

Nusantara modding guide

This document shows how to modify existing and implement new features using the [Nusantara](#) engine. Almost all content in the base game is implementable using these methods.

In general, modding boils down to modifying **config files** and **scripts**. [VSCode](#) works well enough for both purposes.

1. Config files are written in [YAML](#) and their extension is **.yaml**. They contain "static" data that describes, e.g, all the units in a faction or the contents of a map.
 - [VSCode extension for YAML files](#)
1. Script files are written in [NusantaraScript language](#) and their extension is **.ns**. They describe dynamic behaviours at game's runtime, such as effects, scenarios, AI behavior, etc.
 - [VSCode extension for NS files](#)

Table of contents:

- [Nusantara Data](#)
- [Nusantara modding guide](#)
 - [The *data* folder](#)
 - [Meta.yaml](#)
 - [Bots](#)
 - [Config files](#)
 - [Script files](#)
 - [Campaigns](#)
 - [Effects](#)
 - [Factions](#)
 - [Playables](#)
 - [Unit](#)
 - [Hero](#)
 - [Spell](#)
 - [Upgrade](#)
 - [Upgrade Tree](#)
 - [Maps](#)
 - [Movement Types](#)
 - [Projectiles](#)
 - [Resources](#)
 - [Rulesets](#)
 - [Rules](#)
 - [Win conditions](#)
 - [Scenarios](#)
 - [Skills](#)
 - [Stances](#)
 - [Tags](#)

- Terrains
 - Terrain Blending
- Translations
- Script Objects
 - Triggers
 - Targets
 - CombatTypes
 - SoundTypes
 - PlayerStates
 - ObjectiveStates
 - LogLevels
 - Game
 - Server
 - Script Player
 - Script Cost
 - Script Card
 - Script Card Type
 - Script Unit Type
 - Script Hero Type
 - Script Spell Type
 - Script Upgrade Type
 - Script Unit
 - Script Hero
 - Script Upgrade
 - Script Upgrade Tree
 - Script Upgrade Tree Node
 - Script Spell
 - Script Grid
 - Script Cell
 - Script Extras
 - Script Scenario
 - Script Scenario Objective
 - Script Campaign
 - Script Campaign Scenario Progress
 - Script Data
 - Script Bot
 - Script Terrain
 - Script Movement Type
 - Script Resource
 - Script Tag
 - Script Skill
 - Script Faction
 - Script Ruleset

The *data* folder

At startup, Nusantara loads its content from the **data** folder, which has to be located in its persistent storage location. E.g, on Windows 10, this usually resides at

`C:\Users\User\AppData\LocalLow\Globulus\Nusantara\data`. Modding the game involves making changes to its subfolders and files, all of which is discussed in the rest of this document.

The structure of the **data** folder and its subfolders, as well as names of the files are sometimes important, so make sure not to leave anything out. The details are discussed in the sections below.

Loading might fail due to invalid file structure, compilation errors in scripts or internal validation errors, all of which will be apparent during the loading process. If the game data is corrupt for whatever reason, the game won't be playable, so always be sure to **back up your existing data before modding**.

Meta.yaml

The single standalone file in the **data** folder is **Meta.yaml**. It contains metadata that describe the game data and is used by the game to automatically check for updates.

This file describes a single object whose structure is:

- **version** - the version code of the data file. If modding, bump this up to a high value so that it doesn't get overwritten by game's automatic update system.

Bots

The **Bots** folder contains config and script files that describe the game's bots, i.e AI entities that can play the game and simulate human players.

Config files

There can be as many config (.yaml) files in the folder, each one of them describing a list of bots - `List<{id: string, title: string}>`.

- The **id** of the bot must correspond to an **ai** class defined somewhere in scripts (hence the PascalCase naming).
- The **title** should be a translation key.

Example:

```
- id: EasyAi
  title: easyAiTitle
- id: NormalAi
  title: normalAiTitle
- id: HardAi
  title: hardAiTitle
```

Script files

There can be as many script (.ns) files in the folder. Ideally, they should contain the code that describes **ai** classes corresponding to those listed in config files.

There's no hard and fast rule how an AI should be implemented. The game expects two methods to be implemented:

- **awake** is called when the AI assumes control of a player at the start of their turn. E.g, if your AI has a hard limit of only 10 actions per turn, you can reset the counter in this method.
- **update** is called by the game server repeatedly until it returns **false** - if it returns **true**, it means that AI still has work to do in this turn, so the **update** method will be invoked again. Once this method returns **false**, the game server will end the AI player's turn automatically.

There is the optional third method, **onGameStart(player)**, which is called just once, when the game first starts. Its argument is the player object which the AI will drive. You can use this to perform any additional modifications to the player, such as cheating by giving it extra resources.

The default bots use the "value AI" approach, where a series of possible decisions are weighted, with the one with the highest core executed for the given run. Again, the framework is flexible enough to allow for implementation of different kinds of AIs for different purposes (build tree, passive, etc.).

Campaigns

The **Campaigns** folder contains campaigns, which are collections of scenarios that are linked and share data. Each campaign should be in its own folder. Within it, the campaign should have a config (.yaml) file with the same name as the folder (e.g, the folder **Campaigns/OnTheBanksOfDair** has a config file **OnTheBanksOfDair.yaml**). The same campaign folder should have subfolders for individual campaign scenarios, each of which should have the config and script file for that scenario. All the rules that apply to other **scenarios** apply to campaign scenarios as well.

The campaign config file describes an object with the following schema:

- **id** - the ID of the campaign.
- **title** - should be a translatable key.
- **description** - should be a translatable key.
- **image** - relative path to a 128x128px PNG that is the splash screen of this campaign.
- **difficulties** - list of string IDs of available difficulties for this campaign. The selected difficulty propagates to all of its scenarios.
- **scenarios** - a list of scenarios for this campaign, which should align with the scenario subfolders. Each item has the following schema:
 - **id** - ID of the campaign scenario.
 - **image** - relative path to a 128x128px PNG that is the loading splash screen for this scenario.
 - **loadingText** - the translatable key of the loading text shown when the scenario is selected.

Effects

The **Effects** folder contains only script files (.ns), which should represent generally define different **effect** classes used in the game. Effects are events that happen based on some trigger, such as start of turn, unit attacking or defending, ability being cast, or just by some unit or upgrade having an ability with an effect attached to it. They can also define temporary effects, which are attached to units and players at runtime. The

versatility of effects allows you to do wonders with gameplay, experimenting with weird abilities, auras, attacks, etc.

An `effect` class can implement the following fields/methods:

- **trigger (required)** - describes events that can trigger this effect. Each event can have one or more triggers, all of which are fields on the `Triggers` object.
 - If you have a single trigger, return just its value: `trigger => Triggers.aura`.
 - If you have multiple triggers, return a list: `trigger => [Triggers.attack, Triggers.retaliate]`.
 - Available triggers:
 - `startOfTurn` - target is null.
 - `endOfTurn` - target is null.
 - `onDie` - before the unit actually dies. Target is null.
 - `afterDie` - after the unit dies. Target is the cell on which the unit died.
 - `attack` - after a unit attacks. Target is unit being attacked.
 - `defend` - after a unit survives damage. Target is the attacking unit.
 - `retaliate` - after a unit retaliates. Target is the attacking unit.
 - `kill` - after a kill. Target is the killed unit.
 - `cast` - target varies, can be null, a unit or a cell.
 - `passive` - target is null.
 - `aura` - target is null.
 - `interact` - when a unit is interacted with. Caster is the interactable unit, target is unit that initiated interaction.
- **targets** (optional) - if the effect requires a target (for those with `cast` trigger), defines the possible targets, all of which are fields on the `Targets` object:
 - `none` - the cast happens without targetting in the UI.
 - `unit` (default) - requires a unit target.
 - `cell` - requires a cell target.
- **cost(caster)** (optional) - defines the cost that has to be paid by the caster (or its owner player). If the cost cannot be paid, the effect won't be executed. The expected result is a cost object, which optionally has `g`, `u` and/or `ap` fields, e.g `[g: 3, u: 2, ap: 1]`.
- **range(caster)** (optional) - the range from which the effect can be cast, used in cast and attack/defense/retaliate effects. Should return an `Int` representing the distance in cells between the caster and target, at or under which the effect can be cast.
- **isEnabled(caster)** (optional) - defines a list of conditions that have to be satisfied for the effect to be castable. The list contains lists of two elements, the first of which is a `Bool` expression that checks a necessary precondition for enabling the effect, while the second is an error message that is shown in the UI if that expression is `false`. All pairs in the list must resolve to `true` for the effect to be considered enabled. Example:

```
isEnabled(caster) => [  
  [caster.owner.canAddCards, "Your hand is full!"],  
  [hasMajorUpgrade(caster.owner), "You need a Major Upgrade!"]  
]
```

- `check(caster, target)` (optional) - checks if the effect will run when applied with the given caster and target. It should return a `Bool` value. Example:

```
check(caster, target) => areAllies(caster, target) && !isOrganic(target) &&
target.hp <= target.health - @repairAmount
```

- `run(caster, target)` (**required**) - the heart of the effect, it is the code that is executed when the effect happens. Before its execution, the cost defined in `cost(caster)`, if any, is paid by the caster or its owner. It can return a result that can be used elsewhere in the code.
- `sprite` (optional) - key of the sprite associated with the effect. Must be the `Str id` of an image holder.
- `title` (optional) - the title of the effect that can be shown in the UI.
- `description(caster)` (optional) - description of the effect that can be shown in the UI.
- `flavor(caster)` (optional) - flavor text of the effect that can be shown in the UI.

