Epsitec.WebCore.Server

Le projet Epsitec.WebCore.Server est la couche responsable de la partie serveur de l’application, que le client web accède via des requêtes https.

# Architecture

L’architecture générale du serveur web est décrite par le schéma ci-dessous :

Web Client

Web Server

Nginx server

Epsitec.WebCore.Server.dll

Owin server

File system

Nancy server

https://my.server.url/...

https://my.server.url/...

https://my.server.url/owin/...

https://my.server.urlproxy/...

CoreWorkerPool

Le point d’entrée du serveur est un serveur web Nginx. Toutes les requêtes http du client passent par lui. Suivant leur forme, il va les traiter différemment :

* <https://my.server.url/proxy>/... : le traitement de la requête est délégué à un serveur Nancy intégré dans Epsitec.WebCore.Server.dll. Typiquement, il s’agit des requêtes qui retournent des données ou qui servent à effectuer des modifications sur les données.
* <https://my.server.url/owin>/... : le traitement de la requête est délégué à un serveur owin intégré dans Epsitec.WebCore.Server.dll. Il s’agit des requêtes que le client web fait pour le système de notifications temps réel.
* https://my.server.url/... : le traitement de la requête est effectué directement par le serveur Nginx. Dans ce cas, il s’agit simplement de servir un fichier statique au client, comme un fichier javascript ou une image par exemple. Toutes les requêtes qui ne tombent pas dans les deux cas précédents arrivent ici.

# Nginx[[1]](#footnote-1)

Le serveur Nginx est lancé automatiquement par Cresus.WebCore.Server lors du processus d’initialisation du serveur. De même, il est quitté automatiquement lors du processus de fermeture du serveur.

Le serveur Nginx reçoit toutes les requêtes https du client et sert tous les fichiers statiques. De plus, il sert de proxy à un serveur Nancy et à un serveur Owin qui s’occupent du traitement de toutes les requêtes qui ne retournent pas des fichiers statiques.

En plus de cela, il assure une redirection des requêtes http vers https, de manière à s’assurer que toutes les transmissions de données entre le client et le serveur sont sécurisées.

# Owin

Todo by Samuel ;-)

# Nancy[[2]](#footnote-2)

Le serveur basé sur Nancy est le plus gros morceau de cette partie serveur. C’est elle qui s’occupe du traitement de toutes les requêtes qui servent à obtenir des données ou à les modifier. Son architecture générale est décrite dans le schéma ci-dessous :

Http server

Nancy server

CoreWorkerPool

Nancy module

Nancy module

Nancy module

Nancy module

Le serveur http écoute toutes les requêtes effectuées sur un port de la machine. Ce sont les requêtes transmises par le serveur Nginx qui fait office de proxy avec le client. Il transmet la requête au serveur Nancy. Le serveur Nancy s’occupe de toute la gestion de l’url, pour la parser et déterminer le module à utiliser pour exécuter la requête. Dans la majorité des cas, ce module va accèder aux données ou vouloir faire un traitement dessus. Pour ce faire, il s’adresse au CoreWorkerPool qui va lui fournir un contexte qu’il va pouvoir utiliser.

# CoreWorkerPool

Le but de CoreWorkerPool est de fournir aux modules Nancy un context dans lequel ils peuvent bénéficier de toute l’infrastructure mise en place pour Cresus.Core. Son architecture générale est la suivante :

CoreWorkerPool

Blocking queue

CoreWorker

CoreApp

WorkerThread

Nancy module

CoreWorker

CoreApp

WorkerThread

CoreWorker

CoreApp

WorkerThread

CoreWorker

CoreApp

WorkerThread

L’élément principal de ce schéma est le CoreApp. Une instance de CoreApp contient toute l’infrastructure nécessaire au bon fonctionnement d’une application. C’est une sorte de contexte, dans lequel on une instance de CoreData, une connection à la base de données, tout un tas de services, etc. C’est ce contexte dont les modules ont besoin pour faire leur travail.

Une contrainte liée à CoreApp, est qu’une instance donnée de CoreApp doit toujours être utilisée par le même thread. Cela est dû à des problèmes d’architectures internes. C’est pour cela qu’on associe une instance de WorkerThread à un une instance de CoreApp. Cela permet de toujours utiliser le même thread pour le manipuler. On appelle cette paire CoreApp/WorkerThread un CoreWorker.

Puisqu’on veut autoriser le traitement de plusieurs requêtes en même temps, le CoreWorkerPool contient plusieurs instances de CoreWorker. Elles sont stockées dans une queue. A chaque fois qu’un module a besoin d’un CoreApp, le CoreWorkerPool prend un CoreWorker dans la queue et le lui donne le temps d’effectuer son travail. Une fois le travail terminé, le CoreWorker est réinitialisé et remis dans la queue. La queue est bloquante, de manière à faire attendre un module, dans le cas où il n’y aurait aucun CoreWorker de disponible pour l’instant.

1. http:// nginx.org [↑](#footnote-ref-1)
2. http:// nancyfx.org [↑](#footnote-ref-2)