ChatGPT

Handleiding: FiveM scripting voor client, server en NUI met events, state bags en exports

Inleiding

FiveM maakt het mogelijk om maatwerk **scripts** te schrijven die zowel op de server als op de client (game) draaien. In deze handleiding leer je de belangrijkste bouwstenen kennen om een script te maken, waaronder het onderscheid tussen clientside en serverside logica, het gebruik van **state bags** (gesynchroniseerde data), het werken met **events** (gebeurtenissen) en **NUI** (HTML-gebaseerde user interface) en het delen van functies via **exports**. We behandelen voorbeelden in **Lua** (hoofdzakelijk) en geven waar relevant ook JavaScript-equivalenten. Deze gids is bedoeld als uitgebreide documentatie, bruikbaar voor ontwikkelaars én als naslag voor AI-ondersteuning bij scriptgeneratie.

Client-side vs Server-side scripting

In FiveM bestaat een resource vaak uit **clientscripts** (draaien bij iedere speler) en **serverscripts** (draaien op de server). Beide kanten hebben toegang tot verschillende functies en events. Zo kan een clientscript bijvoorbeeld de positie van een speler of voertuigaansturing regelen, terwijl een serverscript toegang heeft tot alle spelers en gedeelde servergegevens (zoals databases). Sommige functies zijn alleen beschikbaar clientside of serverside – de FiveM documentatie heeft aparte referenties voor Client-side functies en Server-side functies.

Belangrijk verschil: clientscripts kunnen gebruikersinput en UI afhandelen en lokale game natives (zoals het besturen van voertuigen) aanroepen, terwijl serverscripts autoriteit hebben over spelwereld-logica (bijv. geld toekennen, spelers spawnen) en communicatie tussen spelers. Vaak communiceren client en server met elkaar via events (netwerkgebeurtenissen), wat we hieronder toelichten.

State Bags voor data-synchronisatie

State bags zijn een systeem om data (key/value-pairs) aan entiteiten, spelers of global state te hangen die automatisch gesynchroniseerd kunnen worden tussen server en clients [1]. Dit is handig om bij te houden *staat* van bijv. een voertuig of speler, zonder voortdurend expliciete events te sturen. Iedere entiteit in het spel (zoals een voertuig of ped) heeft een `.state` property die je kunt gebruiken om waarden op te slaan.

- **Entiteit state:** Je kunt de state van een entity uitlezen of instellen via de `Entity(...).state` API (Lua). Voorbeeld: in Lua krijg je een entiteit (bijv. een voertuig) en lees je een waarde `mollis` als `local state = Entity(vehicle).state` gevolgd door `print(state.mollis)` [2]. Een waarde zetten doe je met `Entity(vehicle).state.mollis = 'waarde'` [3]. *NB:* Dit mag alleen vanuit de eigenaar van de entity (client) of de server gebeuren [3].
- **Player state:** Ook spelers hebben een state-bag. In Lua kun je bijvoorbeeld `LocalPlayer.state.example = 123` zetten op de eigen client, of op de server `Player(source).state.example` uitlezen voor een bepaalde speler [4].

- **Global state:** Er is tevens een globale state-bag voor de hele server, alleen beschrijfbaar vanaf de server en alleen-lezen op clients. Bijvoorbeeld: `GlobalState.moneyEnabled = true` zet een globale flag aan [5].
- **Replicatie en lokaal:** Standaard worden waardes die de server zet gerepliceerd naar alle clients, en waardes die een client zet **niet** gerepliceerd (lokaal) [6] [7]. Je kunt dit per zet aanpassen met een `set`-functie. Voorbeeld: `Entity(vehicle).state:set('clone', 600, false)` op de server zet de waarde en voorkomt replicatie (alleen server houdt het bij) [8]. Andersom kan een client iets repliceren naar de server door de derde parameter op `true` te zetten (`Entity(obj).state:set('key', value, true)` op client stuurt naar server) [9].
- **Change handlers:** Je kunt handlers registreren om veranderingen in state bags te volgen. Met de native `AddStateBagChangeHandler` kun je bijv. een functie laten aanroepen zodra een bepaalde state-key verandert [10]. Zo kun je realtime reageren op wijzigingen, zowel client- als server-side.

