - In many cases, we can include some inter-NPC behaviours here too, unless they require a more complex sequence of actions.
- A **Sleep** state.
    - Sleep states are often best kept separate from the idle state, both to take advantage of mechanisms for handling the transition from the state to others (e.g. playing wake up animations) and to accommodate slight changes to the NPC's detection capabilities.
- An **Eat** state.
    - With the same reasoning as **Sleep**.
- An **Alerted** state.
    - Almost all NPCs end up having one of these in some form, though they often vary a bit in their intent.
- A **Combat** state.
    - While not always the case, it often makes sense to roll all combat into a single distinct state.
- A **ReturnHome** state.
    - This applies particularly to NPCs that have stationary guard points.
    - Essentially handles getting them home again.

Some of these might break down into a set of additional **substates** handling individual parts of the behavioural logic. For example,

- **Idle**
    - **.Default** (stand guard and do nothing else)
    - **.FindFood** (go search for some nearby food if it exists)
    - **.EatRat** (murder an innocent nearby rat)

Now we have a rough idea of how this NPC is going to be structured at the highest level. With this in mind, we can also see some potential for **state transitions** that make sense.

- **Any State -> Sleep**
    - Play a laydown animation

- **Sleep -> Any State**
    - Play a get up animation
- **Any State -> Eat**
    - Prepare items for eating
- **Eat -> Any State**
    - Pull out weapons again

These particular state transitions are pretty generic and commonly used, but at this stage, I don't see the need for any others.

Step 3: Find existing components we can reuse so we can save ourselves some work!

Straight away, there are a few pre-existing components that will make our lives significantly easier.

- We need to chase and attack other targets. **Component_Instruction_Intelligent_Chase** is built for exactly this purpose and abstracts away a lot of complicated logic needed to make an NPC smartly try to track down a target based on its last seen position.
- **Component_Sensor_Standard_Detection** will help setting up NPC senses, like vision and hearing.
- **Component_Instruction_Soft_Leash** will make sure the ogre doesn't chase its target to the far reaches of Orbis and will eventually give up if the player is just running away.

These will handle a chunk of our generic combat logic. We also need to add supporting files, like an **Appearance** file that describes what model and animations are to be used and an **Attack Interaction** for combat.

The ogre does more than just attack though! Unfortunately, the bulk of the other behaviour is quite specific to the ogre itself. There are existing components for handling sleep states, but this creature needs to be able to swat at other goblins in its sleep so we can't make use of them. The general idle components aren't much use either, but we might be able to handle the simple standing guard by using **Component_Instruction_Intelligent_Idle_Motion_Follow_Path**. There are also some general utility components that might prove useful as we build up the template, such as **Component_Instruction_Damage_Check**.

Step 4 & 5: Identify parts we can reuse in other goblins (or are generally useful to have)

There is a small chance that eating rats and searching for food might be reusable for other goblins, but owing to the uncertainty here, it makes sense to build the NPC without worrying too much about it and then extract the logic out into a component later if required.

3. Getting started with templates

There are no hard and fast rules for making a start on a template. I like to duplicate **BlankTemplate** and start from there because it already provides the basic structure of the file. This leaves me with the following template (**Template_Goblin_Ogre.json**): *(note: the file could be looking different in the current version of the game)*

```
{
 "Type": "Abstract",
   "Parameters": {
    "Appearance": {
      "Value": "Bear_Grizzly",
      "Description": "Model to be used"
    },
    "DropList": {
      "Value": "Empty",
      "Description": "Drop Items"
    },
    "MaxHealth": {
      "Value": 100,
      "Description": "Max health for the NPC"
    },
    "NameTranslationKey": {
      "Value": "server.npcRoles.Template.name",
      "Description": "Translation key for NPC name display"
    }
  },
  "Appearance": { "Compute": "Appearance" },
  "DropList": { "Compute": "DropList" },
  "MaxHealth": { "Compute": "MaxHealth" },
  "MotionControllerList": [
    {
      "Type": "Walk",
      "MaxWalkSpeed": 3,
      "Gravity": 10,
      "MaxFallSpeed": 8,
      "Acceleration": 10
    }
  ],
  "Instructions": [
        {
          "Sensor": {
            "Type": "Any"
          },
          "BodyMotion": {
            "Type": "Nothing"
          }
        }
   ],
  "NameTranslationKey": { "Compute": "NameTranslationKey" }
}
```

And the following variant (**Goblin_Ogre.json**) next to the template. There's a translation key in the template used for localisation of the NPC name, but we won't worry about that for now since it's not required for the NPC to work.

```
{
  "Type": "Variant",
  "Reference": "Template_Goblin_Ogre",
  "Modify": {
    "Appearance": "Goblin_Ogre",
    "MaxHealth": 124
  }
}
```

Our artists have defined the Goblin_Ogre appearance already so I'm going to use that.

4. The importance of being idle

There's no particular rule for the order in which we should tackle states either, but I like to start with the **Idle** state.

First we need to add the state to the template. We do this in two steps.

1. Set up the **Idle** state in the instructions:

```
"Instructions": [
       {
         "Sensor": {
           "Type": "State",
           "State": "Idle"
         },
         "BodyMotion": {
           "Type": "Nothing"
         }
       }
     ]
```

Here we've added the state sensor and corresponding instruction that will hold and define the contents of the **Idle** state.

2. Set the default starting state for the NPC:

```
"Appearance": { "Compute": "Appearance" },
 "DropList": { "Compute": "DropList" },
 "MaxHealth": { "Compute": "MaxHealth" },
 "StartState": "Idle",
```

This is a simple case of adding the **StartState** line alongside the other header fields and setting it to be the **Idle** state.

Now we add the first substate within this, which we already decided would just be a plain **.Default** substate.

```
 "Instructions": [
       {
         "Sensor": {
           "Type": "State",
           "State": "Idle"
         },
        "Instructions": [
           {
             "Sensor": {
               "Type": "State",
               "State": ".Default"
             },
             "Instructions": [
               {
```

```
            }
          ]
        }
      ]
    }
  ]
```

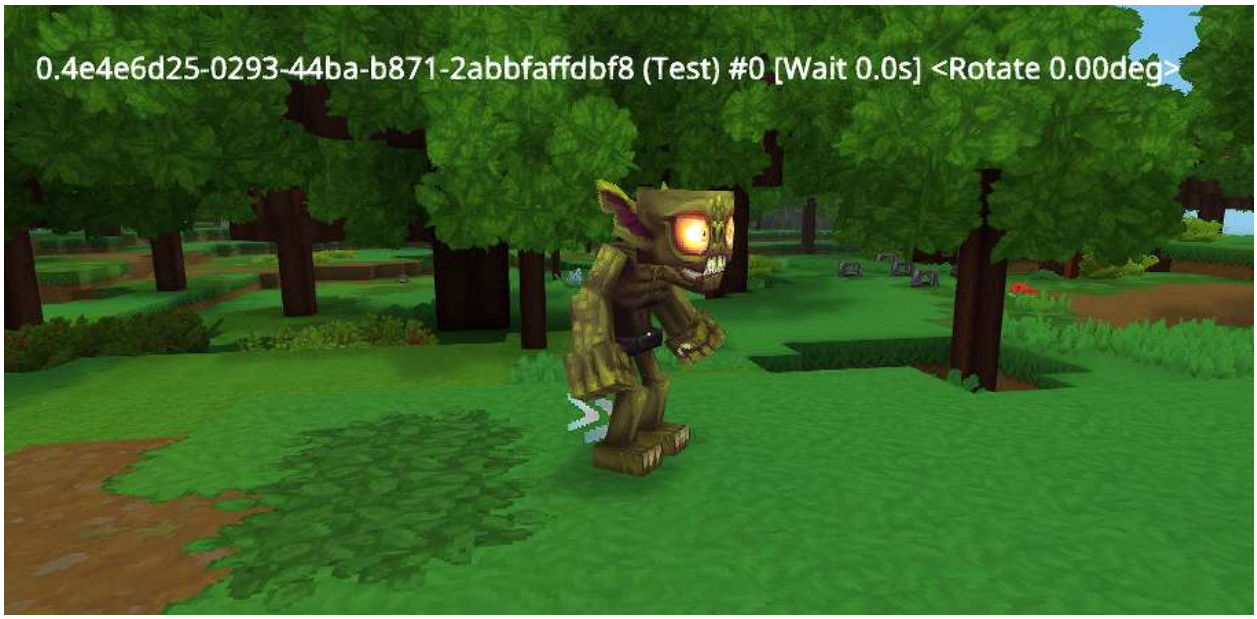
This functions the same way as its parent **Idle** state, but we don't need to specify it as a starting state anywhere because it's using the default name.

The first behaviour we're adding here is the 'protect the entrance of the Goblin POI' behaviour. We already identified an existing component we can use for this, so let's add that immediately.

```
"Instructions": [
        {
          "Sensor": {
            "Type": "State",
            "State": ".Default"
          },
          "Instructions": [
            {
               "Reference": "Component_Instruction_Intelligent_Idle_Motion_Follow_Path"
            }
          ]
        }
      ]
```

If we want, we can use a **Modify** block to set a range within which to follow this path or a particular way to follow it, but since we only want it to stand guard and expect it to spawn near its target marker, we can ignore both of those for now.

Straight away we can jump in-game and spawn our ogre using **/npc spawn Goblin_Ogre**. If we add a single path marker with **/path new Test**, he'll happily go and stand guard there without doing anything else.



And there we go! First idle behaviour is now complete!

5. Drawing the rest of the ogre

We were fortunate, in a sense, that the initial ogre behaviour was simple to handle with existing components. Now the real challenge begins! From here on out, it's critical to remember to test, test, and test some more. **Every time we add a new behaviour it should be tested to ensure it works as expected.** Never leave it until everything is implemented - this just results in unnecessary headaches.

First, we have to figure out how we want to randomise these base idle behaviours. There are three we care about right now since they don't involve interacting with any other types of NPC:

- Standing guard (we already did this!)
- Napping for a while.
- Going off to find some food.

There are a few ways we can approach this, but we'll be using a **Random Action** to make the initial pick and then letting the individual behaviour control its length before resetting and picking a new one.

With this in mind, we can see a minor error in our original judgement - the **Idle** state needs **four** substates, not three:

- **.Default** (this now becomes an entry state for picking the random behaviour state to switch to)
- **.Guard** (this is now our guarding state)
- **.FindFood** (go search for some nearby food if it exists)
- **.EatRat** (murder an innocent nearby rat - we ignore this for now since it relies on another NPC)

So let's quickly refactor the existing **Idle** state to reflect this change, and add a **Random Action** that will, for now, only have a single option.

```
"Instructions": [
          {
            "Sensor": {
              "Type": "State",
              "State": ".Default"
            },
            "Instructions": [
              {
                "Actions": [
                  {
                    "Type": "Random",
                    "Actions": [
                      {
                        "Weight": 100,
                        "Action": {
                          "Type": "State",
                          "State": ".Guard"
                        }
                      }
                    ]
                  }
                ]
              }
            ]
          },
          {
            "Sensor": {
              "Type": "State",
              "State": ".Guard"
            },
            "Instructions": [
              {
                "Reference": "Component_Instruction_Intelligent_Idle_Motion_Follow_Path"
              }
            ]
          }
        ]
```

Looking at the modified **Idle** state instruction, we now have two substates: **.Default** and **.Guard**. If we spawn this NPC next to a path marker, it'll act exactly as it did before. The difference now is that it first executes the **Random** action in the **.Default** substate which picks the only available state action and sends the ogre to **.Guard.** We've set the **Weight** to be 100 for now, but it doesn't matter since there are no other choices anyway and we'll balance these better later (perhaps even for months after initial implementation!)

Now that we have more than one substate, it becomes a little harder to understand what the ogre is doing, so we'll add a **Debug** flag at the top of the file that will tell us what state it's in.

```
"Debug": "DisplayState",
```

Now we can be sure our (presently) friendly goblin ogre is in the correct state.

But wait! There's nothing to send the ogre back to try and do something different!

