

Utilisation

Guide d'utilisation — EasyDB

Ce guide s'adresse à quelqu'un qui écrit du code mission/serveur pour FastRP et veut lire/écrire dans MySQL depuis SQF. Pour builder, packager ou déployer l'addon lui-même, voir le [README](#).

Prérequis

- Le mod `@EasyDB` chargé sur le serveur dédié (`-mod=@EasyDB`), **jamais** nécessaire côté client — tout ce qui suit ne fonctionne que dans du code qui s'exécute côté serveur (`isServer == true`).
- `easydb.json` posé à la racine de `@EasyDB/`, à côté de `EasyDB_x64.so` (voir `easydb.json.example`), avec les identifiants MySQL. L'extension détecte son propre dossier toute seule, rien à configurer côté serveur.

Rien d'autre à initialiser : `EasyDB_fnc_init` tourne automatiquement au démarrage de la mission et enregistre ce qu'il faut pour l'async.

Les deux fonctions

	<code>EasyDB_fnc_query</code>	<code>EasyDB_fnc_queryAsync</code>
Usage	lookups rapides	tout ce qui peut être lent ou volumineux
Bloquant ?	oui (thread appelant, timeout moteur 1s)	non, retourne immédiatement
Résultat	renvoyé directement	livré plus tard via callback
Taille du résultat	refusé si > ~10 Ko	pas de limite (découpé automatiquement)

Règle simple : si c'est un `SELECT` / `UPDATE` sur une ligne avec peu de colonnes (argent, grade, un compteur...), `EasyDB_fnc_query` convient. Dès que ça touche un inventaire complet, une liste de plusieurs joueurs, ou un batch de plusieurs requêtes, utiliser `EasyDB_fnc_queryAsync`.

Toutes les requêtes sont des **statements préparés** : `_sql` contient des `?` comme placeholders, les valeurs vont dans le tableau `_params`. Ne jamais construire `_sql` en concaténant des valeurs venant du joueur — c'est exactement ce que les `?` évitent.

EasyDB_fnc_query — synchrone

```
params ["_sql", ["_params", []]];
// Retourne [true, resultat] ou [false, messageErreur]
```

Exemple — lire l'argent d'un joueur

```
private _result = ["SELECT money FROM players WHERE steam_id = ?", [_steamId]] call
EasyDB_fnc_query;
_result params ["_ok", "_data"];

if (!_ok) exitWith {
    diag_log text format ["[FastRP] échec lecture argent pour %1: %2", _steamId, _data];
};

_data params ["_columns", "_rows"];
if (count _rows == 0) exitWith { /* joueur pas encore en base */ };

private _money = parseNumber ((_rows select 0) select 0); // colonnes = STRING, caster au
besoin
```

Exemple — incrémenter l'argent (opération atomique côté SQL)

```
private _result = [
    "UPDATE players SET money = money + ? WHERE steam_id = ?",
    [_montant, _steamId]
] call EasyDB_fnc_query;
_result params ["_ok", "_data"];
// _data = [affectedRows, lastInsertId] pour un UPDATE/INSERT/DELETE
if (_ok && { (_data select 0) == 0 }) then {
```

```
diag_log text format ["[FastRP] steam_id %1 introuvable, argent non crédité", _steamId];  
};
```

Faire l'incrément dans le `UPDATE (money = money + ?)` plutôt que lire-modifier-écrire depuis SQF évite les races entre deux appels concurrents sur le même joueur.

Exemple — changer le grade/palier d'un job

```
["UPDATE players SET grade = ? WHERE steam_id = ?", [_nouveauGrade, _steamId]] call  
EasyDB_fnc_query;
```

EasyDB_fnc_queryAsync — asynchrone

```
params ["_sql", ["_params", []], "_callback"];  
// Retourne [true, requestId] immédiatement, ou [false, messageErreur] si  
// la requête est rejetée tout de suite (SQL/arguments invalides)
```

`_callback` est le **nom** (STRING) d'une fonction globale déjà compilée (via `CfgFunctions` par exemple), pas une code-string. Elle sera appelée plus tard avec :

```
params ["_ok", "_data"];  
// même format que le retour de EasyDB_fnc_query : [colonnes, lignes] ou  
// [affectedRows, lastInsertId] si _ok, message d'erreur sinon
```