State bags vervangen deels de noodzaak om constant events te sturen voor status. Ze zijn onderdeel van OneSync's *state awareness*. Let wel op de beperkingen: het ophalen van nested waarden werkt niet direct (je moet platte keys gebruiken) en iedere `.state` toegang deserialiseert de hele state [11] [12]. Ook kunnen spelers alleen hun eigen player-state schrijven, en clients alleen entity-state voor entiteiten die ze bezitten (anders moet de server het doen) [13]. Gebruik state bags dus voor gesynchroniseerde attributen, en reguliere events voor andere communicatie.

Eventsystem: luisteren en triggeren van events

FiveM maakt intensief gebruik van een **event-driven** architectuur. Je kunt zowel **ingebouwde events** van FiveM/Game (zoals speler spawnt, resource start, etc.) als **custom events** gebruiken. Het eventmechanisme werkt tussen client en server zodat je berichten kunt uitwisselen.

Event handlers registreren (luisteren naar events)

Om code uit te voeren wanneer een event plaatsvindt, registreer je een event handler. In Lua doe je dit met `AddEventHandler(eventNaam, function(...) end)` [14]. In JavaScript bestaat een vergelijkbare globale functie `on(eventNaam, (args) => { ... })` [15]. Wanneer het event **getriggerd** wordt, zal alle code in dergelijke handlers uitgevoerd worden.

Bij een *netwerk-event* (van de ene runtime naar de andere, bv. client->server) wordt in Lua/JS een globale variabele `source` beschikbaar gesteld binnen de handler, die het speler-ID bevat van degene die het event triggerde [16]. **Let op:** deze `source` is alleen geldig tijdens de directe aanroep van de handler. Als je asynchrone acties doet (bijv. `Wait()` in Lua of een `async/await` in JS), moet je `source` eerst opslaan in een lokale variabele, anders reset `source` terug naar de default waarde [17].

Een voorbeeld van een eenvoudige event-handler in Lua en JS:

```lua
-- Lua: luisteren naar een lokaal of netwerk event "eventName"
AddEventHandler("eventName", function(param1, param2)
    print("Event ontvangen met params:", param1, param2)
    -- Je code hier...
end)
```

```
// JS: luisteren naar hetzelfde event
on('eventName', (param1, param2) => {
    console.log('Event ontvangen:', param1, param2);
    // Je code hier...
});
```

Als je een **custom event over het netwerk** wilt gebruiken (dus een serverevent aanroepen vanaf de client of omgekeerd), moet je dat event eerst *registreren* als netwerk-event. In Lua doe je dat met `RegisterNetEvent("eventNaam")`. Bijvoorbeeld in een serverscript een event registreren dat door clients mag worden aangeroepen, of vice versa. De FiveM docs geven aan dat je bij Lua/JS expliciet moet registreren zodat het event niet geblokkeerd wordt [18] [19]. In JS kun je gebruikmaken van `onNet('eventName', handler)` om zowel lokale als netwerk events onder dezelfde naam te behandelen [20] (intern registreert dit ook het netwerk-event type). C# scripts hoeven dit niet expliciet door de manier waarop events daar werken, maar in Lua/JS is het vereist voor cross-script events [21].

Naast custom events kent FiveM **vooraf gedefinieerde events**. Bijvoorbeeld: - `onClientResourceStart` (clientside) en `onResourceStart` (serverside) worden getriggerd wanneer een resource start [22]. - `playerConnecting` (server event) wordt getriggerd als een speler wil verbinden; hiermee kun je bijvoorbeeld namen loggen of iemand weigeren. Deze levert parameters zoals de spelernaam, een functie om een kick-reden te zetten en een deferrals object voor bijv. whitelist controles [23]. - `playerDropped` (server event) wanneer een speler disconnect; handig om op te ruimen. - `chatMessage` (client event) bij in-game chat. - ...en vele anderen. Een volledig overzicht van events die je kunt afvangen staat in de documentatie onder *Client events* en *Server events*. Elke event-pagina beschrijft de parameters. Zo heeft *playerConnecting* bijvoorbeeld `playerName`, `setKickReason` en `deferrals` als argumenten [23]. Je gebruikt deze net als custom events via `AddEventHandler("eventNaam", function(args...) ...)`.