This is actually pretty simple; we'll just add an instruction with **ActionsBlocking**, a **Timeout**, and **Continue** to the **.Guard** substate that will send it back to **.Default** after a randomised amount of time so it can make a new pick**.**

```
{
      "Sensor": {
        "Type": "State",
        "State": ".Guard"
      },
      "Instructions": [
        {
          "Continue": true,
          "ActionsBlocking": true,
           "Actions": [
           {
              "Type": "Timeout",
              "Delay": [ 15, 30 ]
            },
            {
              "Type": "State",
              "State": ".Default"
            }
          ]
        },
        {
          "Reference": "Component_Instruction_Intelligent_Idle_Motion_Follow_Path"
        }
      ]
 }
```

The **ActionsBlocking** is important here because it ensures we don't execute the switch between states until the first action is complete (the **Timeout**)**.** So now we'll hang around at the path marker for between **fifteen** and **thirty** seconds before going back to pick something else to do.

...not that there's anything else to do yet, so why don't we address that next?

6. Do NPC goblins dream of electric loot?

Let's add a new top level **Sleep** state. For illustrative purposes, the Idle state has been truncated.

```
{
    "Sensor": {
        "Type": "State",
        "State": "Idle"
      },
     "Instructions": [
              . . .
      ]
    },
    {
      "Sensor": {
        "Type": "State",
        "State": "Sleep"
      },
      "Instructions": [
      ]
    }
```

And now we add another action to the **Random** action list to move to the **Sleep** state.

```
      {
          "Sensor": {
            "Type": "State",
            "State": ".Default"
          },
          "Instructions": [
            {
              "Actions": [
                {
                  "Type": "Random",
                  "Actions": [
                    {
                      "Weight": 80,
                      "Action": {
                        "Type": "State",
                        "State": ".Guard"
                      }
                    },
                    {
                      "Weight": 20,
                      "Action": {
                        "Type": "State",
                        "State": "Sleep"
                      }
                    }
                  ]
                }
```

```
                ]
              }
            ]
          },
```

Notice the weights in this case - there's a 20% chance that the ogre will go to sleep any time it decides to switch behaviour, but an 80% chance that it'll just keep standing guard instead.

We still need to actually add the sleeping behaviour, but first we'll add Instruction with the **Timeout** action to the beginning of the **Sleep** state, just like we did with **.Guard.** We just remove **"Continue": true** because we only have one instruction.

7. A brief interlude (or: how I learned to stop worrying and love building NPC components)

We already have the logic to build this component - it's already contained within the **.Guard** state we created earlier. All we need to do is identify which parts of it we need to expose to whoever might want to use this component in the future. That's pretty simple too - we have two:

- The duration of the delay before switching state.
- The state to switch to.

So we put together our new component (**Component_Instruction_State_Timeout**).

```
{
  "Type": "Component",
  "Class": "Instruction",
  "Parameters": {
    "_ImportStates": [ "Main" ],
    "Delay": {
      "Value": [ 3, 5 ],
      "Description": "The amount of time to wait before transitioning"
    }
  },
  "Content": {
    "Continue": true,
    "ActionsBlocking": true,
    "Actions": [
      {
        "Type": "Timeout",
        "Delay": { "Compute": "Delay" }
      },
      {
        "Type": "ParentState",
        "State": "Main"
      }
    ]
  }
}
```

The logic here should be very familiar - the only difference is that we've added and described the parameters in the **Parameters** block, and now use a **ParentState** action to signify that this is going to use one of the **_ImportedStates** from the template itself.

If we would update the template and replace theTimeout block with it, it would look like this, for example, in the **Sleep** state.

```
{
  "Sensor": {
    "Type": "State",
    "State": "Sleep"
  },
  "Instructions": [
      {
        "Reference": "Component_Instruction_State_Timeout",
        "Modify": {
          "_ExportStates": [ "Idle.Default" ],
          "Delay": [ 30, 45 ]
        }
      }
    ]
}
```

We won't replace this everywhere in the tutorial just yet, but we'll use this knowledge to replace even more code with components.

8. An ancient evil ~~awakens~~ hasn't yet fallen asleep

There are still a few more things we need to do before our ogre can get some rest. Until we come to inter-NPC behaviours later, most of this is purely for visual effect and is pretty much entirely restricted to playing animations.

The most basic part is playing the sleep animation. We can use a convenient component for this. Don't forget to put back **"Continue":true**, because now we want to play the animation while the timeout runs.

```
{
          "Sensor": {
            "Type": "State",
            "State": "Sleep"
          },
         "Instructions": [
            {
              "Continue": true,
              "ActionsBlocking": true,
                "Actions": [
                {
                  "Type": "Timeout",
                  "Delay": [ 5, 10 ]
                },
                {
                  "Type": "State",
                  "State": "Idle"
                }
              ]
            },
            {
              "Reference": "Component_Instruction_Play_Animation",
              "Modify": {
                "Animation": "Sleep"
              }
            }
          ]
        }
```

Wait...that's it?

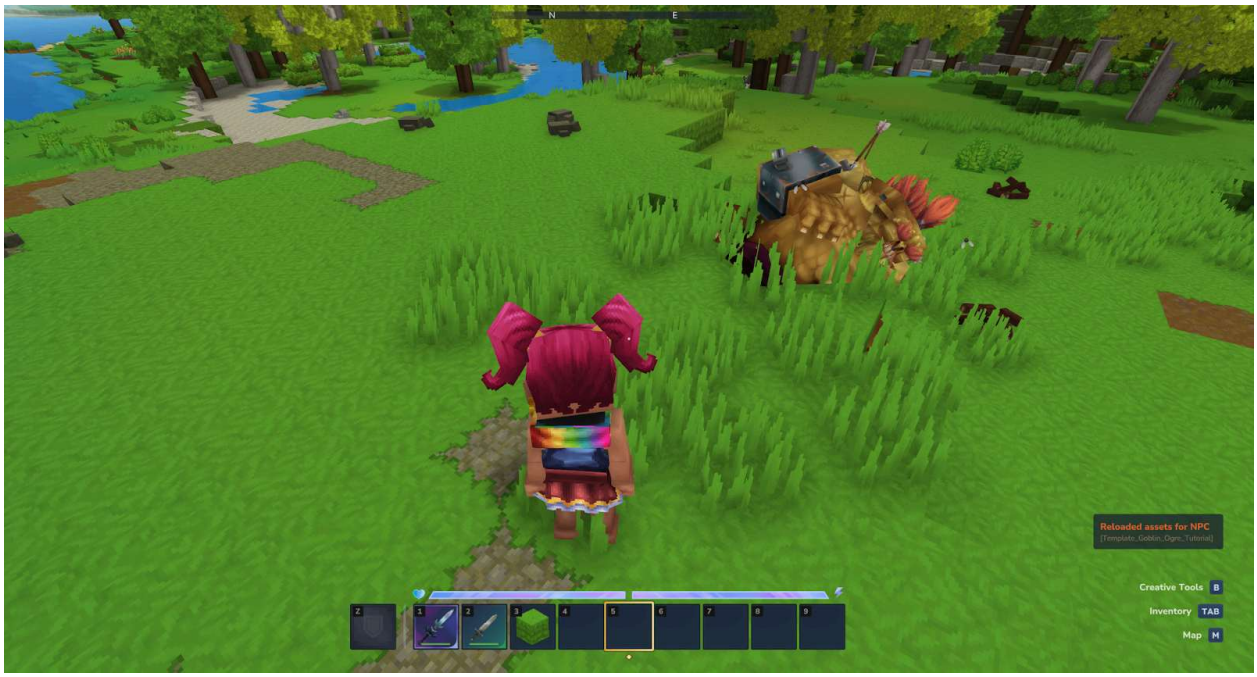
Almost! While our ogre will now happily go to sleep, it's a bit janky. We still need to handle the transitions between states. For this we add a completely new section to the asset above the instructions, called **StateTransitions.**

```
"StateTransitions": [
  {
    "States": [
      {
        "From": [ "Idle" ],
        "To": [ "Sleep" ]
      }
    ],
    "Actions": [
      {
        "Type": "PlayAnimation",
        "Slot": "Status",
        "Animation": "Laydown"
      },
      {
        "Type": "Timeout",
        "Delay": [ 1, 1 ]
      }
    ]
  },
  {
    "States": [
      {
        "From": [ "Sleep" ],
        "To": [ ]
      }
    ],
    "Actions": [
      {
        "Type": "PlayAnimation",
        "Slot": "Status",
        "Animation": "Wake"
      },
      {
        "Type": "Timeout",
        "Delay": [ 1, 1 ]
      }
    ]
  }
],
"Instructions": [
  ...
]
```

State transitions are a bit verbose but relatively simple. Each transition represents a set of actions that will be performed in sequence before we actually start executing the logic of the state itself. An empty array represents all existing states. In this instance, our ogre will play the

Laydown animation when switching from **Idle** to **Sleep** and the **Wake** animation when switching from **Sleep** to any other state**.** We currently have to define a **Timeout** action with a delay the length of the animation to ensure that we don't do anything else until the animation completes. In the future, we might be able to automatically find out the length of the animation and block for its duration, but that's not currently possible**.**

But that's all for the **Sleep** state for now**.** Really this time**.**



9. The ogre who ate everything

Lest we forget, there was another significantly more complex idle behaviour we have to implement:

- Standing guard (we already did this!)
- Napping for a while (this one too!)
- **Going off to find some food.**

This one might be a bit trickier. When we pick this behaviour we need to scan the area to see if there is any food. If we find it, we head over, pretend to grab it, and then set to eating. But what if we *don't* find any food? We should probably still pretend to go looking for it - it's not like this ogre is omniscient and already *knows* there's no food where it expects it because the sensor said there wasn't...right?

We start by creating our **.FindFood** substate immediately after **.Guard.**

```
{
  "Sensor": {
    "Type": "State",
    "State": ".Guard"
  },
  "Instructions": [

      ...
    ]
},
{
  "Sensor": {
   "Type": "State",
    "State": ".FindFood"
    },
   "Instructions": [
    ]
 }
```

We then add it to our list of random actions.

```
            "Type": "Random",
                  "Actions": [
                    {
                      "Weight": 70,
                      "Action": {
                        "Type": "State",
                        "State": ".Guard"
                      }
                    },
                    {
                      "Weight": 20,
                      "Action": {
                        "Type": "State",
                        "State": "Sleep"
                      }
                    },
                    {
                      "Weight": 10,
                      "Action": {
                        "Type": "State",
                        "State": ".FindFood"
                      }
                    }
                  ]
```

Next we want to perform the actual search for food. But what *is* food? And not in a philosophical sense. What kinds of food does this ogre look for? This is another point where we might want to have a chat with the designers. Do they want it to go after multiple different things? Or is there one particular type of food a goblin ogre is interested in?

A quick discussion with our designers reveals we've been working under an incorrect assumption and highlights the importance of ensuring that we thoroughly understand both the contents of the design specifications, and *the intent behind them*. This ogre is meant to be a guard and shouldn't leave his post. He should never actively set off to *find* food, but rather whip something out of a pocket and eat it on the spot.

