Epsitec.Cresus.WebCore.Client

Le projet Epsitec.Cresus.WebCore.Client est le client web qui est utilisé pour communiquer avec le serveur qui se trouve dans Epsitec.Cresus.WebCore.Server. Il s’agit d’un client web destiné à être utilisé dans un navigateur web. La technologie utilisée est une librairie javascript, Ext JS[[1]](#footnote-1). Comme pour toutes les applications Ext JS, l’application se compose d’une page HTML vide qui en fait que référencer des scripts javascript qui vont modifier le DOM de la page.

# Ecran de login

Au lancement de l’application, l’utilisateur arrive sur un écran d’accueil qu’il doit utiliser pour se connecter avec son nom d’utilisateur et son mot de passe. Cet écran est géré par le fichier LoginPanel.js.

# Ecran principal

Une fois connecté, on affiche l’écran principal. Son affichage est complexe et géré par de nombreux fichiers. Le fichier App.js définit le conteneur principal qui va créer les sous-éléments

En haut de l’écran, il y a un menu qui permet à l’utilisateur de se déconnecter ou d’afficher les données des différentes bases. Ce menu est géré par la classe Menu.

Le reste de l’écran est utilisé par une zone dans laquelle on va afficher le contenu des bases. Elle est gérée par la classe TabManager. C’est cet élément qui est utilisé par le menu pour afficher le contenu des bases.

Pour chaque base à afficher, le TabManager va créer un nouvel onglet avec pour unique élément une instance de ColumnManager qui va s’occuper d’afficher tous les sous-éléments.

A gauche, il va créer une liste d’entités qui est gérée par la classe EntityListPanel. A droite, peut afficher des instances de la classe EntityColumn, suivant les actions effectuées par l’utilisateur.

L’affichage de l’écran principal de l’application correspond donc au schéma ci-dessous :

Bouton de sélection des bases

Menu

Bouton de logout

EntityListPanel

ColumnManager

EntityColumn

TabManager

App

# Listes d’entités

Toutes les listes d’entités qui sont afficher à un moment où à un autre à l’utilisateur sont des instances de la classe EntityList ou d’une de ses sous classes. Ces sous classes sont caractérisées par plusieurs éléments.

D’abord, une liste peut être éditable ou non. Si elle est éditable, on peut y rajouter ou en enlever des éléments. Si elle n’est pas éditable, on ne peut qu’en sélectionner des éléments.

Ensuite, les éléments d’une liste peuvent venir des plusieurs endroits différents :

* Les données d’une table d’entités dans la base de données, éventuellement préfiltrées. C’est par exemple le cas de la liste de gauche. Les données sont obtenues par le DatabaseModule du serveur, et elles correspondent aux définitions des DataSetMetadata du fichier de configuration Aider.Metadata.crconfig.
* Les données d’un SetViewController. C’est par exemple le cas des listes qui sont inclues dans les SetColumn. Les données sont obtenues par le SetModule du serveur, et elles correspondent à une vue sur une table d’entités configurée dynamiquement par le SetViewControlle. Par exemple, ces listes sont utilisées pour afficher les participants d’un groupe, donc une vue sur la table d’AiderGroupParticipantEntity, mais contenant seulement les entités liées à un groupe donné.
* Les données d’une FavoritesCollection. C’est par exemple le cas des listes de sélections d’entités dont les entités ne viennent pas directement d’une base de données, mais d’une liste d’entités favorites. Les données sont obtenues par le FavoritesModule du serveur et correspondent à un ensemble d’entités prédéterminée par le serveur.

Un certain nombre d’outils sont disponibles sur toutes ces listes, indépendamment de leur autre caractéristiques. Elles sont décrites dans les paragraphes suivants.

Colonnes

L’utilisateur peut configurer l’affichage des colonnes. C’est-à-dire qu’il peut choisir quelles colonnes sont affichées et quelles colonnes sont cachées parmi une liste de colonnes à disposition.

Tris

L’utilisateur peut trier les listes selon une colonne en cliquant sur son en-tête, ou selon plusieurs colonnes au moyen d’un dialogue prévu à cet effet.

Filtres

L’utilisateur peut filtrer les entités affichées. Pour cela, il peut entrer des critères de filtres sur chaque colonne séparément au moyen du menu contextuel qui apparait quand il clique sur l’en-tête de la colonne.

Un dialogue de configuration des filtres plus complet est en cours de création par Samuel (au 29 juillet 2013).

Exports

L’utilisateur peut exporter le contenu d’une liste, sous plusieurs formats différents :

* Des étiquettes, dont le texte est déterminé sur le serveur
* Un fichier csv dont les colonnes sont déterminées par un dialogue de sélection des colonnes. Toutes les colonnes disponibles à l’affichage le sont aussi à l’export. A terme, il est prévu de pouvoir exporter les mêmes données au format pdf.

Les filtres et les tris définis sur les entités sont utilisés lors de l’export.

