SPECIFICATIONS TECHNIQUES POUR LE DEVELOPPEMENT DES PLUGINS TOURISM SYSTEM CLIENT

V 1.0 27 janvier 2011

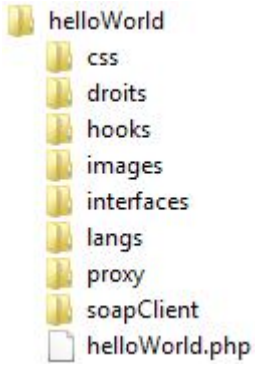
Ce document présente l'utilisation des plugins dans **Tourism System Client**.

Dans le **Client**, un plugin permet d'ajouter de nouvelles interfaces, mais aussi d'enrichir une interface existante.

Pour être fonctionnel, un plugin doit être installé et actif sur le **Server**. La liste des plugins est récupérée une fois sur le **Server**, juste après l'authentification de l'utilisateur, et conservée en mémoire durant toute la durée de la session.

Ils sont stockés dans le répertoire *plugins* se trouvant à la racine du projet. Chaque sous répertoire constitue un plugin. Le nom du répertoire sera le nom du plugin.

Ci dessous, un exemple d'arborescence constituant le plugin *helloWorld* :



Certaines parties du plugin étant chargés automatiquement, celle-ci doivent donc se situer à des endroits précis de l'arborescence. Cependant rien n'est obligatoire :

Le fichier *nomPlugin.php*, se situant à la racine, est inclus à tout moment, dans chaque interface de l'application. Son contenu se retrouve dans la source *HTML* de la page, entre les balises *head*. Il peut par exemple, servir à enrichir le menu principal.

Le répertoire *droits*, contient un fichier php par type d'utilisateur,

permettant de rajouter des constantes de droits, utilisées dans les interfaces. La classe *tsDroits* inclus automatiquement ces fichiers. Les types d'utilisateurs possibles sont : *root*, *superadmin*, *admin*, *manager*, *desk*.

Le répertoire *hooks*, contient des fichiers php permettant d'enrichir des interfaces existantes. Pour enrichir une interface, il suffit de créer un fichier portant le même nom. Celui-ci sera alors inclus automatiquement, dans la source *HTML* de la page, entre les balises *head*.

Le répertoire *interfaces*, contient des fichiers php permettant de créer de nouvelles interfaces. Celle-ci peuvent être appelées depuis le menu.

Le répertoire *langs*, contient autant de sous-dossier que de langues disponibles. Chacun d'eux contenant des fichiers js permettant la traduction des libellés. Pour chaque nouvelle interface proposée par le plugin, le fichier de langue portant le même nom est chargé automatiquement. Il est possible, selon les besoins, d'en créer d'autres qui pourront être appelés manuellement, pour les hooks, ou le menu.

Le répertoire *proxy*, contient des fichier php qui sont les points d'entrée des requêtes vers le serveur. Chaque fichier constitue un service et doit être nommé selon ce modèle : *pNomService.php*.

Le répertoire *soapClient*, contient les clients *SOAP*. Le découpage et le nommage des fichier n'est pas imposé, sachant que les classes sont incluses selon les besoins dans les classes *proxy*.

Les répertoires *css* et *images*, contiennent les fichiers *CSS* et les images. L'organisation de ces dossiers n'est contrainte à aucun modèle.

Présentation du HookMgr

L'objet *Ext.ts.HookMgr* est une instance de la classe *Ext.ts.HookGroup*. Il permet de gérer l'ensemble des hooks, pour les menus et les interfaces.

La méthode *get* permettent d'ajouter des hooks pour une interface. Elle s'utilise de la façon suivante :

```
Ext.ts.HookMgr.add(string nomHook, object definitionHook);
```

Le nom du hook doit être le même que le nom de l'interface qu'il est censé enrichir. De cette façon, durant l'initialisation de l'interface, les hooks référencés vont automatiquement s'appliquer.

