Project CARS 2 - Dedicated Server Scripting Guide

This document describes the scripting capabilities of the dedicated server. For most server administrators, the server configuration and supplied Lua addons described in Dedicated Server User Guide should be sufficient. If further customization or stat tracking are needed, the server offers two APIs - HTTP-based access and Lua-based scripting:

- HTTP API. You can query the server status and send requests to the server using a simple HTTP-based protocol. This API is mostly useful when you want to monitor the server or show the current server's status on your website. The HTTP API is disabled in the default sample config.
- Lua API. The server embeds Lua 5.3 and provides a scripting API which can be used to both monitor and control the server. This API is more useful for controlling the server in a predefined manner, such as automatically rotating tracks. The server ships with several sample addons and the Lua API is enabled by in the default sample config.

Functionality available to the APIs

Before the description of the two APIs, few notes about how the server itself operates. The Project CARS 2 dedicated server is more of a “coordinator” kind of server, the core of the simulation is still running on the individual peers. The gameplay network traffic is sent through the server, which eliminates many issues with peer to peer networking having to punch through user routers and NATs. But the server does not participate in the actual physics simulation. That means that even with the open scriptable architecture, you can’t control the actual simulation and gameplay from the server in any other means that those exposed by the game to the server.

What the server does and what can be extended with scripting is:

- Session management - players joining the server and setting up a session on it. In the API you can monitor players joining/leaving and kick players, with an optional temporary ban. In the future it will be also possible to access and modify the persistent blacklist and whitelist from the scripts.

- Session, player and participant setup and status. The server receives information about the setup and the state of the active race, and individual players (called session members in the API) and participants (vehicles). The setup and status are available to the API via queries and notifications.
- Modify the game's setup. In a peer to peer session with no server, the "host" player can control the game's setup - first when creating a new multiplayer session, then in the lobby. The server can optionally take over this responsibility, and the setup control is then also exposed to the APIs.

Server control modes

The server can be configured to run in different "control modes".

The main setting is available in the server configuration file (server.cfg) as "controlGameSetup" and it defaults to false in the sample config. This means that multiplayer sessions hosted on the server work in a very similar way to peer to peer sessions, the first player that joins the server becomes the "host" and can control the setup in the lobby UI. In this mode the API calls that control the setup of the active multiplayer session will silently fail. The scripts will still be able to configure the default setup that's used when the first player joins an empty server, but the player will always be able to change the settings to anything else.

If the "controlGameSetup" configuration option is set to true, the server will take over the game's setup control. While the game will still have a player host, the player will not be able to modify the game's setup from the lobby anymore. The setup can then be controlled on the server:

- In the server's configuration file, "sessionAttributes" specifies the initial setup.
- HTTP API and Lua API hooks can be used to change the "next session" setup and the "current session" setup. The "next session" setup can be changed at any time, and will apply when the first player joins an empty server, or when the current race finishes and players return back to the lobby. The "current session" setup can be changed only while the players are in that lobby, and will change that lobby's setup directly.
- The sample Lua addon "sms_rotate" uses the Lua API to automatically rotate setups applied by the server. Even if fixed setup is used, it's recommended to use the addon over the "sessionAttributes" configuration option, because the addon allows for more user-friendly attribute values. Where the config accepts only integral id values, the Lua addon can also parse string names and converts them to ids automatically - so for example you can set the "TrackId" attribute by the track's name, instead of the numeric id.
- The "sms_rotate" Lua addon uses a helper Lua library "lib_rotate", which simplifies enforcing track/vehicle/class, and has helper functions for combining multiple setups into on. This library can be used by any other community-developed addon.

Once a server is running in “controls game’s setup” mode, it can enforce even more than that. The additional control options are controlled by these session attributes:

- “ServerControlsTrack”: If set to 0, the host player will be able to change the track, and other players can vote to request a change from the host. If set to 1, the players won’t be able to change the track and it will be controlled only by the server’s “TrackId” attribute.
- “ServerControlsVehicleClass”: Even though the host player can’t change the class directly in the lobby setup when the setup is controlled by the server, it’s still possible to navigate to the garage and select a car from another class. When this attribute is set to 1, this will no longer be possible - while the garage UI will still be enabled, any attempts to change to car from different class will fail. When setting this attribute to 1, it should always be combined with the session flag FORCE_SAME_VEHICLE_CLASS and the correct class id set in the “VehicleClassId” session attribute, or with FORCE_MULTI_VEHICLE_CLASS session flag, and the correct values in “VehicleClassId”, “MultiClassSlots” and “MultiClassSlot[1-3]” session attributes.
- ServerControlsVehicle: Similar to the vehicle class control, but completely disables the vehicle selection UI of all players. It should always be combined with the session flag FOCE_IDENTICAL_VEHICLES and the correct vehicle id set in the “VehicleModelId” session attribute.

Server Attributes

The above section already mentioned “session attributes”. The server attributes store the setup and state of the game on the server. There are three kinds of attributes:

- Session attributes: The setup and state of the multiplayer session. Because each server can host only one multiplayer session, these attributes do not need any additional identification other than their name. Most of the “setup”-style session attributes can be modified through the APIs when the server is running in “controls game’s setup” mode.
- Member attributes (or player attributes): The setup and state of individual session members, or players. These attributes are all read-only in the current version of the server. Each session member is identified by their “reference id”, or “refid”. This is a relatively random 16-bit integer, that changes if the same player leaves and rejoins the session later.
- Participant attributes: State of each participant. Since this is a racing game, each participant corresponds to one vehicle in the race. Participants can be controlled by players, or they can be AI vehicles. Participants are identified by their ids, which are 32-bit integers - since the ids start at 1 and increment by one for each new participant created, you will rarely see large numbers here, but in theory they still require 32-bit data type for storage.

Attributes can be of only two types:

- Integral attributes, with data sizes of 1, 2 or 4 bytes. All attribute integers are signed. Boolean attributes are represented as 1-byte integers, with zero meaning false, and

non-zero meaning true ; the server itself will always use 1 for true, but do not rely on that.
- String attributes. All strings on the server use UTF-8 encoding. In most cases the server does not modify or interpret any strings, so they are treated as arrays of bytes. In some track or vehicle names you will encounter non-ASCII characters, so it's still important to interpret the strings as UTF-8 where necessary. As a side note, all files written and read by the server are also expected to be encoded in UTF-8.

See the Dedicated Server Values and Types document for the full list of attributes.

Extended Session Attributes

The attributes contain various identifiers such as track id, vehicle class id or vehicle model id. Those are stored as 32-bit signed integers in the attribute tables. Also enumeration types such as "weather" are represented by integer values. The values are all available in the Values and Types document.

