Basics Custom Mechanics

By now you have created a custom recipe, or maybe multiple, or even manipulated tags, or created custom items or blocks.

But you want to do more then that, you want to add a custom mechanic, for example milking a goat.

The first step is to break down your idea into smaller pieces, until each piece is something you can code.
One thing to note, is that most all things are caused by some trigger. Such as an entity dieing, or a block being placed. These are detected by events.

Detecting Events

This is just like when we made recipes, but that time the event was triggered not by a players action, but by the game doing internal operations, that being getting to the time that is for registering recipes.

As a refresher, here is detecting the recipes event:

```
onEvent('recipes', event => {
//recipes
})
```

To change the event detected, we need to change what is in the `'`s. But to what? Luckily there is a list of all event page in this wiki!

Searching the `ID` column, we can scroll down and find that there is an event named `item.entity_interact` which happens to be the one that we want for milking the goat.

> Look at the `type` column and it will tell you which folder, you will need to put you code into.

Now we just put that in there, and we can now run code when a player right clicks an entity.

```
onEvent('item.entity_interact', event => {
//code
})
```

To test we can use `Utils.server.tell()` to detect when the event occurs.

> The are many situations that `console.log()`, would be better, which put the result in to `instance/logs/kubejs/server.txt`.

```
onEvent('item.entity_interact', event => {
	Utils.server.tell("Entity Interation Detected!")
})
```

> Now to test you can try right clicking an entity and see you will see a message appears in the chat.

But this occurs to it entities, and want to only affect what happens to goats.
To do this, we need to know information about the context of the event.

Calling Methods of an Event

Up to this point you may have been wondering what the purpose of the `event => {` is.

You can recall that for the custom recipe, used it to call the method that added the recipe.

```
onEvent('recipes', event => {
	event.shapeless('flint', ['gravel', 'gravel', 'gravel'])
}
```

For each event that we detect the variable event will have different methods. The `item.entity_interact` event has methods:

- `.getEntity()`
- `.getHand()`
- `.getItem()`
- `.getTarget()`

> How are you supposed to know this? Using ProbeJS! There is a whole wiki page about this addon!

So in our code we can write:

```
onEvent('item.entity_interation', event => {
	event.getTarget()
})
```

>

> `.getEntity()` gets the player, while `.getTarget()` gets the entity

What does this do?
Nothing!
Why?
Because the `.getEntity()` method does not do anything, but it **returns** the entity.

To see this we can put it into the chat.

```
onEvent('item.entity_interation', event => {
Utils.server.tell( event.getTarget() )
})
```

> Now when you interact with an entity you can see what some details about it!

> What is put in the chat is **not** the actual value is, as **only** Strings can be displayed. All other types (such as EntityJS) have a toString() method that is called which extracts some information and returns a string that is then displayed instead.

Because of this, you might think what we need to do is run `event.getEntity().toString()` to get the entity type.

But this is wrong. You should not be using `.toString()` as there is almost always a better way. In this case its using the method `.getType()` of entity that returns a string of the type of the entity.

```
onEvent('item.entity_interation', event => {
Utils.server.tell( event.getTarget().getType() )
})
```

This code is good, but it can be better because of a feature called **BEANS.**

This feature is very simple:

- Methods that start with `get` and take no parameters, can be shorted from `foo.getBar()` to `foo.bar`
- Methods that start with `set` and take one parameter, can be shorted from `foo.setBar("cactus")` to `foo.bar = "cactus"`

So in our case the code can be shortened to:

```
onEvent('item.entity_interation', event => {
Utils.server.tell( event.target.type )
```

```
})
```

Alright, this is all good, but we want to make the code do stuff, not just tell tell us about the entities type. Notably we want to run code **if** an the type is a certain value.

We do this by using a control structure called: **if**!

If Statements

The basic syntax is as following:
`if (condition) {result}`
The condition is a boolean, which holds a value: true or false.
And if the boolean equates to true, then the code in result runs, otherwise it does not.

Here is an example:

```
onEvent('item.entity_interact', event => {
  Utils.server.tell( event.target.type )
  if (true) {
    Utils.server.tell("True")
    }
    if (false) {
    Utils.server.tell("False")
    }
})
```

> When you interact with an entity in the chat you will be told the **True**, but not the **False**.

Lets make this useful, we need to use a condition to run the code based on the entity type.