La méthode *addMenu* permet de rajouter des éléments au menu principal. Elle s'utilise de la façon suivante :

```
Ext.ts.HookMgr.addMenu(object/array menu);
```

Il est possible d'ajouter un ou plusieurs onglet à la fois, c'est pourquoi le

paramètre menu peut être un objet ou un tableau d'objet.

La méthode findCmp permet d'explorer, et d'accéder aux items d'une interface simplement, afin de les modifier. Elle s'utilise de la façon suivante :

Ext.ts.HookMgr.findCmp(*object/array* items, *string* itemId)

Le premier paramètre est l'élément que l'on veut parcourir, et le deuxième, l'itemId du composant auquel on souhaite accéder. La méthode retourne le composant en question, il est ainsi possible de modifier sa configuration.

Exemples

Tous les exemples suivants seront issus d'un plugin de test *helloWorld*. Ses fonctionnalités sont les suivantes :

- Ajout d'une interface comportant un simple bouton qui affiche une alerte.
- Surcharge d'une interface existante pour y intégrer ce même bouton.

Comment enrichir le menu ?

Pour enrichir le menu, nous allons utiliser le fichier *helloWorld.php* qui se trouve à la racine du projet. Voici un exemple de code ci dessous :

```
<link rel="stylesheet" type="text/css" href="plugins/helloWorld/css/menu.css" />
<script type="text/javascript" src="plugins/helloWorld/langs/<?php echo TS_LANG; ?>/menu.js"></script>
<script type="text/javascript">
    Ext.ts.HookMgr.addMenu({
        text: Ext.ts.Lang.menuBonjour,
        iconCls: 'bonjourMenu',
        url: 'plugin_helloWorld_bonjour.php',
        tsDroits: <?php tsDroits::printDroit('MENU_BONJOUR'); ?>
    });
</script>
```

Ici, on rajoute un seul onglet, on passe donc directement l'objet en paramètre de la fonction *addMenu*. Mais il est également possible de passer un tableau d'objet.

Afin d'accéder à l'interface, il faut renseigner une url de la forme suivante :

plugin_nomPlugin_nomInterface.php

L'exemple ci dessus va donc charger l'interface suivante :

plugins/helloWorld/interfaces/bonjour.php

La config *tsDroits* permet d'afficher ou non l'onglet en fonction des droits de l'utilisateur. La constante *MENU_BONJOUR* doit donc être définit dans les fichiers du répertoire *droits* : *true*, l'onglet s'affiche, *false*, il ne s'affiche pas. On utilisera la méthode statique, *printDroit*, de la classe *tsDroits*. De cette façon, si la constante n'est pas définie, on considérera qu'elle vaut *false*.

Les configs *text* et *iconCls* sont des configs utilisées par la classe *Ext.Button* : le fichier *menu.css* permet d'associer une icône à la classe *bonjourMenu*, et le fichier *menu.js* est le fichier de traduction dans lequel est définit la variable

Ext.ts.Lang.menuBonjour.
Pour charger dynamiquement le bon fichier de traductin en fonction de la langue courante, il suffit d'utiliser la constante *TS_LANG* pour construire l'attribut *href*.

Voici un exemple de fichier de traduction. On utilise la fonction *Ext.apply* pour ajouter des traductions à l'objet *Ext.ts.Lang*.

```
Ext.apply(Ext.ts.Lang, {
    menuBonjour: 'Bonjour'
});
```

Comment créer une nouvelle interface ?

Les pages constituant de nouvelle interfaces doivent se situer dans le répertoire interfaces. La façon de les définir est exactement la même que les interfaces natives :

On peut utiliser la méthode statique *checkDroit* de la classe *tsDroits* au début du fichier, pour vérifier que l'utilisateur courant a accès à l'interface. Si ce n'est pas le cas, l'application renverra le message *Access denied*.

La définition de l'interface doit être encadré par deux inclusions de fichier : le header et le footer de la page qui se trouvent dans le répertoire *include* à la racine du projet (*header.php* et *footer.php*).

