Getting Started

A step by step guide for learning the basics of KubeJS

- Introduction and Installation
- Your First Script
- Basics Custom Mechanics
- Using ProbeJS

Introduction and Installation

Installation

Install the mod and its two dependencies Architectury and Rhino.
Make you use the most resent version of each mods for your version.

If you are using 1.16 fabric then use this instead.

When you first install KubeJS, you will need to launch Minecraft with the mods (and the game not crashing) to generate the some folders and files.

The `kubejs` folder

Finding it

Everything you do in KubeJS in located in the kubejs folder in your instance.

- In PolyMC the file structure will look like `polymc > instances > instance name > minecraft > kubejs`
- In CurseForge launcher the file structure will look like `curseforge > minecraft > instances > instance name > kubejs`
- In all of the above cases the `instance name` is the name of the instance
- In the normal Minecraft launcher it will be `.minecraft > kubejs`, unless you changed the instance folder.

From now on this will be referenced as the kubejs folder.

The contents of it

- `startup_scripts`
  - Scripts that get loaded once during game startup
  - Used for adding items and other things that can only happen while the game is loading
  - Can reload code **not in an event** with `/kubejs reload_startup_scripts`
  - To reload all the code you must restart the game
- `client_scripts`
  - Scripts that get loaded every time client resources reload
  - Used for:
    - JEI events
    - tooltips

        - other client side things
    - Can reload code **not in an event** with `/kubejs reload client_scripts`
    - Can reload all the code in client_scripts with F3+T
- `server_scripts`
    - Scripts that get loaded every time server resources reload (world load, `/reload`)
    - Used for modifying:
        - recipes
        - tags
        - loot tables
        - handling server events
    - Can reload code **not in an event** with `/kubejs reload server_scripts`
    - Can be all the code in server_scripts with `/reload`
- `exported`
    - Data dumps like texture atlases end up here
- `config`
    - KubeJS config storage.
    - This is also the only directory that scripts can access other than world directory
- `assets`
    - Acts as a resource pack
    - you can put any client resources in here, like:
        - textures
            - Example: assets/kubejs/textures/item/test_item.png
        - models
        - lang
        - etc.
    - Can be reloaded by pressing F3 + T
    - Can reload **only** the lang files (so faster) `/kubejs reload lang`
    - Read more about it here.
- `data`
    - Acts as a datapack
    - you can put any server resources in here, like:
        - loot tables
            - Example: data/kubejs/loot_tables/blocks/test_block.json
        - functions
        - etc
    - Can be reloaded with `/reload`
    - Read more about it here.
- `README.txt`
    - Contains the information here

You can find type-specific logs in logs/kubejs/ directory

Other Useful Tools

Code is just a language that computers can understand. However, the grammar of the language, called syntax for code, is very precise. When you code has a syntactical error, the computer does not know what to do and will probably do something that you do not desire.

With KubeJS we will be writing a lot of code, so it important to avoid these errors. Luckily, there are tools called code editors, that can help us write code correctly.

We recommend installing Visual Studio Code as it is light-ish and has great built in JS support. Now when you edit you java script files, it will not only warn you when you make most syntactical errors, but also help you prevent them in the first place.

Your First Script

Writing Your First Script

If you have launched the game at least once before you will find `kubejs/server_scripts/example_server_script.js` It looks like this:

```
// priority: 0

settings.logAddedRecipes = true
settings.logRemovedRecipes = true
settings.logSkippedRecipes = false
settings.logErroringRecipes = true

console.info('Hello, World! (You will see this line every time server resources reload)')

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

onEvent('item.tags', event => {
	// Get the #forge:cobblestone tag collection and add Diamond Ore to it
	// event.get('forge:cobblestone').add('minecraft:diamond_ore')

	// Get the #forge:cobblestone tag collection and remove Mossy Cobblestone from it
	// event.get('forge:cobblestone').remove('minecraft:mossy_cobblestone')
})
```

Lets break it down:

- // priority: 0
  - Makes it so that if you have multiple server scripts, this script gets loaded first
  - If you have only one server_script, this has no effect
- settings.logAddedRecipes = true
  settings.logRemovedRecipes = true
  settings.logSkippedRecipes = false
  settings.logErroringRecipes = true
  - sets settings for what messages are logged

    - You can remove all four of these lines if you want and it will only change what is put into the logs
- console.info('Hello, World! (You will see this line every time server resources reload)')
    - Prints the message in the log
    - This line is useless other then example and should be removed eventually
- onEvent('recipes', event => {
    - This makes an event listener for the `recipes` event, and will run the code inside when and only when the `recipes` event is triggered
    - This is triggered when server resources reload
        - Which happens when the world load or the `/reload` command is used
- // Change recipes here
    - comment, an code in a line following `//` will be considered a comment and will not be run
    - Used for taking notes as you write the code
- })
    - Indicates the end of the 'recipes' event listener
- onEvent('item.tags', event => {
  // Get the #forge:cobblestone tag collection and add Diamond Ore to it
  // event.get('forge:cobblestone').add('minecraft:diamond_ore')
  // Get the #forge:cobblestone tag collection and remove Mossy Cobblestone from it
  // event.get('forge:cobblestone').remove('minecraft:mossy_cobblestone')
  })
    - Same thing as the other one but for the `item.tags` event
    - You can find the list of all event here

Finally Writing Code For Real

Lets start off by adding a recipe to craft flint from three gravel.