Example - the **TrainHero** effect:

```
effect TrainHero {
  init(@type)

  trigger => Triggers.cast

  cost(caster) => [g: 10, ap: 1]

  targets => Targets.none

  isEnabled(caster) => [
    [caster.owner.hero == null, "You already have a hero!"],
    [hasVacantInfluencableCell(caster, 1), "You need free space around!"]
  ]

  check(caster, target) => hasVacantInfluencableCell(caster, 1)

  run(caster, target) {
    player = caster.owner
    targetCell = getVacantInfluencableCell(caster, 1)
    hero = targetCell.summon(type, player)
    player.setHero(hero)
    player.addToHand("heroUpgrade", 0)
  }

  description(caster) => "Summon this Hero at the nearest cell!\n" +
    Game.localizedStr(type + "Description")
  flavor(caster) => Game.localizedStr(type + "Flavor")
}
```

Factions

The **Factions** folder contains data for game's factions, describing their metadata, units, spells and upgrades. Each faction should be in its own folder and it must contain a config (.yaml) file of the same name (e.g, **Frenza** folder contains **Frenza.yaml**).

Sound and image files can be organized in any fashion, but it's common to put them in **Sounds** and **Images** folders, respectively.

A faction config file describes an object with the following schema:

- **id** - the ID of the faction.
- **name** - UI-facing display name, should be a translatable key.
- **flavor** - flavor text of the faction, should be a translatable key.
- **image** - should be of a bit higher resolution, such as 256x256px.
- **playable** - determines if this is a faction playable in custom games or only in a scenario/campaign.
- **uniqueResource** - ID of the unique resource for this faction.
- **music** - list of sound files that consistute the musical back-drop for when this faction is played.
- **units** - list of [unit definitions](#).
- **heroes** - list of [hero definitions](#).
- **spells** - list of [spell definitions](#).
- **upgrades** - list of [upgrade definitions](#).

All factions need to have units. Heroes, spells and upgrades are optional, but only for non-playable factions, such as neutral or campaign factions,

Playables

Units, heroes, spells and upgrades are all *playables*, meaning they share a set of properties. Here's a list of those properties:

- **id (required)**
- **title (required)** - should be a translatable key.
- **description (required)** - should be a translatable key.
- **flavor (required)** - should be a translatable key.
- **image (required)** - relative path to a 64x64px PNG.
- **sounds** (optional) - list of sounds that are played based on triggers. All triggers are optional. All sounds are specified as as relative paths to an OGG file. Sounds play only for the currently active player if they can see the action associated with the sound and if the sound is produced by their playable.
 - **summon** - when a: unit/hero is summoned on the map; spell is played from hand; upgrade is slotted.
 - **select** - when a unit/hero is selected.
 - **attack** - when a unit/hero is instructed to attack another.
 - **cast** - when a unit/hero is instructed to cast an ability.
 - **die** - when a unit/hero dies.
- **playCost** (optional) - cost that the owning player has to pay when playing this playable from hand. It is a cost object with two optional **Int** fields: **g** and **u**, standing for *gold* and *unique resource*, respectively.
- **tags** (optional) - list of IDs of [tags](#) applicable to this playable.
- **requiredUpgrades** (optional) - list of IDs for [upgrades](#) that need to be slotted for this to be playable from hand.

Unit

A unit is a [playable](#) that represents an object on the game board. This includes creatures, buildings, interactable map objects, etc. Here is the list of available properties for a unit type:

- **merit (required)** - describes the importance of a unit and is used by AIs to decide on valuable targets or build orders. This is also the amount of XP a Hero gains when killing a unit. There's no good rule what this number should be, as it is based on the relative strengths of other units in the game.
- **h (required)** - represents the maximum health of a unit. It needs to be a positive number.
- **a** (optional) - attack value for this unit. Units with no attack value (or attack value of zero) cannot attack or retaliate.
- **as** (optional) - attack spread for this unit. When non-deterministic combat is enabled in a [ruleset](#), it represents the range around the attack value from which the actual attack value for each attack will be chosen. E.g, if a unit's attack is 10, and attack spread is 2, each attack will have a value from range [8, 12].
- **ac** (optional) - attack count for this unit, defaults to 1. This is the number of times a unit will (try to) attack when engaging in combat.
- **rc** (optional) - retaliation count for this unit, defaults to 1. This is the number of times a unit can retaliate when attacked before its next turn (when retaliations refresh).
- **d** (optional) - defense value for this unit.
- **s** (optional) - speed value for this unit. It represents the number of APs that a unit regenerates at the start of its turn, as well as its maximum number of available APs.
- **r** (optional) - range value for this unit, defaults to 1. This is the range from which the unit can engage in combat, represented as number of cells between it and the opponent. Valid for both attack and retaliation.
- **reg** (optional) - regeneration value for this unit, defaults to 0. This is the number of HP a unit regenerates at the start of each turn. Can be a negative integer to represent automatic loss of health.
- **movementType (required)** - the [movement type](#) for this unit.
- **projectile** (optional) - the ID of a [projectile](#) this unit uses in combat.
- **sight** (optional) - the sight value for this unit, defaults to 1. This is the number of surrounding cells from which the unit removes the fog of war.
- **influence** (optional) - describes the influence this unit exerts, if any. This is an object with two required properties:
 - **amount** - an integer describing the amount of influence exerted. The actual effect depends on **kind**. If this is zero, the unit doesn't exert any influence.
 - **kind** - the type of influence. Available options:
 - **radial** - influences all cells in **amount** radius around itself.
 - **spiral** - influences **amount** cells around itself, starting with the lower-right one.
 - **line** - influences **amount** cells in a straight line to the right of itself.
- **loadCapacity** (optional) - the number of units this unit can *load*, defaults to 0. Loaded units are gone from the board until unloaded. When a unit does, all of its loaded units die as well (although you can write abilities and effect that mitigate this).
- **applyTeamColor** (optional) - if this is **false**, the unit won't have its team color applied to the white pixels in its image.
- **stances (required)** - the list of IDs for [stances](#). Each unit **must have at least one stance defined**.
- **skills** (optional) - the list of IDs of [skills](#) that this unit has.

- **canInteract** (optional) - if this is **true**, the unit can interact with Neutral units that have an **interactionEffect** defined.
- **interactionEffect** (optional) - the ID of an **effect** that is invoked when another unit tries interacting with this one.
- **extras** (optional) - a list of bundled extra data that you can ship with a unit (and, of course, change at runtime). Each *extra* object has the following structure:
 - **id**
 - **title** - should be a translatable key.
 - **image**
 - **value** - this can be any kind of value - a number, string or boolean - depending on the actual context of the extra itself.
- **animationFrames** (optional) - unit actions can be animate, with extra images acting as animation frames. The number of frames isn't limited. The first frame is always the default **image** frame, so all the other images should be based on it. This property has four optional subproperties, each representing a different animation context - **move**, **attack**, **cast** and **die**. Each subproperty's value should be a list of relative paths to 64x64px PNGs.

