

Debug local des Server Scripts BrainCloud

Pourquoi ce système existe

Le dossier `node.js/backend/braincloud/debugging/` recrée localement une partie de l'environnement d'exécution BrainCloud. C'est l'outil principal pour exécuter un Cloud Script du dépôt, suivre ses `include()`, inspecter ses variables et utiliser de vrais documents sans devoir déclencher toute la simulation depuis Unity.

Fichiers clés

Fichier	Rôle
<code>debugging/server_scripts_debugging_base.js</code>	Bootstrap du runtime local : crée <code>include()</code> , initialise MongoDB et le client BrainCloud, charge le bridge puis le script de debug.
<code>debugging/server_scripts_debugging.js</code>	Scratchpad d'exécution : définir <code>data</code> , appeler un script, tester un module ou écrire une fonction de diagnostic.
<code>debugging/utils/bridge.js</code>	Émule l'objet global <code>bridge</code> disponible dans les Cloud Scripts BrainCloud.
<code>debugging/utils/custom_entity_service_proxy.js</code>	Émule l'accès aux Custom Entities en utilisant MongoDB local et BrainCloud en fallback.
<code>utils/MongoDBClient.js</code>	Connexion au MongoDB local <code>localhost:27017</code> , base <code>uniksim</code> .
<code>utils/BraincloudClient.js</code>	Accès à l'environnement BrainCloud choisi lorsque le document n'existe pas localement.

Bootstrap

```
sequenceDiagram
participant N as Node
participant BASE as server_scripts_debugging_base
participant M as MongoDB local
participant BC as BrainCloudClient
participant BR as bridge local
participant DBG as server_scripts_debugging
N->>BASE: exécution
BASE->>BASE: crée global.include()
BASE->>M: authenticate()
BASE->>BC: crée le client
BASE->>BR: include debugging/utils/bridge.js
BASE->>DBG: include server_scripts_debugging.js
DBG->>DBG: exécute la fonction de test choisie
```

Comment include() fonctionne

`server_scripts_debugging_base.js` expose un `global.include(path)`. Pour un Server Script, le chemin est résolu vers `/server_scripts<path>.js`. Le fichier est lu avec `fs.readFileSync()` puis exécuté dans le même contexte Node avec `vm.runInThisContext()`.

La liste `includedScripts` évite de réinclure deux fois le même script lorsque `includeOnce` est actif. Cela reproduit suffisamment le modèle BrainCloud pour suivre la majorité des dépendances du dépôt.

Lancer le debugger

Depuis le dossier `node.js`, le point d'entrée est :

```
node backend/braincloud/debugging/server_scripts_debugging_base.js
```

Le bootstrap change lui-même le working directory vers `backend/braincloud`.

Préparer server_scripts_debugging.js

Le fichier est volontairement un scratchpad. La méthode la plus simple est :

1. choisir `serverInstanceType` (par défaut `dev`);
2. définir l'objet global `data` exactement comme BrainCloud le recevrait;
3. appeler `include("/_command/...")` ou un script game-specific;
4. inspecter `response`, les variables ou les documents produits.

```
data = {  
  commandUid: "get_module_grade",  
  commandData: {  
    teamUid: "...",  
    actModuleUid: "...",  
    moduleType: "practice",  
    actUid: "..."  
  }  
};  
  
include("/acc1/command/acc1_custom_command");
```

testScript()

Le helper `testScript({ scriptName, scriptData })` copie `scriptData` dans `data` puis appelle directement `include("/") + scriptName)`. C'est utile lorsqu'on veut reproduire une requête Cloud Script presque telle qu'elle apparaît dans les logs.

Tester le moteur By Module sans document existant

`testGeneralFunctionalities(moduleUid)` est particulièrement important. Il reconstruit le pipeline complet :

```
flowchart LR
  SET[Static settings] --> CREATE[Create Module Type Data]
  CREATE --> AK[Answer Key]
  AK --> GRADE[Grade Module]
  GRADE --> RE[Reassemble Answers]
  RE --> OUT[Scores / Errors]
```

Il crée un seed, génère les données, prend les bonnes décisions, grade le résultat puis réassemble les scores. `testAllModuleGeneralFunctionalities()` applique ce flux à tous les modules configurés et permet de trouver rapidement une régression globale.

Accès aux documents : MongoDB d'abord, BrainCloud ensuite

`CustomEntityServiceProxy` cherche d'abord le document dans le MongoDB local. Si rien n'est trouvé, il authentifie le `BraincloudClient` dans `serverInstanceType` et récupère le document distant. Les lectures suivantes peuvent donc être déboguées localement tout en utilisant un vrai document de Dev.

```
flowchart TD
  GET[getDocumentData] --> LOCAL{Document dans MongoDB local ?}
  LOCAL -->|oui| USE[Utiliser document local]
  LOCAL -->|non| BC[Lire depuis BrainCloud]
  BC --> USE
  USE --> SCRIPT[Exécuter le Server Script]
  SCRIPT --> SAVE[Écritures via proxy local]
```

Limites du bridge local

Le bridge n'émule pas tout BrainCloud. Plusieurs proxies retournent volontairement `501` ou ne sont pas implémentés :

- certaines opérations User/Player State;
- scheduled scripts;
- présence réelle;
- messaging réel (le contenu est surtout loggé);
- HTTP client;

- `getProfileId()` et certains contextes utilisateur.

Si un Server Script dépend fortement d'une de ces APIs, le debug local peut nécessiter d'étendre le proxy ou de valider la dernière étape directement sur BrainCloud.

Cas d'usage recommandés

- reproduire un `get_module_grade` avec un vrai `teamUid`;
- inspecter une génération de Module Type Data;
- tester l'answer key d'un module précis;
- chercher un seed qui casse une balance;
- tester tous les modules Accounting en boucle;
- lire un document via `/_command/get_document`;
- reproduire une custom command sans passer par Unity.

Important avant de modifier

Ne pas laisser un appel de test accidentel actif en haut de `server_scripts_debugging.js`. Le fichier exécute directement le code présent lorsqu'il est inclus par le bootstrap. Garder une seule fonction de test explicitement appelée pendant la session de debug.

Revision #1

Created 2026-09-18 19:12:54 UTC by Mathis

Updated 2026-09-18 19:12:55 UTC by Mathis