Exemple — charger l'inventaire (JSON) d'un joueur à la connexion

```
// dans le CfgFunctions du mod FastRP, tag FastRP :  
// TAG_fnc_onInventoryLoaded.sqf  
params ["_ok", "_data"];  
if (!_ok) exitWith { diag_log text format ["[FastRP] échec chargement inventaire: %1", _data];
```

```
};
_data params ["_columns", "_rows"];
if (count _rows == 0) exitWith {};
private _inventoryJson = (_rows select 0) select 0; // colonne JSON déjà décodée en tableau SQF
// _inventoryJson est de la forme [{"weapon","arifle_MX_F"},["items",[...]]] pour
{"weapon":"arifle_MX_F","items":[...]}
```

```
// appel, côté connexion joueur :
[
    "SELECT inventory FROM players WHERE steam_id = ?",
    [_steamId],
    "TAG_fnc_onInventoryLoaded"
] call EasyDB_fnc_queryAsync;
```

Exemple — sauvegarder l'inventaire (écrire du JSON)

Le plus simple est de construire le JSON directement en SQF (ou de le garder tel quel si vous le stockez déjà sous cette forme) et de le passer comme n'importe quel paramètre STRING — MySQL valide et stocke le JSON lui-même :

```
private _json = '{"weapon":"arifle_MX_F","items":["FirstAidKit","HandGrenade"]}';
[
    "UPDATE players SET inventory = ? WHERE steam_id = ?",
    [_json, _steamId],
    "TAG_fnc_onInventorySaved"
] call EasyDB_fnc_queryAsync;
```

(Si vous préférez éviter d'écrire du JSON à la main en SQF, un petit helper `TAG_fnc_arrayToJson` côté mission reste la responsabilité du code FastRP — EasyDB ne fait la conversion JSON→SQF que dans le sens lecture, volontairement, pour rester simple.)

Format des valeurs

- Toute valeur scalaire (colonnes non-JSON) revient en **STRING**, y compris les nombres — faire `parseNumber` si besoin d'un number SQF. Raison : évite les surprises

d'arrondi/précision sur les `DECIMAL`/`BIGINT` et garde le marshaling simple et uniforme.

- `NULL` (SQL) devient `""` (string vide).
- Une colonne `JSON` est décodée récursivement :
 - un objet JSON `{"a":1,"b":[2,3]}` devient `[["a",1],["b",[2,3]]]` (paires clé/valeur triées par clé, puisque SQF n'a pas de type "map")
 - un tableau JSON devient un tableau SQF classique
 - `null` JSON devient `""`

Gestion des erreurs

`EasyDB_fnc_query`/`queryAsync` retournent toujours `[false, message]` en cas de problème, jamais d'exception SQF. Causes possibles :

- SQL invalide ou erreur MySQL (contrainte violée, table inexistante...) → message = l'erreur MySQL brute
- Résultat trop gros pour la voie synchrone → `"result too large for sync query, use queryAsync"` (utiliser la version async)
- Requête mal formée côté SQF (mauvais types d'arguments) → message explicite, à vérifier en dev

Toujours tester `_ok` avant d'utiliser `_data` — ne pas supposer que l'appel a réussi.

Bonnes pratiques

- **Toujours** des `?` + `_params`, jamais de concaténation de valeurs joueur dans `_sql` (injection SQL).
- Préférer les opérations atomiques côté SQL (`money = money + ?`) à un cycle lire/modifier/écrire depuis SQF, pour éviter les races.
- Réserver `EasyDB_fnc_query` aux lookups qui répondent en quelques millisecondes ; dès que le volume de données ou la charge serveur est incertaine, passer en async pour ne pas risquer de geler le serveur pendant 1 seconde (timeout moteur).
- Un callback `queryAsync` peut arriver plusieurs frames après l'appel (découpage en plusieurs morceaux pour les gros résultats) : ne pas supposer qu'il arrive avant la fin du frame courant.

Dépannage

- `diag_log` affiche `[EasyDB] initialized` au démarrage si l'addon a bien démarré côté serveur (sinon : mod pas chargé, ou lancé côté client).

- Si `EasyDB_fnc_query/queryAsync` renvoient systématiquement `[false, "EasyDB_fnc_query must be called server-side"]` : le code est exécuté côté client, il faut le passer côté serveur (ou via `remoteExec` ciblé serveur si l'appel part d'un client).
 - Erreur `init error: ... : easydb.json` introuvable/mal formé (vérifier qu'il est bien à côté de `EasyDB_x64.so`, ou que `EASYDB_CONFIG` pointe au bon endroit), ou connexion MySQL impossible — vérifier les identifiants et les logs du serveur MySQL.
-

Revision #2

Created 2026-09-29 07:01:09 UTC by ProtoMehka

Updated 2026-09-29 07:19:26 UTC by ProtoMehka