# **Hunter’z Persistency Module**

# **Introduction**

Hunter’z Persistency Module is a framework designed for mission-makers looking to run campaign-like scenarios ranging from multi-session operations (MSO or similar), to survival and role-play missions, or anything that you feel would be enhanced by persistency. It is a fully customisable, offline persistency framework. In other words, it does not need complex infrastructure like SQL servers and databases. It works based on file read/writes on server machine and stores full information on vehicles and items, and player units for persistent missions. It offers reconnect persistency for players, and is especially made compatible with ACE and ACRE. It accurately restores the state of vehicles and players down to ammo count upon server restart.

This module doesn’t work by just saving every vehicle/soldier existing on the battlefield. In most cases, that’s not what mission makers are after, and it’s highly inefficient. Instead, it saves whatever you want it to save. This is best described as the vehicles, ammo crates and objects (ranging from fortifications, to lamps and campfires – basically could be anything you want) that you “own”, or otherwise want to save. You achieve this by setting these objects as “persistent” through the calling of API functions described in this manual. Once you set an object as persistent, it will stay as persistent throughout your campaign, until the point it is destroyed or deleted.

In addition, you are able to configure variables as persistent. Variables can be in global (aka mission namespace) or object namespace, and each will have their own API function to easily configure them as persistent. Persistent global variables can be used to store gameplay/campaign information, such as how many civilians have been killed in total, or how much money your side has, while object namespace variables can be made persistent to store “getvariable/setvariable” values together with the persistent vehicles/objects themselves, allowing you to store further information specific to different objects.

Player persistency works automatically. Upon reconnecting, your character’s state will be fully restored, down to how many rounds you have left in your magazines, and your medical state if you enable ACE medical persistency through the module settings.

As this is a persistency framework, it is designed for complex dynamic missions in mind. Anything you want to save, is expected to be dynamically spawned during mission execution, and not placed into the mission using the editor. For example, it’s sensible to save vehicles that you spawn or “buy” using a vehicle spawner/store, but not a vehicle that you place in the editor that always re-appears when you restart the server anyway.

# **A Note on Persistency for User-placed Map Markers**

Hunter’z Persistency Module is also able to save user-placed map markers. Due to the limitations of the Arma 3 engine, only actual markers (and not lines) are able to be saved. However, since a subtle change in the Arma 3 netcode in one of the recent game updates (v1.80 if I remember right…), the server is only able to retrieve user-placed marker information from the *Global Channel*. It is advised to brief players on this matter, so they adjust themselves to using the Global Channel when placing markers. Any marker placed in the global channel will become automatically persistent, with no need to adjust any settings. You can also make things easier for players by disabling other channels from *description.ext* within your mission. For more info:  
<https://community.bistudio.com/wiki/Description.ext#disableChannels>

# **How it Works**

Hunter’z Persistency Module should have no performance impact. The only time it uses resources is when it is saving to file (how often that happens is configured by you), which is a near-instant process, or when dealing with player persistency, which is also well optimised and uses server-side Disconnect/Connect eventhandlers to store and retrieve the information. Currently the only drawback is the pause-menu delay, which you see when you hit the Esc button to open your game menu while in multiplayer. This occurs because the module is configured to send any local variables from your player’s side that you configured as persistent to the server, thinking that you might be attempting to disconnect when you hit the menu. This is a very efficient way of handling the updating on the server of players’ local variables, but is a bit annoying if you for whatever reason constantly want to access your game menu while playing.

In terms of how the saving works, the module uses the Debug Console Extension by Killzonekid, which is already included in this package and you don’t need to obtain it separately. This is a dll extension and allows the game to create a text file on the server’s hard-drive which effectively becomes your “save file”. The principle of saving is by autosaving like in other games. You configure how often the module conducts an autosave, and it will create a new save file every time, while keeping any older files in case they’re needed by you later.

Save files will be created inside the Arma 3 installation folder on the server. The name of these files will be in the following format "debug\_console(\_x64)\_xxxx-xx-xx\_xx-xx-xx". This is a bit awkward, but it’s because that’s how the debug console names its files. The good thing is that you get a timestamp in the file name. The part with all the "x"s will be the date & time. In order to make things easier, when you restart your server, you should rename the latest save file to a name that can be recognised by the module as the save file to load from. You can also move it to another folder inside the Arma 3 installation folder to keep it separated from everything else. This is all explained in the Setup section of the manual. There is also a batch file provided in the Tools folder to allow you to automate this process, with instructions.

