



Composant Serveur API REST pour WinDev®

Léger.

Rapide.

Gratuit & Open Source.

Sans serveur IIS/Apache/...

Sans licence Serveur WebDev®.

Pas de SAAS, pas d'abonnement.

Windows 32/64 & Linux

[Présentation](#)[NEW : LightREST V3](#)[La Documentation](#)[Téléchargements](#)[Un coup de main ?](#)

Être paranoïaque !

Développer une API REST ne consiste pas seulement à exposer des routes HTTP qui renvoient des données. Dès qu'une API devient accessible sur un réseau — et plus encore sur Internet — elle devient immédiatement une surface d'attaque potentielle, un point d'entrée pour des comportements imprévus, ou un vecteur de surcharge involontaire.

Avec **LightREST**, il est très simple de créer rapidement un serveur REST performant. Mais cette simplicité ne doit jamais faire oublier que l'on développe un **composant serveur concurrent**, capable de traiter plusieurs requêtes simultanément, parfois provenant de clients inconnus ou malveillants. Ce qui est valable quelle que soit la technologie ou le protocole de communication utilisés.

Dans ce contexte, la prudence n'est pas une option. Un développeur d'API doit adopter une approche presque **paranoïaque** : considérer que toute entrée peut être invalide, que toute ressource peut être sollicitée en parallèle, et que tout comportement inattendu finira un jour par se produire en production.

Une API publique doit être conçue pour résister à :

- des appels concurrents en grand nombre
- des paramètres erronés ou malformés
- des tentatives d'énumération ou d'exploration des données
- des erreurs applicatives imprévues
- des usages non anticipés par les développeurs

C'est précisément pour cette raison qu'il est essentiel d'appliquer un ensemble de **bonnes pratiques** lors du développement d'un serveur LightREST. Ces pratiques permettent de construire des API plus robustes, plus sûres et plus maintenables dans le temps.

Les recommandations présentées ci-dessous ne sont pas de simples conseils de style. Elles correspondent aux principes fondamentaux de conception des services REST modernes : gestion correcte de la concurrence, isolation des requêtes, validation systématique des entrées, maîtrise des ressources et respect des conventions d'architecture.

Adopter ces pratiques dès le départ permet d'éviter des problèmes difficiles à diagnostiquer en production et garantit que votre serveur LightREST restera stable, sécurisé et performant, même lorsque la charge ou l'exposition publique augmentera.

LightREST est un serveur REST concurrent. Une même route peut être exécutée simultanément par plusieurs requêtes.

Les principes fondamentaux sont :

- éviter les variables globales non protégées
- éviter les connexions base de données partagées entre toutes les requêtes
- utiliser les hooks pour préparer et libérer les ressources
- valider systématiquement les entrées
- produire des réponses HTTP claires et cohérentes
- concevoir des routes rapides, isolées et stateless

Pas de connexion base de données globale

Une connexion base de données unique, partagée entre toutes les routes, est une mauvaise pratique en environnement serveur concurrent.

Cela peut provoquer :

- conflits d'accès
- blocages
- corruption d'état
- comportements aléatoires selon la charge

Même si le handler LightREST est exécuté dans un contexte Hyperfile dédiée, la bonne pratique consiste à ouvrir la connexion pour chaque requête à y affecter les tables de l'analyse, généralement via un hook exécuté avant le handler, puis à la fermer dans un hook de fin. Les moteurs de base de données gérant des pools de connexions disponibles, le processus est très rapide.

Exemple :

Copier

Copier

```
PROCÉDURE HookBeforeMethod(pEventInfo  
lrHook:EventInfo) : lrHook:EventReturn
```

```
cnxDB est une Connexion
```

```
//définir les paramètres de la  
connexion.....
```

```
SI PAS HOuvreConnexion(cnxDB ) ALORS  
    pEventInfo:Response:Status =  
    lrResponse::StatusInternalServerError  
    pEventInfo:Response:ContentType =  
    lrReponse::ContentTXT  
    pEventInfo:Response:Body =  
    Herreurinfo()  
    RENVOYER lrHook:EVE_ABORT  
FIN
```

```
pRequest:DatabaseConnexion["DB"] =  
cnxDB  
hChangeConnexion("*", cnxDB )
```

```
RENVoyer lrHook:EVE_OK
```