Hero

A Hero is a **unit** with some extra properties. The game supports controlling a single Hero at the time. Here is the list of available properties for a hero type:

- **v** (optional) - initial Vitality value for this hero, defaults to 0. Has to be a non-negative integer.
- **f** (optional) - initial Fortitude value for this hero, defaults to 0. Has to be a non-negative integer.
- **g** (optional) - initial Guard value for this hero, defaults to 0. Has to be a non-negative integer.
- **p** (optional) - initial Prowess value for this hero, defaults to 0. Has to be a non-negative integer.
- **vitalityProgression** (optional) - how much vitality does the hero gain by leveling up, defaults to 0. Has to be a non-negative integer.
- **fortitudeProgression** (optional) - how much vitality does the hero gain by leveling up, defaults to 0. Has to be a non-negative integer.
- **guardProgression** (optional) - how much vitality does the hero gain by leveling up, defaults to 0. Has to be a non-negative integer.
- **prowessProgression** (optional) - how much vitality does the hero gain by leveling up, defaults to 0. Has to be a non-negative integer.
- **upgradeTree** (**required**) - this hero's **upgrade tree**, indicating skills that become available by leveling up.

Spell

A Spell is a **playable** that holds a single **skill** which is executed when the spell is played (cast) from the hand. Here is the list of available properties for a spell type:

- **skill** (**required**) - the ID of the **skill** invoked when this is played.

Upgrade

An Upgrade is a **playable** that can be slotted in an **upgrade tree**, be it a Faction or a Hero one. Upgrades have skills that are executed when they're played (for the **summon** trigger), passively or at start/end of turn. Here is the list of available properties for an upgrade type:

- **kind (required)** - the kind of the upgrade which determines which tree can it go into and which slots can it occupy. There are three upgrade kinds:
 - **minor** - minor slots in a faction upgrade tree.
 - **major** - major slots in a faction upgrade tree.
 - **hero** - slots in a hero upgrade tree.
- **skills** (optional) - the list of **skill** IDs associated with this upgrade.

Upgrade Tree

Upgrade trees are properties of both factions (via **rulesets**) and **heroes**. An upgrade tree is, well, a tree structure, which consists of nodes. Those nodes can have other nodes as their subnodes, etc.

Nodes are described by their **type** property, which is the ID of the **upgrade** that can be slotted at that node. Each node can only have a single type.

A node can have multiple parents (as shown below with the *buildWonder* upgrade), as long as all of them are at the same depth of the tree. The engine will automatically resolve the tree and merge all of those subnodes into a single node with multiple parents.

Example upgrade tree:

```
nodes:
- type: obsidianWeapons
- type: slaveAuctions
- type: strengthInUnity
  subnodes:
    - type: ironForging
    - type: ransackers
    - type: karrish
      subnodes:
        - type: thunderousHooves
        - type: onslaught
        - type: buildWonder
    - type: reignOfBeastlords
      subnodes:
        - type: eyeToEyeClawToClaw
        - type: tasteOfManflesh
        - type: buildWonder
- type: ritesOfRogash
  subnodes:
    - type: harbingersOfChange
    - type: theDreamFuel
    - type: excrescence
      subnodes:
        - type: shapingCorral
        - type: toTheSkies
        - type: buildWonder
    - type: beastmastery
      subnodes:
        - type: thickShell
```

- type: `daringTamers`
- type: `buildWonder`

Maps

The **Maps** folder contains map files that can be used and played in the game. You can have as many levels of subfolders as you'd like in this folder, but only the maps at the root level (inside the **Maps** folder itself) will be presented in the dropdown for Custom Games. Likewise, if you have scenario or campaign-specific maps that you don't want listed there, put them in appropriate subfolders. By default, **Scenarios** and **Campaigns** subfolders are already present.

A map file is a YAML document describing an object with the following properties, all of which are required:

- `id`
- `title`
- `description`
- `size` - the size of the map. For hexagonal maps, it indicated the radius length of the map. E.g, a hexagonal map of size 7 has 127 cells.
- `kind` - 0 for hexagonal maps, 1 for rectangular. Only hexagonal maps are supported at the moment.
- `maxPlayers` - the maximum number of players that can play on this map in a custom game.
- `defaultTerrain` - the ID of the `terrain` that is set as default for cells without the `terrain` (`t`) property.
- `hexes` - describes the actual layout of the map, as a list of hexes (cells), all of which have the following properties:
 - `c` (**required**) - the coordinates of this cell, in Cube Coordinates. This is a list with three integers, describing the X, Y and Z values.
 - `t` (optional) - the ID of the `terrain` at this cell. If omitted, `defaultTerrain` is used.
 - `s` (optional) - the ID of the terrain sprite used. Set this if you want to use an alternative sprite for your terrain at this cell. Note that this usually interferes with `terrain blending`.
 - `content` (optional) - the ID of a `unit type` that is present on this cell when map loads.
 - `startingLocation` (optional) - if this cell is a starting location for a player, set this value to the index of that player. E.g, a cell on which the 2nd player starts will have `startingLocation: 2`.
 - `comment` (optional) - a hidden comment associated with this cell, which can be used by scripts at runtime.

Making and editing maps manually is difficult, so it's highly recommended to use the built-in **Map Editor**, available in the **Tools & Extras** menu in the game.

Movement Types

The **MovementTypes** folder contains config (.yaml) files that describe movement types. A movement type describes how a unit's pathing works - if it can move at all, and if yes, how much APs does it spend based on the types of terrain it moves across. It also determines if a movement can traverse obstacles or if it can be blocked by other units at the map.

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of movement type objects, which have the following schema:

- `id` (**required**)

- **title (required)** - should be a translatable key.
- **stationary** (optional) - if set to **true**, it indicated that this is a stationary object which doesn't turn towards its target when attacking, retaliating or casting. It has no bearing on the unit's actual ability to move.
- **costs (required)** - a list of movement costs, which describe how many APs does a unit have to spend to move from its current cell to the target one, depending on the terrain of the target cell. The terrain of the origin cell doesn't matter, since the unit is already on it - only the terrain of the destination cell plays a part. The schema of a movement cost is as following:
 - **terrain (required)** - the ID of the **terrain** this cost is applied to. You can use a **wildcard**, *****, to indicate that the cost is applied to all terrains for which there's no other explicitly listed cost.
 - **mp (required)** - how many movement points does the unit have to expend to arrive at the cell with the given territory. If you want to mark that a unit can't move over a certain terrain, put a large number here - 1000 is used in the default data set.
 - **ignoreObstacles** (optional) - if **true**, the unit will ignore any obstacles on this terrain unless it is the destination hex. For example, flying units can fly over a cell even if there's something on it, but they can't arrive at a cell if it isn't empty.

Projectiles

The **Projectiles** folder contains config (.yaml) files that describe projectiles. Projectiles are visual and audible representations of unit attacks: if a unit type has a **projectile** property defined, that projectile will appear whenever the unit attacks or retaliates, for each attack and retaliation.

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of projectile objects, which have the following schema:

- **id (required)** - this is the ID specified in the unit type's **projectile** property.
- **image (required)** - relative path to a 32x32px PNG that represents the projectile's sprite.
- **speed (required)** - speed multiplier that affects how fast does the projectile fly from caster to target.
- **sounds** (optional) - list of sounds that are played based on triggers. All triggers are optional. All sounds are specified as as relative paths to an OGG file. Sounds play only for the currently active player if they can see the action associated with the sound.
 - **cast** - when the projectile leaves the caster.
 - **attack** - when the projectile hits the target.

Resources

The **Resources** folder contains config (.yaml) files that describe resources. The only customizable resource that players can use is the **unique resource** for their faction - all players have access to **gold** by default.

Generation and spending of a resource is a separate matter that is done via [effects](#).

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of resource objects, which have the following schema (all properties required):

- **id**
- **title** - should be a translatable key.
- **description** - should be a translatable key.
- **image** - relative path to a 32x32px PNG.

Rulesets

The **Rulesets** folder contains config (.yaml) files that describe rulesets, as well as script (.ns) files that describe their initial setup and win conditions.

A ruleset is, as its name says, a set of rules that governs the game. This allows for a wide array of customization of how game mechanics work, which can drastically affect individual games. Rulesets provide a fairly simple way of adding completely new game modes or and replayability to existing ones via small twists that can have a big impact.

Each ruleset should be in its own config file, named according to its ID. Win condition scripts can be spread out over as many script files as needed, since all scripts are compiled at once anyhow.

To reduce clutter, rulesets can copy each other with the **inherits** property and then only override certain other properties.