Okay. That makes things significantly easier! We can now cross off one of our idle substates:

- **.Default** (pick the random behaviour state to switch to)
- **.Guard** (stand guard)
- ~~**.FindFood** (go search for some nearby food if it exists)~~
- **.EatRat** (murder an innocent nearby rat - we're still ignoring this for now)

So let's remove **.FindFood** and jump straight to **Eat**, adding the same logic we added to the other idle states. We also need to remember to update the reference in the list of random actions so that it points to the correct state!

```
      {
        "Sensor": {
          "Type": "State",
          "State": "Eat"
        },
        "Instructions": [
          {
            "ActionsBlocking": true,
              "Actions": [
              {
                "Type": "Timeout",
                "Delay": [ 5, 10 ]
              },
              {
                "Type": "State",
                "State": "Idle"
              }
            ]
          }
        ]
      }
```

We don't actually know for sure what food this ogre has in his pockets and it's probably something that might get changed in the future, or be nice to make *easily changeable* in variants. To accommodate that, let's add an **EatItem** parameter and accompanying description to the parameters block.

```
"Parameters": {
  "Appearance": {
    "Value": "Bear_Grizzly",
    "Description": "Model to be used"
  },
  "EatItem": {
    "Value": "Food_Beef_Raw",
    "Description": "The item this NPC will find when it rummages for food"
  },
  "DropList": {
    "Value": "Empty",
    "Description": "Drop Items"
  }
}
```

Now we just need to set up another **PlayAnimation** action, much like in the **Sleep** state, to handle the eating itself.

```
{
  "Sensor": {
    "Type": "State",
    "State": "Eat"
  },
  "Instructions": [
      {
               "Continue":true,
               "ActionsBlocking": true,
                 "Actions": [
                 {
                   "Type": "Timeout",
                   "Delay": [ 5, 10 ]
                 },
                 {
                   "Type": "State",
                   "State": "Idle"
                 }
               ]
       },

      {
        "Reference": "Component_Instruction_Play_Animation",
        "Modify": {
          "Animation": "Eat"
        }
      }
    ]
  }
}
```

This looks *awfully* familiar…

Could this be something we want to do often in many NPCs? Might we frequently want to play an animation for a specific random duration? I'd say yes, so let's make it a component!

```
{
  "Type": "Component",
  "Class": "Instruction",
  "Parameters": {
    "_ImportStates": [ "Main" ],
    "Animation": {
      "Value": "",
      "Description": "The animation to play"
    },
    "Duration": {
      "Value": [ 3, 5 ],
      "Description": "The amount of time to wait before transitioning"
    }
  },
  "Content": {
    "Continue": true,
    "Instructions": [
      {
        "Reference": "Component_Instruction_State_Timeout",
        "Modify": {
          "_ExportStates": [ "Main" ],
          "Delay": { "Compute": "Duration" }
        }
      },
      {
        "Reference": "Component_Instruction_Play_Animation",
        "Modify": {
          "Animation": { "Compute": "Animation" }
        }
      }
    ]
  }
}
```

I'm calling this one **Component_Instruction_Play_Animation_In_State_For_Duration.** The Timeout instruction is replaced with the component made in the previous interlude. Now we need to apply it to both places in the ogre template.

```
{
  "Sensor": {
    "Type": "State",
    "State": "Eat"
  },
    "Instructions": [
      {
        "Reference": "Component_Instruction_Play_Animation_In_State_For_Duration",
        "Modify": {
          "_ExportStates": [ "Idle.Default" ],
          "Animation": "Eat",
          "Duration": [ 15, 20 ]
        }
      }
    ]
}
```

Only **Eat** is pictured here, but you can imagine the same changes being made to **Sleep** too.

Now why aren't we converting the whole state itself into a component? There's a pretty good reason behind that: we haven't added entity detection yet and we almost surely will. This will need to live in the state alongside the other logic, so we can't exactly push the whole state itself into a component.

There's one last thing we need to do to make this ogre actually eat - take out the food and put it away again. Since we don't want to somehow risk ending up in a state where the ogre is trying to beat its enemies with a chunk of meat or eat its own weapon, we'll do this with state transitions.

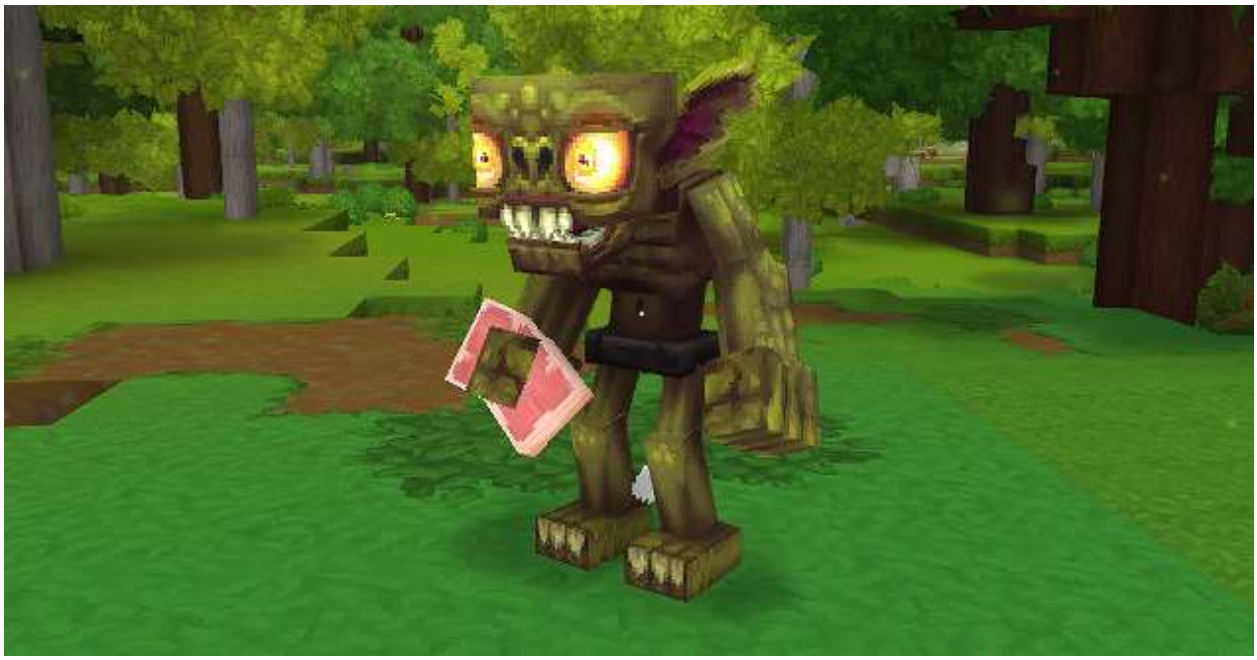
```
"StateTransitions": [
  ...
  {
    "States": [
      {
        "From": [ "Idle" ],
        "To": [ "Eat" ]
      }
    ],
    "Actions": [
      {
        "Type": "Inventory",
        "Operation": "SetHotbar",
        "Item": { "Compute": "EatItem" },
        "Slot": 2,
        "UseTarget": false
      },
      {
        "Type": "Inventory",
        "Operation": "EquipHotbar",
        "Slot": 2,
        "UseTarget": false
      }
    ]
  }
]
```

There's a few important things to note here. This combination of actions will place the item from the **EatItem** parameter we defined earlier into slot 2 of its hotbar, and then switch to using that slot. We have to define it as **UseTarget: false** to ensure that it acts on the ogre itself. By default, NPCs have three hotbar slots, so we've set it to use the last slot for this purpose (slots are numbered starting from zero).

So this handles actually pulling out the food to eat it, but not putting it away again afterwards. Let's add that too.

```
{
  "States": [
    {
      "From": [ "Eat" ],
      "To": []
    }
  ],
  "Actions": [
     {
      "Type": "PlayAnimation",
      "Slot": "Status"
    },
    {
      "Type": "Inventory",
      "Operation": "EquipHotbar",
      "Slot": 0,
      "UseTarget": false
    }
  ]
}
```

Now our ogre will happily pull out a chunk of meat, start eating it, and then put it away again when it's done!



Perfect! The last thing we'll do, just to be sure we don't end up in any strange states if the goblin unloads while eating, is make sure we also switch to the correct weapon at the beginning of the idle state.

```
{
  "Sensor": {
    "Type": "State",
    "State": "Idle"
  },
    "Instructions": [
      {
        "Continue": true,
        "Sensor": {
          "Type": "Any",
          "Once": true
        },
        "Actions": [
          {
            "Type": "Inventory",
            "Operation": "EquipHotbar",
            "Slot": 0,
            "UseTarget": false
          }
        ]
      },
      ...
    ]
  }
```

With that, we have all the idle behaviours that don’t rely on other NPCs set up. We’ll deal with that next!

10. Of rats and goblins (mostly just goblins)

Now we need to start thinking about how to get our ogre to respond to other NPCs in the world. We won't think about combat or dealing with players yet, but there are a couple of other inter-NPC behaviours described in the design specification:

- Whack annoying goblin scrappers while sleeping.
- Grab and eat rats that come too close.

We'll start with the first because there are a number of problems that are going to come into play when we tackle the second.

Whacking goblins takes place during the **Sleep** state, so we'll be doing all our editing there.

We don't actually need to add any detection for this, because we're going to trust the goblin scrappers to *tell* us they're being annoying. This is something pretty neat about using beacons to communicate between NPCs. When one of the NPCs sends a message, it can trigger behaviour in the other. Basically, this means that the **Goblin Scrapper** is going to handle the bulk of this behaviour. When they go into annoy mode, they'll approach the ogre. When they're close enough, they'll send a message telling him they're being annoying, and he'll then randomly swat at them in his sleep.

They're essentially actually annoying him with their message!

```
{
  "Sensor": {
    "Type": "State",
    "State": "Sleep"
  },
    "Instructions": [
      ...
      {
        "Sensor": {
          "Type": "Beacon",
          "Message": "Annoy_Ogre",
          "Range": 5
        },
        "Actions": [

        ]
      }
    ]
}
```

This sensor listens for the **Annoy_Ogre** message and will perform its actions when receiving it, so long as it comes from an NPC that's close by. The nice thing about this is that it can respond to *any* NPC that decides to annoy it! Next we'll add an attack to it. It'll be the animators'

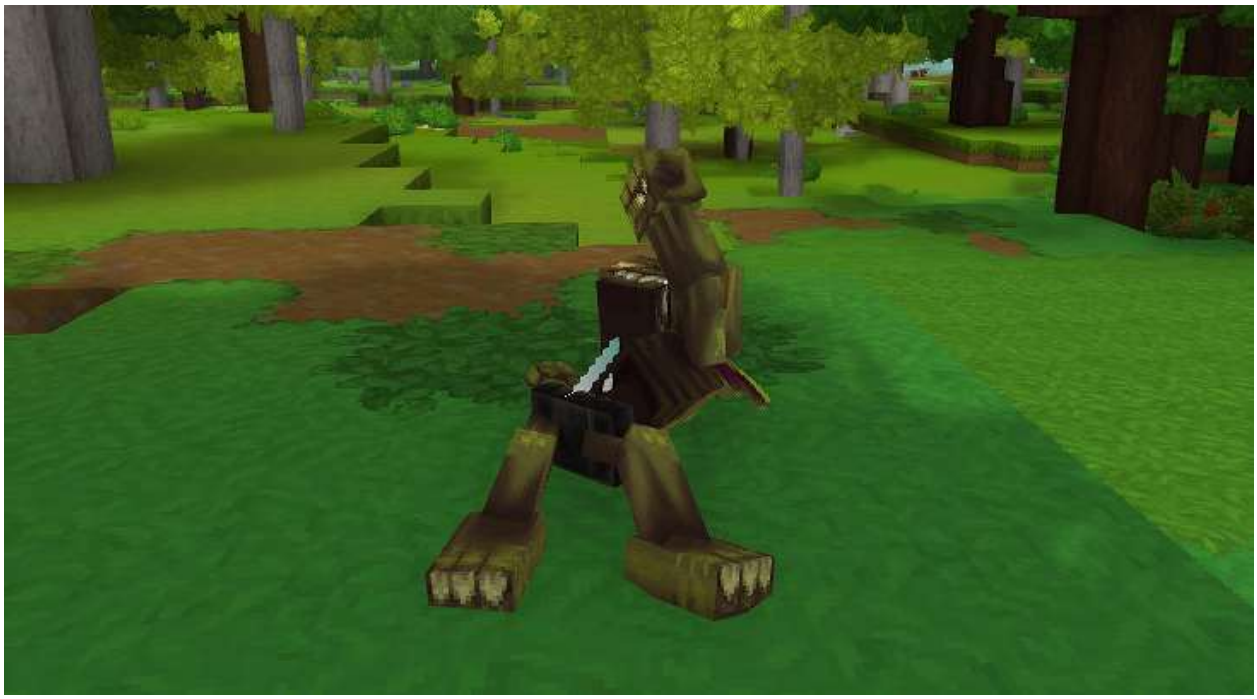
responsibility to ensure that this meshes nicely with the sleeping animation, and we'll also expose the attack name as a parameter in the parameters block.

```
"Parameters": {
  ...
  "SleepingAttack": {
    "Value": "Root_NPC_Attack_Melee",
    "Description": "Attack to use on NPCs that annoy it while sleeping"
  },
  "DropList": {
    "Value": "Empty",
    "Description": "Drop Items"
  }
}
```

We use a placeholder attack because we don't have a real one yet.

```
{
  "Sensor": {
    "Type": "Beacon",
    "Message": "Annoy_Ogre",
    "Range": 5
  },
  "Actions": [
    {
      "Type": "Attack",
      "Attack": { "Compute": "SleepingAttack" },
      "AttackPauseRange": [ 1, 2 ]
    }
  ]
}
```

We don't need another NPC to test this - we can use **/npc message Annoy_Ogre** to trigger this behaviour while looking at the ogre.