> **Tip:** Gebruik voor server events altijd `source` (of de meegeleverde speler object) om te weten *wie* het event veroorzaakte. Bijv. in een serverside `AddEventHandler('chatMessage', function(source, name, message) ... end)` geeft `source` de player ID.

Events triggeren (uitsturen van events)

Naast luisteren moet je ook events **triggeren** (afvuren) om ze daadwerkelijk te gebruiken. Er zijn verschillende situaties: - **Lokaal event triggeren:** Binnen dezelfde runtime (client->zelfde client of server->zelfde server). Gebruik in Lua de functie `TriggerEvent("eventName", param1, param2)` [24]. In JavaScript is het equivalent `emit("eventName", param1, param2)` [24] [25]. Dit roept direct de handlers in dezelfde omgeving aan. - **Server-event triggeren vanaf de client:** Wanneer je vanuit een clientscript een event naar de server wilt sturen, gebruik je `TriggerServerEvent("eventName", param1, ...)` (Lua) [26]. In JS is hiervoor `emitNet("eventName", param1, ...)` [27]. Dit stuurt een bericht naar de server die het betreffende event afhandelt. FiveM biedt ook een variant voor grote datastromen: `TriggerLatentServerEvent` (Lua/JS) waarbij je een extra *bytes per second* parameter meegeeft [28] [29]. Deze latent events blokkeren de netwerkbuffer niet volledig en zijn bedoeld voor bijvoorbeeld het versturen van grote hoeveelheden JSON of bestanden zonder timeouts. - **Client-event triggeren vanaf de server:** Als een serverscript een event op (één of meerdere) clients wil aanroepen, gebruik je `TriggerClientEvent("eventName", doelSpeler, param1, ...)` (Lua). Je moet specificeren naar **welke client(s)** het event gaat. Gebruik een specifieke player ID of gebruik `-1` om **alle**

verbonden clients aan te spreken [30]. Voorbeeld in Lua: `TriggerClientEvent("showHUD", somePlayerId, true)` triggert het *showHUD* event op één speler, terwijl `TriggerClientEvent("showHUD", -1, true)` het naar iedereen stuurt [30]. In JS doe je dit met `emitNet("eventName", targetPlayer, ... )` [31] (waar `targetPlayer` ook een ID of -1 kan zijn). Ook voor client-events bestaat een *latente* variant `TriggerLatentClientEvent` voor grote data, wederom met een BPS parameter [32] [33].

Enkele voorbeelden in code:

```
-- Lua: client -> server event triggeren
TriggerServerEvent("savePlayerScore", score, level)
```

```
-- Lua: server -> alle clients event triggeren
TriggerClientEvent("showAnnouncement", -1, "Server gaat rebooten over 5 min")
```

```
// JS: client -> server event triggeren
emitNet('savePlayerScore', score, level);
```

```
// JS: server -> specifieke client event triggeren
emitNet('showAnnouncement', targetSrc, 'Welkom op de server!');
```

Let op dat je voor het **ontvangen** van deze netwerk-events de events geregistreerd hebt (met `RegisterNetEvent` of `onNet`, zie vorige sectie). Als dat niet gebeurd is, zal een van de bovenstaande triggers niets doen om beveiligingsredenen [18].

Events annuleren (CancelEvent)

In bepaalde gevallen wil je een event *onderscheppen* en de verdere behandeling stoppen. FiveM biedt de native `CancelEvent()` functie die je *binnen een event handler* kunt aanroepen om het event te annuleren [34] [35]. Bijvoorbeeld: in de standaard chatresource wordt een `chatMessage` event geannuleerd om te voorkomen dat een bericht dubbel in de standaard chat komt.

Gebruik in Lua `CancelEvent()` (zonder parameters) in je handler, of in JS `CancelEvent();` binnen de `on('event', ...)` callback [36] [37]. **Belangrijk:** *CancelEvent* stopt **niet** andere handlers van datzelfde event – alle geregistreerde handlers worden nog steeds aangeroepen, maar het markeert bij FiveM dat het event "geannuleerd" is voor de game-engine of default acties [38]. In feite voorkomt het bijvoorbeeld dat een standaard actie wordt uitgevoerd (zoals broadcast naar chat of loggen), maar je eigen scripts die ook luisteren krijgen het event nog wel.