# **Vehicles, Objects and Crates**

When you look at the API functions, you’ll realise the module defines different types of objects that you can set persistent. These go under 3 categories: vehicles, objects and crates. Essentially, there isn’t much of a difference between them, but it still helps to keep them separate. So what you have to watch out for is to use the correct function for the correct type of object. It’s simple: use vehicle functions for vehicles, use crate functions for crates or boxes that you use to hold weapons, ammo or any other items inside, and for everything else like fortifications or miscellaneous items use object functions. You need to be aware that unless you specifically set some object to be persistent by using the appropriate API function, it won’t be persistent. So if you have a vehicle spawner in your mission, for example, and you want all vehicles spawned by it to be persistent, you have to make that happen by calling *Hz\_pers\_API\_addVehicle* for each new vehicle that it spawns. The same goes for a crate spawner, an object spawner, or some object that your team captures after completing a task. All of this needs to be scripted by you, as the module can’t guess what you want to achieve.

# **Persistent Object Namespace Variables**

For true persistency, it’s indeed not enough to just save a bunch of objects and vehicles. While doing so will save their physical state and some other intrinsic information such as ammo content, fuel, etc., in many cases you’ll also want to save some more information along with each object. If you have a lot of custom scripts running, chances are they will be storing variables in object namespace, such as perhaps a vehicle access script that determines who is allowed to drive a particular car, using a variable called “vehicle\_access” which might point to an array with names of different players. You wouldn’t want a server restart to wipe this information. The module is able to handle all of this, by letting you set any object namespace variables such as the one in the example as persistent. Object namespace variables are set globally for all persistent objects, which means when you set such a variable as persistent, it will check the existence of that variable for each persistent object. So you only need to set it once, and not for each and every object individually. It doesn’t matter if not all objects have this variable defined. It will not affect the correct functioning of the module or your scripts in any way. So to go along with the previous example, it doesn’t matter if only one of my cars has restricted access and has the variable “vehicle\_access” defined. However, if I later make another one of my persistent cars have restricted access, I don’t need to redefine the “vehicle\_access” variable as persistent, since it already is on the persistency list. When the module conducts a save, it will pick this up and store the information if suddenly this new vehicle also has this variable defined.

So that’s how object namespace variables are made persistent. You have these kinds of variables divided into the same categories as in the previous section, with vehicles, crates and objects keeping a different list of persistent variables. In addition, you also have player variables which are object namespace variables specific to players. They work in exactly the same way, storing further information on player units if needed. All of these have their respective API commands.

# **Persistent Mission Namespace Variables**

The module is also able to save any global variables, which are sometimes called “mission namespace variables”. An API command exists to configure global variables as persistent.

**A word of caution!**

Although object namespace variables don’t need to be defined for all objects (or any object for that matter), mission namespace variables that you set as persistent should always be defined! If the module attempts to save, and you previously added a global variable to be persistent, and that variable does not exist (or is nil) at the time of saving, the behaviour of the module is undefined. You will most likely get errors and the save may be unsuccessful, or even corrupted. It is your responsibility to ensure any global variables you make persistent are defined and “known” by the server at the time of saving.

# **Undefined Variable in Logs**

It is advisable to check the logs for testing after you set up the module for the first time. Even if things appear to be working, there might easily be one or two things you haven’t noticed and are causing problems. **However, one important thing to be aware of is that the module by default will log some errors in the RPT during loading from a save file, warning you repeatedly of an undefined return value from some array parsing functions.** This is because those functions are configured to pass nil values as part of their normal operation. Those aren’t errors, so you can ignore them, but be on the lookout for any other errors mixed up in between that might be caused by your setup.

# **Setup**

## Setting Up the Environment

Because saving is done through a dll extension, you have to make sure your server has the required C++ runtime libraries installed. The extension uses Microsoft Visual C++ 2013 Redistributable Libraries. You can obtain this from here: <https://www.microsoft.com/en-gb/download/details.aspx?id=40784>. You need to make sure both the x86 and x64 versions are installed, so be sure to download and install both of them from the link provided. You might find that you already have them installed, but it’s good to verify.

One more thing to note, to get the module to work correctly, is to make sure your server has “filepatching” enabled. This is to ensure that your server is allowed to read from files outside of the mission folder, i.e. from directly inside the Arma 3 installation folder. Filepatching is disabled by default, and you must enable it manually on the server. If you don’t, your server will give you an error telling you it can’t find the save file, since it’s not allowed to look in there. To enable filepatching, use the “-filepatching” startup parameter for your server. If you have no idea what startup parameters are, then it’s time to read the wiki <https://community.bistudio.com/wiki/Arma_3_Startup_Parameters>.