Now that our ogre can whack those pesky scrappers, the time has come to address the rat in the room: grabbing other NPCs is **hard.** Just in general. We don't have that kind of capability in tech and there's no guarantee we will either. A quick chat with our designers resulted in the following specification:

- Spawn a rat at some location.
- Have it run past the goblin.
- Have it animate to grab the rat and eat it.

This clarifies things, but doesn't resolve the problems. To do this, we're going to have to be creative.

First we need to actually spawn the rat somewhere near the ogre and get it to head over to be seized and eaten. We can do something like this by using a form of **manual spawn beacon** - an entity type which has to be placed in the world and can be triggered by other nearby NPCs on demand. We'll also need to build a small template for the rat itself to get it to move to the ogre. This seems like a behaviour that could be pretty reusable, so we'll make this template as simple and generic as possible so that it can be used in conjunction with all sorts of NPCs that are meant to grab small creatures and eat them.

We'll call this **Template_Edible_Critter.**

```
{
  "Type": "Abstract",
  "KnockbackScale": 0.5,
  "Parameters": {
    "Appearance": {
      "Value": "Rat",
      "Description": "Model to be used"
    },
    "WalkSpeed": {
      "Value": 3,
      "Description": "How fast this critter moves"
    },
    "SeekRange": {
      "Value": 40,
      "Description": "How far this NPC is allowed to be from the target that will eat it"
    },
    "MaxHealth": {
      "Value": 100,
      "Description": "Max health for the NPC"
    },
    "NameTranslationKey": {
      "Value": "server.npcRoles.Template.name",
      "Description": "Translation key for NPC name display"
    }
  },
  "Appearance": { "Compute": "Appearance" },
  "StartState": "Idle",
  "MotionControllerList": [
    {
      "Type": "Walk",
      "MaxWalkSpeed": { "Compute": "WalkSpeed" },
      "Gravity": 10,
      "MaxFallSpeed": 8,
      "Acceleration": 10
    }
  ],
  "MaxHealth": { "Compute": "MaxHealth" },
  "Instructions": [
    {
      "Instructions": [
        {
          "Sensor": {
            "Type": "State",
            "State": "Idle"
          },
          "Instructions": [
            {
              "Sensor": {
                "Type": "Beacon",
                "Message": "Approach_Target",
                "TargetSlot": "LockedTarget"
              },
              "Actions": [
                {
                  "Type": "State",
                  "State": "Seek"
```

```
            }
          ]
        },
        {
          "ActionsBlocking": true,
          "Actions": [
            {
              "Type": "Timeout",
              "Delay": [ 1, 1 ]
            },
            {
              "Type": "Despawn"
            }
          ]
        }
      ]
    },
    {
      "Sensor": {
        "Type": "State",
        "State": "Seek"
      },
      "Instructions": [
        {
          "Sensor": {
            "Type": "Target",
            "TargetSlot": "LockedTarget",
            "Range": { "Compute": "SeekRange" }
          },
          "BodyMotion": {
            "Type": "Seek",
            "SlowDownDistance": 0.1,
            "StopDistance": 0.1
          }
        },
        {
          "ActionsBlocking": true,
          "Actions": [
            {
              "Type": "Timeout",
              "Delay": [ 1, 1 ]
            },
            {
              "Type": "State",
              "State": "Idle"
            }
          ]
        }
      ]
    }
  ],
  "NameTranslationKey": { "Compute": "NameTranslationKey" }
}
```

This is a really simple NPC. It allows setting an **Appearance**, **WalkSpeed**, and **SeekRange** so that it can encompass a variety of different critter types. If idle for too long, it'll despawn, otherwise it'll try and seek towards the NPC that's meant to eat it, either from being signalled with a beacon or from being spawned in that specific state with a specific target already defined.

In this case, we want to do the latter, so let's define a **variant** (**Edible_Rat**)..

```
{
  "Type": "Variant",
  "Reference": "Template_Edible_Critter",
  "Modify": {
    "Appearance": "Rat",
    "WalkSpeed": 5,
    "MaxHealth": 100,
    "NameTranslationKey": { "Compute": "NameTranslationKey" }
  },
  "Parameters": {
    "NameTranslationKey": {
      "Value": "server.npcRoles.Rat.name",
      "Description": "Translation key for NPC name display"
    }
  }
}
```

...and a **spawn beacon** (also called **Edible_Rat**) that can create it for us.

```
{
  "Environments": [],
  "NPCs": [
    {
      "Weight": 1,
      "Id": "Edible_Rat"
    }
  ],
  "SpawnAfterGameTimeRange": [
    "PT5M",
    "PT10M"
  ],
  "NPCSpawnState": "Seek",
  "TargetSlot": "LockedTarget"
}
```

This simple config will only spawn the rat NPC we defined and will set it to the correct state. The **SpawnAfterGameTimeRange** parameter is required for the configuration to be used in other contexts, but isn't actually useful to us here.

Now, if we use **/spawning beacons add Edible_Rat --manual** we can create this spawn beacon ready for our ogre to use! Level designers who want to use this behaviour will need to add this beacon somewhere in the prefab so that the rats can be spawned, but multiple ogres can share the same beacon for this purpose. We can use **/spawning beacons trigger** to make sure it works!

Now we just need to set up our ogre to actually spawn the rat and fake the interaction between them. As always, we start by adding the state (**CallRat** - we're actually going to make it a main state instead of using the originally planned **.EatRat** substate to take advantage of some existing states and state transitions as we talked about earlier)...

```
{
  "Sensor": {
    "Type": "State",
    "State": "Eat"
  },
  ...
},
{
  "Sensor": {
    "Type": "State",
    "State": "CallRat"
  },
   "Instructions": [
        ]
}
```

...and an action to the random list.

```
{
  "Type": "Random",
  "Actions": [
    {
      "Weight": 60,
      "Action": {
        "Type": "State",
        "State": ".Guard"
      }
    },
    ...
    {
      "Weight": 10,
      "Action": {
        "Type": "State",
        "State": "CallRat"
      }
    }
  ]
}
```

Our **CallRat** state will just handle triggering the beacon and then waiting for a little while to see if the rat successfully arrives. If it doesn't, we'll reset and go back to pick another idle state.

Faking picking up and eating the rat we'll handle by **removing** the rat once it gets close enough and then giving the ogre an **item to hold** just like we did with the previous eat state transition.

The only difference here is the item being eaten (and thus the specific transition itself), so we can actually just make use of the previous **Eat** state for handling that side of the logic!

Before we can go any further though, we need to define an **NPC group** (**Edible_Rat** *again*) containing our edible rat and make it a parameter on the ogre so it can find it!

```
{
  "IncludeRoles": [
    "Edible_Rat"
  ]
}
```

And now we make this an exposed parameter on the ogre template. While we're at it, let's also add a parameter containing the name of the manual spawn beacon we'll trigger.

```
"Parameters": {
  ...
  "FoodNPCGroups": {
    "Value": [ "Edible_Rat" ],
    "Description": "The groups of edible NPCs that will come from triggering the beacon"
  },
  "FoodNPCBeacon": {
    "Value": "Edible_Rat",
    "Description": "The spawn beacon to trigger to create an edible NPC"
  },
  ...
}
```

With this done, we can implement the **CallRat** state logic and test that it works.

```
{
  "Sensor": {
    "Type": "State",
    "State": "CallRat"
  },
    "Instructions": [
      {
        "Continue": true,
        "Sensor": {
          "Type": "Any",
          "Once": true
        },
        "Actions": [
          {
            "Type": "TriggerSpawnBeacon",
            "BeaconSpawn": { "Compute": "FoodNPCBeacon" },
            "Range": 15
          }
        ]
      },
      {
```

```
      "Sensor": {
        "Type": "Mob",
        "Range": 2.5,
        "Filters": [
          {
            "Type": "NPCGroup",
            "IncludeGroups": { "Compute": "FoodNPCGroups" }
          },
          {
            "Type": "LineOfSight"
          }
        ]
      },
      "HeadMotion": {
        "Type": "Watch"
      },
      "ActionsBlocking": true,
      "Actions": [
        {
          "Type": "PlayAnimation",
          "Slot": "Status",
          "Animation": "Swipe"
        },
        {
          "Type": "Timeout",
          "Delay": [ 0.1, 0.1],
          "Action": {
            "Type": "Sequence",
            "Actions": [
              {
                "Type": "Remove"
              },
              {
                "Type": "State",
                "State": "Eat"
              }
            ]
          }
        }
      ]
    },
    {
      "Continue": true,
      "Sensor": {
        "Type": "Mob",
        "Range": 5,
        "Filters": [
          {
            "Type": "NPCGroup",
            "IncludeGroups": { "Compute": "FoodNPCGroups" }
          },
          {
            "Type": "LineOfSight"
          }
        ]
      },
```

```
        "HeadMotion": {
          "Type": "Watch"
        }
      },
      {
        "Reference": "Component_Step_State_Timeout",
        "Modify": {
          "_ExportStates": [ "Idle" ],
          "Delay": [ 10, 15 ]
        }
      }
    ]
}
```

The logic here is pretty straightforward - we start by triggering the beacon we defined in the parameters exactly once and continuing (as defined by the **Continue** and **Once** flags). We then wait for the edible NPC matching the list of groups we provided in the parameter (which contains only our edible rat in this instance) to get close enough to grab and then play an animation with a brief delay before removing it and moving to the **Eat** state. We watch it for a little bit until it gets close enough so it doesn't look too bad. Note how that last pair of actions (enclosed in a **sequence**) is not marked as blocking - both will execute in the same tick.

If the rat never arrives, we give it about 10-15 seconds before giving up and returning to **Idle**.

Now that it's working, let's implement the state transitions to make this goblin actually 'grab' the rat and start eating it. We don't have the 'rat' item yet, so we'll just use a different item as a placeholder for now. We need to define this in the parameters too.

```
"Parameters": {
  ...
  "FoodNPCItem": {
    "Value": "Food_Cheese",
    "Description": "The edible NPC in item form"
  },
  ...
}
```

And then a single state transition from **CallRat** to **Eat**.

```
{
  "States": [
    {
      "From": [ "CallRat" ],
      "To": [ "Eat" ]
    }
  ],
  "Actions": [
    {
      "Type": "Inventory",
      "Operation": "SetHotbar",
```

```
      "Item": { "Compute": "FoodNPCItem" },
      "Slot": 2,
      "UseTarget": false
    },
    {
      "Type": "Inventory",
      "Operation": "EquipHotbar",
      "Slot": 2,
      "UseTarget": false
    }
  ]
}
```

This isn't perfect, but it'll do for now and we can always come back and tweak things later.

With that, we're done with both the idle and inter-NPC behaviours. Next we move on to combat and reacting to players!

11. To kill a player

There are actually two parts to dealing with the player in this design description and the resulting state machine. We have the initial **Alerted** state, followed by combat itself. For now, we'll start with implementing all the awareness checks and the **Alerted** state, which acts as a transitional state to the combat behaviours themselves.