Je kunt ook checken of een event geannuleerd is. Als je vanuit een script een (lokaal) event triggert met `TriggerEvent`, kun je daarna de native `WasEventCanceled()` aanroepen om te zien of een van de handlers het event gecanceld heeft [39] [40]. In Lua retourneert `TriggerEvent()` zelfs direct *true* als het event geannuleerd is, zodat je eenvoudig kunt doen: `if TriggerEvent("naam", args) then ... end` [41]. Dit werkt alleen voor lokale events (binnen dezelfde runtime) [42].

Voorbeeld: Stel je hebt een event `onPlayerDeath` met meerdere handlers en één daarvan roept `CancelEvent()` aan (bijv. om respawn te blokkeren). Andere handlers krijgen het event nog, maar de game weet dat default respawn niet moet plaatsvinden. Je kunt dan bijvoorbeeld in een script doen:

```
local canceled = TriggerEvent("onPlayerDeath", playerId)
if canceled then
  print("Death event was cancelled, doe alternatieve verwerking.")
end
```

User Interface (NUI) integreren

Naast gameplay-logica wil je vaak een gebruikersinterface toevoegen, bijvoorbeeld een HUD, menu of telefoon. FiveM biedt hiervoor **NUI (New UI)**: een op **HTML/CSS/JS** gebaseerde UI die in het spel gerenderd wordt met behulp van Chromium Embedded Framework [43]. Je kunt hiermee complexe interfaces bouwen (inclusief frameworks zoals React/Angular) die bovenop het spel getoond worden of zelfs op textures in-world. We onderscheiden twee hoofdvormen: - **Fullscreen NUI:** Een volledige overlay bovenop de game (bijv. een schermvullend menu of HUD). Dit is de meestgebruikte vorm. - **Direct-rendered UI (DUI):** Een UI die op een *runtime texture* wordt gerenderd, bijvoorbeeld om webcontent op een object in de spelwereld te tonen (denk aan een in-game televisie of billboard) [44]. DUI is geavanceerder gebruik; we zullen dit kort benoemen.

NUI draait in feite als een ingebedde browser. Je front-end code communiceert met de script-runtime via berichten (events en callbacks). Hieronder behandelen we het opzetten van een NUI pagina, communicatie van script -> UI (NUI messages) en UI -> script (NUI callbacks).

Fullscreen NUI instellen (resource manifest)

Om een NUI pagina aan je resource toe te voegen moet je in het **fxmanifest.lua** (resource manifest) aangeven welk HTML-bestand als UI dient. Dit doe je met de directive `ui_page`:

```
ui_page 'html/index.html'
```

Hiermee wijs je een HTML-bestand (in de resource bestanden) aan als de hoofdpagina voor NUI [45]. Vergeet niet dit HTML-bestand (en eventuele JS/CSS bestanden die het gebruikt) ook op te nemen in de `files { ... }` sectie van het manifest, zodat clients het downloaden [46]. Bijvoorbeeld:

```
files {
  'html/index.html',
  'html/style.css',
  'html/app.js'
}
ui_page 'html/index.html'
```

Je kunt in plaats van een lokaal bestand ook een externe URL opgeven als ui_page (bijv. voor ontwikkeldoeleinden) [47], maar doorgaans bundel je de UI met de resource.

Zodra `ui_page` is ingesteld en de resource start, wordt de NUI pagina geladen (maar nog niet per se zichtbaar). Standaard heeft de UI geen focus en is niet interactief totdat je dat aangeeft.

NUI Focus: Met de native `SetNuiFocus(bool focus, bool cursor)` kun je de muis- en toetsenbordfocus op jouw NUI zetten [48]. Bijvoorbeeld: `SetNuiFocus(true, true)` geeft de speler een muiscursor en richt alle input op de UI (game-controls worden dan genegeerd). Je gebruikt dit wanneer je een UI wilt openen of sluiten. Denk eraan om focus ook weer uit te zetten (`SetNuiFocus(false, false)`) als de UI gesloten wordt, anders blijft de speler "vast" met een muiscursor.

Ontwikkelaarstools: Omdat NUI een echte browser gebruikt, kun je de Chrome DevTools gebruiken om je UI te debuggen. Start FiveM en voer `nui_devTools` in de F8-console in, of open in je externe browser de URL http://localhost:13172/ zodra de UI geladen is [49]. Hier kun je de HTML inspecteren, console logs bekijken, etc., net als bij normale webontwikkeling.