Rules

Here's the full list of ruleset properties:

- **id (required)** - this is usually capitalized (PascalCase).
- **title (required)** - should be a translatable key.
- **description (required)** - should be a translatable key.
- **inherits** (optional) - the ID of the other ruleset that this ruleset inherits/copies.
- **playable** (optional) - if **true**, this ruleset will appear as selectable for custom games. Otherwise, it is meant for scenarios/campaigns.
- **maxTeams** (optional) - the maximum number of teams that players can be distributed into.
- **teamsShareVision** (optional) - if **true**, players can see everything that their teammates can as well.
- **fogOfWar** (optional) - if **true**, the map has fog of war, which prevents seeing cells which are out of your sight.
- **timeoutLength** (optional) - indicates how much time does a player have before the server automatically ends their turn, in *seconds*. If **null** or zero, indicates that players have infinite turns. The actual behavior tightly correlates with the **timerKind** property.
- **timerKind** (optional) - indicates how do timeouts work. Possible values:
 - **perTurn** (default) - **timeoutLength** is reset for each turn and the timeout ends the turn. E.g, if **timeoutLength** is 120, each player has 2 minutes to play each of their turns, after which their turn is automatically ended.
 - **chess** - **timeoutLength** spans the entire game, counting down for each player on their turn. Timeout kicks the user from the game (meaning they lose). E.g, if **timeoutLength** is 600, each player has only 10 minutes in total for everything they wish to play, regardless of how many turns they take. This is akin to how timers work during chess games, hence the name.
- **maxHandSize** (optional) - the maximum number of playables that a player can have in their hand at any given time. Effectively limits the building queue for a player and limits how fast they can grow and expand.
- **deterministicCombat** (optional) - if **true**, attack spread (**as**) isn't applied when units attack/retaliate, meaning that the combat is entirely deterministic and can be computed solely based on attack/defense/attack count/retaliation count values of units involved (not counting combat **effects**, which can be coded to have random outcomes).

- **autoCameraReposition** (optional) - if **true**, the camera will jump to the current player when their turn starts. This is usually set only for scenarios and campaigns.
- **randomizedFirstPlayer** (optional) - if **true**, the first player (the one which starts on the first turn) is chosen randomly. Otherwise, players always play in the order which is set by the game rules, i.e the first player goes first, the second one second, etc.
- **winCondition** (optional) - the class name of the **scenario** the game runs in. Each game, even a custom one, runs in a scenario, which responds to game events in order to set the game up, check if anyone won, etc.
- **factionData** (optional) - determines the setup for each individual faction. It is a map where the key is the faction ID, mapping to the following schema:
 - **startingData (required)** - the initial conditions for each faction when the game starts. Has the following required properties:
 - **resources** - the resource distribution this factions starts with, as an object with **g** and **u** properties.
 - **cards** - the **playables** that start in the player's hand. It is a map, where the key is the playable's ID and the value is the quantity the player starts with.
 - **units** - a list of **unit** IDs that will be spawned at the player's starting location. Add an ID multiple times to indicate multiple units of the same type.
 - **playerEffects** (optional) - list of class names of **effects** that are attached to the player at the start of the game.
 - **upgradeTree (required)** - the **faction upgrade tree** for this faction.
- **bans** (optional) - list of banned skills and stances that can't be used in a game. It has two optional properties:
 - **skills** - a list of IDs of **skills** that can't be used in the game.
 - **stances** - a list of IDs of **stances** that can't be used in the game.
- **otokRuleset** (optional) - the set of rules for Otok minigames when played within a Nusantara game with this ruleset. It has the following required properties:
 - **rounds** - total number of rounds for a game of Otok. This should be an odd integer.
 - **initialResources** - the amount of gold each player received at the start of the game.
 - **resourcesPerDraft** - the additional gold each player receives before each draft (including the first one).
 - **draftDuration** - duration of the draft phase, in seconds.
 - **combatTurnsPerRound** - the number of combat turns per each round, after which the winner is determined.
 - **combatTurnDuration** - duration of each combat turn, in seconds.

Win conditions

Ruleset win conditions are **scenarios** that govern how does the game go by responding to different in-game events, just like regular scenarios. They're usually placed in script files in the **Rulesets** folder.

The basic **DestroyAll** scenario highlights important event methods that every win condition should have:

- **onStart** to set the game up, by adding creeps and sites, adding the objective, etc.
- **onStateChange** to figure out if any player lost all of their units and therefore kick them out of the game.
- **checkVictory** to see if all remaining players are from the same team and declare the winner.

Just as rulesets can inherit other rulesets, it's common for win conditions to extend each other to reuse script code.

Scenarios

The **Scenarios** folder contains scenario config (.yaml) and script (.ns) files. You can have as many scenarios, spread out over as many subfolders. Each scenario should have both a config and a script file in the same folder, which must have the **same name** (different extensions, obviously). Likewise, the **id** in the config file must correspond to the name of the **scenario** class in the script file.

A scenario config file has the following properties:

- **id (required)**
- **title (required)** - should be a translatable key.
- **description (required)** - should be a translatable key.
- **difficulties** (optional) - array of strings, listing available difficulties for this scenario that the user can choose when starting it.
- **map** (optional) - ID of the **map** that this scenario uses. If it is omitted, a map with an ID equal to scenario ID is expected. Not having such a map will result in an error.
- **ruleset (required)** - a **ruleset** object, describing the rules for this scenario. It obviously must have a unique ID and it can also make use of all the other ruleset properties, including **inherits**.
- **players (required)** - list of players for this scenario. Scenario players are hardcoded and can't be altered by the player when starting the scenario. Each player object has the following properties:
 - **name (required)**
 - **color (required)** - available values are **red**, **blue**, **green**, **yellow**, **orange**, **purple**, **cyan** and **black**.
 - **team (required)** - number of the team this player is on.
 - **faction (required)** - ID of the **faction** this player belongs to.
 - **ai** (optional) - if set, it should be the ID of a **bot** that controls this player. Otherwise, the controller is assumed to be the local (human) player (Host).

The corresponding script file contains the **scenario** class. Its purpose is to contains callbacks for various in-game events and modify the game according to the scenario rules. The selected difficulty will be available in the class instance via the **difficulty** field (string).

The only required method is **onCheckVictory**, which should return the winning player if conditions are met. It is invoked after every state update. If no player won yet (or if you'll trigger the victory manually using **Server.win**), return **null**.

Optional methods:

- **onStart** - called when the game starts, before the first turn. It is also executed before **[bot][#bots]** **onStart** method.
- **onFirstRender** - called when the local client renders the scenario for the first time (executed locally, and not on the server).
- **onTurnStart** - calleed when a new turn starts for another player.
- **onStateChange** - called after the state changes and it is broadcasted to the clients.
- **onUnitSelect(unit)** - called when a unit is selected.

- `onChangeStance(unit, previousStance)` - called when a unit assumes a new stance. Old stance is provided as a string ID.
- `onChangeAutoCast(unit, skillId, isAutoCast)` - called when auto-cast is set on a unit's skill. Boolean parameter `isAutoCast` tells if auto-cast is active now or not.
- `onMove(unit, from, to)` - called after a unit moves from one cell to another.
- `onAttack(attacker, defender)` - called after a unit attacks another one.
- `onInteract(unit, target)` - called after a unit interacts with another one.
- `onSkillSelect(unit, skillId)` - called when a skill is selected on a unit.
- `onSkillCast(unit, skillId)` - called after a unit casts as skill.
- `onCardSelect(card)` - called when the player selects a playable card in their hand.
- `onCardPlay(card)` - called after a card is played from hand.

Skills

The **Skills** folder contains config (.yaml) files that describe skills. [Playables](#) can have skills, which are shown in the UI and each wraps a single [effect](#). Skills can also be banned by [rulesets](#).

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of skill objects, which have the following schema:

- **id (required)**
- **title (required)** - should be a translatable key.
- **image (required)** - relative path to a 64x64px PNG.
- **effect (required)** - the class name of an [effect](#) that this skill wraps.
- **params** (optional) - this is the list of parameters that are passed to the effect `init` method when the skill effect is instantiated. The presence and format of this property depends on the effect and its initializer. The purpose is to allow reuse of effect classes (e.g, reuse of **TrainHero** or **GenerateFavorMonument** effects).

Stances

The **Stances** folder contains config (.yaml) and script (.ns) files that describe stances.

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of tag objects, which have the following schema (all properties required):

- **id** - must correspond to a `stance` class name from scripts, hence it's capitalized (PascalCase).
- **title** - should be a translatable key.
- **image** - relative path to a 64x64px PNG.

Units assume stances during gameplay and it alters their stats and skills. That behavior which is described by the corresponding `stance` class in the script files.

Every `stance` instance automatically has the `unit` field, which holds a reference to the unit that this stance is applied to. You can use this field to make the stance tailored to different units.

All the fields and methods discussed in this section are **optional**.