The Lua API adds several helper functions that allow you to use the corresponding names in place of the integers. So for example, the TrackId attribute can be set to integer `1988984740` meaning "Brands Hatch Indy". In many contexts in Lua, the attribute can actually be set to string "Brands Hatch Indy" and the code will automatically convert the value to the integral id. In the context of session attributes these automatic conversions apply to:

- TrackId
- VehicleClassId
- VehicleModelId
- Flags
- All "enum" attributes

Id and enum conversion is straightforward - instead of the numeric identifier/value, the name string can be used. Flags can combine multiple flag names into one integer, the string representation is then the value names concatenated by commas. For example, this is a valid string-form value of the Flags attribute:

```
FILL_SESSION_WITH_AI,ABS_ALLOWED,SC_ALLOWED,TCS_ALLOWED
```

No spaces are allowed in the string Flags format. Numbers are still allowed and can be combined with string flag values.

The "sms_base" Lua addon that's always loaded will overload the following Lua API builtins: SetSessionAttributes, SetNextSessionAttributes and SetSessionAndNextAttributes. These overridden builtins can access ids, enums and flags in string forms. It also defines several functions which help with the "normalization" (converting string values to numeric values) and "stringification" (converting numbers to strings), as documented in Attribute Normalization and Stringification.

Server Events

Another way of communicating the server's and game's state from on the server are "events". While attributes are used to read and modify the current state, the events are stored in a log and can be examined later.

Similar to attributes, events come in three kinds: session events, session member events, and participant events. Each event is identified by its kind, name, and attributes specific to that event. Session member events include the member's refid, participant events include the participant id and the refid of the member owning the participant.

See the Dedicated Server Values and Types document for the full list of all events.

HTTP API

Please refer to the official API forum thread at http://forum.projectcarsgame.com/showthread.php?26520-Dedicated-Server-API, that should contain all necessary information about the API.

Lua API

The dedicated server includes support for scripting the server in Lua. Lua is a popular scripting language used in many games, so it should not be hard to find enough documentation and help with the language itself on the Internet. To be a bit more specific, the server uses Lua version 5.3. It's compiled with all standard Lua library functions enabled, integer types are 64-bit and floating-point numbers are doubles (also 64-bit). Few links to start with:

- Main Lua web page: http://www.lua.org/
- Lua 5.3 readme - changes compared to version 5.2: http://www.lua.org/manual/5.3/readme.html#changes
- Lua 5.3 reference manual - http://www.lua.org/manual/5.3/

Please be aware that most information found on the Internet might be about Lua 5.1 or Lua 5.2. There are few significant changes in Lua 5.3, mostly the addition of integers, and few incompatibilities when compared to older versions. These are described in Section 8 of the Lua 5.3 reference manual.

Lua API - Server Configuration

To enable the Lua API, set the server configuration option "enableLuaApi" to true. This is the default value in the sample config. The server will load all addons specified in array "luaApiAddons" on startup, as well as any of their dependencies. The default server config file enables several sample addons that ship with the server. More information about these addons can be found in Dedicated Server User Guide, in section Server Addons. If you change the list of addons to load or want to reload an addon after updating it, the server needs to be restarted.

The config can be also used to control the location of several directories related to Lua addons:

- “luaAddonRoot”: The root directory from which the addons are loaded. It defaults to “lua”, which is relative to the current directory when starting the server. This will work when running the server executable directly from its directory when the directory structure is preserved after downloading the server from Steam. But you might have to customize it if running the server from another directory.
- “luaConfigRoot”: The directory where the server stores Lua addon configuration files and persistent data files. Defaults to “lua_config”.
- “luaOutputRoot”: Currently not used.

Lua Addon Structure

All Lua scripts are separated into “addons”. Each addon consist of a metafile with basic information about the addon, default addon configuration, and the addon’s Lua scripts. When the server starts, it loads addons specified in the configuration’s “luaApiAddons” list, and all addons required by the addons in the list as their dependencies.

The addons shipped with the server are all located in the “lua” subdirectory below the executable, and it’s recommended to just copy any custom addons into the same directory.

The recommended naming conventions for the Lua addons are:

- sms_ADDON: Addon developed by SMS and shipped with the server.
- lib_ADDON: Library addon that does not do anything on its own, but provides useful functionality that can be used by other addons
- xxx_ADDON: It’s recommended to use similar naming conventions for addons developed by community members. Use a short prefix based on the developer’s name or nickname followed by the addon name.

Addon Metafile

Each addon must include addon metafile stored in file `<lua_root>/<addon_name>/<addon_name>.json`, where `<lua_root>` is specified in the server config’s “luaAddonRoot”, and `<addon_name>` is the addon’s name. The syntax of the file is the same extended JSON syntax as that of the the server configuration file. The contents of the metafile are:

- “version”: List with the major and minor version number of the addon. The server only remembers the version, it’s not used for anything specific. It can be queried from the Lua addon script by builtin GetAddonVersion(). Example “`version : [ 1, 0 ]`”
- “description”: Short description text. This is only stored but not used by the server.

- “apiVersion”: List with the major and minor version of the The Lua API required by the addon. If this version number is not compatible with the API version provided by the server, the addon will not be loaded. The current API version is 3.1 (written as “`[ 3, 1 ]`” list in the metafile). The compatibility checks as follows:
    - The major version needs to match exactly. The current major version is 3, and it’s increased when a compatibility-breaking change is introduced to the API.
    - The requested major version must be equal to or greater than the current API version. The current minor version is 1, and it’s increased when a new feature is added to the API, but the API remains backwards compatible.

  So for example, addon requesting API version 3.1 would load on server providing API 3.1 and API 3.2, but would not load on server providing API 2.x, 3.0 or 4.x.
- “depends”: List of addon names this addon requires for its functionality. Leave this empty if the addon has no dependencies. All addons implicitly depend on “sms_base”.
- “files”: The list of Lua files to be loaded, relative to the addon’s root directory.

Default Addon Configuration

In addition to the metafile, each addon needs to include default configuration file in `<lua_root>/<addon_name>/<addon_name>_default_config.json`. This file also needs to use the usual extended JSON syntax and contain exactly two entries:

- “version”: The configuration file’s version. Start at 1 for any new addon.
- “config”: A JSON object with the default configuration. This will be available to the addon as a Lua table.

When the server loads an addon for the first time, the default configuration will be automatically copied over into `<lua_config>/<addon_name>_config.json`, which can be then customized by the addon’s user. This is the file which the server loads, and the passes to all addon’s scripts (see Addon Loading below).

If the “version” of the default config is increased, the server will notice that it’s not equal to the customized user’s addon. In this case, it will print an information message, backup the customized config (using its version number in the name), and copy over the new default config file. Increase the version whenever the addon’s configuration changes in a significant incompatible way. But try to avoid doing this too often, because it forces users to edit the configuration again.

