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 générale du serveur

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 des entités 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 accéder 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 contexte dans lequel ils peuvent bénéficier de toute l’infrastructure mise en place pour Cresus.Core. Son architecture générale est la suivante :

CoreWorkerPool

CoreWorker

CoreWorker

CoreWorker

CoreWorker

Blocking bag

WorkerApp

WorkerThread

Nancy module

WorkerApp

WorkerThread

WorkerApp

WorkerThread

WorkerApp

WorkerThread

L’élément principal de ce schéma est le WorkerApp. Une instance de WorkerApp contient toute l’infrastructure nécessaire au bon fonctionnement d’une application. C’est une sorte de contexte, dans lequel on a une instance de CoreData, une connexion 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 WorkerApp, est qu’une instance donnée de WorkerApp 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 WorkerApp. Cela permet de toujours utiliser le même thread pour le manipuler. On appelle cette paire WorkerApp/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 collection synchronisés et bloquante. A chaque fois qu’un module a besoin d’un WorkerApp, le CoreWorkerPool prend un CoreWorker dans la collection et le lui donne le temps d’effectuer son travail. Une fois le travail terminé, le CoreWorker est réinitialisé et remis dans la collection. La collection est bloquante, de manière à faire attendre un module, dans le cas où il n’y aurait aucun CoreWorker de disponible pour l’instant.

# NancyHosting.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.NancyHosting s’occupent de la configuration de Nancy et fournissent de la fonctionnalité en rapport avec le serveur Nancy.

# NancyModules.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.NancyModules sont tous les modules utilisés par Nancy pour traiter les requêtes http envoyée par le client javascript et transmises par le serveur NGinx.

Chaque module a un but bien précis et sont API est documentée dans le code source.

# Layout.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Layout servent à sérialiser les définitions d’interface graphique depuis le serveur vers le client javascript. Il y a deux classes clés dans cette hiérarchie :

* Mason : le but de cette classe est d’instantier les instances d’EntityViewController et de les utiliser pour créer des instances de BrickWall. Il joue donc le même rôle que la class Bridge définie dans Cresus.Core.Library.
* Carpenter : le but de cette classe est de transformer les instances de BrickWall (et de Brick) en instances de EntityColumn (et de AbstractTile). Il joue donc le même rôle que la classe Bridge<T> définie dans Cresus.Core.Library.

Toutes les autres classes de la hiérarchie servent à représenter les éléments d’interface qui seront sérialisés vers le client javascript.

Une des difficultés avec ces classes est qu’il n’y a pas de spécification formelle de ce qu’il est possible de faire avec les Bricks et que leur interprétation exacte n’est pas non plus définie formellement.

# Core.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Core fournissent diverses fonctionnalités utiles pour le serveur.

En particulier, il y a le triplet WorkerApp, CoreWorker et CoreWorkerPool. Je ne vais pas le décrire plus ici, je l’ai déjà fait plus haut.

De plus, il y a la classe ItemCache dont héritent IdCache et AbstractLambdaCache<T> (dont héritent PropertyAccessorCache et AutoCreatorCache). L’implémentation de ces classes est un peu compliquée, probablement que si c’était à refaire, je les referai un peu différemment, mais dans le fond, leur fonctionnalité est relativement simple :

* Le but d’IdCache est de fournir un mapping biridectionnel entre des instances d’un type donné et un id autoincrémenté. De cette manière on peut donner une référence sur un de ces objets au client javascript simplement en lui passant son id dans le cache, et on peut ensuite retrouver simplement cet élément.
* Le but de PropertyAccessorCache et d’AutoCreatorCache est de créer des objets de ce type en se basant sur des lambdas expressions et de pouvoir ensuite retrouver ces instances en se basant uniquement sur une lambda expression ou sur un id. Donc, quand on donne une lambda expression à un de ces caches, il va créer l’instance de l’objet correspondant si elle n’existe pas et la retourner, ou directement retourner celle qui est dans le cache, s’il y en a déjà une. Lors de la création, il assigne un id à cet objet, et on peut ensuite retrouver l’objet en utilisant cet id. Pouvoir retrouver un objet avec une lambda expression est utile pour le serveur et pouvoir retrouver un objet avec un id est utile pour fournir des références de ces objets au client javascript.

Finalement, il y a les classes suivantes :

* AuthenticationManager qui s’occupe de la gestion de l’authentification des utilisateurs.
* BackupManager qui s’occupe de faire des backups de la base de données à intervalles réguliers.
* ConfigurationFileGenerator qui est responsable de créer un fichier de configuration javascript qui sera disponible pour le client javascript, de manière à avoir des paires clés/valeurs qui lui permettront de configurer le client.
* CoreServer qui instancie et garde des références sur de nombreuses classes dont le serveur a besoin pour fonctionner.
* ExpressionNormalizer dont le but est de normaliser des lambdas expressions qu’on utilise comme clé de dictionnaire (par exemple pour les caches), de manière à ce que deux lambdas qui ont la même structure mais pas exactement les mêmes noms de symboles soient considérées comme identiques (par exemple (Person x) => x.Comment et (Person p) => p.Comment.
* IconManager qui s’occupe de la conversion des icônes au format natif de Cresus.Core en fichier png et qui les met à disposition du client via un fichier css.

# Core.Databases.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Core.Databases servent à représenter les définitions des DataSetMetadata et des EntityTableMetadata qui vont être transmises au client javascript.

De plus, elles contiennent un certain nombre de méthodes utilitaires en relations avec ces schémas, comme pour la création d’une entité dans un DataSet, pour obtenir les données d’une entité dans un DataSet, etc.

# Core.Extraction.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Core.Extraction servent à la production des exportations, comme par exemple des fichers pdf ou csv.

# Core.IO.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Core.IO servent à faire la serialization et la deserialization de données entre le serveur et le client. En particulier, certain objets savent se sérialiser tout seuls (ceux dans Core.Database.\* et ceux dans Layout.\*) et ne sont donc pas directement concernés par ces classes, mais les autres n’ont pas de méthodes pour ce faire.

# Core.PropertyAccessor.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Core.PropertyAccessor servent à obtenir les valeurs des propriétés des entités, à les assigner, ou à vérifier qu’une valeur est compatible avec. On a un besoin spécifique pour ces classes, car il faut parfois effectuer une conversion spécifique à la propriété avant de traiter la valeur. C’est particulièrement vrai pour les champs de type « valeur » auxquels on peut associer une instance de IFieldBinder qui va effectuer des conversions entre la valeur stockée dans l’entité et la valeur affichée à l’utilisateur.

# Core.PropertyAutoCreator.\*

Les classes définies dans le namespace Epsitec.Cresus.WebCore.Server.Core.AutoCreator servent à créer des entités qui n’existent pas et à les assigner à une autre entité, uniquement en se basant sur la définition de la propriété. C’est cela qui va servir à créer automatiquement le commentaire d’une personne, lors que l’utilisateur clique sur la tuile d’une personne qui n’a pas encore de commentaire.

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