Note that if you want to conduct tests with the module in singleplayer or in the editor, you need to enable filepatching for your game too, from the Arma 3 Launcher *parameters* tab, since in that case you are also considered to be a “server”.

Configuring Module Parameters Through the Editor

Setting up the module is done through the configuration of parameters through the mission editor. To enable the module in your mission, find it in the editor modules list under the “Hz” title, and deploy it. You will see a few options here, which need to be configured correctly for the module to work.

## First Time Launch Handler Function

The first item you see when you open up the module options is the *First Time Launch Handler Function*. This is the name of a function you must write and define in global namespace inside your mission. The name comes from the idea that you’re launching your persistent mission for the very first time. It is basically the main thing that will allow you to suit the module for your mission’s needs, during the mission’s first launch. This function will only be called by the module on the server the first time you run your mission, or when the module does not find a save file and assumes you want to start fresh. If it finds a save file, it will instead load your progress from there, and not call this function. You don’t have to worry about the module being too early to check for the existence of the function. In fact, the module will keep waiting and won’t proceed until this function is defined during your mission’s first launch.

So what is this function really about, and what is it supposed to contain? It is inside this function that you can put your API calls to set up your persistent variables during your first launch. This function can contain any number of calls to set mission, object, crate, vehicle or player variables as persistent. These API functions are described later in this manual, and you can understand better how this all works by checking out the example mission provided. If you are reading this for the first time, I advise you to go and check out the example mission provided right now and see what the *First Time Launch Handler Function* is about. You can find explanations to what the API calls inside do in the later part of this manual.

**Important Note**

If you want to set more variables persistent later as your campaign progresses, changing the contents of your *First Time Launch Handler Function* will have no effect, since as mentioned above this function is only called at first launch of your mission, or when no save file is found. So if you want to both keep your progress, and add some new variables that should be saved, you need to be able to execute code in real-time on the server during runtime, for example by logging in as admin and using the debug console. The API functions can still be called while the mission is loaded in and running, and it will influence the next autosave. So, let’s say you have a new variable called “captured\_outposts”, which you added into your mission in your latest version, and you didn’t have this in your *First Time Launch Handler Function* during your first launch. If you want to set this to be a persistent variable, you will have to start up your server, wait for it to load, and then use the in-game debug console, or otherwise, and execute the appropriate API call at runtime. You will notice that the next time the server saves to file, you will have this variable stored.

Autosaving

It is possible to set the frequency of autosaving through the module options. By default, this is 1 hour. Remember that each autosave will create a new save file, without overwriting the old one. It is advisable to keep some old save files handy, in case something goes wrong down the line and you can revert your progress without losing everything and starting again.

## Autoloading from save file

Loading is done automatically at mission start after you restart your server. As mentioned earlier, the server will attempt to load from a save file when starting up, assuming that it exists. If it can’t be found, it will run the *First Time Launch Handler Function* instead. You can set the delay after mission start for the autoloading to take place through the module options. It might be useful to include a relatively longer delay, if your mission has many initialisations that it needs to go through, that you want to be finished before the save file is loaded.

You will have to configure the module correctly such that it finds the save file you want to load, which will in most cases be the latest save file. Since all the old saves will also be kept, you will normally have a list of save files in the Arma 3 folder after a session, each created after an autosave. The normal procedure is to take the latest save file that your last session created (the " debug\_console(\_x64)\_xxxx-xx-xx\_xx-xx-xx” thingy), rename it to something more sensible, like “lastSave” or whatever, and if you want, move it to another folder (which still should be inside the Arma 3 root folder). The critical thing here, is to make the module “know” what your save file name is and where it can be found, so that it can load it. This is done through the configuration of the parameter “Path to Save File”. A batch program exists in the tools folder of this addon, which will allow you to automate this procedure. Instructions on how to use it can also be found there.

## Client Loading on Reconnect

Client loading describes how a player loads their saved state upon reconnecting to the server. This can be done in 2 different ways. The first mode works similarly to how the server autoloads game progress from a save file, i.e. using a delay. You configure this delay in seconds through the module options, which defines how long your game will wait after joining the game, for it to load your previous state. Again, longer delays are useful if you have loads of scripts initialising when you join in and you want them to finish initialising before loading.

The second method you can use is an optimised version of the above explained method. Instead of waiting for a set amount time, the game waits for a variable on the client to become true before loading. This is useful if you don’t want to guesstimate the delay, and just want manual control over when to let it load. If you want to use this method instead, simply tick “Enable Client Manual Load” in the module options. This will override your delay setting and instead wait for you to set the variable “Hz\_pers\_clientReadyForLoad = true;”. You can use this at the end of your mission init script, for example, to make loading happen without any unnecessary delay, while also ensuring everything that should happen before loading has already finished.