Addon Loading

All Lua addons are loaded at server’s startup, in order specified by the “luaApiAddons” list from the server configuration. For each addon in the list the server goes through these steps:

1. Load and parse the addon’s metafile, fail loading the addon if the metafile is not present or can’t be parsed.
2. Check the “apiVersion” required by the metafile, and fail loading the addon if it does not match the server’s Lua API version.

3. Load the addon's default configuration file and parse its "version". Fail loading the addon if there are any problems with the file.
4. Check if custom configuration file for the file exists
    a. If yes, parse it and check if its version matches the default configuration file version. If yes, remember the custom config, otherwise backup the custom file and continue to b.
    b. If no (or if there was a version mismatch), copy the default configuration file to the custom file's location, and remember the default config.
5. Check if persistent addon data file file exists and load it if it does.
6. Go through all "depends" of the addon, and if any of those addons have not been loaded yet, load them first. Fail loading of this addon if any of the depends fail to load. Also fail loading the addon if a circular dependency is detected.
7. Now finally load the addon scripts from "files".

The script files themselves are loaded in the order as specified by the "files" option, and each is ran individually. The server passes the "addon storage" table to the addon as the only argument through the … parameter. The addon storage is a table created for the addon, with two items stored in it by the server:

- "config": A table stored at the config key contains the configuration table remembered in step 4. So if the user never customized the config, this will be the default addon config JSON converted to Lua table following the rules described in Data Type Conversions, otherwise it will be the customized config. Because the structure of the config can be modified by users the addon code should always check its content for validity, in case the user removed something by mistake or used a wrong data type somewhere, and fix it if possible or at least print a user-friendly warning message.
- "data": Addon's persistent data loaded from `<lua_config>/<addon_name>_data.json`. If the addon needs to store any data that persists through server restart, it should store them into this table, and then call builtin SavePersistentData().

The addon is free to use any other keys in the storage table for whatever it wants, there are currently no plans to reserve any other keys for the server's use.

Example 1: Access addon configuration

```
local addon_storage = ...
local addon_config = addon_storage.config
-- Now read the config from the addon_config table
```

Example 2: Access persistent addon data

```
local addon_storage = ...
local addon_data = addon_storage.data
-- Store anything in addon_data as the addon runs
-- And then tell the server to save the data
SavePersistentData()
```

Also see Data Type Conversions - JSON for some additional information about storing complex nested types in the addon persistent data table.

Special Core Addon: sms_base

While loading dependencies of an addon, the special addon “sms_base” is always checked as the first dependency. Therefore this addon is always loaded first, no matter what. The addon defines various helper functions and data, and it is considered the integral part of the server’s Lua API. Any data structures it adds are listed in Server Lists Exposed to Lua, and any functions it defines are included in Server Lua Library. You can also have a look at its source to see what exactly defines and how.

Data Type Conversions

All Lua data types will be converted to and from internal server’s data types by the API at several points. These are:

Session Attributes

Session attributes, session member attributes and participant attributes are used to store the setup and status of the session, players and cars. The attributes are identified by names, which translate to strings in Lua. Their values can be only integers or strings, no other data types are allowed. These translate to and from Lua directly. Be aware that “true/false” kinds of attributes are represented as “1/0” integers, and that in Lua the number zero evaluates to true in boolean context. This can be a source of unexpected errors, so be extra careful when checking the values of those attributes. Always write code like:

```
    if session.attributes.ServerControlsSetup ~= 0 then … end
```

and never as

```
    if session.attributes.ServerCotnrolsSetup then … end
```

JSON

The server stores configuration files and persistent addon data as JSON. Conversions between JSON and Lua data types can be a bit tricky in some cases. The scalar conversion rules are quite simple:

- Lua strings are converted to/from JSON strings.
- Lua boolean values are converted to/from JSON “true” and “false” identifiers.
- Lua integral or floating point numbers are converted to/from JSON’s numbers. Because JSON does not specify the exact numeric data types and does not distinguish integers and floating-point numbers, this conversion is a bit more complicated.
    - To JSON: The only numbers that are converted to JSON numbers directly are integers that fit into 32 bits. Big integers and floating-point numbers are formatted

as strings into JSON, to prevent precision loss when reading them back in various parsers.
    - From JSON: Quoted strings that contain only numbers remain strings after conversion to Lua, the reader does not attempt to guess whether “1.23” should be converted to a number, it never does that. Proper JSON numbers are converted to 64-bit Lua integers or double-precision floating-point numbers depending on their value.

  This means that non-integral or large 64-bit integral values can change their Lua data type after being saved in the persistent addon storage, they can become a string when read back into the addon. Each addon is responsible for dealing with this potential issue if it needs to store large/floating-point values in the persistent storage.

Even a bit more tricky than the number conversion are Lua tables. Tables are a weird and not very sensible Lua container type that can be used as a map (hash table, associative array, key-value list, whatever you want to call it), or as an array, or as both in a mixed way. On the other hand, most sane languages and also the JSON data format clearly distinguish between maps/objects and arrays/lists.

When converting a Lua table to JSON the server tries to automatically guess whether the table represents an array or an object, if not given depending on the context ; for example a config table is always converted to an object. The rules for generic table->JSON conversion are:
- Empty table is “object”
- Table with keys being only 1, 2, …, N with no gaps is “array”
- Anything else is “object”

This automatic guess should not cause any problems with data stored and then read back by a Lua addon, for example when using the persistent storage. After reading the JSON back it will end up a table again no matter what. But for JSON output that is then processed by external tools, the automatic conversion of empty tables to objects might be something to look out for.

The bigger issue with table conversion is the fact that JSON objects allow only strings as keys, while Lua can support pretty much any data types as keys. The conversion to JSON will always try to convert all keys to strings, and fail if there are any keys in the table that can’t be converted to strings. The conversion back to Lua table then does not try to do any guessing, and will leave all keys as strings. The Lua addon code needs to be aware of this, and for example when storing sparse array tables indexed by integers in the persistent storage, fix the data on startup to contain the expected key types. Otherwise you risk the danger of ending up with mixed strings and integers as the keys in the tables, which are different things in Lua (you can have table with two keys “123” and 123).

The Server Lua Library extends the Lua “table” object with several useful functions, one of them being `table.deep_copy_normalized`. This can be used to create a copy of the persistent data table structure in one call, automatically converting some sub-tables to use integral keys instead of string keys. You can check the sources of the “sms_stats” addon for a sample usage.

Note that passing arguments and receiving results from Lua API builtins performs similar conversions, because internal server's data structures closely mirror the JSON restrictions ; to be specific, arrays and maps are different container types, and maps can have only string keys. This is usually not an issue in the API as the interface does not use tables in a manner that could cause unexpected conversions anywhere.

