Events

List of Events

This is a list of all events. It's possible that not all events are listed here, but this list will be updated regularly.

Click on event ID to open its class and see information, fields, and methods.

Type descriptions:

- **Startup**: scripts go into the `/kubejs/startup_scripts/` folder. Startup scripts run once, at startup, on both the client and server. Most events that require registering or modifying something at game start (like custom blocks, items, and fluids) will be Startup events.
- **Server**: scripts go into the `/kubejs/server_scripts/` folder. It will be reloaded when you run `/reload` command. Server events are always accessible, even in single-player worlds. Most events that make changes to the world while the game is running (such as breaking blocks, teleporting players, or adding recipes) will go here.
- **Server Startup**: same as Server, and the event will be fired at least once when the server loads.
- **Client**: scripts go into the `/kubejs/client_scripts/` folder. Will be reloaded when you press `F3+T`. Most changes that are per-client (such as resource packs, Painter, and JEI) are client events.
- **Client Startup**: Same as Client, and the event will be fired at least once when the client loads.

Base KubeJS Events

Folder	Method	Cancellable
`/startup_scripts/`	`StartupEvents.init` (link)	□
`/startup_scripts/`	`StartupEvents.postInit` (link)	□

Folder	Method	Cancellable
/startup_scripts/	StartupEvents.registry (fluid) StartupEvents.registry (block) StartupEvents.registry (item) StartupEvents.registry (enchantment) StartupEvents.registry (mob effects) StartupEvents.registry (sound event) StartupEvents.registry (block entity type) StartupEvents.registry (potion) StartupEvents.registry (particle type) StartupEvents.registry (painting variant) StartupEvents.registry (custom stat) StartupEvents.registry (point of interest type) StartupEvents.registry (villager type) StartupEvents.registry (villager profession)	☐
/client_scripts/	ClientEvents.highPriorityAssets (link)	☐
/client_scripts/	ClientEvents.init (link)	☐
/client_scripts/	ClientEvents.loggedIn (link)	☐
/client_scripts/	ClientEvents.loggedOut (link)	☐
/client_scripts/	ClientEvents.tick (link)	☐
/client_scripts/	ClientEvents.painterUpdated (link)	☐
/client_scripts/	ClientEvents.leftDebugInfo (link)	☐
/client_scripts/	ClientEvents.rightDebugInfo (link)	☐
/client_scripts/	ClientEvents.paintScreen (link)	☐
/server_scripts/	ServerEvents.lowPriorityData (link)	☐
/server_scripts/	ServerEvents.highPriorityData (link)	☐
/server_scripts/	ServerEvents.loaded (link)	☐

Folder	Method	Cancellable
/server_scripts/	ServerEvents.unloaded (link)	☐
/server_scripts/	ServerEvents.tick (link)	☐
/server_scripts/	ServerEvents.tags (link)	☐
/server_scripts/	ServerEvents.commandRegistry (link)	☐
/server_scripts/	ServerEvents.command (link)	☐
/server_scripts/	ServerEvents.customCommand (link)	☐
/server_scripts/	ServerEvents.recipes (link)	☐
/server_scripts/	ServerEvents.afterRecipes (link)	☐
/server_scripts/	ServerEvents.specialRecipeSerializers (link)	☐
/server_scripts/	ServerEvents.compostableRecipes (link)	☐
/server_scripts/	ServerEvents.recipeTypeRegistry (link)	☐
/server_scripts/	ServerEvents.genericLootTables (link)	☐
/server_scripts/	ServerEvents.blockLootTables (link)	☐
/server_scripts/	ServerEvents.entityLootTables (link)	☐
/server_scripts/	ServerEvents.giftLootTables (link)	☐
/server_scripts/	ServerEvents.fishingLootTables (link)	☐
/server_scripts/	ServerEvents.chestLootTables (link)	☐
/server_scripts/	LevelEvents.loaded (link)	☐
/server_scripts/	LevelEvents.unloaded (link)	☐
/server_scripts/	LevelEvents.tick (link)	☐
/server_scripts/	LevelEvents.beforeExplosion (link)	☐

Folder	Method	Cancellable
/server_scripts/	LevelEvents.afterExplosion (link)	☐
/startup_scripts/	WorldgenEvents.add (link)	☐
/startup_scripts/	WorldgenEvents.remove (link)	☐
/client_scripts/	NetworkEvents.fromServer (link)	☐
/server_scripts/	NetworkEvents.fromClient (link)	☐
/startup_scripts/	ItemEvents.modification (link)	☐
/startup_scripts/	ItemEvents.toolTierRegistry (link)	☐
/startup_scripts/	ItemEvents.armorTierRegistry (link)	☐
/server_scripts/	ItemEvents.rightClicked (link)	☐
/server_scripts/	ItemEvents.canPickUp (link)	☐
/server_scripts/	ItemEvents.pickedUp (link)	☐
/server_scripts/	ItemEvents.dropped (link)	☐
/server_scripts/	ItemEvents.entityInteracted (link)	☐
/server_scripts/	ItemEvents.crafted (link)	☐
/server_scripts/	ItemEvents.smelted (link)	☐
/server_scripts/	ItemEvents.foodEaten (link)	☐
/client_scripts/	ItemEvents.tooltip (link)	☐
/startup_scripts/	ItemEvents.modelProperties (link)	☐
/client_scripts/	ItemEvents.clientRightClicked (link)	☐
/client_scripts/	ItemEvents.clientLeftClicked (link)	☐
/server_scripts/	ItemEvents.firstRightClicked (link)	☐

Folder	Method	Cancellable
/server_scripts/	ItemEvents.firstLeftClicked (link)	☐
/startup_scripts/	BlockEvents.modification (link)	☐
/server_scripts/	BlockEvents.rightClicked (link)	☐
/server_scripts/	BlockEvents.leftClicked (link)	☐
/server_scripts/	BlockEvents.placed (link)	☐
/server_scripts/	BlockEvents.broken (link)	☐
/server_scripts/	BlockEvents.detectorChanged (link)	☐
/server_scripts/	BlockEvents.detectorPowered (link)	☐
/server_scripts/	BlockEvents.detectorUnpowered (link)	☐
/server_scripts/	BlockEvents.farmlandTrampled (link)	☐
/server_scripts/	EntityEvents.death (link)	☐
/server_scripts/	EntityEvents.hurt (link)	☐
/server_scripts/	EntityEvents.checkSpawn (link)	☐
/server_scripts/	EntityEvents.spawned (link)	☐
/server_scripts/	PlayerEvents.loggedIn (link)	☐
/server_scripts/	PlayerEvents.loggedOut (link)	☐
/server_scripts/	PlayerEvents.respawned (link)	☐
/server_scripts/	PlayerEvents.tick (link)	☐
/server_scripts/	PlayerEvents.chat (link)	☐
/server_scripts/	PlayerEvents.decorateChat (link)	☐
/server_scripts/	PlayerEvents.advancement (link)	☐
/server_scripts/	PlayerEvents.inventoryOpened (link)	☐

