Crésus WebCore

Table des matières

[1 Infrastructure générale 2](#_Toc302130983)

[1.1 Client 2](#_Toc302130984)

[1.2 Serveur web 2](#_Toc302130985)

[1.3 Serveur Core 2](#_Toc302130986)

[2 Client 2](#_Toc302130987)

[2.1 Javascript 3](#_Toc302130988)

[2.1.1 app.js 3](#_Toc302130989)

[*2.1.2* Dossier *js* 3](#_Toc302130990)

[2.2 CSS 3](#_Toc302130991)

[3 Serveur Web 3](#_Toc302130992)

[3.1 Proxy PHP 4](#_Toc302130993)

[4 Serveur Core 4](#_Toc302130994)

[4.1 IconsBuilder 4](#_Toc302130995)

[4.2 PanelBuilder 4](#_Toc302130996)

[4.3 CoreSession 4](#_Toc302130997)

[4.4 Nancy 5](#_Toc302130998)

[4.4.1 CoreModule 5](#_Toc302130999)

[5 Problèmes connus 5](#_Toc302131000)

[5.1 Emptysummary pas recréée 5](#_Toc302131001)

[5.2 Commentaire pas affiché correctement 5](#_Toc302131002)

[5.3 Propriété Template mal gérée 5](#_Toc302131003)

[5.4 Sessions Core non supprimées 5](#_Toc302131004)

[5.5 Modification d’entités « on the fly » 5](#_Toc302131005)

[5.6 Les collections sont envoyées directement 6](#_Toc302131006)

[5.7 Entités pas synchronisées 6](#_Toc302131007)

[5.8 Serveur avec un seul thread 6](#_Toc302131008)

[5.9 Boutons reset/save mal placés 6](#_Toc302131009)

[5.10 Tri des entités très lent 6](#_Toc302131010)

[5.11 Base de données pas mise à jour 7](#_Toc302131011)

[5.12 Création d’entité 7](#_Toc302131012)

[5.13 Authentification 7](#_Toc302131013)

[5.14 Langues 7](#_Toc302131014)

[5.15 Code JavaScript pas minifié ni obfusqué 7](#_Toc302131015)

[6 Lancement en production 7](#_Toc302131016)

[6.1 TODO 7](#_Toc302131017)

[6.2 Client 8](#_Toc302131018)

[6.3 Serveur web 8](#_Toc302131019)

[6.4 Server Core 8](#_Toc302131020)

# Infrastructure générale

Nous considérons trois entités; l’utilisateur, le serveur web et le serveur Core. L’utilisateur se sert d’un navigateur web pour accéder au serveur web. Le serveur web se connecte au serveur Core pour obtenir les informations nécessaires.

La logique est séparée en deux parties. L’interaction avec l’utilisateur est entièrement gérée par du JavaScript présent sur le serveur web et exécuté par le navigateur. La gestion des entités (ajout, suppression, modification, ...) et l’authentification sont déléguées au serveur Core.

## Client

Le client n’a besoin que d’un navigateur web. Il accède à une adresse web donnée, se connecte en utilisant les identifiants qu’il connait, et peut travailler.

## Serveur web

Le serveur web est le point d’entrée pour le client. Le client interagit uniquement avec ce serveur, et n’a pas conscience du deuxième serveur.

Il fourni les fichiers statiques nécessaires au bon fonctionnement du site. Il s’agit de la homepage en HTML, des fichiers JavaScript, CSS et des images. Il gère aussi le cache afin de rendre le site le plus efficace possible.

Le serveur web sert de proxy pour accéder au serveur Core. Lorsque le client souhaite une information provenant du serveur Core, il effectue sa demande auprès du serveur web, qui se charge d’aller chercher les données sur le serveur Core et de les rendre au client.

## Serveur Core

Le serveur Core est utile pour attaquer la base de données. C’est lui qui permet d’ajouter, supprimer, modifier les entités.

Le serveur est lui aussi un serveur web, même s’il n’est utilisé que par le proxy. Ceci est pratique car le code peut utiliser les mécanismes connus pour le web (sessions, cookies, formulaires, ...)

# Client

Le client est entièrement développé grâce à la librairie ExtJS 4 [2].

Une seule page HTML est présente (*index.html*), elle sert à charger les fichiers CSS et JavaScript.

## Javascript

L’interface est entièrement gérée en JavaScript grâce à ExtJS 4. Des connaissances basiques d’ExtJS sont requises pour comprendre le fonctionnement de l’application. Les guides ExtJs [3] expliquent très bien les bases.