Server Lists Exposed to Lua

Server attributes contain values of many different types. In many cases, enumeration-style types use integral values or ids. The server defines global tables which list all available id, enumeration and flags values. Those can be used to determine which id means which track, what's the value of the "force identical vehicles" flag and so on.

All builtin lists follow the same schema. They are all stored in the global table with a simple name "lists". Each list in the table is then another table with these contents:

- "description": String describing the use of the list
- "list": Table with the list itself. Each list is an array of structures. The format of the structures is different for each list, but all lists contain at least the "name" field.

The addon "sms_base" defines several additional tables, extracting information from the array-style builtin lists. These extra tables provide quick lookup from names to values or values to names for the most useful data types.

Note that all these lists should be used as read-only, never write anything into them. Currently the server does not protect them against unexpected writes, but modifying them will do no good and will mostly likely break not yours but also any other running addons (the global namespace and therefore the tables are shared between all addons).

The lists are also available in separate document Dedicated Server Values and Types.

Track List

The list of all known tracks is available in `lists.tracks`, with `lists.tracks.description` containing the description text, and the `lists.tracks.list` being the actual list. Each structure in the list has these fields:

- "name": The track's name, as UTF-8 string.
- "id": The track's id, a 32-bit signed integer.

There are also helper global tables `id_to_track` and `name_to_track`, allowing a simple key-value lookup to tracks by either their id or name.

Vehicle List

The list of all known vehicles is available in `lists.vehicles`. Each structure in the list has these fields:

- “name”: The vehicle’s name, as UTF-8 string.
- “id”: The vehicle’s id, a 32-bit signed integer.
- “class”: The vehicle class, using its name (not the class’s id!)
- “liveries”: A table-array with the list of all vehicle’s liveries. Each element of the array is a structure with “id” and “name” of the livery.

There are also helper global tables `id_to_vehicle` and `name_to_vehicle`, allowing a simple key-value lookup to vehicles by either their id or name.

Vehicle Class List

The list of all known vehicle classes is available in `lists.vehicle_classes`. Each structure in the list has these fields:

- “name”: The class name, as UTF-8 string.
- “id”: The class id, a 32-bit signed integer.

There are also helper global tables `id_to_vehicle_class` and `name_to_vehicle_class`, allowing a simple key-value lookup to vehicle classes by either their id or name.

Server Attribute List

The lists of all server attributes are stored in:

- Session attributes: `lists.attributes.session`
- Session member (player) attributes: `lists.attributes.member`
- Participant (car) attributes: `lists.attributes.participant`

Each of these lists contains structures with these fields:

- “name”: The attribute’s name.
- “type”: The type, one of “int8”, “int16”, “int32” or “string”.
- “access”: Access rights to the attribute. “ReadOnly” attributes can be only read, while “ReadWrite” can be also modified. The ServerControlsSetup attribute is “ReadWrite” but it can be modified only by the server itself.
- “description”: A short description.

You can use these helper global tables to look up individual attributes by their names: `name_to_session_attribute`, `name_to_member_attribute`, `name_to_participant_attribute`. See Session Status and Attributes for more information about the attribute structures.

Events List

While server attributes store the current status and setup, events describe changes in the server’s or session’s state. A Lua addon can monitor events by utilizing Addon Callbacks, events are also stored in a server log which can be read later by Lua builtin functions (see GetEventLogInfo and GetEventLogRange).

The list of all event types is stored in `lists.events`. Each structure in the list contains these fields:

- “name”: The event’s name
- “type”: The type, one of “Session”, “Player” or “Participant”.
- “description”: A short description.
- “attributes”: List of attributes associated with the event. Each element of the list is a structure describing the attribute, with “name”, “type” and “description”. Unlike session/member/participant attributes, there is no “access” field - all events are read-only.

Enums Lists

Several attributes have “enum” type. The attribute values are integers, and their enum type can be used to translate the integer to a more user-friendly name. So for example, the attribute WeatherSlot1 might have value 3360755426, which translates to “Clear”.

There are multiple enum types, each having its own list as well as two global helper tables - one converting enum value to name, and one converting enum name to value. The following table summarizes all enum lists:

List	Name->Value	Value->Name
`lists.enums.damage`	`Damage.NAME`	`value_to_damage.N`
`lists.enums.tire_wear`	`TireWear.NAME`	`value_to_tire_wear.N`
`lists.enums.fuel_usage`	`FuelUsage.NAME`	`value_to_fuel_usage.N`
`lists.enums.penalties`	`Penalties.NAME`	`value_to_penalties.N`
`lists.enums.allowed_view`	`AllowedView.NAME`	`value_to_allowed_view.N`
`lists.enums.weather`	`Weather.NAME`	`value_to_weather.N`
`lists.enums.game_mode`	`GameMode.Name`	`value_to_game_mode.N`

The lists themselves contain two fields: “name” and “value”. So for example the list `lists.enums.weather.list` contains array of all weather enum names and values, `Weather.CLEAR` evaluates to 3360755426, and `value_to_weather[3360755426]` evaluates to “CLEAR”.

Flags Lists

Similar to “enum” attribute types there are also “flags” types. Flags also associate names to values, but the attribute value itself can combine several flag values together using binary “or” (or to be less precise, adding them together).

There are multiple flags types, with lists and tables similar to the enums tables:

List	Name->Value	Value->Name
`lists.flags.session`	`SessionFlags.NAME`	`value_to_session_flag.N`
`lists.flags.player`	`PlayerFlags.NAME`	`value_to_player_flag.N`

For example you could combine several session flags like this:
```
      local flags = SessionFlags.ABS_ALLOWED | SessionFlags.TCS_ALLOWED
```
Which yields the same result as
```
      local flags = 0x20 | 0x80
```
Which in decimal notation is the same value as “32 | 128” or “160”.

Callback List

During various in-game and server-side events the server can call a “callback” function registered by a Lua addon. This is documented more in Addon Callbacks. Each callback type is identified by an integral value, and the server provides mapping between these values and names, very similar to the enum lists:

- “lists.callbacks”: The callbacks list itself. Each structure contains “name”, “value” and “description”.
- “Callback”: Global table that maps callback name to value.
- “value_to_callback”: Reverse global table mapping callback value to name.

Tables with Server and Session Status

In addition to global static list tables the server also maintains global tables with the run-time state of the server and the multiplayer session hosted on the server.

Similar to the lists tables, these tables should be only read and never written to, even if the server does not protect them against writes yet.