Puis dans le handler :

Copier

Copier

```
PROCÉDURE HookAfterMethod(pEventInfo  
  lrHook:EventInfo) : lrHook:EventReturn
```

```
SI
```

```
pRequest:DatabaseConnexion["DB"]..exis  
te ALORS
```

```
hFermeConnexion(pRequest:DatabaseConne  
xion["DB"])
```

```
FIN
```

```
RENVoyer lrHook:EVE_OK
```

Éviter les variables globales dans les handlers

Dans LightREST, plusieurs requêtes peuvent accéder au même moment au même code et aux mêmes données.

Une variable globale non protégée peut donc devenir une source :

- d'incohérences
- de corruption mémoire
- de bugs intermittents
- de crashes difficiles à reproduire

Exemple à éviter :

Copier

```
gCompteur++
```

Copier

Si un état partagé est réellement nécessaire, il doit être protégé par un mécanisme de synchronisation ou stocké dans une structure adaptée.

Partager proprement des données globales avec LightREST

LightREST fournit les membres **CustomData** à plusieurs niveaux :

- serveur
- route
- requête

C'est la méthode recommandée pour associer des données de contexte sans multiplier les variables globales dispersées.

Exemple :

Copier

```
oServer:CustomData["Config"] =  
maConfiguration
```

Copier

Puis dans le handler :

Copier

Copier

```
cfg est un Config =  
poRequest:ServerCustomData["Config"]
```

Cette approche rend le code plus clair, mieux structuré et plus sûr.

Limiter l'utilisation des objets ou ressources partagés entre plusieurs handlers

Le plus sûr est d'éviter ce cas de figure. Si toutefois c'est incontournable, il faut que leur accès soit maîtrisé.

Toute ressource partagée doit être :

- thread-safe par conception
- ou protégée par une synchronisation explicite

Par exemple, si plusieurs handlers modifient un même compteur ou une même structure, il faut sécuriser l'accès par section critique (<https://doc.pcsoft.fr/fr-FR/?1000021292>), sémaphore (<https://doc.pcsoft.fr/fr-FR/?3077013>) ou autre mécanisme approprié.

Copier

Copier

```
SectionCritiqueDébut(gsLock)  
gCompteur++  
SectionCritiqueFin(gsLock)
```

Ne pas exposer les identifiants techniques dans les routes

Exposer directement un identifiant base de données dans une URL facilite :

- l'énumération des ressources
- l'exploration des données
- certaines tentatives d'accès non autorisé

Exemple à éviter :

Copier

```
GET /clients/1287
```

Copier

Il est préférable d'exposer un identifiant chiffré ou obfusqué via les fonctions :

- CipherID()
- DecipherID()

NB : L'algorithme CipherID ne produira pas le même identifiant chiffré pour 2 valeurs identiques sur 2 tables différentes.

Exemple :

Copier

[Copier](#)

```
GET /clients/03980608-5abe9581-7fee65e9-194ae106
```

Puis dans le handler :

[Copier](#)

[Copier](#)

```
nID est entier =  
DecipherID(pRequest:GetUrlValue("id"))  
  
//Ici nID = 1287
```

Rester stateless

Oui. Une API REST doit rester aussi stateless que possible.

Une requête ne doit pas dépendre d'un état conservé implicitement dans le serveur ou dans un handler précédent.

Il faut éviter par exemple :

- de mémoriser un utilisateur courant dans une variable globale
- de supposer qu'une requête précédente a déjà préparé un contexte
- de conserver un état métier caché entre deux appels

Chaque requête doit porter elle-même les informations nécessaires via :

- son token
- ses headers
- ses paramètres
- ou son contexte de requête

Éviter les variables globales dans les handlers

Un handler REST doit répondre rapidement.

Un traitement long monopolise inutilement les ressources du serveur et dégrade la capacité à traiter d'autres requêtes.