Testing equality:

```
//GOOD
"foo" == "foobar" // this is false
"foo" == "foo" // this is true

//BAD
"foo" = "foobar" // a single '=' does assignment (we will get to this later) NOT equality
```

So our code can look like:

```
onEvent('item.entity_interact', event => {
  if (event.target.type == "minecraft:goat") {
    Utils.server.tell("Is a Goat")
    }
})
```

> Now interacting with a goat will provide the message **Is a Goat** when interacting with a goat!

Now any code that we want to run when a goat is interacted with, we will place inside of this if statement.

This works, but it can be better.

Something that as you write more code will become increasingly important is **code readability**. In this case it can be improved with what is know as **guard statements**. In this case it will look like:

Guard Statements

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat") return
    Utils.server.tell("Is a Goat")
})
```

This might look more confusing at first but is really quite simple.

Firstly, I am using `!=` instead of `==`, which is the same as, except it returns the opposite, so **true** if they are unequal, and **false** if they do equal.

Secondly, id you do not include `{}` then the if will only apply to the next line immediately after, and everything after is considered to be out of the if.

Thirdly, `return` in this context will end the execution of the code.
So if the entity type is not a goat, then execution will not get passed line 2.

> Learn more about ifs **here**.

The next step is to take a bucket, but before we can do that, we need to ensure the player is holding a bucket.

Getting the item in the players hand

We can use the method `.getItem()` so `event.getItem()` which can be beaned to `event.item`.

Now we get the type of item we can use `.getId()` so `event.item.getId()` so `event.item.id`.

We could use another if, but I want to show you a different option, the OR boolean operator:

```
true || true // is true
true || false // is true
false || true // is true
false || false // is false

false || false || true || false // is true
false || false || false || false // is false
```

so we can put this in or code to be:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
    Utils.server.tell("Is a Goat and is Holding a Bucket")
})
```

> This should say in the chat Is a Goat and is Holding a Bucket if you right click a goat with a bucket

Now to take the item, we will manipulate the count of it. We can get the count of the item, subtract one from it, then set the count to the result.

- `.getCount()`
  - get the count of an item
- `.setCount()`
  - sets the count of an item

We can write the code:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
    Utils.server.tell("Is a Goat and is Holding a Bucket")

  event.item.setCount( event.item.getCount() - 1)
})
```

Now we **bean** it to:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
    Utils.server.tell("Is a Goat and is Holding a Bucket")

  event.item.count = event.item.count - 1
})
```

But there is a better way to write this using something know as **syntactical sugar**. This is just a fancy term for using symbols in a special order that lets you write a piece of code with less total characters to do a different thing with under the hood.

In the example above we used the *basic* assignment operator `=`.
But there are other assignment operators! Such as the *subtraction* assignment operator `-=`.

Here is it in the code:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
    Utils.server.tell("Is a Goat and is Holding a Bucket")

  event.item.count -= 1
})
```

Instead of getting the value, then subtracting one, it can now be thought of as simply reducing the value by 1.

> There are other assignment operators, such as one for addition, `+=`, multiplication, `*=`, division, `/=`, modulo, `%=`, logical or `||=`, logical and, `&&==`, bitwise xor, `^=`, bitwise and, `&=`, bitwise or, `|=`, left bitshift, `<<=`, right bitshift, `>>>=`, signed right bitshift `>>=`, and of course minus, `-=`

But wait, there's more! For adding or subtracting **by 1**, you can make the code even smaller, appending `++` or `--` to then end.

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
    Utils.server.tell("Is a Goat and is Holding a Bucket")

  event.item.count--
})
```

Epic, we made it smaller! None of that was required, but it looks a lot nicer.

Giving the Player Items

We can go quick cause we know all the steps for all that is left.

`event.getPlayer()` for player but `event.player` because of **beans**. Player has a method called `.give(ItemStack)` to give an item so `event.player.give(ItemStack)`. And in our case `ItemStackJS` is `'milk_bucket'`. So our final code:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
    Utils.server.tell("Is a Goat and is Holding a Bucket")

  event.item.count--

  event.player.give('milk_bucket')
})
```

Now we can remove the debugging line `Utils.server.tell("Is a Goat and is Holding a Bucket")`.

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
  event.item.count--
  event.player.give('milk_bucket')
})
```

> When holding a stack of buckets and right clicking a goat, a bucket will be consumed and you gain a milk bucket.

It seem good. Right? All done. Wrong!

When programming, you always have to be careful about **edge-cases**. These are situations that are typically at extremes on situations. For example you write some code to function differently if you have 5 or more levels, but when you have 5 levels exactly, some logic differently causing an expected result.

Our code currently mishandles an edge case. The edge case is when the player has one bucket in their hand.