Folder	Method	Cancellable
/server_scripts/	PlayerEvents.inventoryClosed (link)	☐
/server_scripts/	PlayerEvents.inventoryChanged (link)	☐
/server_scripts/	PlayerEvents.chestOpened (link)	☐
/server_scripts/	PlayerEvents.chestClosed (link)	☐

Mod Compatibility

> These events are enabled when certain other mods are present.

Just Enough Items (JEI)

Folder	Method	Cancellable
/client_scripts/	JEIEvents.subtypes (source)	☐
/client_scripts/	JEIEvents.hideItems (source)	☐
/client_scripts/	JEIEvents.hideFluids (source)	☐
/client_scripts/	JEIEvents.hideCustom (source)	☐
/client_scripts/	JEIEvents.removeCategories (source)	☐
/client_scripts/	JEIEvents.removeRecipes (source)	☐
/client_scripts/	JEIEvents.addItems (source)	☐
/client_scripts/	JEIEvents.addFluids (source)	☐
/client_scripts/	JEIEvents.information (source)	☐

Roughly Enough Items (REI)

Folder	Method	Cancellable
/client_scripts/	REIEvents.hide (source)	☐

/client_scripts/	REIEvents.add (source)	☐
/client_scripts/	REIEvents.information (source)	☐
/client_scripts/	REIEvents.removeCategories (source)	☐
/client_scripts/	REIEvents.groupEntries (source)	☐

GameStages

Folder	Method	Cancellable
/server_scripts/	GameStageEvents.stageAdded (source)	☐
/server_scripts/	GameStageEvents.stageRemoved (source)	☐

Recipes

> This page is still under development. It is not complete, and subject to change at any time.

> The recipe event is a server event.

Contents

Recipes, Callbacks, and You ↑

The recipe event can be used to add, remove, or replace recipes.

Any script that modifies recipes should be placed in the `kubejs/server_scripts` folder, and can be reloaded at any time with `/reload`.

Any modifications to the recipes should be done within the context of a `recipes` event. This means that we need to register an "event listener" for the `ServerEvents.recipes` event, and give it some code to execute whenever the game is ready to modify recipes. Here's how we tell KubeJS to execute some code whenever it's recipe time:

```
/*
 * ServerEvents.recipes() is a function that accepts another function,
```

```
 * called the "callback", as a parameter. The callback gets run when the
 * server is working on recipes, and then we can make our own changes.
 * When the callback runs, it is also known as the event "firing".
*/

ServerEvents.recipes(event => { //listen for the "recipes" server event.
  // You can replace `event` with any name you like, as
  // long as you change it inside the callback too!

  // This part, inside the curly braces, is the callback.
  // You can modify as many recipes as you like in here,
  // without needing to use ServerEvents.recipes() again.

  console.log('Hello! The recipe event has fired!')
})
```

In the next sections, you can see what to put inside your callback.

Adding Recipes ↑

The following is all code that should be placed *inside* your recipe callback.

Shaped ↑

Shaped recipes are added with the `event.shaped()` method. Shaped recipes must have their ingredients in a specific order and shape in order to match the player's input. The arguments to `event.shaped()` are:

1. The output item, which can have a count of 1-64
2. An array (max length 3) of crafting table rows, represented as strings (max length 3). Spaces represent slots with no item, and letters represent items. The letters don't have to mean anything; you explain what they mean in the next argument.
3. An object mapping the letters to Items, like `{letter: item}`. Input items must have a count of 1.

If you want to force strict positions on the crafting grid or disable mirroring, see Methods of Custom Recipes.

```
event.shaped('3x minecraft:stone', [// arg 1: output
    'A B',
```

```
    ' C ', // arg 2: the shape (array of strings)
    'B A'
  ], {
    A: 'minecraft:andesite',
    B: 'minecraft:diorite',  //arg 3: the mapping object
    C: 'minecraft:granite'
  }
)
```

Shapeless↑

Shapeless recipes are added with the `event.shapeless()` method. Players can put ingredients of shapeless recipes anywhere on the grid and it will still craft. The arguments to `event.shapeless()` are:

1. The output item
2. An array of input items. The total input items' count must be 9 at most.

```
event.shapeless('3x minecraft:dandelion', [ // arg 1: output
  'minecraft:bone_meal',
  'minecraft:yellow_dye', //arg 2: the array of inputs
  '3x minecraft:ender_pearl'
])
```

Smithing↑

Smithing recipes have 2 inputs and one output and are added with the `event.smithing()` method. Smithing recipes are crafted in the smithing table.

```
event.smithing(
  'minecraft:netherite',  // arg 1: output
  'minecraft:iron_ingot', // arg 2: the item to be upgraded
  'minecraft:black_dye'   // arg 3: the upgrade item
)
```

Smelting & Cooking↑

Cooking recipes are all very similar, accepting one input (a single item) and giving one output (which can be up to 64 of the same item). The fuel cannot be changed in this recipe event and should be done with tags instead.

- Smelting recipes are added with `event.smelting()`, and require the regular Furnace.
- Blasting recipes are added with `event.blasting()`, and require the Blast Furnace.
- Smoking recipes are added with `event.smoking()`, and require the Smoker.

- Campfire cooking recipes are added with `event.campfireCooking()`, and require the Campfire.

```
// Cook 1 stone into 3 gravel in a Furnace:
event.smelting('3x minecraft:gravel', 'minecraft:stone')

// Blast 1 iron ingot into 10 nuggets in a Blast Furnace:
event.blasting('10x minecraft:iron_nugget', 'minecraft:iron_ingot')

// Smoke glass into tinted glass in the Smoker:
event.smoking('minecraft:tinted_glass', 'minecraft:glass')

// Burn sticks into torches on the Campfire:
event.campfireCooking('minecraft:torch', 'minecraft:stick')
```

Stonecutting ↑

Like the cooking recipes, stonecutting recipes are very simple, with one input (a single item) and one output (which can be up to 64 of the same item). They are added with the `event.stonecutting()` method.

```
//allow cutting 3 sticks from any plank on the stonecutter
event.stonecutting('3x minecraft:stick', '#minecraft:planks')
```

Custom/Modded JSON recipes ↑

If a mod supports Datapack recipes, you can add recipes for it without any addon mod support! Unfortunately, we can't give specific advice because every mod's layout is different, but if a mod has a GitHub (most do!) or other source code, you can find the relevant JSON files in `/src/generated/resources/data/<modname>/recipes/`. Otherwise, you may be able to find it by unzipping the mod's `.jar` file.