Much like an effect, a stance can't be assumed if its `isEnabled` method fails. This method works exactly the same as in `effects`, holding a list of condition-description pairs, all of which have to pass for the stance to be enabled.

To modify a unit's movement type, add the `movementType` method to the `stance` class and have it return the ID of the `movement type` that the unit will have while in this stance:

```
movementType => "jungleWalk"
```

You can modify the following stats by using the corresponding fields/methods:

- `speed`
- `sight`
- `loadCapacity`
- `attack`
- `defense`
- `attackRange`
- `retaliationRange`
- `regeneration`
- `attackCount`
- `retaliationCount`
- `retaliationAttackModifier` - how much does a unit's `attack` value change when it's retaliating instead of attacking.
- `influence` - alters only the `amount` change of influence, not the `kind`.

Also, `heroes` can also have the following stats modified:

- `vitality`
- `fortitude`
- `guard`
- `prowess`

Each of the above should return a pair (list with two elements). The first one should be the operator, while the second one is the value. Available operators are:

- `"+"` - add the value to the stats base value.
- `"-"` - subtract the value from the stats base value.
- `"*"` - multiply the stats base value by the value.
- `"/"` - divide the stats base value by the value.
- `"%"` - multiply the stats base value by the value, expressed as percentage.
- `"="` - set the stats value to the value.

Examples:

```
stance Camp {  
  attack = ["*", 0.5] # half the attack  
  speed = ["=", 1] # set to 1
```

```
retaliationCount = ["+", 2] # add retaliations
regeneration => ["+", Math.max(1, @unit.health * 0.1)] # increase regeneration
by 10% of max health
}
```

You can also **add and remove skills** via stances using the `addedSkills` and `removedSkills` methods, both of which should return a list of strings, marking IDs of `skills` that the unit should gain or lose, respectively:

```
removedSkills = ["poisonedBlades"]
```

Tags

The **Tags** folder contains config (.yaml) files that describe tags. A `unit` can have multiple tags, and tags can be reused between multiple unit types.

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of tag objects, which have the following schema (all properties required):

- `id`
- `title` - should be a translatable key.

Terrains

The **Terrains** folder contains config (.yaml) files that describe terrains, as well as their sprite files. All terrain sprites are 66x64px PNGs, and they should have a unique name that qualifies as their ID.

You can have as many config files in this folder (or its subfolders) as you'd like, as all are read and merged when data is loaded. Each file needs to be a list of terrain objects, which have the following schema:

- `id (required)`
- `title (required)` - should be a translatable key.
- `influencable (required)` - `true` if players can exert influence over this terrain type. Set to `false` if you don't want to have players, say, summon units on certain terrain types (like watery terrains).
- `layer (required)` - index of the layer in which this terrain is rendered. There are five rendering layers for terrains, 0-4. Most terrains are rendered in layer 2, with some more prominent ones, such as Hills, rendered in layer 4. The layers with higher index are rendered closer to the player/above those in lower layers.
- `sprites (required)` - sprites object for this terrain. Has the following properties:
 - `base (required)` - relative path to the base sprite for this terrain, which is also used for base during `terrain blending`.
 - `useAltAsBase` (optional) - if `true`, the game will randomly swap out the base sprite for some terrains with one from the `alt` list. Only select this for terrains that aren't bases for `terrain blending`.
 - `alt` (optional) - list of relative paths to alternative sprites for this terrain, available in the map editor.

- **mirrorable** (optional) - IDs for terrain sprites that can be mirrored. The game will randomly mirror some terrain horizontally sprites to make the map more lively. If the horizontal orientation of your sprite matters, omit it from this list.
- **border** (optional) - IDs for terrain sprites that have border regions where they are blended with another terrain type. The usual naming convention is *baseTerrainBorderTerrainIndex*, where index is 1-6, with 1 being top right side of a cell hex, 2 being the right one, 3 the lower right one, etc. E.g, **beachGrass5** means that the base beach terrain has a slight grass blend on its leftmost edge. Border terrain sprites are a must have for **terrain blending** to work.
- **spriteRules** (optional) - defines rules that govern which **border** sprites to use, depending on the terrain sprites of adjacent cells. It is a list of objects with the following schema (all properties required):
 - **sprite** - the sprite to consider.
 - **rules** - list of rules: These have the following schema (all properties required):
 - **neighbor** - list of terrain IDs that this rule applies to when they're adjacent to the base cell.
 - **sprites** - list of border sprite IDs for this rule. For clarity, consider these rule. They say that, when the cell has terrain sprite of type *beach*, in the case adjunct cells are of type *grass*, *forest*, *jungle* or *swamp*, we should use *beachGrass1-6* as borders during **terrain blending**. Otherwise, if the adjunct cell has terrain *river*, use *beachRiver1-6*.

```

spriteRules:
- sprite: beach
  rules:
    - neighbor: [grass, forest, jungle, swamp]
      sprites: [beachGrass1, beachGrass2, beachGrass3, beachGrass4,
beachGrass5, beachGrass6]
    - neighbor: [river]
      sprites: [beachRiver1, beachRiver2, beachRiver3, beachRiver4,
beachRiver5, beachRiver6]

```

- **debrisFrequency** (optional) - if set, it tells how often (as X : 1) does debris appear on terrain sprites. E.g, if this is 0.4, 40% of cells with that terrain will have debris on them, while 60% will not.
- **debris** (optional) - list of relative paths towards debris sprites. Debris sprites contain small images that are overlaid on terrain sprites to provide additional content on them. The debris is placed randomly and is there just for the sake of making the map more appealing to look at.

Terrain Blending

Nusantara engine can improve the visual quality of the rendered map by using terrain blending. This process smooths the borders between cells, making it seem as if cells with different terrains are melting together to form a continuous transition between the two sprites.

To make terrain blending work, you need to create a variants of the base terrain, called border terrains. These have borders on one of the cell edges (1-6) altered to make it seem as if it's a part of another terrain. Then, at runtime, the game will impose the border terrains onto the base terrain depending on adjacent cells and sprite rules.

E.g, if you have a cell whose sprite is *beach*, and it is surrounded by two *grass* and one *forest* terrains to the right, and three *river* terrains on the left. Terrain blending will read the sprite rules and create a composite terrain sprite, in which 7 sprites will be overlaid on top of each other sequentially - *beach*, *beachGrass1*, *beachGrass2*, *beachGrass3*, *beachRiver4*, *beachRiver5* and *beachRiver6*. In-game, this cell's terrain sprite will be that composite sprite.

Terrain blending happens at game startup and can take a few minutes to complete, but doesn't prevent the player from starting the game. The blended terrain sprites can be stored locally. If you make any changes to your terrain sprites or their rules, make sure to remove the **temp/_blended** folder.

There is a setting in the game called **Terrain Blending Level**. It basically says how many blended terrains to pre-make. E.g, if this value is 3, the game will blend the base terrain with up to and including 3 borders. If you set this level to 6 (max), all terrains will have variants for any combination of adjacent terrains.

Translations

The **Translations** folder contains translations for game data. Each supported locale should be in its own folder, named after its code (e.g, **en-US**). Inside each locale folder, you can have as many YAML files, each of which should contain translatable strings in the key-value format. E.g:

```
# Rulesets
deathmatchTitle: Deathmatch
deathmatchDescription: Destroy all enemies.
regicideTitle: Regicide
regicideDescription: Destroy the enemy banner to defeat them!
```

Script Objects

Nusantara Scripts interact with the game through script objects. They allow you to query the current game state, as well as to alter it to your wishes. Below is the comprehensive reference for all script objects and their schemas. In general, use fields are read-only and are used to inspect the state, while methods mutate it - e.g, `player.state` reads the state of that player, but in order to change it, use `player.setState(newState)`.

Triggers

Triggers object holds effect triggers as its fields.

- `startOfTurn`
- `endOfTurn`
- `summon`
- `onDie`
- `afterDie`
- `attack`
- `defend`
- `retaliate`
- `kill`
- `cast`
- `passive`

- `aura`
- `interact`

Targets

Targets object holds effect targets as its fields.

- `none`
- `unit`
- `cell`

CombatTypes

CombatTypes object holds combat types as its fields.

- `attack`
- `retaliation`

SoundTypes

SoundTypes object holds sound types as its fields. These are used when playing sounds from the [Game](#) object.

- `summon`
- `select`
- `attack`
- `cast`
- `die`

PlayerStates

PlayerStates object holds player states as its fields.

- `active`
- `inactive`

ObjectiveStates

ObjectiveStates object holds objective states as its fields.