To do so, insert this code right after the **recipes event.**

```
event.shapeless("flint", ["gravel", "gravel", "gravel"])
```

It should look like this:

```
// priority: 0

settings.logAddedRecipes = true
settings.logRemovedRecipes = true
settings.logSkippedRecipes = false
settings.logErroringRecipes = true

console.info('Hello, World! (You will see this line every time server resources reload)')
```

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

onEvent('item.tags', event => {
// Get the #forge:cobblestone tag collection and add Diamond Ore to it
// event.get('forge:cobblestone').add('minecraft:diamond_ore')

// Get the #forge:cobblestone tag collection and remove Mossy Cobblestone from it
// event.get('forge:cobblestone').remove('minecraft:mossy_cobblestone')
})
```

Now lets test it!

Run the command `/reload` in game, then try crafting three gravel together in any order.

But how does it work?

- event
  - This is a variable that created with the arrow expression in `onEvent('recipes', event => {`...
    - You can have the name be what every you choose, as long as it matches everywhere
- .
  - The dot operator is used for calling a method of an object
  - In this case event is the object and shapeless is the method
- shapeless(
  - This is the method that is called by the dot operator on the event
  - It is taking two arguments, that being an item result and a array input
- "
  - Indicates the start of a string
- flint
  - The contents of the string
  - You can use `create:flour` , if it is from a different mod (`flint` is the same as `minecraft:flint`, and both are valid)
- "
  - Signifies the end of the string.
  - A string is simply a sequence of characters, or letters
  - You can read more about strings in JS here.
- ,
  - separates different arguments in the method.
- [

    - Signifies the start of the array.
    - An array holds multiple values or any type, including other arrays.
    - You can read more about arrays in JS here.
- "gravel", "gravel", "gravel"
    - The contents of the array
    - Arrays can hold an indefinite number of elements
-]
    - Closing the array
-)
    - Closing the method

There you go! You can make custom shapeless recipes!

If you want to make other types of recipes, learn about it here, and if you have an addon that adds more recipe types, loot at its mod page, or here.

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.

Using ProbeJS

ProbeJS is an add-on that is built exclusively to help you program.

What it does:

It generates documentation files from digging around in the game code itself. So, you get all the methods, not only from KubeJS, but also from base Minecraft, no matter they're added by modloader, or from the other mods you install. Not only can you view these docs, but they are also formatted in a way that a sufficiently advanced code editor, like VSCode, can understand. So, you will now get more relevant code suggestions too.

Installation:

Find ProbeJS on the 3rd Party addons list and download the relevant version for you.
Once you've installed it and relaunched your game, run the command `/probejs dump`.
Now you will need to wait a little while, but after some time, you should see a message alerting you that the dump is complete.

What just happened?

You can now look and see that there is a new folder located at `instance/kubejs/probe/` and inside of here there are a more folders and files. These are your docs.

Setting up VS Code

1. In VS Code select `file > open folder`
2. This opens up a file explore window, select the KubeJS folder (`instance/`) and choose select folder.

You're done!

Troubleshooting

For many people, autocompletions won't be popped up as they type. You need to configure your VSCode to setup a valid JavaScript IDE so you can get 100% power of ProbeJS!

No Intellisense at All

For some reason, VSCode downloaded by some people are not having builtin JavaScript/TypeScript support. To check if you have such support enabled, search `@builtin JavaScript` in the extension tab in your VSCode, you should see a plugin named `TypeScript and JavaScript Language Features`, that's the builtin extension for VSCode to support JS/TS.

If not, then you'll have to install the `JavaScript and TypeScript Nightly` to get JS/TS support.

Downloading Intellisense Models

If your ISP is weird, downloading Intellisense models for enabling support can take a long time. You can consider switch to proxy or some other methods to change your Internet environment, maybe even changing a WiFi can work. If not, then sorry, it's an Internet problem, there's no way to solve it on VSCode's end.

Too Many Mods

Completion takes a significant amount of performance. You can't expect VSCode to run super-fast on some ATM8-like modpacks, that's not possible.

For less than 150 mods, VSCode should run at a decent speed, for more than 300 mods, completions are taking >10s since now VSCode need to examine over 100k item/block/entity entries before telling you what to type next.

Usage

Properties and Methods of a Class

To know the methods of a class just type in the class name, like `Item` or `BlockProperties`, then type a `.` now you will see a list of the public methods and properties.

ProbeJS will display the **bean**ed accessors and mutators. However, due to the limitation of JavaScript syntax, if there's a method having same name with a field/bean, then the name will always be resolved to the method.

Type Checking and JSDoc

To add type checking for extra safety when coding JavaScript, add `//@ts-check` to the first line of a JS file, then you will have VSCode guarding your types for the rest of the file. It's extremely useful when you're working with some dangerous code which is likely to crash the game if you have a mistake in type.

Sometimes, due to limitations of TypeScript, you might need to persuade VSCode to skip checking for some part of your code. Adding `//@ts-ignore` would help you to do that.

Or maybe you want to tell VSCode: "This should be a list of item names!", or "This method should have ... as params, and ... as return types!". Then you can add `JSDoc` to tell VSCode to do that:

```
/**
 * @type {Special.Item[]}
 */
let consumableItems = []


ServerEvents.recipes(event => {
/**
 *
 * @param {Internal.Ingredient_} input
 * @param {Internal.ItemStack_} output
 * @returns {Internal.ShapedRecipeJS}
 */
let make3x3Recipe = (input, output) => {
return event.recipes.minecraft.crafting_shaped(output, ["SSS", "SSS", "SSS"], { S: input })
}
})
```

Sometimes, if with `//@ts-check` enabled, you will need to add `//@ts-ignore` to calm VSCode to accept your docs.

Searching by Keyword

If you are in VSCode press the explorer button in the top-ish left to open up the explorer pane.

Now navigate to `probe > generated > globals.d.ts`.
Press `Ctrl + F` and a little search window should pop up in your editor.
Now type in you key word and look through all the matches.

Tips

If you append `class` to the front and ` ` to the end then you will look for classes so like Item has 8635 results for me, but if I type `class Item` then the one I want!

In `events.d.ts` you will find events but only basic information about them.

In `constants.d.ts` you can see different pieces that you can use whereever.

If you want to find the methods of an event, say `item.pickup` find it in one of the files (In this case `events.documented.d.ts`) and here is the line describing it:

```
declare function onEvent(name: 'item.pickup', handler: (event: Internal.ItemPickupEventJS) =>
void)
```

Look closely and find `Internal.ItemPickupEventJS`. Since it says `Internal`, we will look in the the `globals.d.ts` file, but if it says `Registry` then we use `registries.d.ts`.

Now we will go to the generated file and search `ItemPickupEventJS`.
Then we find:

```
/**
* Fired when an item is about to be picked up by the player.
* @javaClass dev.latvian.mods.kubejs.item.ItemPickupEventJS
*/
class ItemPickupEventJS extends Internal.PlayerEventJS {
    getItem(): Internal.ItemStackJS;
    getEntity(): Internal.EntityJS;
    getItemEntity(): Internal.EntityJS;
    canCancel(): boolean;
    get item(): Internal.ItemStackJS;
    get itemEntity(): Internal.EntityJS;
    get entity(): Internal.EntityJS;
    /**
    * Internal constructor, this means that it's not valid unless you use `java()`.
    */
    constructor(player: Internal.Player, entity: Internal.ItemEntity, stack:
Internal.ItemStack);
}
```

This means that we can use the methods `.getItem()` `.getEntity()` `.getItemEntity()` `.canCancel()` `.item` `.itemEntity` and `.entity`.

But if we did `potion.registry` then we get `Registry.Potion` which brings us to:

```
class Potion extends Internal.RegistryObjectBuilderTypes$RegistryEventJS<any> {
create(id: string, type: "basic"): Internal.PotionBuilder;
create(id: string): Internal.PotionBuilder;
}
```

So we can use `event.create('cactus_juice')` but that does not do much so we need to follow one step further and go to the potion builder, which you see is `Internal.PotionBuilder`. Now we search

`PotionBuilder` in `globals.d.ts` then we see:

```
/**
* @javaClass dev.latvian.mods.kubejs.misc.PotionBuilder
*/
class PotionBuilder extends Internal.BuilderBase<Internal.Potion> {
    getRegistryType(): Internal.RegistryObjectBuilderTypes<Internal.Potion>;
    effect(effect: Internal.MobEffect_, duration: number, amplifier: number, ambient: boolean,
visible: boolean): this;
    effect(effect: Internal.MobEffect_, duration: number, amplifier: number, ambient: boolean,
visible: boolean, showIcon: boolean): this;
    effect(effect: Internal.MobEffect_, duration: number, amplifier: number, ambient: boolean,
visible: boolean, showIcon: boolean, hiddenEffect: Internal.MobEffectInstance_): this;
    effect(effect: Internal.MobEffect_, duration: number): this;
    effect(effect: Internal.MobEffect_, duration: number, amplifier: number): this;
    effect(effect: Internal.MobEffect_): this;
    addEffect(effect: Internal.MobEffectInstance_): this;
    createObject(): Internal.Potion;
    get registryType(): Internal.RegistryObjectBuilderTypes<Internal.Potion>;
    /**
    * Internal constructor, this means that it's not valid unless you use `java()`.
    */
    constructor(i: ResourceLocation);
}
```

Now we see the methods that we can call after this.

So in our code we could write:

```
onEvent('potion.registry', event => {
  event.create('cactus_juice').effect('speed', 10, 5)
})
```