Here's an example of adding a Farmer's Delight cutting board recipe, which defines an input, output, and tool taken straight from their GitHub. Obviously, you can substitute any of the items here to make your own recipe!

```
// Slice cake on a cutting board!
event.custom({
  type: 'farmersdelight:cutting',
  ingredients: [
    { item: 'minecraft:cake' }
  ],
  tool: { tag: 'forge:tools/knives' },
```

```
  result: [
    { item: 'farmersdelight:cake_slice', count: 7 }
  ]
})
```

Here's another example of `event.custom()` for making a Tinkers' Construct alloying recipe, which defines inputs, an output, and a temperature straight from their GitHub (conditions removed):

```
// Adding the Molten Electrum alloying recipe from Tinkers' Construct
event.custom({
  type: 'tconstruct:alloy',
  inputs: [
    { tag: 'forge:molten_gold', amount: 90 },
    { tag: 'forge:molten_silver', amount: 90 }
  ],
  result: { fluid: 'tconstruct:molten_electrum', amount: 180 },
  temperature: 760
})
```

Removing Recipes ↑

Removing recipes can be done with the `event.remove()` method. It accepts 1 argument: a Recipe Filter. A filter is a set of properties that determine which recipe(s) to select. There are many conditions with which you can select a recipe:

- by output item `{output: '<item_id>'}`
- by input item(s) `{input: '<item_id>'}`
- by mod `{mod: '<mod_id>'}`
- by the unique recipe ID `{id: '<recipe_id>'}`
- combinations of the above:
  - Require ALL conditions to be met: `{condition1: 'value', condition2: 'value2'}`
  - Require ANY of the conditions to be met: `[{condition_a: 'true'}, {condition_b: 'true'}]`
  - Require the condition NOT be met: `{not: {condition: 'requirement'}}`

All of the following can go in your recipe callback, as normal:

```
// A blank condition removes all recipes (obviously not recommended):
event.remove({})


// Remove all recipes where output is stone pickaxe:
```

```
event.remove({ output: 'minecraft:stone_pickaxe' })

// Remove all recipes where output has the Wool tag:
event.remove({ output: '#minecraft:wool' })

// Remove all recipes where any input has the Redstone Dust tag:
event.remove({ input: '#forge:dusts/redstone' })

// Remove all recipes from Farmer's Delight:
event.remove({ mod: 'farmersdelight' })

// Remove all campfire cooking recipes:
event.remove({ type: 'minecraft:campfire_cooking' })

// Remove all recipes that grant stone EXCEPT smelting recipes:
event.remove({ not: { type: 'minecraft:smelting' }, output: 'stone' })

// Remove recipes that output cooked chicken AND are cooked on a campfire:
event.remove({ output: 'minecraft:cooked_chicken', type: 'minecraft:campfire_cooking' })

// Remove any blasting OR smelting recipes that output minecraft:iron_ingot:
event.remove([{ type: 'minecraft:smelting', output: 'minecraft:iron_ingot' }, { type: 'minecraft:blasting', output: 'minecraft:iron_ingot' }])

// Remove a recipe by ID. in this case, data/minecraft/recipes/glowstone.json:
// Note: Recipe ID and output are different!
event.remove({ id: 'minecraft:glowstone' })
```

To find a recipe's unique ID, turn on advanced tooltips using the `F3` + `H` keys (you will see an announcement in chat), then hover over the [+] symbol (if using REI) or the output (if using JEI).

Modifying & Replacing Recipes ↑

You can bulk-modify supported recipes using `event.replaceInput()` and `event.replaceOutput()`. They each take 3 arguments:

1. A filter, similar to the ones used when removing recipes
2. The ingredient to replace
3. The ingredient to replace it with (can be a tag)

For example, let's say you were removing all sticks and wanted to make people craft things with saplings instead. Inside your callback you would put:

```
event.replaceInput(
  { input: 'minecraft:stick' }, // Arg 1: the filter
  'minecraft:stick',            // Arg 2: the item to replace
  '#minecraft:saplings'         // Arg 3: the item to replace it with
  // Note: tagged fluid ingredients do not work on Fabric, but tagged items do.
)
```

Advanced Techniques ↑

Helper functions ↑

Are you making a lot of similar recipes? Feel like you're typing the same text over and over? Try helper functions! Helper functions will perform repeated actions in much less space by taking in only the parts that are relevant, then doing the repetitive stuff for you!

Here's a helper function, which allows you to make items by crafting a flower pot around the item you specify.

```
ServerEvents.recipes(event => {
  let potting = (output, pottedInput) => {
    event.shaped(output, [
      'BIB',
      ' B '
    ], {
      B: 'minecraft:brick',
      I: pottedInput
    })
  }

  //Now we can make many 'potting' recipes without typing that huge block over and over!
  potting('kubejs:potted_snowball', 'minecraft:snowball')
  potting('kubejs:potted_lava', 'minecraft:lava_bucket')
  potting('minecraft:blast_furnace', 'minecraft:furnace')
})
```

Looping ↑

In addition to helper functions, you can also loop through an array to perform an action on every item in the array.

Tags

> The tag event is a server event.

> Tags are per item/block/fluid/entity_type and as such cannot be added based on things like NBT data!

The tags event takes an extra parameter that determines which registry it's adding tags to. You will generally only need item, block, and fluid tags. However, it does support adding tags to any registry, including other mods ones. For mod ones make sure to include the namespace!