The tables are:

Server Status

The global table “server” stores the server status. It contains these fields:

- “name”: The name of the server, as set in the server config.
- “max_player_count”: Maximum number of players on the server, again from the config.
- “password_protected”: Boolean set to true if and only if the server has a password set in the config.
- “state”: Current state of the server. See below.
- “secure”: Is the server configured as secure? This enables the usual Steam’s user authorization checks for any joining players.
- “connected_to_steam”: Is the server currently connected to Steam?
- “server_id”: The server’s 64-bit id as assigned by Steam.

The valid values of the “state” field are:

- “Idle”: The server is not running yet. You will most likely never see this value in Lua.
- “Starting”: The server is starting but not everything has been initialized yet. When Lua addons are already running this most likely means that the server has not connected to Steam yet.
- “Running”: The server is up and running but not hosting any session.
- “RunningActive”: The server is running and hosting a multiplayer session.

Session Status and Attributes

The global table “session” stores all information about the multiplayer session hosted on the server. Most of its content make sense only while there is an active session running on the server, i.e. only when `server.state == “RunningActive”`. The most important exception to this is the `session.next_attributes` table which (if no session is active) contains the setup of the next session to be created on the server. The session table contains these fields:

- “manager_state”: The state of the session manager. Be aware that this is not the same thing as the session’s gameplay state, stored in session attribute “SessionState”.
- “name”: The name assigned to this session.
- “lobbyid”: Steam’s 64-bit id assigned to the lobby used by the session. Note that in the future we might remove this field, as we plan to change the server to no longer rely on Steam lobbies, which should improve networking stability when the server itself is running fine but Steam servers have problems.
- “joinable”: Is the session hosted on the server currently joinable? The game itself decides the value, you can’t modify this from the server. Sessions are joinable in the lobby, and also during practice/qualification sessions as long enough time is left until the end of the session.

- “max_member_count”: Maximum number of players that can join the session. Can be lower than the number configured for the server if the session is set up that way, but never can be higher.
- “attributes”: The session attributes, see below for more information.
- “next_attributes”: Attributes that will be applied “next”, which is either when the current race finishes and everyone loads back into the lobby, or if the session completely ends, when the next session is created on this server. This field is accessible even when there is no session running on the server.
- “members”: Table with all session members (players), identified by their “refid”. See below for more information.
- “participants”: Table with all participants (vehicles), identified by their participant id. See below for more information.

The valid values of the “manager_state” field are:
- “Idle”: No session is running on the server.
- “Allocating”: First member is joining the server. The server is busy and no longer available to new sessions, but it’s not available yet for join until the first member is fully authenticated and joined.
- “Running”: The server is hosting an active session.

Session Attributes

All attributes of the current multiplayer session are stored in `session.attributes`. Separate document Dedicated Server Values and Types lists all available session attributes. The list is also available via the HTTP API at the /api/list/attributes/session endpoint, and in Lua global tables `lists.attributes.session` and `name_to_session_attributes`, as documented in Server Attribute List.

While the current version of the server does not prevent modification of this table, it should be used as read-only. Modifying contents of the table **will not** modify the game’s setup, it will just confuse other addons. In the future the table will most likely become write-protected.

Any changes to the attributes table will be also notified to the game via the SessionAttributesChanged callback, see Addon Callbacks for more info.

In addition to current session attributes, there is another table at `session.next_attributes`. This stores attributes that will be applied to the “next session”:
- When the current race finishes and everyone loads back to the lobby, next attributes will be applied to the session.
- When everyone leaves the server, next attributes will be applied and show in the in-game browser, and new session created on the server will use those attributes.

This table is also read-only, you need to request changes to attributes via builtin functions documented in Server Lua Library: SetSessionAttributes, SetNextSessionAttributes, SetSessionAndNextAttributes. Changing the attributes directly has undefined behavior, and will be forbidden in the future.

Unlike `session.attributes`, the table `session.next_attributes` is available even when there is no session running on the server, and can be used to reconfigure the attributes for the next lobby that will be created on the server. Addons that rotate settings will therefore most likely request changes to “next attributes”, rather than modifying the attributes of current session, which works only while players are waiting in the lobby. On the other hand for example an addon that listens to player’s chat, and handles commands to modify or vote to modify certain settings will want to modify the active setup - and also probably the “next” setup as well, so the changes are remembered for the next session.

Session Members

The `session.members` table contains all members (players) joined in the session. Each member is identified by their “refid”, which is a unique 16-bit integer assigned to each new joining member. If a player leaves the session and then rejoins later, they will be assigned a new refid. If you want to treat a player rejoining the session multiple times as a single entity, use their Steam ID instead.

The table maps member refids to structures with these fields:

- “index”: The member’s index. This index is zero-based, so the values will be between 0 and the session size - 1. The game’s lobby UI displays players ordered by this index.
- “refid”: The refid again.
- “steamid”: The player’s Steam ID, a 64-bit integer.
- “state”: The player’s state. Can be one of “Idle”, “Authenticating”, “Connected” or “Disconnected”. Disconnected session members will be removed from the session rather quickly, so you will usually not see their state as such.
- “name”: Steam profile name of the player.
- “jointime”: Time when the player joined, UTC Unix time in seconds.
- “host”: Is the player the session’s host? This is a true boolean value (unlike boolean-like attributes which are 0/1 integers).
- “attributes”: The member’s attributes.

Similar to session attributes, member attributes are documented in Dedicated Server Values and Types, and the list is available via HTTP API at /api/list/attributes/member, and in Lua global list tables `lists.attributes.member` and `name_to_member_attribute`. The attributes are read-only, and so is each member table.

Changes to members are notified by these callbacks: MemberJoined, MemberStateChanged, MemberAttributesChanged, HostMigrated, MemberLeft.

Session Participants

Session participants sub-table works similar to members, and it lists the vehicles in the game. It's available at `session.participants`. Participants are identified by 32-bit integers assigned by the game. The table maps these ids to structures with these fields:

- "id": The participant's id again
- "attributes": The participant attributes

Again the participant attribute list is available in the usual places - in document Dedicated Server Values and Types, via HTTP API at /api/list/attributes/participant, and in Lua global list tables `lists.attributes.participant` and `name_to_participants_attribute`. The attributes are read-only, and so is each participant table.

Participants are created by the game, and won't be available until it starts loading into a race. When a race is finished, all participants are destroyed by the game - participants do not persist between consecutive races. Even if the players all return back to the lobby and then start another race (without restarting the session), brand new participants with unique ids will be created for them.

Usually there is one participant for each player and each participant corresponds to an in-game vehicle. But in the future there might be either participant-less players or vehicle-less participants if dedicated spectator slots are implemented.

Two participant attributes are important in mapping participants to session members:

- "RefId": This is the refid of the member that "owns" the participant.
- "IsPlayer": This is a 0/1 integral boolean. For each refid only one participant associated with that refid will be tagged with IsPlayer equal to 1, and this is that player's participant car. Participants with IsPlayer equal to 0 are AI vehicles that are assigned to that member for game simulation purposes.