NUI berichten (script -> UI communicatie)

Vanuit je script (clientside Lua/JS) kun je berichten sturen naar de UI. FiveM biedt hiervoor de native `SEND_NUI_MESSAGE` (C function) en in Lua een handigere wrapper `SendNUIMessage(table)` die een Lua-table als JSON naar de UI stuurt [50]. In JavaScript gebruik je `SendNuiMessage(jsonString)` op soortgelijke wijze [51].

Meestal stuur je een JSON-payload waarin je bv. een *type* of *actie* aangeeft en eventuele data. De UI (frontend JS) kan dit bericht opvangen met de standaard `window.postMessage` API. Een voorbeeld uit de docs:

```
-- Lua: een bericht sturen naar de NUI pagina
SendNUIMessage({
    type = 'open',
    text = 'Hallo wereld'
})
```

```
// JS: equivalent (hier moet je zelf JSON string maken)
SendNuiMessage(JSON.stringify({
    type: 'open',
    text: 'Hallo wereld'
}));
```

Aan UI-zijde (bijvoorbeeld in `app.js` van je UI) kun je dit zo opvangen:

```
window.addEventListener('message', (event) => {
  const data = event.data;
  if (data.type === 'open') {
    // Doe iets in de UI, bv. open menu en toon tekst
    document.getElementById('msg').innerText = data.text;
  }
});
```

De `event.data` bevat de JSON die je stuurde (automatisch geparsed). In bovenstaand voorbeeld checken we of `type === 'open'` en voeren dan een UI-actie uit [52].

Met dit mechanisme kun je vanuit je script de UI aansturen: menus openen/sluiten, informatie updaten, etc. Dit werkt realtime en asynchroon.

Opmerking: De NUI message wordt alleen ontvangen door de UI van de *huidige resource*. Als je meerdere resources met NUI hebt, is er een focus-stack waarbij meestal maar één resource focus heeft [53]. De protocollink `https://cfx-nui-[resourceName]/...` wordt gebruikt door FiveM intern; referenties naar bestanden in je HTML (scripts, afbeeldingen) moeten dit schema gebruiken als je naar resource-bestanden verwijst [54]. Bijvoorbeeld `<script src="https://cfx-nui-mijnresource/app.js">`.

NUI callbacks (UI -> script communicatie)

Interactie van de UI terug naar je script (bijvoorbeeld de speler klikt een knop in de HTML en je wilt dat de scriptkant daarop reageert) verloopt via **NUI callbacks**. Dit werkt door een HTTP-achtige *POST* vanuit de UI naar de script runtime. FiveM voorziet daarvoor een eenvoudige API.

Lua (RegisterNUICallback): In Lua kun je met `RegisterNUICallback("naam", function(data, cb) ... end)` een callback registreren [55] [56]. `naam` is een string die de actie identificeert. In je UI doe je dan een JavaScript `fetch` call naar een URL `https://<resourceName>/<naam>` om de callback te triggeren. FiveM koppelt dat aan je geregistreerde functie. Bijvoorbeeld, stel je registreert:

```
RegisterNUICallback('getItemInfo', function(data, cb)
    local itemId = data.itemId  -- data is direct geparsed JSON van UI
    if not itemCache[itemId] then
        cb({ error = 'Item bestaat niet' })
    else
        cb(itemCache[itemId])  -- stuur informatie terug
    end
end)
```

Wanneer de UI nu een POST request doet naar dit callbackkanaal, wordt je Lua-functie aangeroepen. Een voorbeeld van de UI-kant (JavaScript in de browser context):

```
fetch(`https://${GetParentResourceName()}/getItemInfo`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json; charset=UTF-8' },
    body: JSON.stringify({ itemId: 'weapon_1' })
}).then(resp => resp.json()).then(data => {
    if (data.error) {
      console.error('Error:', data.error);
    } else {
      console.log('Item info ontvangen:', data);
    }
});
```

Hierbij is `GetParentResourceName()` een globale die FiveM beschikbaar maakt binnen de UI; het geeft de resource-naam zodat de fetch naar de juiste resource gaat [57]. In de Lua-callback functie ontvang je het JSON-body als table `data` (FiveM parse dat automatisch) en een `cb` functie. **Belangrijk:** je móet altijd `cb(...)` aanroepen om een response terug te sturen naar de UI, ook al is het een lege table (`cb({})`) of enkel `{ ok = true }`. Doe je dit niet, blijft de fetch *hangen* (Promise blijft onvervuld) [58]. In het voorbeeld hierboven stuurt de script terug ofwel een error of de gevraagde item-info, waarna de UI dat in `.then` verwerkt.