# Colonnes et tuiles

L’affichage d’une entité sélectionnée dans la liste de gauche se fait à l’aide d’une succession de colonnes. On a deux types de colonnes :

* TileColumn. Son contenu est une série des tuiles
* SetColumn. Son contenu est une liste d’entités gérée par une instance de SetEditableEntityList

Les tuiles sont des petits bouts d’éléments d’interface graphique qui forment une TileColumn. Il y en a plusieurs types :

* SummaryTile : les tuiles de résumé d’une seule entité, qui affichent un texte.
* GroupedSummaryTile : les tuiles de résumé d’une collection d’entités dont le template a l’auto grouping activé. Il s’agit s’une séquence d’éléments contenant chacun le résumé d’une entité.
* CollectionSummaryTile : les tuiles de résumé d’une seule entité qui fait partie d’une collection d’entités dont le template a l’auto grouping désactivé. Elles affichent juste un texte, avec un bouton pour ajouter une nouvelle entité dans la collection et un autre pour la supprimer.
* EmptyCollectionSummaryTile : les tuiles utilisées pour les collections d’entités dont le template a l’auto grouping désactivé qui n’ont aucun élément. Elle contient juste un bouton qui permet d’ajouter une nouvelle entité à la collection.
* EditionTile : les tuiles utilisées pour l’édition d’entités. Elles contiennent des champs éditables et un bouton pour sauver ces champs dans la base de données.

C’est la classe ColumnManager qui est responsable de l’affichage de ces colonnes et de faire les requêtes sur le serveur pour obtenir leur définitions. La classe BrickWallParser a, quant à elle, le rôle de décoder la réponse du serveur pour créer les objets de configuration javascript qui vont servir à instancier les classes Ext JS.

# Actions

L’utilisateur peut effectuer des actions pour modifier une entité, mais aussi pour modifier une paire d’entités (dans le cas où l’entité secondaire fait partie d’une collection appartenant à l’entité principale) ou ajouter/enlever une entité d’une liste. Pour ce faire, on présente à l’utilisateur une fenêtre avec un formulaire dont le contenu est déterminé par le serveur. Lorsqu’il clique sur OK, le contenu du formulaire est envoyé au serveur qui effectue l’action.

Les étapes de ce processus sont relativement similaires à ce qui se passe pour les tuiles d’éditions et sont gérées par les classes Action, EntityAction et TypeAction. L’interface se compose aussi de tuiles et de briques, ce qui fait que l’interprétation du layout se fait par le BrickWallParser.

# Champs d’édition spécifiques

De manière à éditer les entités, il faut des champs standards pour éditer des textes, nombres, dates etc. Il faut également des champs plus particuliers pour répondre à des besoins spécifiques. Il y en a trois types :

* EnumerationField. C’est un champ d’édition qui permet de sélectionner une valeur parmi une énumération de valeurs obtenue depuis le serveur.
* EntityReferenceField. C’est un champ d’édition qui permet d’éditer une référence sur une autre entité. On sélectionne la référence au moyen d’un EntityPicker qui présente à l’utilisateur une liste d’entités.
* EntityCollectionField. C’est un champ d’édition qui permet d’éditer une collection d’entités. On peut supprimer des entités de la liste et en rajouter au moyen d’un EntityPicker.

En plus de cela, il y a les champs d’édition spéciaux. Le but de ces champs, est de pouvoir définir des champs d’édition personnalisés spécifiques à une application. Pour ce faire, on a une implémentation dédiée dans le serveur qui fournit des méthodes que le client peut appeler au moyen d’une url, ce qui lui permet d’obtenir des données qui répondent à ces besoins. Par exemple pour le projet AIDER, il y a un champ spécifique qui permet de sélectionner des groupes au sein de leur arbre, ce qui met bien en évidence leur structure hiérarchique. Tous les champs spéciaux doivent hériter de la classe Epsitec.SpecialField, qui met à disposition de quoi appeler les méthodes définies sur le serveur.

# Localisation

La localisation des textes internes à l’application se fait de manière relativement simple. Absolument tous les textes affichés à l’utilisateur qui sont définis par le client doivent venir de la classe Texts. Cette classe utilise un dictionnaire pour stocker les textes, et ce dictionnaire est rempli par un fichier javascript qui se trouve dans le dossier locale. Il suffit d’inclure le bon fichier de locale dans la page index.html pour avoir la locale dans la langue désirée.

Si on veut que l’application puisse être utilisée avec plusieurs langues à la fois, il faut soit rendre dynamique la page index.html avec du php par exemple, ou bien avoir une règle dans le serveur NGinx pour servir une page différente en fonction de l’url par exemple. Dans tous les cas, il est obligatoire de recharger la page web pour changer la locale. Le framework Ext JS ne supporte pas le changement à chaud de la locale.

# Notifications

Todo by Samuel ;-)

1. http://www.sencha.com/products/extjs [↑](#footnote-ref-1)