A partir de la version 3.3, LightREST propose un système de Workers qui permettent d'exécution des traitements longs de façon asynchrone et de libérer immédiatement le handler en cours.

Il est préférable de :

- déléguer les travaux lourds à une file de traitement
- lancer un traitement asynchrone
- renvoyer immédiatement un accusé de prise en compte

Exemple de réponse :

Copier

```
{  
  "status": "accepted",  
  "jobId": "12345"  
}
```

Copier

Centraliser l'authentification

Centraliser l'authentification avec un Hook

EVE_BEFORE_HANDLER évite la duplication de code dans chaque route et réduit le risque d'oublis ou d'incohérences.

La fonction dédiée permet d'appliquer une logique homogène sur l'ensemble du serveur ou sur un groupe de routes.

Copier

```
SetAuthenticationCheckFunction()
```

Copier

Une authentification centralisée améliore :

- la lisibilité
- la maintenabilité
- la sécurité globale

Centraliser les logs

Une API en production doit permettre de comprendre ce qu'il se passe en cas d'erreur, de lenteur ou d'usage anormal.

Il faut journaliser au minimum :

- les erreurs
- les avertissements
- les échecs d'authentification
- les appels reçus
- la durée des traitements

Hooks utiles :

- EVE_ERROR
 - EVE_WARNING
 - EVE_AUTH_FAILED
 - EVE_RECEIVED
-

Limiter le volume des réponses

Retourner des volumes massifs de données ralentit :

- le serveur
- le réseau
- le client

Il est préférable d'utiliser :

- la pagination
- des filtres
- une limite explicite

Copier

```
GET /clients?page=1&limit=50
```

Copier

Cette pratique améliore à la fois les performances et l'expérience d'intégration côté client.

Valider systématiquement les entrées

Un bon développeur d'API se doit d'être PARANOÏAQUE ! Une API ne doit jamais faire confiance aux données reçues.

Il faut valider :

- les paramètres d'URL
- les payloads JSON
- les headers
- les types et formats attendus

Copier

Copier

```
sID = pRequest:GetUrlValue("id")  
  
SI sID = "" ALORS  
    oResponse:Status =  
    lResponse::StatusBadRequest  
FIN
```

Ne jamais exposer le détail des erreurs techniques

C'est **très important pour une API publique**.

Une API ne doit jamais renvoyer directement des erreurs internes du serveur ou de la base de données, qui peuvent contenir des extraits de code, le contenu de variables, des requêtes SQL ou des données.

Exemples dangereux :

Copier

[Copier](#)

```
SQL error near column CLIENT_ID
File not found: /var/data/config.ini
Stack trace...
```

Ces informations peuvent révéler :

- la structure interne de l'application
- la base de données utilisée
- l'architecture du serveur

La bonne pratique consiste à renvoyer des messages d'erreur contrôlés.

Exemple :

Copier

[Copier](#)

```
{
  "error": "internal_error",
  "message": "An unexpected error
occurred"
}
```

Les détails techniques doivent être enregistrés uniquement dans les logs serveur.

NB : Le niveau de détail des erreurs gérées automatiquement par LightREST est réglable par la propriété `IrServer:WindevErrorDetail`.

Ajouter un endpoint de santé (health check)

Très utilisé en production.

Un serveur API devrait exposer une route simple permettant de vérifier son état.

Exemple :

Copier

Copier

```
GET /health
```

Réponse :

Copier

Copier

```
{
  "status": "ok"
}
```

Ce type de route est utilisé par :

- les systèmes de supervision
- les load balancers
- les orchestrateurs
- les outils de monitoring

Respecter les conventions REST

Action	Route
liste	GET /clients
détail	GET /clients/{id}
création	POST /clients
modification	PUT /clients/{id}
suppression	DELETE /clients/{id}

Une API cohérente réduit la documentation nécessaire et facilite l'adoption. LightREST V3 marque une évolution majeure : **plus ouvert, plus flexible, plus sécurisé, et plus robuste en production.**

Cette page récapitule les nouveautés principales et surtout **ce qu'elles apportent concrètement** au quotidien.

© CODE LINE 2018-2026