First we'll just create the empty **Alerted** state after the **CallRat** state with a timeout to send it back to **Idle.** The template will stop compiling until we add references to it via the awareness checks, but that's fine.

```
{
  "Sensor": {
    "Type": "State",
    "State": "Alerted"
  },
    "Instructions": [
      {
        "Reference": "Component_Instructions_State_Timeout",
        "Modify": {
          "_ExportStates": [ "Idle" ],
          "Delay": [ 10, 15 ]
        }
      }
    ]
}
```

There are a number of good components for handling awareness checks. For this we are going to use a handy Sensor component: **Component_Sensor_Standard_Detection.** To make this work, we'll set up an attitude group (**Goblin**) that will likely be shared by all other goblins too and references a pre-existing **Goblin NPC group.** We can detect different NPCs depending on the attitude and react accordingly.

```
{
  "Groups": {
    "Friendly": [
      "Goblin"
    ],
    "Hostile": [
    ]
  }
}
```

We don't actually have much in it for now since there aren't many other related NPCs implemented yet, but when there are, we can add any hostiles as hostile. We'll reference this in the parameters and assign it to the attitude group for the ogre, as well as adding a default

attitude towards the player (and ignoring most NPCs by default). I'm not really expecting 'friendly' ogres, so I won't bother making the default player attitude a parameter for now.

```
"Parameters": {
  ...
  "AttitudeGroup": {
    "Value": "Empty",
    "Description": "This NPCs attitude group"
  },
  ...
},
...
"DefaultPlayerAttitude": "Hostile",
"DefaultNPCAttitude": "Ignore",
"AttitudeGroup": { "Compute": "AttitudeGroup" },
```

In order to configure 'sight' and 'hearing' on the ogre, we also need to add a few more parameters relating to this.

```
"Parameters": {
  ...
  "ViewRange": {
    "Value": 15,
    "Description": "View range in blocks"
  },
  "ViewSector": {
    "Value": 180,
    "Description": "View sector in degrees"
  },
  "HearingRange": {
    "Value": 8,
    "Description": "Hearing range in blocks"
  },
  "AlertedRange": {
    "Value": 30,
    "Description": "A range within which the player can be seen/sensed when the NPC is
alerted to their presence"
  },
  "AbsoluteDetectionRange": {
      "Value": 4,
      "Description": "The range at which a target is guaranteed to be detected. If zero, absolute
detection will be disabled."
    },

  ...
}
```

These parameters are pretty self explanatory, but the **ViewRange/ViewSector** handle how far the ogre can see and his view cone, **HearingRange** handles how far his hearing extends, and

AlertedRange handles how far away the target can go while still being tracked after the ogre has been alerted to their presence. **AbsoluteDetectionRange** allows us to set a distance at which the Ogre will definitely react to us, regardless of any other conditions. This Sensor has a few other parameters, for example we can explicitly exclude NPC groups from detection too! But for our purposes, just a simple Attitude filter will work.

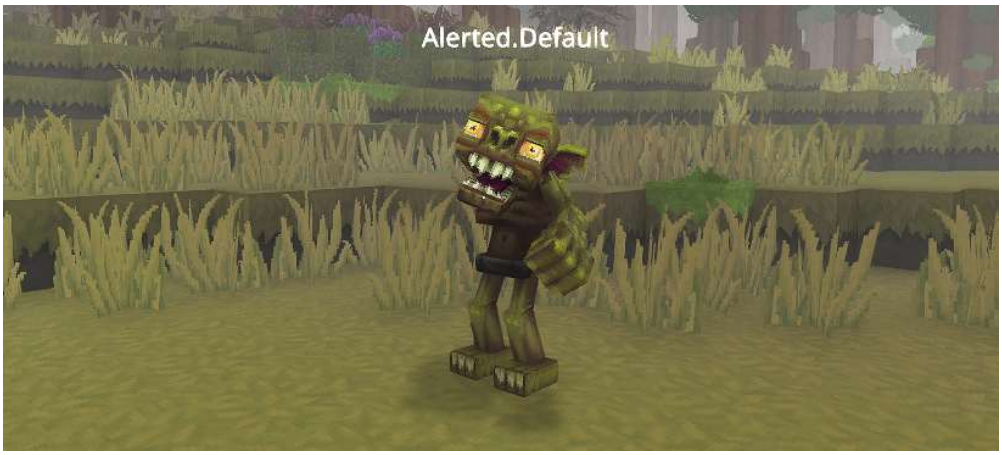
Let's talk for a moment about how this works. The sensor has different checks:

- First it checks if there is something in **absolute detection range**, then if nothing satisfies the filter, it checks further.

- It then **checks if there is a potential target in ViewRange/Sector.** If there's a target within the view cone and range and there's an unobstructed line of sight to it, then we can 'see' it.
- If the NPC couldn't see anything, it will **try to 'listen'.** This is a little more specific - if a target is walking or running and isn't crouching, we can 'hear' it, regardless of most other factors (though it won't 'hear' through walls). This makes it pretty perceptive, which is why we tend to use it with a much lower detection radius than the view range.

```
{
  "$Comment": "Check for any hostile targets in range that could alert the NPC",
  "Sensor": {
       "Reference": "Component_Sensor_Standard_Detection",
       "Modify": {
          "ViewRange": { "Compute": "ViewRange" },
          "ViewSector": { "Compute": "ViewSector" },
          "HearingRange": { "Compute": "HearingRange" },
          "ThroughWalls": false,
          "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
           "Attitudes": [ "Hostile" ]
        }
       },
  "Actions": [
    {
     "Type": "State",
      "State": "Alerted"
     }
  ]
},
{
  "Sensor": {
    "Type": "State",
    "State": ".Default"
  },
```

At this point, the template will compile again and we can observe in-game that approaching the ogre in various ways results in it switching to the **Alerted** state.



We want to add this component to the other idle states too, like sleeping and eating, but we probably want to reduce their detection capabilities a little bit in both cases. Let's add some kind of factor as a parameter.

```
"Parameters": {
  ...
  "DistractedPenalty": {
    "Value": 2,
    "Description": "A factor by which view range and hearing range will be divided when this NPC is distracted"
  },
  ...
}
```

Then we'll use the previous configuration for the **CallRat** state, but a modified version for both the **Sleep** and **Eat** state. Only the **Sleep** state is shown in this next snippet but the others should follow suit as required. We usually place these checks immediately after any instructions that trigger only **Once** and **Continue** (these are basically initialisation instructions).

```
{
  "Sensor": {
    "Type": "State",
    "State": "Sleep"
  },
    "Instructions": [
     {
              "$Comment": "Check for any hostile targets in range that could alert the NPC",
              "Sensor": {
                "Reference": "Component_Sensor_Standard_Detection",
                "Modify": {
                  "ViewRange": { "Compute": "ViewRange / DistractedPenalty" },
                  "ViewSector": { "Compute": "ViewSector" },
                  "HearingRange": { "Compute": "ViewRange / DistractedPenalty" },
                   "ThroughWalls": false,
                  "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
                  "Attitudes": [ "Hostile"]
```

```
            }
          },
          "Actions": [
            {
              "Type": "State",
              "State": "Alerted"
            }
          ]
        },
..
```

Here we can see a feature we haven't used before - the computed value actually encompasses a computation: we're dividing the **ViewRange** and **HearingRange** by the **DistractedPenalty** to result in a restricted detection radius.

Now, regardless of state, the ogre can react to any threats! Next we'll flesh out the **Alerted** state a bit so it actually does what's required of it. Let's recap what that was:

- Stand up if seated (this is covered by the state transitions already)
- Roar, ordering nearby goblin scrappers to attack
- Start slowly walking towards the player to attack (we'll consider this part of combat)

In that case, all we need to do here is make the ogre look at its target, play a roaring animation and maybe some particles, and send out a beacon to alert nearby goblin scrappers. First we need a quick **NPC group** (**Goblin_Scrapper**) to define goblin scrappers so that we send our beacon to them specifically.

```
{
  "IncludeRoles": [
    "Goblin_Scrapper"
  ]
}
```

We'll add a parameter referencing this too.

```
"Parameters": {
  ...
  "WarnGroups": {
    "Value": [ "Goblin_Scrapper"],
    "Description": "The groups to warn when spotting an enemy"
  },
  ...
}
```

And then implement the basic parts of the **Alerted** state, ending in a transition into a **Combat** state for combat.

```
{
  "Sensor": {
    "Type": "State",
    "State": "Alerted"
  },
    "Instructions": [
      {
        "Reference": "Component_Instruction_Play_Animation",
        "Modify": {
          "Animation": "Alerted"
        }
      },
      {
        "Continue": true,
        "Sensor": {
          "Type": "Target",
          "Range": { "Compute": "AlertedRange" },
          "Filters": [
            {
              "Type": "LineOfSight"
            }
          ]
        },
        "HeadMotion": {
          "Type": "Watch"
        }
      },
      {
        "Sensor": {
          "Type": "Target",
          "Range": { "Compute": "AlertedRange" }
        },
        "ActionsBlocking": true,
        "Actions": [
          {
            "Type": "Timeout",
            "Delay": [ 1, 1 ]
          },
          {
            "Type": "State",
            "State": "Combat"
          }
        ]
      },
      {
        "Actions": [
          {
            "Type": "State",
            "State": "Idle"
          }
        ]
      }
    ]
```

```
},
{
  "Sensor": {
    "Type": "State",
    "State": "Combat"
  },
    "Instructions": [
    ]
}
```

There's a fair bit of logic involved, but it's pretty straightforward. We play an animation, then we watch the target as long as there's line of sight to it. We then have a very short delay before switching to the **Combat** state. We'll handle actually sending the beacon message out to the other goblins using a state transition (and clear the animation at the same time).

```
{
  "States": [
    {
      "From": [ ],
      "To": [ "Combat" ]
    }
  ],
  "Actions": [
    {
      "Type": "PlayAnimation",
      "Slot": "Status"
    },
    {
      "Type": "Beacon",
      "Message": "Goblin_Ogre_Warn",
      "TargetGroups": { "Compute": "WarnGroups" },
      "SendTargetSlot": "LockedTarget"
    }
  ]
}
```

Before we move on to the actual combat behaviours in the **Combat** state, there's one more type of awareness check we need to implement: **damage**. It wouldn't do if a player could just hide away somewhere and snipe at our ogre without him reacting.

We can accomplish this very easily using the **Component_Instruction_Damage_Check** component, which is designed to assess if the NPC has received damage and respond accordingly. If the target is known and within a reasonable distance, it switches to the combat **Chase** state, otherwise it switches to a **Panic** state. Our ogre is pretty tough and has an important job, so we won't actually make him panic - if he takes damage, he's just going to switch to **Alerted** to warn others nearby and then run in and smash the threat!

Though only one instance is shown in the example, we place this in each of the places where we put the sight/sound checks, and we give it a higher priority by putting it first.

```
{
  "Reference": "Component_Instructions_Damage_Check",
  "Modify": {
    "_ExportStates": [ "Alerted", "Alerted" ],
    "AlertedRange": { "Compute": "AlertedRange" }
  }
},
{
  "$Comment": "Check for any hostile targets in range that could alert the NPC",
   "Sensor": {
        ...
```

With that done, our ogre should respond to everything we need it to! Now we can focus on getting it to actually attack its target!

12. Melee Combat

For the purpose of this Tutorial we'll focus only on the melee attack.

We'll start by making an **attack sequence.** Our NPCs can attack using smart decision making with a feature called the Combat Action Evaluator (CAE), but for this tutorial we'll just stick to simple sequences. This will also allow for a quick look into the adjacent Interactions system.

First we need to create a **Root Interaction** in **HytaleAssets/Server/Item/RootInteractions**, together with other Root interactions. Let's make a **Root_NPC_Goblin_Ogre_Attack** interaction:

```
{
  "Interactions": [
    {
      "Type": "Chaining",
      "ChainId": "Slashes",
      "ChainingAllowance": 15,
      "Next": [
        "Goblin_Ogre_Swing_Left",
        "Goblin_Ogre_Swing_Right",
        "Goblin_Ogre_Swing_Down"
      ]
    }
  ],
  "Tags": {
    "Attack": [
      "Melee"
    ]
  }
}
```

This is a simple chaining Interaction where the ogre will perform **Swing_Left** and, so long as the next attack happens within 15 seconds, he will use **Swing_Right.** If he then manages to attack us a third time within 15 seconds he'll use the **Swing_Down** attack.

These swing interactions need to be created in the **HytaleAssets/Server/Item/Interactions** folder.