L’application fait, en particulier, usage de :

* Définition de classe, chargement dynamique, alias
* Layout & containers
* Data & Stores
* Ajax / JSON
* Formulaires

L’application utilise le fichier *app.js* à la racine du site, puis les fichiers présents dans le dossier /*js*.

### app.js

Ce fichier est le fichier principal, chargé par la page HTML. Il charge les bons fichiers JavaScript et affiche la page de login. Une fois l’utilisateur connecté, il crée l’interface globale de l’application.

### Dossier *js*

Le dossier /*js* contient les fichiers JavaScript utilisés pour créer l’interface. En général, ils héritent de composants de base d’ExtJS pour définir un affichage spécialisé.

Ces fichiers utilisent le système de chargement dynamique d’ExtJS ; de par leur nom, ExtJS est capable de charger les bons fichiers lorsqu’il en a besoin.

De cette façon, les fichiers sont chargés au fur et à mesure que l’utilisateur avance dans l’application. Une fois tous les fichiers chargés, l’affichage devient plus fluide.

Certains fichiers sont préchargés par *app.js*. Ceux-ci sont requis car ils utilisent le système d’alias d’ExtJS. Voir la documentation d’ExtJS sur les classes pour plus d’informations.

## CSS

L’application contient deux fichiers CSS. *style.css* contient les styles édité manuellement pour les besoins de l’application. icons.css est généré automatiquement au lancement du serveur. Le chapitre 4.1 en parle plus précisément.

# Serveur Web

Le serveur web est un serveur Nginx [5]. Ce serveur est réputé être plus performant qu’Apache pour fournir des fichiers statiques (en vitesse et en utilisation mémoire). Dans cette application, le serveur web ne fourni que des fichiers statiques, Nginx est donc une bonne solution.

Nginx se charge aussi de faire un proxy vers le serveur Core. Chaque requête débutant par */proxy/* est transférée au serveur Core puis retournée au client.

Le serveur doit être configuré pour n’accepter que les requêtes en HTTPS. Une redirection est mise en place si l’utilisateur tente d’accéder au site en HTTP.

Le proxy utilise par contre une connexion vers le serveur Core en HTTP. Ceci est plus rapide, et ne comporte pas de risque si les deux serveurs sont sur la même machine physique ou dans le même réseau.

Un exemple de configuration de Nginx est présent dans le répertoire de la documentation.

## Proxy PHP

Dans le cas où la solution ne peut pas être hébergée sur un serveur Nginx mais doit passer par un serveur Apache/PHP, un exemple de proxy PHP est disponible dans le dossier *proxy\_apache\_php*. Celui-ci est moins performant mais 100% fonctionnel.

Les fichiers contenus doivent simplement être copiés dans un dossier nommé *proxy* à la racine du serveur web.

Apache semble aussi capable de faire du *reverse proxy*, mais ceci n’a pas été testé.

# Serveur Core

Le serveur Core n’est accédé que par le serveur web.

Il fait partie de la solution Epsitec.Cresus, dans le projet Cresus.Core.Server. Il peut être lancé grâce au projet App.CresusCore et le paramètre de commande *–server*. Si le paramètre *–server* n’est pas spécifié, c’est la version desktop de Core qui se lance.

Pour lancer le serveur, l’application core crée une instance de classe *Epsitec.Cresus.Core.Server.CoreServerProgram*.

La création de l’instance lance le serveur web et attend pour répondre aux requêtes.

## IconsBuilder

Pour faciliter l’affichage des images au sein de l’application, celles-ci sont toutes générées au lancement du serveur. La classe *Iconsbuilder* est appelée et liste toutes les icônes disponibles dans l’assembly. Elle crée une version 16 et 32 pixels de chaque icône et les sauvegarde dans un dossier qui doit être défini à la compilation.

La classe génère aussi un fichier CSS qui lie les icônes à une classe CSS, *icons.css*. Lors du chargement de la page web, ce fichier est lu par le navigateur, il devient donc très facile d’utiliser les icônes.

## PanelBuilder

Le *PanelBuidler* est utilisé pour afficher les entités sur le navigateur. Lorsque que le client demande d’afficher une entité (que se soit en mode résumé ou édition), le serveur effectue une requête au *PanelBuilder* afin que celui-ci génère une description d’interface que ExtJS pourra interpréter.

## CoreSession

La *CoreSession* permet de faire l’interface entre le code C# du serveur et les entités/relations de l’application. Lors de sa connexion, l’utilisateur se voit assigner une *CoreSession*. Cette dernière lui permet d’effectuer des modifications d’entité en son nom, ou alors lui interdit de faire des modifications le cas échéant.