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

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

  // Get #forge:ingots/copper tag and remove all entries from it
  event.removeAll('forge:ingots/copper')

  // Required for FTB Quests to check item NBT
  event.add('itemfilters:check_nbt', 'some_item:that_has_nbt_types')

  // You can create new tags the same way you add to existing, just give it a name
  event.add('forge:completely_new_tag', 'minecraft:clay_ball')

  // It supports adding tags to tags as well. Just prefix the second tag with #
  event.add('c:stones', '#forge:stone')

  // Removes all tags from this entry
  event.removeAllTagsFrom('minecraft:stick')

  // Add all items from the forge:stone tag to the c:stone tag, unless the id contains diorite
  const stones = event.get('forge:stone').getObjectIds()
  const blacklist = Ingredient.of(/.*diorite.*/)
```

```
  stones.forEach(stone => {
    if (!blacklist.test(stone)) event.add('c:stone', stone)
  })
})
```

> Recipes use item tags, not block or fluid tags. Even if items representing those are blocks, like `minecraft:cobblestone`, it still uses an item tag for recipes.

```
// Listen to the block tag event
ServerEvents.tags('block', event => {
  // Add tall grass to the climbable tag. This does make it climbable!
  event.add('minecraft:climbable', 'minecraft:tall_grass')
})
```

Custom Items

> The custom item event is a startup event.

Custom items are created in a startup script. They cannot be reloaded without restarting the game. The event is not cancellable.

```
// Listen to item registry event
StartupEvents.registry('item', e => {
  // The texture for this item has to be placed in kubejs/assets/kubejs/textures/item/test_item.png
  // If you want a custom item model, you can create one in Blockbench and put it in kubejs/assets/kubejs/models/item/test_item.json
  e.create('test_item')

  // If you want to specify a different texture location you can do that too, like this:
  e.create('test_item_1').texture('mobbo:item/lava') // This texture would be located at kubejs/assets/mobbo/textures/item/lava.png

  // You can chain builder methods as much as you like
  e.create('test_item_2').maxStackSize(16).glow(true)

  // You can specify item type as 2nd argument in create(), some types have different available methods
  e.create('custom_sword', 'sword').tier('diamond').attackDamageBaseline(10.0)
})
```

Valid item types:

- basic (this is the default)
- sword
- pickaxe
- axe
- shovel
- shears
- hoe
- helmet
- chestplate
- leggings

- boots

Other methods item builder supports: [you can chain these methods after create()]

- `maxStackSize(size)`
- `displayName(name)`
- `unstackable()`
- `maxDamage(damage)` This is the item's durability, not actual weapon damage.
- `burnTime(ticks)`
- `containerItem(item_id)`
- `rarity('rarity')`
- `tool(type, level)`
- `glow(true/false)`
- `tooltip(text...)`
- `group('group_id')`
- `color(index, colorHex)`
- `texture(customTextureLocation)`
- `parentModel(customParentModelLocation)`
- `food(foodBuilder => ...)` For full syntax see below

Methods available if you use a tool type (`'sword'`, `'pickaxe'`, `'axe'`, `'shovel'` or `'hoe'`):

- `tier('toolTier')`
- `modifyTier(tier => ...)` Same syntax as custom tool tier, see Custom Tiers
- `attackDamageBaseline(damage)` You only want to modify this if you are creating a custom weapon such as Spear, Battleaxe, etc.
- `attackDamageBonus(damage)`
- `speedBaseline(speed)` Same as attackDamageBaseline, only modify for custom weapon types
- `speed(speed)`

Default available tool tiers:

- wood
- stone
- iron
- gold
- diamond
- netherite

Methods available if you use an armour type ('helmet', 'chestplate', 'leggings' or 'boots'):

- `tier('armorTier')`
- `modifyTier(tier => ...)` Same syntax as custom armor tier, see Custom Tiers

Default available armor tiers:

- leather
- chainmail
- iron
- gold
- diamond
- turtle
- netherite

Vanilla group/creative tab IDs:

- search
- buildingBlocks
- decorations
- redstone
- transportation
- misc
- food
- tools
- combat
- brewing

Custom Foods

```
StartupEvents.registry('item', event => {
  event.create('magic_steak').food(food => {
    food
      .hunger(6)
      .saturation(6)//This value does not directly translate to saturation points gained
        //The real value can be assumed to be:
        //min(hunger * saturation * 2 + saturation, foodAmountAfterEating)
        .effect('speed', 600, 0, 1)
        .removeEffect('poison')
        .alwaysEdible()//Like golden apples
        .fastToEat()//Like dried kelp
        .meat()//Dogs are willing to eat it
        .eaten(ctx => {//runs code upon consumption
          ctx.player.tell(Text.gold('Yummy Yummy!'))
```

```
        //If you want to modify this code then you need to restart the game.
        //However, if you make this code call a global startup function
        //and place the function *outside* of an event handler
        //then you may use the command:
        //  /kubejs reload startup_scripts
        //to reload the function instantly.
        //See example below
      })
})

  event.create('magicer_steak').unstackable().food(food => {
    food
      .hunger(7)
      .saturation(7)
      // This references the function below instead of having code directly, so it is reloadable!
      .eaten(ctx => global.myAwesomeReloadableFunction(ctx))
  })
})

global.myAwesomeReloadableFunction = ctx => {
  ctx.player.tell('Hello there!')
  ctx.player.tell(Text.of('Change me then reload with ').append(Text.red('/kubejs reload
startup_scripts')).append(' to see your changes!'))
}
```

Item modification

> Item modification is a startup event.

`ItemEvents.modification` is a startup script event used to modify various properties of existing items.