These callbacks are associated with participants: ParticipantCreated, ParitiicipantAttributesChanged, ParticipantRemoved.

Server Lua Library

The sections above documented global Lua tables maintained by the server - static lists and dynamic attributes. All those tables are read-only and are used only to communicate the state from the server to Lua.

The API of course also implements several functions that let you communicate "from Lua to server". These functions can be split into two sets - extensions of the basic Lua libraries, not related to the server's functionality, and the Lua API itself - server "builtins".

Sandboxing

The server does not use any kind of sandboxing when running Lua addons, and has all standard Lua libraries enabled. These functions are documented in Lua 5.3 - Standard Libraries. While the server allows the addons to call any standard Lua functions, including those in “io” and “os” packages, we might decide to restrict some of that functionality in the future. In general it’s recommended to follow these rules and your addon should be ok even after future updates:

- “dofile”, “load”, “loadfile”: Do not use this, instead just list your addon files and any dependency addons in the metafile
- ”print”: Feel free to use this for debugging purposes, the sms addons already print quite a lot. In the future we plan to implement proper logging builtins instead, using the server’s logging subsystems.
- Coroutine Manipulation: The way the server calls registered Lua callbacks is most likely not compatible with Lua coroutines, do not try to yield from a callback function.
- Modules: Use addon file list and dependencies to load other Lua addons. There is a high chance that we might not allow loading any C libraries from Lua, unless some public addons make a very good use of this feature, or at least require the server’s user to explicitly allow this for select addons, to prevent potential security issues.
- I/O library: We might modify its functionality to allow opening files only in the `<lua_output>/<addon_name>` directory.
- OS library: Do not use `os.exit`, `os.execute`, `os.remove`, `os.rename`, `os.setlocale` - those will most likely be disabled.

Lua Library Extensions

The “sms_base” addon extends the base Lua libraries with several helper functions:

dump(table [, indent])

Prints content of given table using the standard `print` Lua function, recursively. Indent is a string prepended to each row, and it will be automatically extended by two spaces for each sub-table.

dump_typed(table [, indent])

Similar to `dump` but also prints the types of all keys and values. This is useful when debugging issues with tables that contain mixed keys - in Lua string key “123” is different from integer key 123, and you can run into this problem when working with tables persisted in JSON files. See Data Type Conversions - JSON and table.deep_copy_normalized.

string:split(pattern [, results])

Split string into substrings at “pattern”, create an array of the substrings and return it. The pattern is just an ordinary string, the function does not support splitting by regular expressions. If the optional table “results” is passed to the function, the function will append the substrings into this table rather than creating a new table.

Example:

```
      local s = “foo,bar,baz”
      s:split( “,” )
```

returns array `{ “foo”, “bar”, “baz” }`

table.shallow_copy(other_table)

Creates a shallow copy of given table and returns it. Shallow copy means that the function just creates a new table and copies the keys and values from the `other_table` as is, so if the other table contains any sub-tables, those sub-tables will be also referenced from the new table.

table.deep_copy(other_table)

Creates a deep copy of given table and returns it. Unlike shadow copy, deep copying a table will recursively deep copy all sub-tables, therefore creating a completely “independent” clone of any complex nested table structures.

In addition to that, this function also assigns metatable to the new table, equal to the metatable of the `other_table` (the metatable itself will not be a clone).

table.deep_copy_normalized(other_table [, intkey_table_names [, this_table_name]])

This a special extension of the `deep_copy` function. The base functionality is the same, but it can also transform keys in some tables. When this function creates a clone of table which has name that’s included in `intkey_table_names`, the cloning of such table will automatically convert all keys of the table to numbers, using Lua’s `tonumber.`

The `intkey_table_names` should be a “set” table - the keys should be table names, and values set to true. You call table.list_to_set to create such “set” table from a simple “list of names” table (array). The names will be used to determine which tables should be transformed to numeric keys. I.e. the cloning logic is “`if intkey_table_names[ table_name ] then` convert keys of the table to numbers”. “Table name” is defined as:

- The name of the table passed to this function is specified in `this_table_name`
- The name of a nested table is the key at which the table is stored

This is a very specialized function that will be mostly useful only at addon “startup” code to normalize persistent data saved by the addon previously. Because the JSON storage supports only string keys, any tables with integral keys that are persisted by SavePersistentData will have their keys converted to strings when the server loads the data on startup. This is documented in Data Type Conversions - JSON.

Note that this function can be used to fix only integer-string key type mismatches. It's recommended to use tables with only string and integer keys in persistent data, you will have to convert any other data types manually in your addon code.

Example: The "sms_rotate" maintains server statistics and stores them in persistent tables. Some of those tables use integral keys - the per-user tables (keys are player Steam IDs), per-track tables (keys are track ids) and so on. Simplified, its code does something like this on startup:

```
	local addon_storage = ...
	local addon_data = addon_storage.data
	local table_names = { players = true, tracks = true }
	addon_data = table.deep_copy_normalized( addon_data, table_names )
	addon_storage.data = addon_data
```

This fixes the key types after load, and then the addon uses `addon_data` directly.

table.list_to_set(list)

Coverts array-style table to set-style table. An array table is a table with keys being indices from 1 to N, and any values. The corresponding set will have those values as keys, all mapped to "true" value. This representation greatly simplifies queries of type "does table contain X". While for arrays you have to search through all elements, for sets you can simply call "`if set[ x ] then ... end`".

table.add(table, key, delta)

If given table contains the `key`, adds `delta` to its value. Otherwise creates new value at the `key` with value set to `delta`. Or in other words, same as `table[ key ] = table[ key ] + value`, but behaves nicely even if no such key is in the table and works as if the value was zero then.

table.subtract(table, key, delta)

Same as table.add but subtracts the value instead.

Server Builtins

Server builtins are the core of the API - the Lua functions defined by the server which let the addon code communicate with the server. The server defines these builtin functions:

GetBuildVersion()

Returns the server built version, as a single integer. This number is increased with each public release by at least one.

GetProtocolVersion()

Return the game-server protocol version. This version matches the in-game number and usually increases when a new major patch is released.

GetApiVersion()

Returns the API version of the server. The addon should use the metafile to specify with which API versions it's compatible (see Addon Metafile). It can use this function to query the exact version. The version will be returned as a table-array with two fields - the major and minor version numbers.

GetAddonVersion()

Returns the addon version as specified in the addon metafile. The version will be returned as a table-array with two fields - the major and minor version numbers.