> When holding one bucket in your hand, not in the first slot, and with nothing else in the first slot. When right-clicking a goat the milk bucket does not stay in your hand as is intuitive, but instead get placed in the first slot.

To resolve this bug, we could add an if to check if the count is one, then change the logic, but this is not required because their is method that does everything for us. The method `.giveInHand()` is identical to the `.give()` except it first attempts to put the item in the players hand if it is empty.

Putting this in our code looks like:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
  event.item.count--
  event.player.giveInHand('milk_bucket')
})
```

Now it seems like we are done! But compare with milking a cow, its just not as satisfying.

Adding Sound

Although adding **feedback** in to you creations, usually in the form of sound effects, and particles, does not change the effect or your creation, it has a major effect on how engaging, polished, and interesting your creation appears.

Luckily playing sound is really easy with KubeJS, because many different classes have a `.playSound()` method.
We want the sound to originate from the goat being milked so we can use `event.target` to get the goat, then just call `.playSound()`.

`.playSound()` takes some parameters:
Either the `id` of the sound, or the id, the `volume`, and the `pitch`.
Lets keep thing simple by only using the `id`.

Although you can register new sounds with KubeJS, it would be easier to use the existing cow milking sound. The `id` of this sound is `entity.cow.milk`.

Putting this into the code looks like:

```
onEvent('item.entity_interact', event => {
  if (event.target.type != "minecraft:goat" || event.item.id != "minecraft:bucket") return
  event.item.count--
  event.player.giveInHand('milk_bucket')
  event.target.playSound('entity.cow.milk')
})
```

> Now when milking a goat, you hear the milking sound.

There we go! We are done!

Recap

Now that we implemented a feature together you will be able to make some of your own basic custom features too!

Don't be too intimidated by how long it took us, we went through every single detail, but you already know those so it will take you a fraction of the time it took to make this.

Here is a step by step list of how you can make your own mechanic:

1. Determine what triggers the mechanic.
    1. This is the event.
    2. In this example we did `item.entity_interact`.
    3. A list of all events is here.
2. Narrow down when the code of you event runs with guard statements.
    1. Use an if and return.
    2. In our case it is detecting the entity as a goat and the item as a bucket.
    3. Use ProbeJS or the second wiki or the source code to get the information you need.
3. Break down what you want to do as code you can write.
    1. In our case instead of the idea of filling a bucket with milk, the code takes one of the item and give the player a bucket of milk.
    2. Use ProbeJS or the second wiki or the source code to get the information you need.
4. Double check edge cases.
    1. **You should be always testing you code with most every change you make.**
    2. You need to be extra careful with edge cases, when making changes too.
    3. In our case we replaced `player.give()` with `player.giveInHand()`.
5. Add Polish.
    1. This includes fixing minor bugs on edge cases.
    2. This also involves making sure the player gets feedback such as sound or particles.
    3. In our case this is the milking sound.

Other Helpful Things to Know

Although we did not get to it with the example, here are some simple things that would be helpful to know:

- Cancelling events:
    - Sometimes you want the default action of an event to not occur.
    - An example is maybe if you wanted to add milking of horses.
        - There already is an interaction for right clicking horses, getting on them.
        - The player would both milk and be put on the horse if the event is not canceled.
    - The syntax is `event.cancel()`.

    - You can place it anywhere in your code and the effect will be the same, the default action will not occur.
  - Only some events are cancel-able.
    - The non-cancel-able events are listed in the list of all events.
    - You can tell if an event is cancel-able with `event.isCancelable()`.
  - Some events are partly cancel-able.
    - They are listed as cancel-able, but don't completely undo the default action.
    - For example entity.death event, canceling it will not prevent the entity from dying, but will prevent loot, and statistics.
- While loops
  - They syntax is the same as an if.
  - The function is the same, except the code inside of the loop will repeat until the condition becomes false.
  - Learn more here.
- Variables
  - Using `let foo = bar` will make a variable named foo and set it to the contents of bar.
    - To change the value of foo later use `foo = bar` if `foo` is already made.
  - A common use is to reduce repeated code.
    - So in our example we could have placed `let t = event.target` at the beginning.
    - Then every use of `event.target` could have been replaced with t so `event.target.type` become `t.type`.
  - Variables massively increase what is possible, and as begin to reveal a lot more hidden complexities (such as scope, reference vs value and more) that we not gonna get into right now.
  - Learn more here.

Revision #4
Created 2022-10-07 20:01:44 UTC by Q6
Updated 2024-02-19 06:37:08 UTC by Q6