Let's make the **Goblin_Ogre_Swing_Left** interaction:

```
{
  "Type": "Simple",
  "Effects": {
    "ItemPlayerAnimationsId": "Goblin_Club",
    "ItemAnimationId": "SwingLeft"
  },
  "$Comment": "Prepare Delay",
  "RunTime": 0.2,
```

```
        "Next": {
            "Type": "Selector",
            "$Comment": "Length of Combat",
            "RunTime": 0.25,
            "Selector": {
              "Id": "Horizontal",
              "Direction": "ToLeft",
              "TestLineOfSight": true,
              "ExtendTop": 0.5,
              "ExtendBottom": 2,
              "StartDistance": 0.1,
              "EndDistance": 3.5,
              "Length": 60,
              "RollOffset": 0,
              "YawStartOffset": -30
            },
            "HitEntity": {
              "Interactions": [
                {
                  "Parent": "DamageEntityParent",
                  "DamageCalculator": {
                    "BaseDamage": {
                      "Physical": 8
                    }
                  },
                  "DamageEffects": {
                    "Knockback": {
                      "Force": 0.5,
                      "RelativeX": -5,
                      "RelativeZ": -5,
                      "VelocityY": 5
                    },
                    "WorldSoundEventId": "SFX_Unarmed_Impact",
                    "WorldParticles": [
                      {
                        "SystemId": "Impact_Blade_01"
                      }
                    ]
                  }
                }
              ]
            },
            "Next": {
              "Type": "Simple",
              "$Comment": "Pad the interaction length",
              "RunTime": 0.1
              }
            }
        }
```

Now we need to approach the target and use our fancy attack sequence. Again, there are a few components that will make most of this much simpler to do:

- **Component_Instruction_Soft_Leash** will work in conjunction with an external **ReturnHome** state to send the ogre back to his start point if we get him too far away.
- **Component_Instruction_Intelligent_Chase** will handle smartly chasing the target based on its last known position and will trigger pathfinding where necessary. We might want to add an extra **Search** state for this.

Let's create the states we need. First, Combat itself, but since we're going to be using **Component_Instruction_Intelligent_Chase**, this component needs to know where to switch when the Target is lost or if the NPC is too far away. Let's create two states: **Search** and **ReturnHome** that will be used there.

```
    {
      "Sensor": {
        "Type": "State",
        "State": "Combat"
      },
      "Instructions": []
    },
    {
      "Sensor": {
        "Type": "State",
        "State": "ReturnHome"
      },
      "Instructions": []
    },
    {
      "Sensor": {
        "Type": "State",
        "State": "Search"
      },
      "Instructions": []
    }
  ]
```

Let’s start with chasing the target. We wrap that behavior in sub state **.Chase** so we can fall back to it when the target gets out of attack range and the NPC needs to run after it in a somewhat intelligent manner.

```
{
      "Sensor": {
        "Type": "State",
        "State": "Combat"
      },
      "Instructions": [
         {
          "Sensor": {
            "Type": "State",
            "State": ".Chase"
          },
          "Instructions": [
            {
              "Sensor": {
                "Type": "Target",
                "Range": { "Compute": "AttackDistance" },
                "Filters": [
                  {
                    "Type": "LineOfSight"
                  }
                ]
              },
              "Actions": [
                {
                  "Type": "State",
                  "State": ".Default"
                }
              ]
            },
            {
              "Reference": "Component_Instruction_Soft_Leash",
              "Modify": {
                "_ExportStates": [ "ReturnHome" ],
                "LeashDistance": { "Compute": "LeashDistance" },
                "LeashMinPlayerDistance": { "Compute": "LeashMinPlayerDistance" },
                "LeashTimer": { "Compute": "LeashTimer" },
                "HardLeashDistance": { "Compute": "HardLeashDistance" }
              }
            },
            {
              "Reference": "Component_Instruction_Intelligent_Chase",
              "Modify": {
                "_ExportStates": [ "Search", "Search", "ReturnHome" ],
                "ViewRange": { "Compute": "AlertedRange * 2" },
                "HearingRange": { "Compute": "HearingRange * 2" },
                "StopDistance": 0.1,
                "RelativeSpeed": 0.5
              }
            }
          ]
        }
```

- First, it checks if the target is within AttackDistance. If so, we switch to the default combat state.
- If the NPC gets too far away from its leash position (often the spawn position), we use **Component_Instruction_Soft_Leash** to send it home. We put this sensor in front of the chase state so that we don't continue chasing if the leash kicks in. There's a **Leash Timer** on this component that decides when it's time to give up.
- And finally the chase component itself, which performs 'intelligent' chasing of the target.

When the Target is within the **AttackDistance**, we perform the attack, otherwise we switch to the chase state. Let's start with the case where the target is within a melee range.

```
  {
          "Sensor": {
            "Type": "State",
            "State": "Combat"
          },
          "Instructions": [
             {
              "Sensor": {
                "Type": "State",
                "State": ".Chase"
              },
              "Instructions": [...
                  ]
             },
             {
              "$Comment": "NPC melee attack",
              "Sensor": {
                "Type": "Target",
                "Range": { "Compute": "AttackDistance" },
                "Filters": [
                  {
                    "Type": "LineOfSight"
                  }
                ]
              },
              "ActionsBlocking": true,
              "Actions": [
                {
                  "Type": "Attack",
                  "Attack": { "Compute": "Attack" },
                  "AttackPauseRange": { "Compute": "AttackPauseRange" }
                },
                {
                  "$Comment": "Brief delay to prevent the NPC potentially strafing/backing away and
missing the shot.",
                  "Type": "Timeout",
                  "Delay": [ 0.2, 0.2 ]
                }
              ],
              "HeadMotion": {
                "Type": "Aim",
                "RelativeTurnSpeed": { "Compute": "CombatRelativeTurnSpeed" }
              }
```

```
            },
            {
             "Actions": [
               {
                 "Type": "State",
                 "State": ".Chase"
               }
             ]
            }
          ]
        },
```

The important things to note here are the **AttackDistance, AttackPauseRange** and **CombatRelativeTurnSpeed**. We're going to configure both of these via the template itself. **AttackDistance** refers to the distance at which the ogre will attempt to perform melee attacks to hit the target (though the actual range of the attack itself is defined in the **Attack** interaction). **CombatRelativeTurnSpeed** allows us to define how quickly (or slowly) the ogre rotates while in combat. **AttackPauseRange** defines how often the attacks will be performed - a bit like an attack cooldown. Let's make sure we have all these parameters added in the list:

```
   "Attack": {
      "Value": "Root_NPC_Goblin_Ogre_Attack",
      "Description": "The attack to use."
    },
    "AttackDistance": {
      "Value": 2,
      "Description": "The distance at which an NPC will execute attacks"},
    "AttackPauseRange": {
      "Value": [ 1.5, 2 ],
      "Description": "The range for absolute minimum time before an NPC can execute a second attack (or
block)."
    },
    "CombatRelativeTurnSpeed": {
      "Value": 1.5,
      "Description": "Modifier that decides turn speed difference in combat."},
    "LeashDistance": {
      "Value": 20,
      "Description": "The range after which an NPC will start to want to return to their spawn point."
    },
    "LeashMinPlayerDistance": {
      "Value": 4,
      "Description": "The minimum distance from the player before the NPC will be willing to give up on
the chase."
    },
    "LeashTimer": {
      "Value": [ 3, 5 ],
      "Description": "How long the NPC must be more than the minimum distance form the player and too
far from leash before giving up."
    },
    "HardLeashDistance": {
      "Value": 60,
      "Description": "An absolute maximum from the the leash position the NPC can go before turning
back."
    },
```

Now the NPC will attack, but we still need to add Search and ReturnHome logic. We can do that right away, so we have a fully functional melee ogre.

```
{
        "Sensor": {
          "Type": "State",
          "State": "ReturnHome"
        },
        "Instructions": [
          {
            "Sensor": {
              "Type": "And",
              "Sensors": [
                {
                  "Type": "Damage",
                  "Combat": true,
                  "TargetSlot": "LockedTarget"
                },
                {
                  "Enabled": { "Compute": "AbsoluteDetectionRange > 0" },
                  "Type": "Target",
                  "TargetSlot": "LockedTarget",
                  "Range": { "Compute": "AbsoluteDetectionRange" }
                }
              ]
            },
            "Actions": [
              {
                "Type": "State",
                "State": "Combat"
              }
            ]
          },
          {
            "Sensor": {
              "Type": "Leash",
              "Range": { "Compute": "LeashDistance * 0.3" }
            },
            "BodyMotion": {
              "Type": "Seek",
              "SlowDownDistance": { "Compute": "LeashDistance * 0.4" },
              "StopDistance": { "Compute": "LeashDistance * 0.2" },
              "RelativeSpeed": 0.8,
              "UsePathfinder": true
            }
          },
          {
            "Actions": [
              {
                "Type": "SetStat",
                "Stat": "Health",
                "Value": 1000000
              },
              {
                "Type": "State",
```

```
                "State": "Idle"
              }
            ]
          }
        ]
      },
```

There are 3 parts to this state: first, it checks if there is incoming combat damage and will switch to Combat if so. We don't want to leave NPCs exploitable as they're running home by having them ignore all attacks. If there's no incoming damage it will find its way home using **BodyMotion: Seek.** We need to be careful here though, since it might become expensive due to **"UsePathfinder": true** turning on complex pathfinding. The last block simply heals the NPC to full health and moves it to idle once it's found its home spot again.

The Search state makes use of **Component_Sensor_Lost_Target_Detection.**

```
{
      "Sensor": {
        "Type": "State",
        "State": "Search"
      },
      "Instructions": [
        {
          "Sensor": {
            "Type": "Damage",
            "Combat": true,
            "TargetSlot": "LockedTarget"
          },
          "Actions": [
            {
              "Type": "State",
              "State": "Alerted"
            }
          ]
        },
        {
          "Instructions": [
            {
              "Sensor": {
                "Reference": "Component_Sensor_Lost_Target_Detection",
                "Modify": {
                  "ViewRange": { "Compute": "ViewRange" },
                  "ViewSector": { "Compute": "ViewSector" },
                  "HearingRange": { "Compute": "HearingRange" },
                  "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
                  "TargetSlot": "LockedTarget"
                }
              },
              "Actions": [
                {
                  "Type": "State",
```

```
        "State": "Combat"
      }
    ]
  },
  {
    "Sensor": {
      "Reference": "Component_Sensor_Standard_Detection",
      "Modify": {
        "ViewRange": { "Compute": "ViewRange" },
        "ViewSector": { "Compute": "ViewSector" },
        "HearingRange": { "Compute": "HearingRange" },
        "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
        "Attitudes": [ "Hostile", "Neutral" ]
      }
    },
    "Actions": [
      {
        "Type": "State",
        "State": "Alerted"
      }
    ]
  },
  {
    "BodyMotion": {
      "Type": "Sequence",
      "Motions": [
        {
          "Type": "Timer",
          "Time": [ 3, 6 ],
          "Motion": {
            "Type": "Wander",
            "MaxHeadingChange": 1,
            "RelativeSpeed": 0.5
          }
        },
        {
          "Type": "Sequence",
          "Looped": true,
          "Motions": [
            {
              "Type": "Timer",
              "Time": [ 3, 6 ],
              "Motion": {
                "Type": "WanderInCircle",
                "Radius": 10,
                "MaxHeadingChange": 60,
                "RelativeSpeed": 0.5
              }
            },
            {
              "Type": "Timer",
              "Time": [ 2, 3 ],
              "Motion": {
                "Type": "Nothing"
              }
            }
```

```
                    ]
                  }
                ]
              },
              "ActionsBlocking": true,
              "Actions": [
                {
                  "Type": "Timeout",
                  "Delay": [4,5]
                },
                {
                  "Type": "State",
                  "State": "Idle"
                }
              ]
            }
          ]
        }
      ]
    }
```

The search state will first check if there is incoming damage and will switch to the alerted state if so.

If not it'll execute the next block of instructions using pre-existing components:

- Check if the lost target is detected with the **Lost Target detection sensor**. If so, switch to combat.
- Check if any other target is available through the **Standard detection sensor**. If so, switch to the alerted state.
- Otherwise the NPC will move around using the **Wander** motion, stopping briefly in between until it gives up and…
- ...goes back to being idle.

