Worldgen Events

> These following examples will only work on **1.18+!** If you need examples for 1.16, you can look *here* if you want to add new features to world generation and *here* if you want to remove features from it.

General Notes

Biome Filters:

Biome filters work similarly to *recipe filters*, and can be used to create complex and exact filters to fine tune exactly 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:

```
onEvent('worldgen.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 (here: anything that starts with minecraft:)
    ore.biomes = '#minecraft:is_forest' □// [1.19+] spawn in all biomes tagged as 'minecraft:is_forest'
    ore.biomes = '^nether' □□□□// [1.18 only!] spawn in all biomes in the 'nether' category (see Biome Categories)
    ore.biomes = '$hot'□□□□□// [Forge 1.18 only!] spawn in all biomes that have the 'hot' biome type (see Biome Dictionary)
    // filters can be arbitrarily combined using AND, OR and NOT logic
    ore.biomes = {}□□□□□□// empty AND filter, always true
    ore.biomes = []□□□□□□// empty OR filter, also always true
    ore.biomes = {
      not: 'minecraft:ocean'□□□□// spawn in all biomes that are NOT 'minecaraft: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',
      category: '^nether',
      biome_type: '$hot',
    }
    // note all of the above syntax may be mixed and matched individually
    // for example, this will spawn the feature in any biome that is
    // either plains, or a hot biome that is not in the nether or savanna categories
    ore.biomes = [
      'minecraft:plains', {
        biome_type: '$hot',
        not: [
          '#nether',
          { category: 'savanna' }
        ]
      },
    ]
  })
})
```

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. (This is used to replace stone with the normal ore and deepslate with the deepslate ore variant, for example).

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. On the KubeJS script side, 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, so you would use something like `"minecraft:furnace[lit=true]"` to match only furnace blocks that are active (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 (as you might expect, these are parsed in the "familiar" tag syntax, so you could for example 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, so 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, for example!

More examples on 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: To use the two most common types of ore placement, *uniform* (the feature has the same chance to spawn anywhere in between the two anchors) and *triangle* (the feature is more likely to spawn in the center of the two anchors than it is to spawn further outwards), 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, or, in newer KubeJS versions, the builtin class wrapper for `VerticalAnchor` (Note that the former has been deprecated in favour of the latter), as well as if you want to specify absolute heights as simple numbers, instead.

Once again, see the example script for more information!

(1.18 only!) Biome Categories:

Biome categories are a vanilla system that can be used to roughly sort biomes into predefined categories, which are noted below. Note that other mods *may* add more categories through extending the enum, however since there is no way for us to know this we will only provide you with the vanilla IDs here:

- taiga
- extreme_hills
- jungle
- mesa

- plains
- savanna
- icy
- the_end
- beach
- forest
- ocean
- desert
- river
- swamp
- mushroom
- nether

(1.18 and Forge only!) Biome Dictionary:

Much like Vanilla biome categories, Forge uses a "Biome Dictionary" to sort biomes based on their properties. Note that this system is *designed* to be extended by mods, so there is no way for us to give a complete list of all categories to you, however some of the ones you might commonly find yourself using are listed here:

- hot
- cold
- wet
- dry
- sparse
- dense
- spooky
- dead
- lush
- etc.... see BiomeDictionary for more

> In 1.19, both of these systems have been removed **with no replacement** in favour of biome tags!

Example script: (kubejs/startup_scripts/worldgen.js)

```
onEvent('worldgen.add', event => {
  // use the anchors helper from the event (NOTE: this requires newer versions of KubeJS)
  // if you are using an older version of KubeJS, you can use the methods in the ore properties class
  const { anchors } = event

  event.addOre(ore => {
    ore.id = 'kubejs:glowstone_test_lmao' // (optional, but recommended) custom id for the feature
```

```
    ore.biomes = {
      not: '^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 = (96 - 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'
```

```
  })
})

onEvent('worldgen.remove', event => {
  console.info('HELP')
  //console.debugEnabled = true;

  // 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. vegetal_decoration
10. 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. So you may have to change the layer to one of the above generation steps by calling `ore.worldgenLayer = 'top_layer_modification'`. This is, however, not recommended.

Revision #5
Created 17 July 2022 15:14:53 by Max
Updated 27 January 2023 21:18:26 by Lexxie