JavaScript/C#: FiveM NUI callbacks zijn oorspronkelijk ontworpen voor Lua, maar in JS/C# kun je ze ook gebruiken met een kleine omweg [59]. In JS registreer je eerst het callback type met `RegisterNuiCallbackType('naam')` en luister je vervolgens naar een speciaal event `__cfx_nui:naam`. Voorbeeld in JS:

```
RegisterNuiCallbackType('getItemInfo');
on('__cfx_nui:getItemInfo', (data, cb) => {
  const itemId = data.itemId;
  if (!itemCache[itemId]) {
    cb({ error: 'No such item!' });
  } else {
    cb(itemCache[itemId]);
  }
});
```

Dit doet in essentie hetzelfde als het Lua voorbeeld hierboven [60] [61]. In C# is het vergelijkbaar: `RegisterNuiCallbackType("naam")` en een event handler toevoegen op `__cfx_nui:naam`, waarbij je gebruik maakt van een `CallbackDelegate` voor de `cb` (zie docs voor de exacte syntax) [62] [63].

Kort samengevat: 1. Registreer op scriptzijde de NUI callback (Lua direct met `RegisterNUICallback`, JS met bovengenoemde constructie). 2. Roep vanuit de UI een `fetch('https://resourceNaam/actie', {...})` aan om data te sturen. 3. Handelaar wordt uitgevoerd; verwerk `data` en roep de meegegeven `cb` functie om een antwoord terug te geven. 4. Verwerk in de UI het antwoord (Promise `.then(response)`).

Met NUI callbacks kun je dus bijvoorbeeld formulieren verwerken, knoppen laten leiden tot script-acties (bijv. een voertuig spawnen via een UI menu, door in de callback server events of natives aan te roepen), enzovoort.

Direct-Rendered UI (DUI)

Naast fullscreen overlays kun je ook webcontent renderen op texturen in de wereld. Dit valt buiten de meeste standaard UIs, maar FiveM biedt natives als `CREATE_DUI(url, width, height)` om een onzichtbare browser te maken en `CREATE_RUNTIME_TEXTURE_FROM_DUI_HANDLE` om die content op een texture te tekenen [44]. Dit kan gebruikt worden voor dingen als **in-game TV-schermen, beveiligingscamera feeds of interactieve billboards**. DUI vereist clientside scripting en gebruik van teken-natives zoals `DrawSprite` [64] of gebruik van *Scaleform*. De handleiding van FiveM noemt dat je hier creatieve toepassingen mee kunt bouwen, zoals bioscoopschermen of asynchronous overlays in de gamewereld [65].

Omdat DUI meer een specialistische techniek is, gaan we er niet diep op in. De kern is dat je een DUI aanmaakt met een URL (die kan wijzen naar een externe site of een lokale UI pagina), en vervolgens de output daarvan op een object of HUD-element tekent. Bijvoorbeeld kun je een live website op een in-game tv laten zien. Zie de FiveM docs voor meer details mocht je dit nodig hebben.

Exports: functies delen tussen resources

Grote FiveM servers bestaan vaak uit meerdere resources die elkaar moeten kunnen aanspreken. **Exports** zijn de manier om functies publiek beschikbaar te maken over resources heen. Wanneer je een functie exporteert, kunnen andere resources deze aanroepen via het `exports` object.

Export definiëren (Lua): In Lua kun je exports op twee manieren definiëren: - Via het fxmanifest: met `export 'functieNaam'` of meerdere in een `exports { ... }` lijst [66]. Dit maakt de globale functie met die naam beschikbaar als export. Zorg dat de daadwerkelijke functie in je script in de *global scope* bestaat (of aan `_G` is toegevoegd). - Via code (FXv2): FiveM heeft een nieuwe manier geïntroduceerd waarbij je in Lua een functie direct kunt exporteren door deze aan te melden. Dit werkt vergelijkbaar met JS: `exports('functieNaam', function(...) ... end)`. Intern doet dit iets soortgelijks als manifest export, maar is dynamischer. (Deze methode is relatief nieuw; de documentatie raadt het aan boven manifest-exports [67] [68].)