```
ItemEvents.modification(event => {
  event.modify('minecraft:ender_pearl', item => {
    item.maxStackSize = 64
    item.fireResistant = true
    item.rarity = "UNCOMMON"
  })
  event.modify('minecraft:ancient_debris', item => {
    item.rarity = "RARE"
    item.burnTime = 16000
  })
  event.modify('minecraft:turtle_helmet', item => {
    item.rarity = "EPIC"
    item.maxDamage = 481
    item.craftingRemainder = Item.of('minecraft:scute').item
  })
})
```

Available properties:

Property	Value Type	Description
maxStackSize	int	Sets the maximum stack size for items. Default is 64 for most items.
maxDamage	int	Sets the maximum damage an item can take before it is broken.
craftingRemainder	Item	Sets the item left behind in the crafting grid when this item is used as a crafting ingredient (like milk buckets in the cake recipe). Most items do not have one.

Property	Value Type	Description
fireResistant	boolean	If this item burns in fire and lava. Most items are false by default, but Ancient Debris and Netherite things are not.
rarity	Rarity	Sets the items rarity. This is mainly used for the name colour. COMMON by default. Nether Stars and Elytra are UNCOMMON, Golden Apples are RARE and Enchanted Golden Apples are EPIC.
burnTime	int	Sets the burn time (in ticks) in a regular furnace for this item. Note that Smokers and Blast Furnaces burn fuel twice as fast. Coal is 1600.
foodProperties	FoodProperties	Sets the items food properties to the provided properties. Can be `null` to remove food properties.
foodProperties	Consumer<FoodBuilder>	Sets the properties according to the consumer. See below for more info.
digSpeed	float	Sets the items digging speed to the number provided. See table below for defaults.
tier	Consumer<MutableToolTier>	Currently BROKEN! Sets the tools tier according to the consumer. See below for more info.
attackDamage	double	Sets the attack damage of this item.
attackSpeed	double	Sets the attack speed of this item
armorProtection	double	Sets the armor protection for this item. 20 is a full armour bar.
armorToughness	double	Adds an armor toughness bonus.
armorKnockbackResistance	double	Add an armor knockback resistance bonus. Can be negative. 1 is full knockback resistance.

Tool defaults

Tier	level	maxDamage	digSpeed	attackDamage (this is a bonus modified by the tool type value, not the final value)	enchantmentValue
Wood	0	59	2	0	15
Stone	1	131	4	1	5
Iron	2	250	6	2	14
Diamond	3	1561	8	3	10
Gold	0	32	12	0	22
Netherite	4	2031	9	4	15

Armor defaults

All boxes with multiple values are formatted [head, chest, legs, feet]. Boxes with single values are the same for every piece.

Tier	maxDamage	armourProtection	armorToughness	armorKnockbackResistance
Leather	[65, 75, 80, 55]	[1, 2, 3, 1]	0	0
Chain	[195, 225, 240, 165]	[1, 4, 5, 2]	0	0
Iron	[195, 225, 240, 165]	[2, 5, 6, 2]	0	0
Gold	[91 ,105, 112, 77]	[1, 3, 5, 2]	0	0
Diamond	[429, 495, 528, 363]	[3, 6, 8, 3]	2	0
Turtle (only has helmet)	[325, nil, nil. nil]	[2, nil, nil, nil]	0	0
Netherite	[481, 555, 592, 407]	[3, 6, 8, 3]	3	0.1
Elytra (not actually armor)	[nil, 432, nil, nil]	0	0	0

Tier

> Broken at the moment! https://github.com/KubeJS-Mods/KubeJS/issues/662. Use the non tier methods instead.

Tools

```
ItemEvents.modification(event => {
  event.modify('golden_sword', item => {
    item.tier = tier => {
        tier.speed = 12
        tier.attackDamageBonus = 10
        tier.repairIngredient = '#forge:storage_blocks/gold'
        tier.level = 3
    }
  })
  event.modify('wooden_sword', item => {
    item.tier = tier => {
        tier.enchantmentValue = 30
    }
  })
})
```

Property	Value Type	Description
uses	int	The maximum damage before this tool breaks. Identical to maxDamage.
speed	float	The digging speed of this tool.
attackDamageBonus	float	The bonus attack damage of this tool.
level	int	The mining level of this tool.
enchantmentValue	int	The enchanting power of this tool. The higher this is, the better the enchantments at an Enchanting Table are.
repairIngredient	Ingredient	The material used to repair this tool in an anvil.

Armor

Doesnt actually exist/work at the moment. Sorry.

Food

```
ItemEvents.modification(event => {
  event.modify('minecraft:diamond', item => {
    item.foodProperties = food => {
        food.hunger(2)
        food.saturation(3)
```

```
        food.fastToEat(true)
        food.eaten(e => e.player.tell('you ate')) // this is broken, use ItemEvents.foodEaten instead.
    }
  })

  event.modify('pumpkin_pie', item => {
    item.foodProperties = null // make pumpkin pies inedible
  })
})
```

Method	Parameters	Description
hunger	int h	Sets the hunger restored when this item is eaten
saturation	float s	Sets the saturation mulitplier when this food is eaten. This is not the final value, it goes through some complicated maths first
meat	boolean flag (optional, true by default)	Sets if this item is considered meat. Meat can be fed to wolves to heal them.
alwaysEdible	boolean flag (optional, true by default)	If this item can be eaten even if your food bar is full. Chorus Fruit has this true by default.
fastToEat	boolean flag (optional, true by default)	If this item is fast to eat, like Dried Kelp.
effect	ResourceLocation mobEffectId, int duration, int amplifier, float probability	Adds an effect to the entity who eats this, like a Golden Apple
removeEffect	MobEffect mobEffect	Removes the effect from the entity who eats this, like Honey Bottles (poison).
eaten	Consumer<FoodEatenEventJS> e	BROKEN! Use ItemEvents.foodEaten in server scripts instead.

Custom Blocks

> This is a startup script, meaning that you will need to *restart your game* each time you want to make changes to it.

You can register many types of custom blocks in KubeJS. Here's the simplest way:

```
StartupEvents.registry("block", (event) => {
  event.create("example_block") // Create a new block with ID "kubejs:example_block"
})
```

That's it! Launch the game, and assuming you've left KubeJS's auto-generated resources alone, there should be a fully-textured block in the Creative menu under KubeJS (purple dye). KubeJS will also generate the name "Example Block" for you.

To make modifications to this block, we use the **block builder** returned by the `event.create()` call. The block builder allows us to chain together multiple modifications. Let's try making some of the more common modifications:

```
StartupEvents.registry("block", (event) => {
    event.create("example_block") // Create a new block
    .displayName("My Custom Block") // Set a custom name
    .material("wood") // Set a material (affects the sounds and some properties)
    .hardness(1.0) // Set hardness (affects mining time)
    .resistance(1.0) // Set resistance (to explosions, etc)
    .tagBlock("my_custom_tag") // Tag the block with `#minecraft:my_custom_tag` (can have multiple tags)
    .requiresTool(true) // Requires a tool or it won't drop (see tags below)
    .tagBlock("my_namespace:my_other_tag") // Tag the block with `#my_namespace:my_other_tag`
    .tagBlock("mineable/axe") //can be mined faster with an axe
    .tagBlock("mineable/pickaxe") // or a pickaxe
    .tagBlock('minecraft:needs_iron_tool') // the tool tier must be at least iron
})
```

All Block Builder Methods

In case it wasn't covered above, here's list of each method you can use when building a block.

- `displayName('name')`

    - Sets the item's display name.
- `material('material')` (No longer supported in 1.20+, see `mapColor` and `soundType` below!)
    - Set the item's material to an available material from the Materials List:

Materials List

air
amethyst
bamboo
bamboo_sapling
barrier
bubble_column
buildable_glass
cactus
cake
clay
cloth_decoration
decoration
dirt
egg
explosive
fire
froglight
frogspawn
glass
grass
heavy_metal
ice
ice_solid
lava
leaves
metal
moss
nether_wood
piston
plant
portal
powder_snow
replaceable_fireproof_plant
replaceable_plant
replaceable_water_plant
sand
sculk
shulker_shell
snow

sponge
stone
structural_air
top_snow
vegetable
water
water_plant
web
wood
wool

- `mapColor(MapColor)` (1.20.1+ only)
  - Set block map color, you can find the entire list here, use ID in lowercase, e.g. `'color_light_green'`.
- `soundType(SoundType)` (1.20.1+ only)
  - Set block sound type:

SoundType List

Instead of using `soundType(SoundType)` you can also use one of these shortcut methods:

- `noSoundType()`
- `woodSoundType()`
- `stoneSoundType()`
- `gravelSoundType()`
- `grassSoundType()`
- `sandSoundType()`
- `cropSoundType()`
- `glassSoundType()`

wood
gravel
grass
lily_pad
stone
metal
glass
wool
sand
snow
powder_snow
ladder
anvil
slime_block

honey_block
wet_grass
coral_block
bamboo
bamboo_sapling
scaffolding
sweet_berry_bush
crop
hard_crop
vine
nether_wart
lantern
stem
nylium
fungus
roots
shroomlight
weeping_vines
twisting_vines
soul_sand
soul_soil
basalt
wart_block
netherrack
nether_bricks
nether_sprouts
nether_ore
bone_block
netherite_block
ancient_debris
lodestone
chain
nether_gold_ore
gilded_blackstone
candle
amethyst
amethyst_cluster
small_amethyst_bud
medium_amethyst_bud
large_amethyst_bud
tuff
calcite
dripstone_block
pointed_dripstone
copper

cave_vines
spore_blossom
azalea
flowering_azalea
moss_carpet
pink_petals
moss
big_dripleaf
small_dripleaf
rooted_dirt
hanging_roots
azalea_leaves
sculk_sensor
sculk_catalyst
sculk
sculk_vein
sculk_shrieker
glow_lichen
deepslate
deepslate_bricks
deepslate_tiles
polished_deepslate
froglight
frogspawn
mangrove_roots
muddy_mangrove_roots
mud
mud_bricks
packed_mud
hanging_sign
nether_wood_hanging_sign
bamboo_wood_hanging_sign
bamboo_wood
nether_wood
cherry_wood
cherry_sapling
cherry_leaves
cherry_wood_hanging_sign
chiseled_bookshelf
suspicious_sand
suspicious_gravel
decorated_pot
decorated_pot_cracked

You can construct your own sound type with `new SoundType(volume, pitch, breakSound, stepSound, placeSound, hitSound, fallSound)` where volume and pitch are floats 0.0 - 1.0 (usually leave it as 1.0) and all sounds are SoundEvents.

- `property(BlockProperty)`
  - Adds more blockstates to the block, like being waterlogged or facing a certain direction. A full list of properties is available in the Properties List:

Properties List

Usage: `.property(BlockProperties.PICKLES)`

Boolean Properties (true/false):

attached,
berries,
bloom,
bottom,
can_summon,
conditional,
disarmed,
down,
drag,
east,
enabled,
extended,
eye,
falling,
hanging,
has_book,
has_bottle_0,
has_bottle_1,
has_bottle_2,
has_record,
inverted,
in_wall,
lit,
locked,
north,
occupied,
open,
persistent,
powered,
short,

shrieking,
signal_fire,
snowy,
south,
triggered,
unstable,
up,
vine_end,
waterlogged,
west

Integer properties:

age_1,
age_2,
age_3,
age_4,
age_5,
age_7,
age_15,
age_25,
bites,
candles,
delay,
distance,
eggs,
hatch,
layers,
level,
level_cauldron,
level_composter,
level_flowing,
level_honey,
moisture,
note,
pickles,
power,
respawn_anchor_charges,
rotation_16,
stability_distance,
stage

Directional Properties:

facing,
facing_hopper,
horizontal_facing,
vertical_direction

Other (`enum`) Properties:

attach_face,
axis,
bamboo_leaves,
bed_part,
bell_attachment,
chest_type,
door_hinge,
double_block_half,
dripstone_thickness,
east_redstone,
east_wall,
half,
horizontal_axis,
mode_comparator,
north_redstone,
north_wall,
noteblock_instrument,
orientation,
piston_type,
rail_shape,
rail_shape_straight,
sculk_sensor_phase,
slab_type,
south_redstone,
south_wall,
stairs_shape,
structureblock_mode,
tilt,
west_redstone,
west_wall

- `tagBlock('namespace:tag_name')`
  - adds a tag to the block
- `tagItem('namespace:tag_name')`
  - adds a tag to the block's item, if it has one
- `tagBoth('namespace:tag_name')`
  - adds both block and item tag if possible
- `hardness(float)`

    - Sets the block's Hardness value. Used for calculating the time it takes for the block to be destroyed.
- `resistance(float)`
    - Set's the block's resistance to things like explosions
- `unbreakable()`
    - Shortcut to set the resistance to MAX_VALUE and hardness to -1 (like bedrock)
- `lightLevel(number)`
    - Sets the block's light level.
    - Passing an integer (0-15) will set the block's light level to that value.
    - Passing a float (0.0-1.0) will multiply that number by 15, then set the block's light level to the nearest integer
- `opaque(boolean)`
    - Sets whether the block is opaque. Full, opaque blocks will not let light through.
- `fullBlock(boolean)`
    - Sets whether the block renders as a full block. Full blocks have certain optimizations applied to them, such as not rendering terrain behind them. If you're using `.box()` to make a custom hitbox, please set this to `false`.
- `requiresTool(boolean)`
    - If `true`, the block will use certain block tags to determine whether it should drop an item when mined. For example, a block tagged with `#minecraft:mineable/axe`, `#minecraft:mineable/pickaxe`, and `#minecraft:needs_iron_tool` would drop nothing unless it was mined with an axe or pickaxe that was at least iron level.
- `renderType('solid'|'cutout'|'translucent')`
    - Sets the render type.
        - `cutout` is required for blocks with texture like glass, where pixels are either transparent or not
        - `translucent` is required for blocks like stained glass, where pixels can be semitransparent
        - otherwise, use `solid` if all pixels in your block are opaque.
- `color(tintindex, color)`
    - Recolors a block to a certain color
- `textureAll('texturepath')`
    - Textures all 6 sides of the block to the same texture.
    - The path must look like `kubejs:block/texture_name` (which would be included under `kubejs/assets/kubejs/textures/block/texture_name.png`).
    - Defaults to `kubejs:block/<block_name>`
- `texture('side', 'texturepath')`
    - Texture one side by itself. Valid sides are `up`, `down`, `north`, `south`, `east`, and `west`.
- `model('modelpath')`
    - Specify a custom model.
    - The path must look like `kubejs:block/texture_name` (which would be included under `kubejs/assets/kubejs/models/block/texture_name.png`).
    - Defaults to `kubejs:block/<block_name>`.
- `noItem()`
    - Removes the associated item. Minecraft does this by default for a few blocks, like nether portal blocks. Use this if the player should never be able to hold or place the block.

- `box(x0, y0, z0, x1, y1, z1, boolean)`
- `box(x0, y0, z0, x1, y1, z1)` // defaults to true
  - Sets a custom hitbox for the block, affecting collision. You can use this multiple times to define a complex shape composed of multiple boxes.
  - Each box is a rectangular prism with corners at (x0,y0,z0) and (x1,y1,z1)
  - You will probably want to set up a custom block model that matches the shape you define here.
  - The final boolean determines the coordinate scale of the box. Passing in `true` will use the numbers 0-16, while passing in `false` will use coordinates ranging from 0.0 to 1.0
- `noCollision()`
  - Removes the default full-block hitbox, allowing you to fall through the block.
- `notSolid()`
  - Tells the renderer that the block isn't solid.
- `waterlogged()`
  - Allows the block to be waterloggable.
- `noDrops()`
  - The block will not drop itself, even if mined with silk touch.
- `slipperiness(float)`
  - Sets the slipperiness of the block. Affects how much entities slide while moving on it. Almost every block in Vanilla has a slipperiness value of 0.6, except slime (0.8) and ice (0.98).
- `speedFactor(float)`
  - A modifier affecting how quickly players walk on the block.
- `jumpFactor(float)`
  - A modifier affecting how high players can jump off the block.
- `randomTick(consumer<randomTickEvent>)`
  - A function to run when the block recieves a random tick.
- `item(consumer<itemBuilder>)`
  - Modify certain properties of the block's item (see link)
- `setLootTableJson(json)`
  - Pass in a custom loot table JSON directly
- `setBlockstateJson(json)`
  - Pass in a custom blockstate JSON directly
- `setModelJson(json)`
  - Pass in a custom model JSON directly
- `noValidSpawns(boolean)`
  - If `true`, the block is not counted as a valid spawnpoint for entities
- `suffocating(boolean)`
  - Whether the block will suffocate entities that have their head inside it
- `viewBlocking(boolean)`
  - Whether the block counts as blocking a player's view.
- `redstoneConductor(boolean)`
  - Sets whether the block will conduct redstone. True by default.
- `transparent(boolean)`
  - Sets whether the block is transparent or not

- `defaultCutout()`
  - batches a bunch of methods to make blocks such as glass
- `defaultTranslucent()`
  - similar to defaultCutout() but using translucent layer instead

Block Modification

> The block modification event is a startup event.

`BlockEvents.modification` event is a startup script event that allows you to change properties of existing blocks.