RegisterCallback(function)

Registers function that should be called as the addon callback. Each addon can have only one callback function registered, its arguments can be used to determine the type of the callback and the callback-specific arguments. See Addon Callbacks for more information about callbacks.

UnregisterCallback()

Unregisters the function previously registered by RegisterCallback.

EnableCallback(callback_id)

Enables callbacks with given id. A callback function must be registered by RegisterCallback, this function then tells the server which callbacks should be actually delivered to this addon. Initially all callbacks are disabled.

DisableCallback(callback_id)

Disables callback previously enabled by EnableCallback.

SavePersistentData()

Saves addon persistent data to JSON file stored in `<lua_config>/<addon_name>_data.json`. These data are initially passed to the addon via the … arguments, and can be accessed like this:

```
local addon_storage = ...
local addon_data = addon_storage.data
```

… use addon_data table in your code to store anything …

Then calling this function will save the contents of the `addon_data` table. Be aware that in JSON “tables” can have only string keys. It's recommended to use only strings as keys in the persistent addon table and its subtables, or alternatively integers and then fixup the data at addon startup (see table.deep_copy_normalized).

GetUtcUnixTime()

Retrieves server “Unix time” - system time in seconds since the “epoch”, which is defined as 1 January 1970, 00:00:00 UTC. This time is synchronized to Steam servers and does not rely on the system time of the server on which the dedicated server is running.

Be aware that in the future we might support running the server in “offline” mode where it would not require stable Steam connection, then this value might be rather unreliable. Use this function to get something resembling “current date and time”. For more detailed timings in the addon, use the monotonic timer provided by GetServerUptimeMs().

GetServerUptimeMs()

Returns current server time in milliseconds. This is a monotonic timer, i.e. it will always tick up, even if the local OS time jumps backwards. The initial value when the server starts is undefined, do not assume it starts at zero. It has 64-bit precision, so there should be no issues with potential overflow.

SendChatToAll(message)

Sends chat text to all members of the session. The chat message will be displayed as is.

SendChatToMember(refid, message)

Sends chat text to single member of the session, identified by their refid.

KickMember(refid [, ban_seconds])

Removes session member from the session. The member is identified by their refid. Optionally this function can also apply a temporary ban if `ban_seconds` is specified. This ban does not persist through server restarts.

Only players who are members of the current session can be kicked and temp-banned by this call, it can’t be used to ban players “in advance”. In the future we might add runtime access to server blacklists and whitelists to provide more persistent banning options..

StopSession()

Removes everyone from the current session, which effectively destroys the session and makes the server available for a new session.

SetSessionAttributes(attributes)

Requests session attribute change from the game. This call changes the attributes of the current session, and can be used only while the session is active and in lobby. Only writable attributes can be modified by this call.

The server needs to communicate with the game to apply the changes, so they won't be reflected in `session.attributes` immediately. Some of the changes might be rejected by the game, such as requesting a non-existing track id - but do not rely on "nonsense" attributes to be always correctly handled by the game. Also even if the game is in the lobby state it might transition to loading before the server communicates the change request, in which case the request will be ignored by the game.

See Session Attributes for more information about the attributes this call can modify. The "sms_base" addon overrides this builtin to apply "session attribute normalization" - it automatically converts string values to numeric identifiers where possible before passing the request to the server. Therefore attributes such as track id, various enums or flags can be set using the track/enum/flag names instead of the numeric values. See Extended Session Attributes for more information.

There is no way to modify individual attributes of session members or participants.

SetNextSessionAttributes(attributes)

Requests change of "next session attributes". Unlike SetSessionAttributes, this call does not communicate with the game and the changes are applied instantly to the global table `session.next_attributes`. These attributes are then automatically applied to the game when:

- A new session is created on the server
- Current race ends and everyone loads back to lobby

This call also changes the information displayed about the server in the game browser. The "sms_base" addon overrides this builtin to support the automatic session attribute normalization (see Extended Session Attributes).

SetSessionAndNextAttributes(attributes)

This call combines SetSessionAttributes and SetNextSessionAttributes. If you want to modify the current lobby attributes, and also have them apply to any lobbies that will follow, use SetSessionAndNextAttributes. Just SetSessionAttributes alone modifies only the the current lobby, and the next lobby is then overwritten with "next attributes" unless those are updated as well.

This call will succeed even when the game is not in a lobby state or when there is no session hosted on the server at all. Then it will behave exactly the same way as SetNextSessionAttributes.

GetEventLogInfo()

Returns information about the event log.

Whenever the server generates an event, it notifies all addons that enabled the EventLogged callback. The server also remembers the event in its internal log, and this function returns a table with basic information about the log. The table contains these fields:

- “first”: The index of the oldest event in the log. Once the server’s log gets full it starts discarding old events, and this number will start increasing. The log size can be changed by the server configuration option “eventsLogSize”.
- “count”: The number of events stored in the log.

GetEventLogRange(offset [, count])

Returns log entries from the event log. The `offset` argument specifies the index of the first event to retrieve. It can be also negative, in which case it’s relative to the log’s end. The `count` argument specifies how many events should be retrieved. If it’s not set, all events until the end of the log will be retrieved.

Example 1: Retrieve 10 latest events:

```
      local events = GetEventLogRange( -10, 10 )
```

Example 2: Retrieve all available log events:

```
      local log_info = GetEventLogInfo()
      local events = GetEventLogRange( log_info.first )
```

The call returns a table with three fields. Fields “first” and “count” contain the same information as what GetEventLogInfo returns. Field “events” contains the returned events, it’s an array table with these structure:

- “index”: Index of the event.
- “time”: Unix time in seconds when the event was generated.
- “type”: Even type, one of “Session”, “Player” or “Participant”.
- “name”: The event’s name.
- “refid”: If this is a “Player” or “Participant” event, the refid of the player associated with the event.
- “participantid”: If this is a “Participant” event, the id of the participant associated with the event.
- “attributes”: Table with event attribute values.

Please refer to event lists for information about the individual event types and their attributes. Some basic information is available in Events List, the full list is available in a separate document Dedicated Server Values and Types, via HTTP API at /api/list/events and in Lua global tables `lists.events`.

Attribute Normalization and Stringification

Server attributes use numeric values to represent various identifiers, enums and flags. The “sms_base” addon implements several helper functions that can be used to convert these numeric values to the corresponding name strings (stringification) and from strings to values (normalization). This is documented in Extended Session Attributes. The addon also overrides the builtins used to manipulate session attributes to automatically apply attribute normalization.

While you can use the lists tables to do these conversion manually, the helper functions simplify this task.

Normalization

To convert session attributes with string values of ids, enums and flags to attributes with numeric values, simply call

```
normalize_session_attributes( attributes )
```

Be aware that this normalizes the attributes in-place, if you want to preserve the original table use table.deep_copy first.