- `hidden`
- `active`
- `completed`
- `failed`

LogLevels

LogLevels object holds log levels as its fields. These can be used to alter the color of the text when it is logged from the [Game](#) object.

- `info`
- `warn`

- error
- positive

Game

Game object is the centerpiece of scripting, containing all the methods to query and alter the state, either directly or via the objects it returns.

- **turn** - returns the current turn as **Int**.
- **currentPlayerIndex** - returns the index of the current player as **Int**.
- **currentPlayer** - returns the current player as **script player**.
- **players** - returns a **List** of all players as **script players**.
- **nature** - returns the default Nature/Nusantara passive player as **script player**.
- **activePlayers** - returns a **List** of all players whose state is **active** as **script players**.
- **currentScenario** - returns the currently active scenario as **script scenario**.
- **currentCampaign** - returns the currently active campaign as **script campaign**.
- **data** - returns locally available data as **script data**.
- **grid** - returns the **script grid** representing the current state on the board.
- **unitsInRange(origin, range)** - takes **origin** as either a **script cell** or coordinates list and returns a **List** of **script units** in radius **range** around the **origin**.
- **castTargets(originCoordinates, caster, skillId)** - takes **originCoordinates** as coordinates list, **caster** as **script unit** and **skillId** as **Str** and returns a **List** or **script cells** on which that unit can cast that skill if it is on those coordinates.
- **spellTargets(card)** - takes **card** as **script card** and assumes it is a **spell**. Finds possible targets for playing that spell, and returns:
 - **null** if the spell can't be cast.
 - **true** if the spell doesn't require targetting.
 - **List** of **script cells** on which the spell can be cast.
- **combat(attacker, defender)** - takes **attacker** and **defender** as **script units** and makes the former attack the latter, playing out a full combat between them.
- **animateAttack(attacker, defender)** - takes **attacker** and **defender** as **script units** and plays attack animations between the two.
- **animateVisualFx(fx)** - takes a **List** of **Objects** that describe the visual effects to animate sequentially. Those objects can have the following properties:
 - **coordinates (required)** - a coordinates list at which the effect takes place.
 - **type (required)** - the type of the effect, as **Str**. Possible values are:
 - aura
 - blood
 - bonfire
 - boost
 - earn
 - electric
 - explosion
 - hit
 - poison
 - ripple
 - circle

- **color** (optional) - color of the effect, as a **Str** of format **#RRGGBB**. This is only applicable to some effect types.
- **playSound(sourceCoordinates, sourceTypeId, soundType, waitForEnd)** - plays a sound. It must be one of the pre-loaded sounds from the game's data. The parameters are **sourceCoordinates**, a coordinates list showing where the sound originated, **sourceTypeId** which tells which unit produced the sound, **soundType** which is a **sound type** value, while **waitForEnd** is a **Bool** indicating if the game should wait for the sound to be played entirely before resuming.
- **localizedStr(key)** - returns the translated (localized) text based on the provided **Str** key.
- **describeAsStance(caster, object)** - provides a custom description of a provided object based on its properties, which must match those of a **stance** instance.
- **showMessage(message)** - displays the message to the currently active player.
- **showMessageToAll(message)** - displays the message to all players.
- **showDialog(portrait, title, content)** - displays the campaign dialog using the following **playable's** ID (**portrait**) for the portrait image.
- **logEvent(message, level, positionCoordinates)** - logs the given **message** with the given **log level** to the logs of all players who can see the cell whose coordinates are **positionCoordinates** (if **null**, all players will get this message).
- **wait(seconds)** - pauses the game for the given **seconds** (as **Float**).
- **zoomCamera(amount)** - zoom the game to the given **amount** (as **Float**). The limits of this parameter depend on screen and map size.
- **positionCamera(cellCoordinates)** - positions the camera so that the cell with the given coordinates list is in the center of the screen.
- **refreshState** - forces the server to compute the current state and send it to all clients. This is sometimes necessary to refresh the game after an objective is changed in a scenario.
- **lose(player)** - makes the given **script player** lose, i.e be deactivated and kicked from the game.

Server

Server object contains methods that also affect the game, but they represent state changes that are normally computed on the server. The reason these are separate from the **Game** methods is that **Game** methods are enqueued on the server, while **Server** methods are computed synchronously.

- **move(fromCoordinates, toCoordinates, cost)** - moves the unit located at **fromCoordinates** to the empty cell at **toCoordinates** and makes it pay **cost** APs for that.
- **changeStance(unit, newStance)** - makes the **script unit** change stance to **newStance** (as **Str** ID of the new stance).
- **cast(caster, targetCoordinates, skillId)** - makes the **script unit** cast the skill with the given ID at the provided coordinates.
- **play(card, targetCoordinates, targetId)** - plays the provided **script card** from its owner's hand at the target coordinates/ID.
- **attack(fromCoordinates, toCoordinates, viaCoordinates, cost)** - makes the unit at **fromCoordinates** attack the unit at **targetCoordinate**, optionally first moving to **viaCoordinates**, and paying **cost** APs in the process.
- **interact(fromCoordinates, toCoordinates)** - makes the unit at **fromCoordinates** interact with unit at **toCoordinates**.
- **concede(player)** - makes the provided **script player** concede and become deactivated. Potentially triggers victory checks.

- `win(player)` - makes the provided `script player` win.

Script Player

In-game players are represented as `Objects` in scripts. Each has the following fields/methods:

- `id` - `Str`
- `name` - `Str`
- `color` - `Str`
- `state` - returns `player state Str`.
- `setState(newState)` - sets this player's state to the provides `player state`.
- `controller` - `Str` ID of the controller entity for this player (can be `null`).
- `team` - `Int`
- `faction` - `Str` ID of the `faction`.
- `resources` - `script resources` that this player currently owns. Can be `null`.
- `canAddCards` - `true` if player can add cards to their hand based on the current `ruleset`.
- `hand` - `List` of `script cards` that are in this player's hand currently.
- `upgradeTree` - `script upgrade tree` for this player.
- `hero` - `script hero` that this player controls. Is set even if the Hero is currently dead, otherwise, if this player never had a Hero is `null`.
- `setHero(hero)` - assigns the provided `script hero` to this player.
- `currentStats` - returns the stats for this player for the current game. It has the following fields:
 - `unit` - unit stats. It has the following fields (all `Ints`):
 - `unitsPlayed`
 - `unitsDestroyed`
 - `unitsLost`
 - `buildingsPlayed`
 - `buildingsDestroyed`
 - `buildingsLost`
 - `largestArmy`
 - `hero` - hero stats. It has the following fields (all `Ints`):
 - `casts`
 - `deaths`
 - `kills`
 - `maxLevel`
 - `xpGained`
 - `economy` - economy stats. It has the following fields (all `Ints`):
 - `goldGained`
 - `goldSpent`
 - `uniqueGained`
 - `uniqueSpent`
 - `influencedCells`
 - `influencedMap`
 - `majorUpgrades`
 - `minorUpgrades`
 - `techtreeComplete`
- `addResources(resources)` - adds the provided `script cost` to this player's resource pool.

- `canPay(cost)` - returns `true` if the provided `script cost cost` is not `null` and the player has more resources than it.
- `pay(cost)` - makes the player pay the provided `script cost cost`.
- `addToHand(cardTypeId, downtime)` - adds a `playable` of type `Str cardTypeId` to the player's hand, making it unavailable for `Int downtime` turns.
- `removeFromHand(id)` - removes the card with the provided ID as `Str` (card ID, not card type ID) from the player's hand.
- `isAllyOf(player)` - returns `true` if this player is on the same team as the provides `script player`.
- `ownedUnits` - returns a `List` of `script units` on the board owned by this player. Doesn't include the Hero if it is not on the board.
- `owns(unitId)` - returns `true` if this player owns a unit with the provided `Str ID`.
- `hasUpgrade(upgradeId)` - returns `true` if this player has an `upgrade type` with the provided `Str ID` slotted in their upgrade tree.
- `addEffect(effectId)` - adds the effect with the provided ID (class name) to the list of this player's permanent effects.
- `hasEffect(effectId)` - returns `true` if this player has an effect with the provided ID in their permanents effects.
- `removeEffect(effectId)` - removes the effect with the provided ID (class name) from the list of this player's permanent effects.

On top of this, players contain `extras`, so they have all the additional `script extras` methods.

Script Cost

In scripts, costs are represented as `Objects` with the following optional fields:

- `g` - `Int` representing amount of Gold.
- `u` - `Int` representing amount of Unique Resource.
- `ap` - `Int` representing amount of Action Points.