```
BlockEvents.modification(e => {
  e.modify('minecraft:stone', block => {
    block.destroySpeed = 0.1
    block.hasCollision = false
  })
})
```

All available properties:

- `String material`
- `boolean hasCollision`
- `float destroySpeed`
- `float explosionResistance`
- `boolean randomlyTicking`
- `String soundType`
- `float friction`
- `float speedFactor`
- `float jumpFactor`
- `int lightEmission`
- `boolean requiresTool`

Custom Tiers

> The custom tier event is a startup event.

You can make custom tiers for armor and tools in a startup script. They are not reloadable without restarting the game. The events are not cancellable.

Tool tiers

```
ItemEvents.toolTierRegistry(event => {
  event.add('tier_id', tier => {
    tier.uses = 250
    tier.speed = 6.0
    tier.attackDamageBonus = 2.0
    tier.level = 2
    tier.enchantmentValue = 14
    tier.repairIngredient = '#forge:ingots/iron'
  })
})
```

Armor tiers

```
ItemEvents.armorTierRegistry(event => {
  event.add('tier_id', tier => {
    tier.durabilityMultiplier = 15 // Each slot will be multiplied with [13, 15, 16, 11]
    tier.slotProtections = [2, 5, 6, 2] // Slot indicies are [FEET, LEGS, BODY, HEAD]
    tier.enchantmentValue = 9
    tier.equipSound = 'minecraft:item.armor.equip_iron'
    tier.repairIngredient = '#forge:ingots/iron'
    tier.toughness = 0.0 // diamond has 2.0, netherite 3.0
    tier.knockbackResistance = 0.0
  })
})
```

Worldgen

General Notes

Biome Filters:

Biome filters work similarly to *recipe filters* and can be used to create complex and exact filters to fine-tune where your features may and may not spawn in the world. They are used for the `biomes` field of a feature and may look something like this:

```
WorldgenEvents.add(event => {
  event.addOre(ore => {
    // let's look at all of the 'simple' filters first
    ore.biomes = 'minecraft:plains' □□// only spawn in exactly this biome
    ore.biomes = /^minecraft:.*/□□□// spawn in all biomes that match the given pattern
    ore.biomes = '#minecraft:is_forest' □// spawn in all biomes tagged as 'minecraft:is_forest'

    // filters can be arbitrarily combined using AND, OR and NOT logic
    ore.biomes = {}□□□□□□□// empty AND filter, always true
    ore.biomes = []□□□□□□□// empty OR filter, always true
    ore.biomes = { not: 'minecraft:ocean' }□// spawn in all biomes that are NOT 'minecraft:ocean'