Appendix

Template_Goblin_Ogre_Tutorial (dependant on Root_NPC_Goblin_Ogre_Attack - you need this file to exist, because template is referencing it)

```
{
  "$Comment": "Debug: DisplayState",
  "Debug": "DisplayState",
  "Type": "Abstract",
  "Parameters": {
    "Appearance": {
      "Value": "Bear_Grizzly",
      "Description": "Model to be used"
    },
    "DropList": {
      "Value": "Empty",
      "Description": "Drop Items"
    },
    "MaxHealth": {
      "Value": 100,
      "Description": "Max health for the NPC"
    },
     "EatItem": {
      "Value": "Food_Beef_Raw",
      "Description": "The item this NPC will find when it rummages for food"
    },
     "SleepingAttack": {
      "Value": "Root_NPC_Attack_Melee",
      "Description": "Attack to use on NPCs that annoy it while sleeping"
    },
    "FoodNPCGroups": {
      "Value": [ "Edible_Rat" ],
      "Description": "The groups of edible NPCs that will come from triggering the beacon"
    },
    "FoodNPCBeacon": {
      "Value": "Edible_Rat",
      "Description": "The spawn beacon to trigger to create an edible NPC"
    },
    "FoodNPCItem": {
      "Value": "Food_Cheese",
      "Description": "The edible NPC in item form"
    },
    "ViewRange": {
      "Value": 15,
      "Description": "View range in blocks"
    },
    "ViewSector": {
      "Value": 180,
      "Description": "View sector in degrees"
    },
    "HearingRange": {
      "Value": 8,
      "Description": "Hearing range in blocks"
    },
```

```
    "AlertedRange": {
      "Value": 30,
      "Description": "A range within which the player can be seen/sensed when the NPC is alerted to
their presence"
    },
    "DistractedPenalty": {
      "Value": 2,
      "Description": "A factor by which view range and hearing range will be divided when this NPC is
distracted"
    },
   "AbsoluteDetectionRange": {
      "Value": 4,
      "Description": "The range at which a target is guaranteed to be detected. If zero, absolute
detection will be disabled."
    },
    "WarnGroups": {
      "Value": [ "Goblin_Scrapper" ],
      "Description": "The groups to warn when spotting an enemy"
    },
    "AttitudeGroup": {
      "Value": "Empty",
      "Description": "This NPCs attitude group"
    },
    "Attack": {
      "Value": "Root_NPC_Goblin_Ogre_Attack",
      "Description": "The attack to use."
    },
    "AttackDistance": {
      "Value": 2,
      "Description": "The distance at which an NPC will execute attacks"
    },
    "AttackPauseRange": {
      "Value": [ 1.5, 2 ],
      "Description": "The range for absolute minimum time before an NPC can execute a second attack (or
block)."
    },
    "CombatRelativeTurnSpeed": {
      "Value": 1.5,
      "Description": "Modifier that decides turn speed difference in combat."
    },
    "LeashDistance": {
      "Value": 20,
      "Description": "The range after which an NPC will start to want to return to their spawn point."
    },
    "LeashMinPlayerDistance": {
      "Value": 4,
      "Description": "The minimum distance from the player before the NPC will be willing to give up on
the chase."
    },
    "LeashTimer": {
      "Value": [ 3, 5 ],
      "Description": "How long the NPC must be more than the minimum distance form the player and too
far from leash before giving up."
    },
    "HardLeashDistance": {
      "Value": 60,
```

```
      "Description": "An absolute maximum from the the leash position the NPC can go before turning
back."
    },
    "NameTranslationKey": {
      "Value": "server.npcRoles.Template.name",
      "Description": "Translation key for NPC name display"
    }
  },
  "Appearance": { "Compute": "Appearance" },
  "DropList": { "Compute": "DropList" },
  "MaxHealth": { "Compute": "MaxHealth" },
  "StartState": "Idle",
  "DefaultPlayerAttitude": "Hostile",
  "DefaultNPCAttitude": "Ignore",
  "AttitudeGroup": { "Compute": "AttitudeGroup" },
  "KnockbackScale": 0.5,

  "MotionControllerList": [
    {
      "Type": "Walk",
      "MaxWalkSpeed": 3,
      "Gravity": 10,
      "MaxFallSpeed": 8,
      "Acceleration": 10
    }
  ],
  "InteractionVars": {
    "Melee_Damage": {
      "Interactions": [
        {
          "$Comment": "Config for a successful melee hit, default values for Effects and HitShape are
not overwritten here, see each NPC_ file",
          "Parent": "NPC_Attack_Melee_Damage",
          "DamageCalculator": {
            "Type": "Absolute",
            "BaseDamage": {
              "Physical": 10
            },
            "RandomPercentageModifier": 0.1
          }
        }
      ]
    }
  },
  "StateTransitions": [
    {
      "States": [
        {
          "From": [ "Idle" ],
          "To": [ "Sleep" ]
        }
      ],
      "Actions": [
        {
          "Type": "PlayAnimation",
          "Slot": "Status",
```

```
        "Animation": "Laydown"
      },
      {
        "Type": "Timeout",
        "Delay": [ 1, 1 ]
      }
    ]
  },
  {
    "States": [
      {
        "From": [ "Sleep" ],
        "To": [ ]
      }
    ],
    "Actions": [
      {
        "Type": "PlayAnimation",
        "Slot": "Status",
        "Animation": "Wake"
      },
      {
        "Type": "Timeout",
        "Delay": [ 1, 1 ]
      }
    ]
  },
  {
    "States": [
      {
        "From": [ "Idle" ],
        "To": [ "Eat" ]
      }
    ],
    "Actions": [
      {
        "Type": "Inventory",
        "Operation": "SetHotbar",
        "Item": { "Compute": "EatItem" },
        "Slot": 2,
        "UseTarget": false
      },
      {
        "Type": "Inventory",
        "Operation": "EquipHotbar",
        "Slot": 2,
        "UseTarget": false
      }
    ]
  },
  {
    "States": [
      {
        "From": [ "CallRat" ],
        "To": [ "Eat" ]
      }
```

```
    ],
    "Actions": [
      {
        "Type": "Inventory",
        "Operation": "SetHotbar",
        "Item": { "Compute": "FoodNPCItem" },
        "Slot": 2,
        "UseTarget": false
      },
      {
        "Type": "Inventory",
        "Operation": "EquipHotbar",
        "Slot": 2,
        "UseTarget": false
      }
    ]
  },
  {
    "States": [
      {
        "From": [ "Eat" ],
        "To": [ ]
      }
    ],
    "Actions": [
      {
        "Type": "PlayAnimation",
        "Slot": "Status"
      },
      {
        "Type": "Inventory",
        "Operation": "EquipHotbar",
        "Slot": 0,
        "UseTarget": false
      }
    ]
  },
  {
    "States": [
      {
        "From": [ ],
        "To": [ "Combat" ]
      }
    ],
    "Actions": [
      {
        "Type": "PlayAnimation",
        "Slot": "Status"
      },
      {
        "Type": "Beacon",
        "Message": "Goblin_Ogre_Warn",
        "TargetGroups": { "Compute": "WarnGroups" },
        "SendTargetSlot": "LockedTarget"
      }
    ]
```

```
    }
  ],
  "Instructions": [
    {
      "Sensor": {
        "Type": "State",
        "State": "Idle"
      },
      "Instructions": [
        {
          "Continue": true,
          "Sensor": {
            "Type": "Any",
            "Once": true
          },
          "Actions": [
            {
              "Type": "Inventory",
              "Operation": "EquipHotbar",
              "Slot": 0,
              "UseTarget": false
            }
          ]
        },
        {
          "Reference": "Component_Instruction_Damage_Check",
          "Modify": {
            "_ExportStates": [ "Alerted", "Alerted" ],
            "AlertedRange": { "Compute": "AlertedRange" }
          }
        },
        {
          "$Comment": "Check for any hostile targets in range that could alert the NPC",
          "Sensor": {
            "Reference": "Component_Sensor_Standard_Detection",
            "Modify": {
              "ViewRange": { "Compute": "ViewRange" },
              "ViewSector": { "Compute": "ViewSector" },
              "HearingRange": { "Compute": "HearingRange" },
              "ThroughWalls": false,
              "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
              "Attitudes": [ "Hostile"]
            }
          },
          "Actions": [
            {
              "Type": "State",
              "State": "Alerted"
            }
          ]
        },
        {
          "Sensor": {
            "Type": "State",
            "State": ".Default"
          },
```

```
      "Instructions": [
        {
          "Actions": [
            {
              "Type": "Random",
              "Actions": [
                {
                  "Weight": 10,
                  "Action": {
                    "Type": "State",
                    "State": ".Guard"
                  }
                },
                {
                  "Weight": 10,
                  "Action": {
                    "Type": "State",
                    "State": "Sleep"
                  }
                },
                {
                  "Weight": 10,
                  "Action": {
                    "Type": "State",
                    "State": "Eat"
                  }
                },
                {
                  "Weight": 10,
                  "Action": {
                    "Type": "State",
                    "State": "CallRat"
                  }
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "Sensor": {
        "Type": "State",
        "State": ".Guard"
      },
      "Instructions": [
        {
          "Continue": true,
          "ActionsBlocking": true,
            "Actions": [
            {
              "Type": "Timeout",
              "Delay": [ 5, 10 ]
            },
            {
              "Type": "State",
```

```
                "State": ".Default"
              }
            ]
          },
          {
            "Reference": "Component_Instruction_Intelligent_Idle_Motion_Follow_Path"
          }
        ]
      }
    ]
  },
  {
    "Sensor": {
      "Type": "State",
      "State": "Sleep"
    },
    "Instructions": [
      {
        "Reference": "Component_Instruction_Damage_Check",
        "Modify": {
          "_ExportStates": [ "Alerted", "Alerted" ],
          "AlertedRange": { "Compute": "AlertedRange" }
        }
      },
      {
        "$Comment": "Check for any hostile targets in range that could alert the NPC",
        "Sensor": {
          "Reference": "Component_Sensor_Standard_Detection",
          "Modify": {
            "ViewRange": { "Compute": "ViewRange / DistractedPenalty" },
            "ViewSector": { "Compute": "ViewSector" },
            "HearingRange": { "Compute": "ViewRange / DistractedPenalty" },
              "ThroughWalls": false,
            "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
            "Attitudes": [ "Hostile"]
          }
        },
        "Actions": [
          {
            "Type": "State",
            "State": "Alerted"
          }
        ]
      },
      {
        "Reference": "Component_Instruction_Play_Animation_In_State_For_Duration",
        "Modify": {
        "_ExportStates": [ "Idle.Default" ],
        "Animation": "Sleep",
        "Duration": [ 5, 6 ]
        }
      },
      {
        "Sensor": {
          "Type": "Beacon",
          "Message": "Annoy_Ogre",
```

```
          "Range": 5
        },
        "Actions": [
          {
            "Type": "Attack",
            "Attack": { "Compute": "SleepingAttack" },
            "AttackPauseRange": [ 1, 2 ]
          }
        ]
      }
    ]
  },
  {
    "Sensor": {
      "Type": "State",
      "State": "Eat"
    },
    "Instructions": [
        {
        "Reference": "Component_Instruction_Damage_Check",
        "Modify": {
          "_ExportStates": [ "Alerted", "Alerted" ],
          "AlertedRange": { "Compute": "AlertedRange" }
        }
      },
      {
        "$Comment": "Check for any hostile targets in range that could alert the NPC",
        "Sensor": {
          "Reference": "Component_Sensor_Standard_Detection",
          "Modify": {
            "ViewRange": { "Compute": "ViewRange / DistractedPenalty" },
            "ViewSector": { "Compute": "ViewSector" },
            "HearingRange": { "Compute": "ViewRange / DistractedPenalty" },
              "ThroughWalls": false,
            "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
            "Attitudes": [ "Hostile"]
          }
        },
        "Actions": [
          {
            "Type": "State",
            "State": "Alerted"
          }
        ]
      },
      {
        "Reference": "Component_Instruction_Play_Animation_In_State_For_Duration",
        "Modify": {
        "_ExportStates": [ "Idle.Default" ],
        "Animation": "Eat",
        "Duration": [ 5, 6 ]
        }
      }
    ]
  },
  {
```

```
    "Sensor": {
      "Type": "State",
      "State": "CallRat"
    },
    "Instructions": [
      {
        "Continue": true,
        "Sensor": {
          "Type": "Any",
          "Once": true
        },
        "Actions": [
          {
            "Type": "TriggerSpawnBeacon",
            "BeaconSpawn": { "Compute": "FoodNPCBeacon" },
            "Range": 15
          }
        ]
      },
      {
        "Reference": "Component_Instruction_Damage_Check",
        "Modify": {
          "_ExportStates": [ "Alerted", "Alerted" ],
          "AlertedRange": { "Compute": "AlertedRange" }
        }
      },
      {
        "$Comment": "Check for any hostile targets in range that could alert the NPC",
        "Sensor": {
          "Reference": "Component_Sensor_Standard_Detection",
          "Modify": {
            "ViewRange": { "Compute": "ViewRange" },
            "ViewSector": { "Compute": "ViewSector" },
            "HearingRange": { "Compute": "HearingRange" },
            "ThroughWalls": false,
            "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
            "Attitudes": [ "Hostile"]
          }
        },
        "Actions": [
          {
            "Type": "State",
            "State": "Alerted"
          }
        ]
      },
      {
        "Sensor": {
          "Type": "Mob",
          "Range": 2.5,
          "Filters": [
            {
              "Type": "NPCGroup",
              "IncludeGroups": { "Compute": "FoodNPCGroups" }
            },
            {
```

```
            "Type": "LineOfSight"
          }
        ]
      },
      "HeadMotion": {
        "Type": "Watch"
      },
      "ActionsBlocking": true,
      "Actions": [
        {
          "Type": "PlayAnimation",
          "Slot": "Status",
          "Animation": "Swipe"
        },
        {
          "Type": "Timeout",
          "Delay": [ 0.1, 0.1 ],
          "Action": {
            "Type": "Sequence",
            "Actions": [
              {
                "Type": "Remove"
              },
              {
                "Type": "State",
                "State": "Eat"
              }
            ]
          }
        }
      ]
    },
    {
      "Continue": true,
      "Sensor": {
        "Type": "Mob",
        "Range": 5,
        "Filters": [
          {
            "Type": "NPCGroup",
            "IncludeGroups": { "Compute": "FoodNPCGroups" }
          },
          {
            "Type": "LineOfSight"
          }
        ]
      },
      "HeadMotion": {
        "Type": "Watch"
      }
    },
    {
      "Reference": "Component_Instruction_State_Timeout",
      "Modify": {
        "_ExportStates": [ "Idle" ],
        "Delay": [ 10, 15 ]
```

```
        }
      }
    ]
  },
  {
    "Sensor": {
      "Type": "State",
      "State": "Alerted"
    },
    "Instructions": [
      {
        "Reference": "Component_Instruction_Play_Animation",
        "Modify": {
          "Animation": "Alerted"
        }
      },
      {
        "Continue": true,
        "Sensor": {
          "Type": "Target",
          "Range": { "Compute": "AlertedRange" },
          "Filters": [
            {
              "Type": "LineOfSight"
            }
          ]
        },
        "HeadMotion": {
          "Type": "Watch"
        }
      },
      {
        "Sensor": {
          "Type": "Target",
          "Range": { "Compute": "AlertedRange" }
        },
        "ActionsBlocking": true,
        "Actions": [
          {
            "Type": "Timeout",
            "Delay": [ 1, 1 ]
          },
          {
            "Type": "State",
            "State": "Combat"
          }
        ]
      },
      {
        "Actions": [
          {
            "Type": "State",
            "State": "Idle"
          }
        ]
      }
```

```
    ]
  },
  {
    "Sensor": {
      "Type": "State",
      "State": "Combat"
    },
    "Instructions": [
        {
        "Sensor": {
          "Type": "State",
          "State": ".Chase"
        },
        "Instructions": [
          {
            "Sensor": {
              "Type": "Target",
              "Range": { "Compute": "AttackDistance" },
              "Filters": [
                {
                  "Type": "LineOfSight"
                }
              ]
            },
            "Actions": [
              {
                "Type": "State",
                "State": ".Default"
              }
            ]
          },
          {
            "Reference": "Component_Instruction_Soft_Leash",
            "Modify": {
              "_ExportStates": [ "ReturnHome" ],
              "LeashDistance": { "Compute": "LeashDistance" },
              "LeashMinPlayerDistance": { "Compute": "LeashMinPlayerDistance" },
              "LeashTimer": { "Compute": "LeashTimer" },
              "HardLeashDistance": { "Compute": "HardLeashDistance" }
            }
          },
          {
            "Reference": "Component_Instruction_Intelligent_Chase",
            "Modify": {
              "_ExportStates": [ "Search", "Search", "ReturnHome" ],
              "ViewRange": { "Compute": "AlertedRange * 2" },
              "HearingRange": { "Compute": "HearingRange * 2" },
              "StopDistance": 0.1,
              "RelativeSpeed": 0.5
            }
          }
        ]
        },
        {
        "$Comment": "NPC melee attack",
        "Sensor": {
```

```
            "Type": "Target",
            "Range": { "Compute": "AttackDistance" },
            "Filters": [
              {
                "Type": "LineOfSight"
              }
            ]
          },
          "ActionsBlocking": true,
          "Actions": [
            {
              "Type": "Attack",
              "Attack": { "Compute": "Attack" },
              "AttackPauseRange": { "Compute": "AttackPauseRange" }
            },
            {
              "$Comment": "Brief delay to prevent the NPC potentially strafing/backing away and missing
the shot.",
              "Type": "Timeout",
              "Delay": [ 0.2, 0.2 ]
            }
          ],
          "HeadMotion": {
            "Type": "Aim",
            "RelativeTurnSpeed": { "Compute": "CombatRelativeTurnSpeed" }
          }
          },
          {
          "Actions": [
            {
              "Type": "State",
              "State": ".Chase"
            }
          ]
          }
      ]
    },
    {
      "Sensor": {
        "Type": "State",
        "State": "ReturnHome"
      },
      "Instructions": [
        {
          "Sensor": {
            "Type": "And",
            "Sensors": [
              {
                "Type": "Damage",
                "Combat": true,
                "TargetSlot": "LockedTarget"
              },
              {
                "Enabled": { "Compute": "AbsoluteDetectionRange > 0" },
                "Type": "Target",
                "TargetSlot": "LockedTarget",
```

```
              "Range": { "Compute": "AbsoluteDetectionRange" }
            }
          ]
        },
        "Actions": [
          {
            "Type": "State",
            "State": "Combat"
          }
        ]
      },
      {
        "Sensor": {
          "Type": "Leash",
          "Range": { "Compute": "LeashDistance * 0.3" }
        },
        "BodyMotion": {
          "Type": "Seek",
          "SlowDownDistance": { "Compute": "LeashDistance * 0.4" },
          "StopDistance": { "Compute": "LeashDistance * 0.2" },
          "RelativeSpeed": 0.8,
          "UsePathfinder": true
        }
      },
      {
        "Actions": [
          {
            "Type": "SetStat",
            "Stat": "Health",
            "Value": 1000000
          },
          {
            "Type": "State",
            "State": "Idle"
          }
        ]
      }
    ]
  },
  {
    "Sensor": {
      "Type": "State",
      "State": "Search"
    },
    "Instructions": [
      {
        "Sensor": {
          "Type": "Damage",
          "Combat": true,
          "TargetSlot": "LockedTarget"
        },
        "Actions": [
          {
            "Type": "State",
            "State": "Alerted"
          }
```

```
      ]
    },
    {
      "Instructions": [
        {
          "Sensor": {
            "Reference": "Component_Sensor_Lost_Target_Detection",
            "Modify": {
              "ViewRange": { "Compute": "ViewRange" },
              "ViewSector": { "Compute": "ViewSector" },
              "HearingRange": { "Compute": "HearingRange" },
              "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
              "TargetSlot": "LockedTarget"
            }
          },
          "Actions": [
            {
              "Type": "State",
              "State": "Combat"
            }
          ]
        },
        {
          "Sensor": {
            "Reference": "Component_Sensor_Standard_Detection",
            "Modify": {
              "ViewRange": { "Compute": "ViewRange" },
              "ViewSector": { "Compute": "ViewSector" },
              "HearingRange": { "Compute": "HearingRange" },
              "AbsoluteDetectionRange": { "Compute": "AbsoluteDetectionRange" },
              "Attitudes": [ "Hostile", "Neutral" ]
            }
          },
          "Actions": [
            {
              "Type": "State",
              "State": "Alerted"
            }
          ]
        },
        {
          "BodyMotion": {
            "Type": "Sequence",
            "Motions": [
              {
                "Type": "Timer",
                "Time": [ 3, 6 ],
                "Motion": {
                  "Type": "Wander",
                  "MaxHeadingChange": 1,
                  "RelativeSpeed": 0.5
                }
              },
              {
                "Type": "Sequence",
                "Looped": true,
```

```
                    "Motions": [
                      {
                        "Type": "Timer",
                        "Time": [ 3, 6 ],
                        "Motion": {
                          "Type": "WanderInCircle",
                          "Radius": 10,
                          "MaxHeadingChange": 60,
                          "RelativeSpeed": 0.5
                        }
                      },
                      {
                        "Type": "Timer",
                        "Time": [ 2, 3 ],
                        "Motion": {
                          "Type": "Nothing"
                        }
                      }
                    ]
                  }
                ]
              },
              "ActionsBlocking": true,
              "Actions": [
                {
                  "Type": "Timeout",
                  "Delay": [4,5]
                },
                {
                  "Type": "State",
                  "State": "Idle"
                }
              ]
            }
          ]
        }
      ]
    }
  ],
  "NameTranslationKey": { "Compute": "NameTranslationKey" }
}
```