Script Card

In scripts, cards in player's hand are represented as `Objects` with the following fields/methods:

- `id` - `Str`
- `type` - `script card type` of this card.
- `downtime` - `Int` representing the number of turns remaining before this card can be played.

Script Card Type

`Card types`, as defined in the game's data, are represented in scripts as `Objects` with the following fields/methods:

- `id` - `Str`
- `title` - `Str`
- `playCost` - `script cost` or `null`.
- `tags` - `List` of `Str` representing `tag` IDs, or `null`.
- `requiresUpgrades` - `List` of `Str` representing `upgrade` IDs, or `null`.

Script Unit Type

On top of all [script card type](#) fields/methods, [unit](#) types contain the following additional ones:

- `type` - always returns `"unit"`.
- `merit` - `Int`
- `h` - `Int`
- `a` - `Int`
- `ac` - `Int`
- `rc` - `Int`
- `d` - `Int`
- `s` - `Int`
- `reg` - `Int`
- `m` - `Str` of [movement type](#) ID.
- `projectile` - `Str` of [projectile](#) ID.
- `sight` - `Int`
- `loadCapacity` - `Int`
- `influence` - `Object` with the following fields, or `null` if not set:
 - `kind` - `Str`
 - `amount` - `Int`
- `stances` - `List` of `Str` of registered stances.
- `skills` - `List` of `Str` of registered skills, or `null`.
- `canInteract` - `Bool`
- `interactionEffect` - `Str` ID of interaction effect, or `null`.

Script Hero Type

On top of all [script unit type](#) fields/methods, [hero](#) types contain the following additional ones:

- `vitality` - `Int`
- `fortitude` - `Int`
- `guard` - `Int`
- `prowess` - `Int`
- `vitalityProgression` - `Int`
- `fortitudeProgression` - `Int`
- `guardProgression` - `Int`
- `prowessProgression` - `Int`

Script Spell Type

On top of all [script card type](#) fields/methods, [spell](#) types contain the following additional ones:

- `type` - always returns `"spell"`.
- `skills` - `List` of `Str` of IDs of skills associated with this spell type.

Script Upgrade Type

On top of all [script card type](#) fields/methods, [upgrade](#) types contain the following additional ones:

- `type` - always returns "upgrade".
- `kind` - `Str` (minor, major, hero).
- `skills` - `List` of `Str` of IDs of skills associated with this upgrade type.

Script Unit

In-game units are represented as `Objects` in scripts. Note that most stats on units are a result of applying multiple factors to them, like the type base values, stance, active effects, player effects, etc. Each unit object has the following fields/methods:

- `id` - `Str`
- `type` - script unit type.
- `hp` - current HP of the unit as `Int`.
- `ap` - current AP of the unit as `Int`.
- `retaliationPoints` - remaining retaliation points of the unit as `Int`.
- `health` - maximum health of this unit, as `Int`.
- `speed` - speed of this unit, as `Int`.
- `sight` - sight of this unit, as `Int`.
- `loadCapacity` - load capacity of this unit, as `Int`.
- `regeneration` - regeneration of this unit, as `Int`.
- `attackCount` - attack count of this unit, as `Int`.
- `retaliationCount` - retaliation count of this unit, as `Int`.
- `influenceAmount` - influence amount of this unit, as `Int`.
- `movementType` - movement type ID of this unit, as `Str`.
- `attack(type, target)` - attack value when attacking script unit `target` using `combat type`, as `Int`.
- `defense(attacker)` - defense value when defending from script unit `attacker`, as `Int`.
- `range(type, target)` - range value when combatting script unit `target` using `combat type`, as `Int`.
- `activeStance` - ID of the `stance` assumed by this unit.
- `availableStances` - list of IDs of stances that this unit can assume (those whose `isEnabled` is `true`).
- `skills` - list of effect instances based on this unit's skills.
- `addSkill(id)` - adds the `skill` with the provided ID to this unit.
- `removeSkill(id)` - removes the `skill` with the provided ID from this unit.
- `owner` - script player that owns this unit.
- `setOwner(player)` - sets the owner of this unit to the provided script player.
- `destroy(attacker)` - destroys the current unit, counting the provided script unit as the one that destroyed it (for player stats and XP purposes).
- `takeDamage(damage, attacker)` - takes `Int` `damage`, potentially destroying the unit, and counting the provided script unit as the one that dealt damage/destroyed it (for player stats and XP purposes). Returns `true` if the unit was destroyed.
- `setHealth(hp)` - sets the current HP to the provided `Int`.
- `restoreHealth(amount)` - restores `Int` `amount` HP to this unit, capped at its health.
- `setActions(ap)` - sets the current AP to the provided `Int`.
- `spendActions(amount)` - subtracts the provided `Int` from current APs. Can't go below zero APs.
- `restoreActions(amount)` - restores `Int` `amount` AP to this unit, capped at its speed.
- `tempEffects` - list of effect instances that affect this unit temporarily.
- `addTempEffect(effect, caster, duration)` -
- `hasTempEffect(effectId)` -

- `removeTempEffect(effectId)` -
- `isBuilding` - `true` if this unit has the `building` tag.
- `isHero` - `true` if this unit is a Hero.
- `setType(newTypeId)` - changes the type of this unit. All active and autocast skills, as well as temp effects, are cleared, and the default stance for the new type is assumed. Doesn't work on Heroes.
- `loadedUnits` - returns a `List` of `script units` currently loaded into this unit.
- `canLoad` - returns `true` if this unit has fewer units loaded than its load capacity is.
- `load(unit)` - loads the provided `script unit` from its cell to this unit. Returns `true` if the load was successful.
- `unload(targetCoordinates)` - unloads the last loaded unit to the cell described by the provided coordinates list. Returns `true` if the unload was successful.
- `coordinates` - returns coordinates list of the cell this unit is on. It will be `null` if the unit is not currently on the board, e.g, if it's dead or loaded into another unit.
- `moveTo(coordinates)` - teleports the unit to the cell with the provided coordinates list. It doesn't make any checks, so it's possible to overwrite an existing unit on the target coordinates.
- `movementCost(toCoordinates)` - computes the number of APs as `Int` needed for this unit to move to the cell marked with the provided coordinates list. If the unit can't move there, it'll return `1000`.
- `movementRange` - returns a `List` of `script cells` that this unit can currently move to.
- `attackRange` - returns a `List` of `Objects` that describe potential attack targets for this unit. Each object has the following fields:
 - `targetCell` - `script cell` containing the potential attack target.
 - `viaCell` - `script cell` that the unit would have to move to in order to attack.
 - `movementCost` - `Int` describing the number of APs the unit would have to expend to move next to the potential target.

On top of this, units contain `extras`, so they have all the additional `script extras` methods.

Script Hero

On top of all `script unit` fields/methods, hero objects contain the following additional ones:

- `xp` - `Int`
- `upgradeTree` - this hero's `script upgrade tree`.
- `level` - `Int`
- `vitality` - `Int`
- `fortitude` - `Int`
- `guard` - `Int`
- `prowess` - `Int`
- `hasUpgrade(upgradeId)` - returns `true` if this hero has an `upgrade type` with the provided `Str` ID slotted in their upgrade tree.
- `isAlive` - returns `true` if this hero is present on the board or loaded into another unit.

Script Upgrade

In-game upgrades are represented as `Objects` in scripts, each having the following fields/methods:

- `id` - `Str`
- `type` - `script upgrade type`.

- **owner** - **script player** that has this upgrade slotted in their upgrade tree (or their hero's).

Script Upgrade Tree

In-game upgrade trees are represented as **Objects** in scripts, each having the following fields/methods:

- **nodes** - returns a **List** of root **script upgrade tree nodes**.
- **allNodes** - returns a **List** of all **script upgrade tree nodes** in this tree, flattened out.
- **allUpgrades** - returns a **List** of root **script upgrades** slotted in this tree.
- **emptyNodesForKind(kind)** - returns a **List** of all empty **script upgrade tree nodes** in which slots of **Str kind** (minor, major, hero) can be slotted.
- **addNode(node)** - adds a new **script upgrade tree node** at the root level of this tree.

Script Upgrade Tree Node

In-game nodes of upgrade trees are represented as **Objects** in scripts, each having the following fields/methods:

- **id** - **Str**
- **upgrade** - **script upgrade** slotted into this node, or **null** if the node is empty.
- **slot(player, trigger)** - slots a new upgrade according to this node's type for the given player, and optionally **trigger** its summon effects. The actual tree used (faction or hero) is determined by node's type.
- **unslot** - removes the upgrade from this node.
- **subnodes** - returns a **List** of all **script upgrade tree nodes** that are direct children of this node.
- **addSubnode(node)** - adds a new **script upgrade tree node** as a direct child of this node.