    // since AND filters are expressed as maps and expect string keys,
    // all of the 'primitive' filters can also be expressed as such
    ore.biomes = {□□□□□// see above for an explanation of these filters
      id: 'minecraft:plains',
      id: /^minecraft:.*/,□□□// regex (also technically an ID filter)
      tag: 'minecraft:is_forest',
    }
    // note all of the above syntax may be mixed and matched individually
  })
})
```

Rule Tests and Targets:

In 1.18, Minecraft WorldGen has changed to a "target-based" replacement system, meaning you can specify specific blocks to be replaced with specific other blocks within the same feature

configuration. (For example, this is used to replace Stone with the normal ore and Deepslate with the Deepslate ore variant).

Each target gets a "rule test" as input (something that checks if a given block state should be replaced or not) and produces a specific output block state. While scripting, both of these concepts are expressed as the same class: `BlockStatePredicate`.

Syntax-wise, `BlockStatePredicate` is pretty similar to biome filters as they too can be combined using `AND` or `OR` filters (which is why we will not be repeating that step here), and can be used to match one of three different things fundamentally:

1. **Blocks:** these are simply parsed as strings, so for example `'minecraft:stone'` to match Stone
2. **Block States:** these are parsed as the block ID followed by an array of properties. You would use something like `'minecraft:furnace[lit=true]'` to match only Furnace blocks that are lit. You can use `F3` to figure out a block's properties, as well as possible values through using the debug stick.
3. **Block Tags**: these are parsed in the "familiar" tag syntax, so you could use `'#minecraft:base_stone_overworld'` to match all types of stone that can be found generating in the ground in the Overworld.

> Note that these are **block** tags, not **item** tags. They may (and probably will) be different! (F3 is your friend!)

> You can also use regular expressions with block filters, so `/^mekanism:.+_ore$/` would match any block from Mekanism whose ID ends with `_ore`. Keep in mind this will *not* match block state properties!

> When a `RuleTest` is required instead of a `BlockStatePredicate`, you can also supply that rule test directly in the form of a JavaScript object (it will then be parsed the same as vanilla would parse JSON or NBT objects). This can be useful if you want rule tests that have a random chance to match.

More examples of how targets work can be found in the example script down below.

Height Providers:

Another system that may appear a bit confusing at first is the system of "height providers", which are used to determine at what Y level a given ore should spawn and with what frequency. Used in tandem with this feature are the so-called "vertical anchors", which may be used to get the height of something relative to a specific anchor point (for example the top or bottom of the world).

In KubeJS, this system has been simplified a bit to make it easier to use for script developers. There are two common types of ore placement:

1. **Uniform**: has the same chance to spawn anywhere in between the two anchors
2. **Triangle**: is more likely to spawn in the center of the two anchors than it is to spawn further outwards

To use these two, you can use the methods `uniformHeight` and `traingleHeight` in `AddOreProperties`, respectively. Vertical anchors have also been simplified, as you can use the `aboveBottom` / `belowTop` helper methods in `AddOreProperties`.

Once again, see the example script for more information!

Example script

```
WorldgenEvents.add(event => {
  // use the anchors helper from the event
  const { anchors } = event

  event.addOre(ore => {
    ore.id = 'kubejs:glowstone_test_lmao' // (optional, but recommended) custom id for the feature
    ore.biomes = {
      not: 'minecraft:savanna' // biome filter, see above (technically also optional)
    }

    // examples on how to use targets
    ore.addTarget('#minecraft:stone_ore_replaceables', 'minecraft:glowstone') // replace anything in the vanilla stone_ore_replaceables tag with Glowstone
    ore.addTarget('minecraft:deepslate', 'minecraft:nether_wart_block')       // replace Deepslate with Nether Wart Blocks
    ore.addTarget([
      'minecraft:gravel',     // replace gravel...
      /minecraft:(.*)_dirt/   // or any kind of dirt (including coarse, rooted, etc.)...
    ], 'minecraft:tnt')       // with TNT (trust me, it'll be funny)

    ore.count([15, 50])            // generate between 15 and 50 veins (chosen at random), you could use a single number here for a fixed amount of blocks
      .squared()                   // randomly spreads the ores out across the chunk, instead of generating them in a column
      .triangleHeight(□□□□      // generate the ore with a triangular distribution, this means it will be more likely to be placed closer to the center of the anchors
        anchors.aboveBottom(32),    // the lower bound should be 32 blocks above the bottom of the world, so in this case, Y = -32 since minY = -64
```

```
      anchors.absolute(96)      // the upper bound, meanwhile is set to be just exactly at Y = 96
    )                 // in total, the ore can be found between Y levels -32 and 96, and will be most likely at Y = (9
32) / 2 = 32

    // more, optional parameters (default values are shown here)
    ore.size = 9                        // max. vein size
    ore.noSurface = 0.5                 // chance to discard if the ore would be exposed to air
    ore.worldgenLayer = 'underground_ores'  // what generation step the ores should be generated in (see below)
    ore.chance = 0              // if != 0 and count is unset, the ore has a 1/n chance to generate per chunk
  })

  // oh yeah, and did I mention lakes exist, too?
  // (for now at least, Vanilla is likely to remove them in the future)
  event.addLake(lake => {
    lake.id = 'kubejs:funny_lake' // BlockStatePredicate
    lake.chance = 4
    lake.fluid = 'minecraft:lava'
    lake.barrier = 'minecraft:diamond_block'
  })
})

WorldgenEvents.remove(event => {
  // print all features for a given biome filter
  event.printFeatures('', 'minecraft:plains')

  event.removeOres(props => {
    // much like ADDING ores, this supports filtering by worldgen layer...
    props.worldgenLayer = 'underground_ores'
    // ...and by biome
    props.biomes = [
      { category: 'icy' },
      { category: 'savanna' },
      { category: 'mesa' }
    ]

    // BlockStatePredicate to remove iron and copper ores from the given biomes
    // Note tags may NOT be used here, unfortunately...
    props.blocks = ['minecraft:iron_ore', 'minecraft:copper_ore']
  })

  // remove features by their id (first argument is a generation step)
```

```
  event.removeFeatureById('underground_ores', ['minecraft:ore_coal_upper', 'minecraft:ore_coal_lower'])
})
```

Generation Steps

1. `raw_generation`
2. `lakes`
3. `local_modifications`
4. `underground_structures`
5. `surface_structures`
6. `strongholds`
7. `underground_ores`
8. `underground_decoration`
9. `fluid_springs`
10. `vegetal_decoration`
11. `top_layer_modification`

> It's possible you may not be able to generate some things in their layer, like ores in Dirt, because Dirt hasn't spawned yet. You may have to change the layer to one of the above generation steps by calling `ore.worldgenLayer = 'top_layer_modification'`. However, this is not recommended.

> Nether ores are generated in `underground_decoration` step!