The "sms_base" addon also exposes these functions that are used by `normalize_session_attributes` internally:

- `normalize_session_attribute( key, value )`
  Returns normalized session attribute value, given the attribute name as `key`.
- `normalize_track( value )`
  `normalize_vehicle( value )`
  `normalize_vehicle_class( value )`
  Converts track, vehicle or class name to id, if possible.
- `normalize_session_enum( name_to_value, value )`
  Converts the string-form value to numeric value using the `name_to_value` table. For example to normalize a weather value, you would use:
  ```
  normalize_session_enum( Weather, "Clear" )
  ```
  Even though the function is called normalize "session" enum, it can be used on player and participant enums as well if the appropriate `name_to_value` table is supplied.
- `normalize_session_flags( name_to_value, value )`
  Similar to the enum normalization function, but supports comma-separated list of multiple flags, and will convert each of them individually then OR them together:
  ```
  normalize_session_flags( SessionFlags, "ABS_ALLOWED,SC_ALLOWED" )
  ```

All of the functions above accept either stringified values, and then they will attempt to convert them to corresponding numbers, but also already numeric values and then they will just return them as is. The flags conversion supports any combination of string-form and numeric flags. The functions return the normalized value.

Stringification

To convert session attributes table to a table with stringified values, call

```
stringify_session_attributes( attributes )
```

This conversion will modify the attributes in-place. It still returns the table reference for easier call chaining, but the return value is always the same as the attributes argument. So never call it on `session.attributes` directly, that structure is read-only! Instead for example to print the current attributes when debugging you addon, use:

```
	local attrs = table.deep_copy( session.attributes )
	dump( stringify_session_attributes( attrs ) )
```

The "sms_base" addon also implements the following conversion functions, all of them are counterparts of the helper functions described in Normalization:

- `stringify_session_attribute( key, value )`
- `stringify_track( value )`
- `stringify_vehicle( value )`
- `stringify_vehicle_class( value )`
- `stringify_session_enum( value_to_name, value )`
- `stringify_session_flags( value_to_name, value )`

Track and Vehicle Names

Two simple helper functions `get_track_name_by_id( track_id )` and `get_vehicle_name_by_id( vehicle_id )` are closely related to the attribute stringification. They are helpful mostly in debugging prints and are almost equivalent to

```
	id_to_track[ track_id ].name
	id_to_vehicle[ vehicle_id ].name
```

with the improvement that they do not crash if the id is not a valid one, they return a "unknown id 123"-style string instead.

Addon Callbacks

The server generates notifications which are delivered to Lua addons via callbacks.

Each addon can register a callback function by calling builtin RegisterCallback. Then the addon should tell the server in which types of notifications it is interested by calling EnableCallback. The server will then call the registered callback function whenever it generates notification of type matching any of the enabled callback types.

The addon callback function is called with one or more arguments. The first argument is always the callback id, same as the argument previously passed to EnableCallback. Any additional arguments depend on the callback types.

Example: Handle callbacks generated when an event is logged, and process "PlayerJoined" events:

```
	local function handle_player_event( event )
		local refid = event.refid
```

```
        local player = session.members[ refid ]
        if event.name == “PlayerJoined” and player then
              print( “Player ” + player.name + “ joined” )
        end
  end

  local function addon_callback( callback, ... )
        if callback == Callback.EventLogged then
              local event = ...
              if event.type == “Player” then
                    handle_player_event( event )
              end
        end
  end

  RegisterCallback( addon_callback )
  EnableCallback( Callback.EventLogged )
```

The supported callback types are:

Tick

Called whenever the server runs its internal tick. This happens less often when the server is idle and is waiting for someone to create a multiplayer session, and the server ticks more often while hosting a session.

This callback type has no arguments.

ServerStateChanged

Called whenever the server's state changes.

This callback type has two arguments:

- Previous server state.
- New server state.

The current server state is always available in `server.state`, see Server Status for the list of possible state values. When the notification is delivered the `server.state` value will contain the new state.

SessionManagerStateChanged

Called whenever the session manager's state changes.

This callback type has two arguments:

- Previous session manager state.
- New session manager state.

The current session manager state is always available in `session.manager_state`, see Session Status and Attributes for the list of possible state values. When the notification is delivered the `session.manager_state` value will contain the new state.

Note that this state is different from the gameplay-related session state attribute available in `session.attributes.SessionState`. Changes to that attribute are notified via session event "StateChanged", and you can use the SessionAttributesChanged callback type to listen for those changes, or alternatively the event notification EventLogged and then check if the event is a "Session" event with name "StateChanged".

SessionAttributesChanged

Called whenever the session's attributes change. The only argument to this callback is an array-style table with the list of attributes that have changed. You can read their new values from `session.attributes`.

NextSessionAttributesChanged

Called whenever the "next session" attributes change, also with a table argument containing the names of the modified attributes.

MemberJoined

Called when a new session member joins the server. The argument is the member's refid.

This notification is generated quite early in the join process, as soon as the server adds the member into its lists. Not many details will be known about the member at this point, and the game might even reject the player. If you are interested in the event that's generated when a player fully joins the game hosted on the server, use the EventLogged callback type instead and listen for "Player" event with name "PlayerJoined".

MemberStateChanged

Called when the state of a session member changes.

This callback type has three arguments:

- Refid of the session member.
- Previous member's state.
- New member's state.

MemberAttributesChanged

Called whenever session member's attributes change.

This callback type has two arguments:

- Refid of the session member.
- Array-style table with the list of attributes that have changed. You can read the attributes from `session.members[ refid ].attributes`

HostMigrated

Called whenever the host migrates to another session member. The only argument is the refid of the new host.

MemberLeft

Called when a session member leaves the session. The only argument is the refid of the leaving member.

ParticipantCreated

Called when a new participant is created by the game. The only argument is the participant's id.

ParitICipantAttributesChanged

Called when participant's attributes change.

This callback has two arguments:

- Id of the participant.
- Array-style table with the list of attributes that have changed. You can read the attributes from `session.participants[ participantid ].attributes`

ParticipantRemoved

Called when a participant is removed from the game. The only argument is the participant's id.

EventLogged

Called when the server logs an event. The callback has one argument, the event itself, which is a table with:

- "index": Index of the event
- "time": Unix time in seconds when the event was generated
- "type": Even type, one of "Session", "Player" or "Participant".
- "name": The event's name.

- “refid”: If this is a “Player” or “Participant” event, the refid of the player associated with the event
- “participantid”: If this is a “Participant” event, the id of the participant associated with the event
- “attributes”: Table with event attribute values.

The individual events are listed in the Dedicated Server Values and Types document, via HTTP API at /api/list/events and in Lua global tables `lists.events`.