Script Spell

In-game spells are represented as **Objects** in scripts, each having the following fields/methods:

- **id** - **Str**
- **type** - **script spell type**.
- **owner** - **script player** that holds this spell in their hand.

Script Grid

The **Grid** object represents the game's board, i.e the grid of cells that the game is happening on. It arguably contains most of the game's data. It is an **Object** with the following fields:

- **size** - **Int map** size.
- **cells** - **List** of all **script cells**.
- **nonEmptyCells** - **List** of all **script cells** that have units on them.
- **visibleCells(player)** - **List** of **script cells** that the provided **script player** can see (i.e, those not hidden by fog of war).
- **hiddenCells(player)** - **List** of **script cells** that the provided **script player** can't see (i.e, those hidden by fog of war).
- **cellsAround(coordinates, range, player)** - **List** of **script cells** around the cells with the provided coordinates list in radius **range** that the provided **script player** can access.

- `ringAround(coordinates, range)` - List of `script cells` around the cells with the provided coordinates list in radius `range`.
- `allUnits` - List of all `script units` present on the board.
- `startingLocations` - List of coordinates lists of all starting locations, sorted by player index.
- `locate(unit)` - returns `script cell` which contains the provided unit, or `null` if none do.
- `elementAt(coordinates)` - returns `script cell` with the provided coordinates list, or `null` if such a cell doesn't exist.

Script Cell

A single cell on the game's `grid` is represented as an `Object` with the following fields:

- `coordinates` - coordinates list of this cell.
- `terrain` - `Str` ID of the `terrain` on this cell.
- `influencer` - `Str` ID of the player influencing this cell, or `null` if none do.
- `isInfluencable` - returns `true` if players can influence this cell based on its terrain type.
- `hasFogOfWar(player)` - returns `true` if the provided `script player` sees fog of war on this cell.
- `content` - `script unit` located on this cell, or `null` if the cell is empty.
- `comment` - `Str` indicating a `map hex` comment.
- `summon(typeId, player)` - summons a unit of the provided `Str typeId`, owned by the provided `script player`.
- `setContent(unit)` - sets the content of this cell to the provided `script unit`, or empties it if `null` is provided.
- `setTerrain(terrainId)` - sets the terrain of this cell to be that with the provided `Str` ID.
- `distance(toCoordinates)` - returns the `Int` distance in number of cells between this cell and the one designated by the provided coordinates list.
- `equals(otherCell)` - returns `true` if both cells have the same coordinates.

On top of this, cells contain `extras`, so they have all the additional `script extras` methods.

Script Extras

Some objects can have the extras bundle, which can contain any data. All those objects have the following fields/methods:

- `extras` - all the current extras on the object, as a `List` of `Objects` with the following fields:
 - `id` - `Str`
 - `title` - `Str`
 - `image` - `Str`
 - `value` - can be of any type, depends on context.
- `getExtraValue(extraId)` - returns the value for the provided `Str` extra ID, or `null` if it doesn't exist.
- `addExtra(id, title, image, value)`
- `updateExtra(id, newValue)`
- `addOrUpdateExtra(id, title, image, newValue)`
- `removeExtra(id)`

Script Scenario

In-game scenario for this given game is represented as an **Object** in scripts, having the following fields/methods:

- **scriptId** - **Str** ID (class name) of the script this scenario runs on.
- **difficulty** - **Str** difficulty this game was started with.
- **wrapper** - references the actual instance of the script scenario class, allowing you to invoke methods on it.
- **objectives** - **List** of **script scenario objectives**.
- **addObjective(objective)** - adds a new objective, provided as an **Object** of type **script scenario objective**.
- **updateObjective(objectiveId, payload)** - updates the objective whose ID is the provided **Str** with the new fields provided as an **Object** of type **script scenario objective**.
- **objectiveState(objectiveId)** - returns **objective state** of the objective whose ID is the provided **Str**.
- **addRevealedCell(coordinates)** - makes the cell as the provided coordinates permanently revealed for all players.
- **removeRevealedCell(coordinates)** - removes the cell with the provided coordinates from the list of permanently revealed cells.

On top of this, scenarios contain **extras**, so they have all the additional **script extras** methods.

Script Scenario Objective

A single objective on the game's **scenario** is represented as an **Object** with the following fields:

- **id** - **Str**
- **state** - **objective state**

When creating objectives, you can also provide the following fields:

- **title** - should be a translatable string.
- **description** - should be a translatable string.
- **image** - ID of a **playable** whose image this objective will use.

Script Campaign

In-game campaign for this given game is represented as an **Object** in scripts, and it's a combination of the campaign data and its current progress. The campaign progress is stored on the disk, persisting between game restarts and allowing for tracking of how far did the player get in a campaign, and what achievements did they acquire while doing so. The object has the following fields/methods:

- **id** - **Str** ID of the currently played campaign.
- **difficulty** - **Str** difficulty the campaign was started with.
- **unlockedScenarios** - **List** of **Str** IDs of currently unlocked scenarios for this campaign (i.e, scenarios you can play).
- **unlock(scenarioId)** - unlocks the campaign scenario with the provided ID, making it playable/selectable from the campaign menu.
- **progressFor(scenarioId)** - retrieves the **script campaign scenario progress** for the provided scenario ID.

- **reload** - reloads the campaign progress from the disk.
- **save** - saves the campaign progress to the disk.

Script Campaign Scenario Progress

Each campaign progress file contains progress for the given scenario. It can hold any data, allowing future scenarios to built upon previous one. E.g, if you hero reached level 3 and had a certain set of upgrades chosen, you can store that and then use that as the starting point in a future scenario. Each scenario progress is represented as an **Object** in scripts, having the following fields/methods:

- **id** - **Str** ID of the campaign scenario this progress is for.

On top of this, campaign scenario progresses contain **extras**, so they have all the additional **script extras** methods.

Script Data

You can access the game data using this **Object** which has the following fields/methods:

- **customGameBots** - **List** of **script bots**.
- **terrains** - **List** of **script terrains**.
- **movementTypes** - **List** of **script movement types**.
- **resources** - **List** of **script resources**.
- **tags** - **List** of **script tags**.
- **skills** - **List** of **script skills**.
- **factions** - **List** of **script factions**.
- **rulesets** - **List** of **script rulesets**.
- **find(id)** - returns the **card type** with the provided **Str** ID, or **null** if it doesn't exist.

Script Bot

Bots, as defined in the game's data, are represented in scripts as **Objects** with the following fields/methods:

- **id** - **Str**

Script Terrain

Terrains, as defined in the game's data, are represented in scripts as **Objects** with the following fields/methods:

- **id** - **Str**
- **title** - **Str**
- **influencable** - **Bool**

Script Movement Type

Movement types, as defined in the game's data, are represented in scripts as **Objects** with the following fields/methods:

- **id** - **Str**

- `title` - `Str`
- `stationary` - `Bool`
- `costs` - `List` of `Objects` with the following fields:
 - `terrain` - `Str`
 - `mp` - `Int`
 - `ignoreObstacles` - `Bool`

Script Resource

[Resources](#), as defined in the game's data, are represented in scripts as `Objects` with the following fields/methods:

- `id` - `Str`
- `title` - `Str`

Script Tag

[Tags](#), as defined in the game's data, are represented in scripts as `Objects` with the following fields/methods:

- `id` - `Str`
- `title` - `Str`

Script Skill

[Skills](#), as defined in the game's data, are represented in scripts as `Objects` with the following fields/methods:

- `id` - `Str`
- `title` - `Str`
- `effect` - `Str` ID (class name) of the skill effect.

Script Faction

[Factions](#), as defined in the game's data, are represented in scripts as `Objects` with the following fields/methods:

- `id` - `Str`
- `name` - `Str`
- `uniqueResource` - `Str`
- `units` - `List` of [script unit types](#).
- `heroes` - `List` of [script hero types](#).
- `spells` - `List` of [script spell types](#).
- `upgrades` - `List` of [script upgrade types](#).

Script Ruleset

[Rulesets](#), as defined in the game's data, are represented in scripts as `Objects` with the following fields/methods:

- `id` - `Str`
- `name` - `Str`

- `fogOfWar` - Bool
- `deterministicCombat` - Bool
- `isBanned(id)` - returns `true` if the playable with the given ID is banned.