In het manifest voorbeeld:

```
exports {
    'setWidget',
    'getWidget'
}
local lastWidget
function setWidget(val) lastWidget = val end
function getWidget() return lastWidget end
```

Hiermee exporteren we `setWidget` en `getWidget` functies aan andere resources [66].

Export definiëren (JavaScript): In een JavaScript script kun je de functie `exports(name, function)` aanroepen [69]. Bijvoorbeeld:

```
// In resource X, file script.js
exports("sayHello", (playerName) => {
  console.log("Hello " + playerName);
});
```

Dit registreert `sayHello` als export. Je kunt ook direct bestaande functies exporteren, bijv. `exports("dropPlayer", DropPlayer)` zoals in de docs [70]. Let op dat in JS **clientscript** exports alleen door andere clientscript-resources aangeroepen kunnen worden, en serverscript-exports alleen door andere serverscripts.

Exports aanroepen (consumeren): - In **Lua** roep je een export van resource *X* aan met `exports.x_naam:functieNaam(args)`. Bijvoorbeeld, stel resource `mijnresource` heeft een export `getWidget`, dan kun je in een andere resource doen: `local val =`

`exports.mijnresource:getWidget()` [71]. (De dubbele punt `:` is hier syntactic sugar zodat de export-functie als method wordt gezien – FiveM zorgt ervoor dat `_G` context goed staat. Je kunt ook `exports.mijnresource.getWidget()` proberen met een punt, maar de conventie is `:`.) - In **JavaScript** gebruik je het globale `exports` object. Bijvoorbeeld: `exports.mijnresource.getWidget()` of met bracket notatie `exports['mijnresource']['getWidget']()` [72]. Beide zijn gelijkwaardig zolang je juiste resource- en functienamen gebruikt. De FiveM JS runtime documentatie bevestigt: *"Exports can be called using* `exports.resourceName.exportName` *(or bracket syntax)"* [73].

Je kunt ook serverexports definieren (in fxmanifest met `server_export` voor functies die in serverscripts zitten) [74]. Het aanroepen werkt hetzelfde concept, maar dan tussen serverscripts onderling.

Praktisch voorbeeld: Stel resource `scoreboard` heeft een serverside functie `AddScore(player, points)` die je wilt hergebruiken. Je zou in *scoreboard* fxmanifest zetten `server_export 'AddScore'` en zorgen dat de functie globaal bestaat in scoreboard's server script. Dan kan een andere resource in zijn serverscript doen: `exports.scoreboard:AddScore(source, 100)` om de speler 100 punten te geven.

Een voorbeeld in JavaScript: de standaard resource **spawnmanager** exporteert functies om een speler te spawnen. In JS kun je bijv. doen:

```
// Aanroepen van exports van spawnmanager (client-side)
exports.spawnmanager.setAutoSpawnCallback(() => {
  exports.spawnmanager.spawnPlayer(...);
});
exports.spawnmanager.setAutoSpawn(true);
exports.spawnmanager.forceRespawn();
```

zoals de docs illustreren [75] [76]. Hier zie je hoe een resource gebruik maakt van functies uit *spawnmanager* via het `exports` object.

Samenvatting exports: - Definieer exports in de resource die de functionaliteit aanbiedt (manifest of via code). - Andere resources kunnen deze functies aanroepen via het globale exports object. - Hiermee kun je modules maken: ene resource als library, andere als gebruiker. - Zorg ervoor dat afhankelijkheden kloppen (je kunt eventueel in fxmanifest `dependency 'andereResource'` zetten om te garanderen dat die gestart is voordat jouw resource start, als je direct bij start exports wilt gebruiken).

Natives en referenties

FiveM geeft scripters toegang tot bijna alle in-game functies van GTA V via **natives** (ingebouwde gamefuncties). Deze zijn beschikbaar als globale functies in Lua/JS, vaak met namen gelijk aan de originele native. Bijvoorbeeld, de GTA native `PLAYER::PLAYER_PED_ID` gebruik je in Lua als `PlayerPedId()` en in JS idem `PlayerPedId()` [77]. Er zijn duizenden natives (voor alles van voertuigdeuren openen tot gameplay statistieken). De lijst is te vinden op de **FiveM Native Reference** pagina. De link docs.fivem.net/natives/ biedt een zoek- en bladerfunctie. Je kunt hier op naam of hash zoeken naar functies. De native doc-pagina's geven parameters en soms beschrijving/voorbeeld.