Chaque utilisateur a une session différente. Les sessions sont mémorisées dans un *CoreServer*. Le *CoreServer* est un singleton contenant un dictionnaire. Cela permet de récupérer facilement la *CoreSession* pour un utilisateur donné.

La session est détruite lorsque l’utilisateur se déconnecte (clic sur le bouton « logout »). S’il ferme son navigateur sans se déconnecter, la session n’est jamais supprimée. Il s’agit bien ici de la session « core » et non de la session « web ». La session « web » est gérée grâce à un cookie transmis à chaque requête par le navigateur. Si l’utilisateur ferme son navigateur ou supprime ses cookies, il apparaîtra comme déconnecté, mais la session « core » ne sera pas supprimée pour autant.

## Nancy

Le projet Nancy [1] est utilisé comme base du serveur Core. Sa documentation explique comment les modules sont créés et utilisés.

Le serveur Core utilise plusieurs modules ayant chacun une utilité très spécifique, expliquée dans leur en-tête.

### CoreModule

Chaque module Core doit étendre la classe *CoreModule*, et non la classe *NancyModule* comme habituellement avec Nancy.

Le *CoreModule* permet de gérer l’authentification sans que chaque module doive le faire. Il permet aussi de récupérer la *CoreSession*.

# Problèmes connus

## Emptysummary pas recréée

Lors de l’affichage d’une collection d’entités (les numéros de téléphone par exemple), chaque entité crée un panel pour son affichage.

Si la collection est vide, on crée un panel vide (*emptysummary*), qui contient un bouton permettant de rajouter une entité à la collection. Ce panel n’est pas présent si la collection contient des éléments.

Lorsque l’utilisateur décide de supprimer une entité d’une collection, le panel est supprimé pour refléter la suppression. S’il ne restait qu’une seule entité et qu’elle est supprimée, le panel lié à cette entité est caché, mais aucun *emptysummary* n’est affiché. Il n’est donc pas possible de rajouter une entité à la collection à ce moment là. Si on rafraichit l’affichage, l’*emptysummary* apparait et l’utilisateur peut rajouter une entité. Sessions Core non supprimées

Lorsqu’un client se connecte, on crée un objet *CoreSession* qui est propre à chaque client. Chaque session est mémorisée dans un singleton *CoreServer*. Si le client ferme son navigateur sans appuyer sur le bouton « Logout », la session n’est jamais supprimée de *CoreServer*.

## Modification d’entités « on the fly »

Lorsque l’on édite une entité, on peut avoir à choisir une option dans une liste (Par exemple « Monsieur, Madame »). La version desktop permet d’éditer cette liste directement (par exemple rajouter « Mademoiselle »), sans devoir quitter l’entité en cours d’édition.

La version web actuelle ne permet pas de le faire. Elle affiche une alerte expliquant qu’il faut sauvegarder les modifications actuelles, choisir la bonne base de données, faire ses modifications puis revenir à l’entité de base pour pouvoir l’éditer.

## Les collections sont envoyées directement

Il existe deux types de menus déroulants lors de l’édition d’une entité.

Le premier est valable pour les énumérations. Lors de l’exécution, les énumérations ne peuvent pas changer. Lors de l’affichage d’une entité, on indique que le champ est de type *enum* ainsi que son type. Le code JavaScript est capable de voir s’il connait déjà ce type et de l’afficher. Si ce n’est pas le cas, il va alors demander les informations au serveur, une seule fois. Il utilise le mécanisme des *Store*s ExtJS.

Le deuxième type de menu déroulant utilise des collections d’entités, par exemple « Monsieur, Madame ». Ces collections peuvent changer au cours de l’exécution (Rajout de « Mademoiselle »). Pour cette raison, le contenu de la collection est envoyé lors de l’affichage de l’entité, à chaque fois.

Cela fonctionne bien pour des petites collections, mais risque de poser problème lorsque l’on souhaite afficher des grosses collections (pays, villes de suisse, ...). De plus, si l’entité contient plusieurs fois une liste déroulante, les données sont aussi envoyées plusieurs fois.

Il faut trouver un moyen d’éviter d’envoyer les données à chaque fois, tout en maintenant la liste à jour.

## Entités pas synchronisées

Si plusieurs *CoreSession*s sont ouvertes en même temps, elles ne sont pas informées des changements des autres.