# **API Functions**

## Hz\_pers\_API\_addVehicle

**Use:** Turns an existing vehicle persistent.

**Argument(s):** \_this: reference to object type -> object

**Example:** \_car **call** Hz\_pers\_API\_ addVehicle;

Hz\_pers\_API\_addVehicleVariable

**Use:** Makes an object namespace variable persistent for all of your persistent vehicles. Can only be called by the server. The variable does not have to be defined for all vehicles.

**Argument(s):** \_this select 0: name of variable type -> string

\_this select 1: variable is publicvariabled type-> bool

**Example:** [“isLocked”, false] **call** Hz\_pers\_API\_ addVehicleVariable;

## Hz\_pers\_API\_addCrate

**Use:** Turns an existing ammo crate persistent.

**Argument(s):** \_this: reference to object type -> object

**Example:** \_crate **call** Hz\_pers\_API\_addCrate;

## Hz\_pers\_API\_addCrateVariable

**Use:** Makes an object namespace variable persistent for all of your persistent ammo crates. Can only be called by the server. The variable does not have to be defined for all crates.

**Argument(s):** \_this select 0: name of variable type -> string

\_this select 1: variable is publicvariabled type-> bool

**Example:** [“canBeMoved”, true] **call** Hz\_pers\_API\_ addCrateVariable;

## Hz\_pers\_API\_addObject

**Use:** Turns an existing object persistent.

**Argument(s):** \_this: reference to object type -> object

**Example:** \_obj **call** Hz\_pers\_API\_addCrate;

## Hz\_pers\_API\_addObjectVariable

**Use:** Makes an object namespace variable persistent for persistent objects added using the Hz\_pers\_API\_addObject function. Can only be called by the server. The variable does not have to be defined for all objects.

**Argument(s):** \_this select 0: name of variable type -> string

\_this select 1: variable is publicvariabled type-> bool

**Example:** [“canBeCarried”, true] **call** Hz\_pers\_API\_ addCrateVariable;

## Hz\_pers\_API\_addPlayerVariable

**Use:** Makes an object namespace variable persistent for all player units. Can only be called by the server. The variable does not have to be defined for all units.

**Argument(s):** \_this select 0: name of variable type -> string

\_this select 1: variable is publicvariabled type-> bool

**Example:** [“isMedic”, false] **call** Hz\_pers\_API\_ addPlayerVariable;

Hz\_pers\_API\_addMissionVariable

**Use:** Makes a global variable persistent. Can only be called by the server. This variable must have been defined before the server attempts to save.

**Argument(s):** \_this select 0: name of variable type -> string

\_this select 1: parsing descriptor type -> number

\_this select 2: variable is publicvariabled type -> bool

The **parsing descriptor** is a number between **0** to **3** that is needed for the module to understand what kind of variable this is.

Use **0** for anything that is not an array.

If it’s an array, depending on its dimension you need to use either:

**1** if it’s a one-dimensional array

**2** if it’s a multidimensional array in the form **[ [ ],[ ],[ ],… ]**

Or **3** if it’s an array of multidimensional arrays, as in **[** **[ [ ],[ ],[ ],… ] , [ [ ],[ ],[ ],… ], … ]**

It’s unlikely that you’ll have to store any data form this complex, so you’ll most likely be using either 0 or 1. However, if you don’t use the correct number, you can corrupt your save file! For example, if you put “0” there and you’ve actually got an array, it might work, but if that array grows to become very large, you’ll eventually corrupt your save file.

**Example:** [“numberOfTasksCompleted”, 0, false] **call** Hz\_pers\_API\_ addMissionVariable;

[“adminNames”, 1, true] **call** Hz\_pers\_API\_ addMissionVariable;

## Hz\_pers\_API\_disablePlayerSaveStateOnDisconnect

**Use:** Allows you to disable saving for a particular player. This is a useful measure to take against possible exploits related to using persistency, or in case you don’t want a player to benefit from it. Must be called from the player’s side.

**Argument(s):** None

**Example:** **call** Hz\_pers\_API\_disablePlayerSaveStateOnDisconnect;

## Hz\_pers\_API\_enablePlayerSaveStateOnDisconnect

**Use:** Allows you to re-enable saving for a player after having disabled it using the previous function. Must be called from the player’s side.

**Argument(s):** None

**Example:** **call** Hz\_pers\_API\_enablePlayerSaveStateOnDisconnect;