Voici ci dessous le code minimal permettant d'afficher une interface vide :

```
Ext.onReady(function(){
    new Ext.ts.Container({
        title: Ext.ts.Lang.titleContainer,
        content: []
    });
});
```

La classe *Ext.ts.Container* gère tout ce qui se trouve au dessus de la barre de titre (bandeau, onglet...). La seule configuration nécessaire est :

- *title* : le texte à afficher dans la barre de titre
- *content* : les composants Ext constituant l'interface

Il est également possible d'ajouter des *tools* dans la barre de titre, qui s'afficheront à coté du bouton permettant de réduire le bandeau.

La gestion des traductions est automatique : le fichier portant le même nom que l'interface dans le répertoire *langs* sera chargé automatiquement. Il est cependant possible d'en inclure d'autres.

Enfin, n'importe quelle fichier (php/js/css) peut être inclus dans la page, en fonction des besoins.

Comment enrichir une interface existante ?

Pour enrichir une interface existante, il faut créer un fichier php dans le répertoire *hooks*, portant le même nom que cette interface.

Dans ce fichier on va utiliser la fonction *add* du *HookMgr* pour déclarer le hook. Voici un exemple :

```
Ext.ts.HookMgr.add('fiches', {
    extendItems: function() {
        var gridFiche = Ext.ts.HookMgr.findCmp(this.items, 'gridFiche');
        gridFiche.bbar.unshift({
            xtype: 'button',
            text: Ext.ts.Lang.clickHere,
            iconCls: 'comment',
            handler: this.bonjour
        });
    },

    bonjour: function() {
        Ext.MessageBox.alert(
            Ext.ts.Lang.alertTitle,
            Ext.ts.Lang.alertMsg
        );
    }
});
```

Ici on va enrichir l'interface *fiches*. L'objet en deuxième paramètre est la définition du hook :

La fonction *extendItems* permet de modifier les composants Ext constituant l'interface. Ceux ci sont disponibles dans l'objet *this.items*. En utilisant la fonction *findCmp* du *HookMgr* il est possible d'accéder facilement aux composants possédant l'attribut *itemId*.

Dans cette exemple, on récupère le composant *gridFiche*, pour ajouter un bouton en première position dans sa *BottomBar*.

Il est également possible de surcharger ou d'ajouter des fonctions. Par exemple la fonction *bonjour* est utilisée ici en tant que *handler* du bouton.

La gestion des traductions n'est pas automatique. De la même manière que pour le menu, il faut inclure manuellement le fichier souhaité en utilisant la constante *TS_LANG*.

Enfin, n'importe quelle fichier (php/js/css) peut être inclus dans la page, en fonction des besoins.

Comment appeler le proxy ?

Dans l'application, toutes les classes permettant d'appeler les proxy utilise la fonction Ext.ts.url pour construire l'url. Cette fonction a besoin de deux informations : le service et l'action. Pour appeler les proxy d'un plugin il spécifier une troisième information : le nom du plugin.

Voici par exemple la fonction *bonjourFromServer*, qui affiche une alerte (comme la fonction bonjour vue précédemment) mais en allant chercher le message sur le serveur.

```
bonjourFromServer: function() {
    Ext.ts.request({
        plugin: 'helloWorld',
        service: 'bonjour',
        action: 'getMessageFromServer',
        success: function(response) {
            var result = Ext.decode(response.responseText);
            Ext.MessageBox.alert(
                Ext.ts.Lang.alertTitle,
                result.reponse
            );
        }
    });
}
```

La config contient bien l'attribut *plugin*. Le fonctionnement est le même avec *Ext.ts.JsonStore*, *Ext.ts.submit* et *Ext.ts.location*.

Ensuite, la construction des fichiers de proxy est semblable aux proxy natifs de l'application. Par exemple :

Le fichier *pBonjour.php* doit contenir la classe *pBonjour*, ce qui constitue le service. Chaque méthode de cette classe déclarée en *public* constitue une action.

Enfin, la construction des fichiers soapClient n'est contrainte à aucun modèle.