Goblin_Ogre_Tutorial role file

```
{
  "Type": "Variant",
  "Reference": "Template_Goblin_Ogre_Tutorial",
  "Modify": {
    "Appearance": "Goblin",
    "MaxHealth": 124,

    "_InteractionVars": {
      "Melee_SwingDown_Damage": {
        "Interactions": [
          {
            "Parent": "Goblin_Ogre_Swing_Down_Damage",
            "DamageCalculator": {
              "Type": "Absolute",
              "BaseDamage": {
                "Physical": 20
              }
            }
          }
        ]
      }
    },
    "NameTranslationKey": {
      "Compute": "NameTranslationKey"
    }
  },
   "Parameters": {
    "NameTranslationKey": {
      "Value": "server.npcRoles.Goblin_Ogre.name",
      "Description": "Translation key for NPC name display"
    }
  }
}
```

Goblin_Ogre_Swing_Down file

```
{
  "Type": "Simple",
  "Effects": {
    "ItemPlayerAnimationsId": "Goblin_Club",
    "ItemAnimationId": "SwingDown"
  },
  "$Comment": "Prepare Delay",
  "RunTime": 0.2,
  "Next":
    {
    "Type": "Selector",
    "$Comment": "Length of Combat",
    "RunTime": 0.25,
    "Selector": {
```

```
      "Id": "Horizontal",
      "Direction": "ToLeft",
      "TestLineOfSight": true,
      "ExtendTop": 0.5,
      "ExtendBottom": 0.5,
      "StartDistance": 1,
      "EndDistance": 2.5,
      "Length": 90,
      "RollOffset": 60,
      "YawStartOffset": -45
    },
    "HitEntity": {
      "Interactions": [
        {
          "Type": "Replace",
          "DefaultValue": {
            "Interactions": [
              "Goblin_Ogre_Swing_Down_Damage"
            ]
          },
          "Var": "Melee_SwingDown_Damage"
        }
      ]
    },
    "Next": {
      "Type": "Simple",
      "$Comment": "Pad the interaction length",
      "RunTime": 0.1
    }
  }
}
```

Root_NPC_Goblin_Ogre_Attack - you need this file to exist, because template is referencing it.

```
{
  "Interactions": [
    {
      "Type": "Chaining",
      "ChainId": "Slashes",
      "ChainingAllowance": 15,
      "Next": [
        "Goblin_Ogre_Swing_Left",
        "Goblin_Ogre_Swing_Right",
        "Goblin_Ogre_Swing_Down"
      ]
    }
  ],
  "Tags": {
    "Attack": [
      "Melee"
    ]
  }
}
```