* Les utilisateurs A et B ouvrent l’application et affichent le client John Doe. Les deux sessions ont ce client en mémoire.
* A modifie le prénom de John en Jack.
* A rafraichit la fiche du client, il voit bien Jack Doe.
* B rafraichit la fiche du client, il voit toujours John Doe, car l’entité était en mémoire.
* Un nouvel utilisateur C ouvre l’application et affiche le même client, il verra son nouveau nom : Jack Doe. Il n’avait pas le client en mémoire et a récupéré les dernières données lors du chargement depuis la base de données.

## Boutons reset/save mal placés

Lors de l’édition d’une entité, chaque panel a ses propres boutons « Reset » et « Save ». Il est possible de laisser comme ceci, de placer un seul groupe de boutons en haut ou en bas de la colonne pour tout sauvegarder, ou alors de mettre les boutons dans le menu comme dans la version desktop.

## Base de données pas mise à jour

Si une mise à jour de la base de données est effectuée, la version desktop affiche une alerte et propose de mettre à jour la base locale. La version web n’affiche pas cette alerte, et la base n’est pas mise à jour. Le code par contre est à jour et ne correspond plus à la structure de la base. Cela ne fonctionne simplement plus.

Il faut trouver un moyen de mettre la base de données à jour automatiquement, ou alors d’afficher la même alerte lors du lancement du serveur.

## Création d’entité

Il n’est pas possible de créer une entité depuis la liste de gauche. Lors du clic sur le bouton « new », le serveur Core doit être capable de créer une nouvelle entité directement pour que l’utilisateur la voie en mode « édition ». Le problème est lié à l’affichage de la liste des *AbstractPerson*s. On ne sait pas si on doit créer une personne physique ou morale. Il n’est actuellement pas possible de présenter ce choix à l’utilisateur.

## Langues

Le code n’est pas traduit. Le code Javascript est écrit et commenté en anglais. Les textes provenant du Javascript sont donc en anglais. Le code C# présent dans Core retourne des textes en français (codés en dur).

ExtJS propose un exemple de traduction de code.

## Code JavaScript pas minifié ni obfusqué

Il est d’usage sur le web de minifier et d’obfusquer le code JavaScript servi aux utilisateurs.

La minification réduit la taille des fichiers en supprimant les commentaires, espaces blancs, retours à la ligne, etc...

L’obfuscation est une technique permettant de modifier le code afin qu’il devienne illisible par un humain, mais toujours compris par la machine. Le nom des méthodes et variables est changé par des noms formés d’un ou deux caractères. Cela évite certaines attaques sur le site car il devient extrêmement difficile pour un pirate de comprendre l’application s’il ne peut plus lire le code. Cela réduit aussi la taille du fichier.

Il est aussi d’usage de compiler tous les fichiers JavaScript en un seul gros fichier JavaScript chargé au lancement de la page.

ExtJS propose les SDK Tools [4] qui devraient obfusquer le code et compiler les fichiers ExtJS en un seul fichier, mais la dernière tentative d’utilisation n’a pas été concluante.

# Lancement en production

Voici une liste de choses à vérifier avant de lancer le serveur en production.

## TODO

A faire avant toute mise en production : résoudre les bugs.

## Client

* Utiliser le bon logo, en fonction du client
* Activer le cache, dans app.js, le *Loader* doit activer le cache pour ne pas recharger le JS à chaque lancement
* Désactiver les contenus de debug, comme les champs de connexion pré-remplis

## Serveur web

* Configurer Nginx pour accéder aux bons fichiers
* Configurer Nginx pour ne répondre qu’aux requêtes en HTTPS (comme dans le fichier d’exemple)
* Configurer Nginx pour faire les appels au serveur Core correctement (IP, port, ...)
* Si le serveur est Apache, activer le proxy PHP ou configurer Apache pour faire du reverse proxy

## Server Core

* Activer l’IconsBuilder. Lancer le serveur une fois afin qu’il génère les icônes et le fichier CSS dans le répertoire du serveur web. Un copier coller des fichiers générés depuis un autre dossier fonctionne aussi. Voir le chapitre 4.1.
* Lancer l’application en version desktop lors d’une mise à jour de la base de données. Voir le bug 5.6

Bibliographie

1. Nancy : <https://github.com/NancyFx/Nancy>
2. ExtJS : <http://www.sencha.com/products/extjs>
3. Guides ExtJS : <http://docs.sencha.com/ext-js/4-0/#!/guide>
4. Sencha SDK Tools : <http://www.sencha.com/products/sdk-tools/>
5. Nginx : <http://nginx.org>