Als je bijvoorbeeld een speler wilt teleporteren, zoek je de native voor *SetEntityCoords*. Je vindt dan dat `SetEntityCoords(entity, x, y, z, ...)` beschikbaar is. Je kunt die direct in je script aanroepen met het gewenste entity (bijv. `SetEntityCoords(PlayerPedId(), 0.0,0.0,100.0,false,false,false,true)` om de eigen speler omhoog te zetten). Gebruik natives zorgvuldig en raadpleeg de documentatie voor de juiste parameters.

Naast natives zijn de **event lijsten** die we eerder noemden ook een handige referentie. De Client events lijst en Server events lijst sommen gebeurtenissen op als *playerConnecting*, *onResourceStart*, *gameEventTriggered* etc. inclusief uitleg van parameters. Gebruik deze lijsten om te weten welke events je kunt afvangen om gameplay-elementen te koppelen.

Tot slot zijn er nog andere referenties zoals de **Resource manifest** documentatie [78] (voor alle manifest opties) en guides voor specifieke onderwerpen (bijv. **Convars**, **commands**). FiveM's docs en community-forums zijn waardevolle bronnen als je verder gevorderde dingen wilt doen.

Conclusie

Met de bovenstaande bouwstenen kun je een FiveM-script opzetten dat zowel op de server als client functioneert, een gebruikersinterface heeft en netjes data en functionaliteit deelt: - Gebruik **state bags** voor gesynchroniseerde data-toestand van entities, spelers of globale flags (scheelt custom sync-code) [5] [8]. - Communiceer tussen client <-> server via **events**: registreer event handlers en trigger events over en weer [18] [30]. Beheer de flow met CancelEvent waar nodig [34]. - Integreer **NUI** voor UI: stel een `ui_page` in, gebruik `SendNUIMessage` voor script-naar-UI en `RegisterNUICallback` / `fetch` voor UI-naar-script communicatie [79] [55]. Geef focus met `SetNuiFocus` wanneer de speler met de UI moet interacteren. - **Exports** laten je je script modulair maken door functies aan andere resources aan te bieden of zelf utilities van anderen te benutten [80]. - Raadpleeg de FiveM documentatie voor specifieke natives en event details – vrijwel alles wat in GTA V kan, kun je via een native of event aanroepen.

Veel succes met het bouwen van je script! Met deze handleiding heb je een naslag die je ook kunt voeden aan AI-tools zodat die je kunnen helpen bij het genereren van code of het beantwoorden van vragen over FiveM scripting. Happy coding! [1] [18]

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] State bags - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/networking/state-bags/

[14] [15] [16] [17] [18] [19] [20] [21] Listening for events - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/working-with-events/listening-for-events/

[22] Client events - Cfx.re Docs
https://docs.fivem.net/docs/scripting-reference/events/client-events/

[23] Server events - Cfx.re Docs
https://docs.fivem.net/docs/scripting-reference/events/server-events/

[24] [25] [26] [27] [28] [29] [30] [31] [32] [33] Triggering events - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/working-with-events/triggering-events/

[34] [35] [36] [37] [38] [39] [40] [41] [42] Event Cancelation - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/working-with-events/event-cancelation/

[43] User interfaces with NUI - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/nui-development/

[44] [64] [65] Direct-rendered UI - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/nui-development/dui/

[45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [79] Fullscreen NUI - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/nui-development/full-screen-nui/

[55] [56] [57] [58] [59] [60] [61] [62] [63] NUI callbacks - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/nui-development/nui-callbacks/

[66] [67] [68] [71] [74] [78] Resource manifest - Cfx.re Docs
https://docs.fivem.net/docs/scripting-reference/resource-manifest/resource-manifest/

[69] [70] Exports - Cfx.re Docs
https://docs.fivem.net/docs/scripting-reference/runtimes/javascript/functions/exports/

[72] [73] [75] [76] [77] [80] Scripting in JavaScript - Cfx.re Docs
https://docs.fivem.net/docs/scripting-manual/runtimes/javascript/