fellerLYnk

Description des widgets

10.FLYNKWIDG-F.1806/180611

by Schneider Electric

1 ID de widget

Chaque widget de type utilisateur doit avoir un identifiant (ID) unique dans un contrôleur. Une chaîne de caractères quelconque peut être utilisée comme ID.

L'ID de widget est utilisé lors de l'installation du widget, apparaît dans son code de programme et ne peut pas être facilement changé après l'installation du widget dans l'application. L'ID de widget n'apparaît dans l'interface utilisateur que pendant l'installation. Pour minimiser le risque de collisions lors de l'utilisation de vos widgets dans diverses installations, nous vous recommandons d'utiliser comme ID un ensemble de caractères contenant une composante unique du développeur de widgets (préfixe).

De plus, certains ID sont déjà réservés pour les widgets système, l'application vous en informera lorsque vous tenterez d'installer le widget.

2 Structure des fichiers de widget

Chaque widget contient l'ensemble de fichiers suivants

- script.js, c'est le fichier requis dans le dossier racine du widget. Il contient du code JavaScript qui décrit l'interface du widget et sa logique de travail. Le script est chargé sur la page une fois, quel que soit le nombre de widgets de ce type créés par l'utilisateur dans une application particulière.
- style.css, c'est un fichier optionnel dans le dossier racine du widget. Il contient des styles CSS qui sont uniques pour votre widget. Si ce fichier apparaît dans le dossier du widget, il sera automatiquement chargé lors du rafraîchissement de la page.
- Tous les autres fichiers graphiques, JavaScript, CSS qui seront utilisés dans votre widget. Vous devez les charger dans l'un des fichiers mentionnés ci-dessus.

Nous recommandons d'utiliser la structure de dossiers suivante pour le widget :

- script.js
- style.css
- images
  fichiers graphiques
- js
  fichiers et bibliothèques JavaScript supplémentaires.

3 Bibliothèques et frameworks disponibles

Dans votre widget, vous pouvez utiliser les bibliothèques et frameworks suivants qui sont inclus dans le projet :

- JQuery (http://jquery.com)
- plugin JQuery noUiSlider (http://plugins.jquery.com/nouislider/)
- plugin JQuery knob (http://plugins.jquery.com/knob/)
- Boostrap v3 (CSS+Javascript) (http://getbootstrap.com)
- Font Awesome v4 (http://fontawesome.io)
- UnderScore.js (http://underscorejs.org/)
- Backbone.js (http://backbonejs.org/)

4 Fichier script.js

Avant de passer à la suite de la documentation, nous vous recommandons de lire la documentation de la bibliothèque Backbone.js http://backbonejs.org/ .

Dans votre code JavaScript, dans la collection de classes de widgets, vous devez ajouter vos propres extensions pour les classes de base – Widget (modèle) et WidgetView (vue) – en réalisant l'ensemble nécessaire de champs et méthodes comme indiqué dans l'exemple ci-dessous :

```
Widget["your widget ID"] = Widget.extend({
    /* description of widget settings*/
    config : {
        ...
    },

    /*is called when objects values are changed on Logic Machine*/
    changeValue : function(name, value) {
        ...
    }
}

WidgetView["your widget ID"] = WidgetView.extend({
    /*width of widget in cells*/
    width : ...,
    /*height of widget in cells*/
    height : ....,

    /* is called when the object is initialized */
    init : function() {
        ....
    }

    /*main template of widget*/
    template : ...,

    /*template for viewing in editor mode*/
    preview : ....,

    /*is called after rendering of widget during initialization*/
    afterRender : function() { ...}

    ...
})
```

Au lieu de "`your widget ID`" vous devez utiliser votre ID de widget et le spécifier également lors de l'installation du widget à partir de l'interface. En plus des attributs de service décrits, lors de l'implémentation de ces classes vous pouvez utiliser toutes vos méthodes et propriétés de helpers qui sont responsables de la conversion des données, de la validation, des calculs, du travail avec le DOM de votre widget, etc.

Les classes de base Widget et WidgetView sont des extensions des classes de base de modèle (Backbone.Model) et de vue (Backbone.View), Backbone inclut un ensemble de méthodes de helper pour travailler avec les objets fellerLYnk qui peuvent être utilisés dans votre implémentation.

Pour chaque type de widget créé par l'utilisateur Touch, les classes suivantes sont créées.

```
model = new Widget["your widget ID"]();
view = new WidgetView["your widget ID"]({model:model});
```

Ainsi, les appels de méthode de modèle à l'intérieur du modèle lui-même doivent être effectués à l'aide de *this* :

```
this.func()
```

L'appel de propriétés et de méthodes de modèle dans la vue doit se faire de la manière suivante :

```
this.model.func()
```

4.1 Configuration du widget

Tous les paramètres du widget nécessaires pour le lier aux objets fellerLYnk ainsi que les autres paramètres et données nécessaires à la création des formulaires et à l'édition dans le constructeur Touch doivent être décrits dans la propriété *config* de votre modèle.

```
Example:
Widget["your widget ID"] = Widget.extend({
    config : {
        title : "Name of widget",
        fields : {
            title : {
                datatype : "string",
                title : "Title"
            }
        },
        objects : {
            key : {
                datatype : dt.bit,
                title : "Object"
            }
            ...
        },
        settings : {
            "min-value" : {
                datatype : "number",
                title : "Minimum value"
            }
            ...
        },
        systems:["blinds", "ac"]
    },
    ...
}
```

Le champ «title» est utilisé pour spécifier le nom du widget. Il sera affiché en mode prévisualisation et en mode édition.

Le champ «fields» est utilisé pour spécifier les champs du widget qui doivent être configurés par l'utilisateur dans le constructeur. Les clés d'objet seront utilisées pour accéder aux valeurs de ces propriétés. L'accès aux champs à l'intérieur du modèle lui-même peut se faire avec la commande suivante *this.field*. Nous recommandons d'utiliser ce champ uniquement pour éditer la propriété *title* et de définir tous les autres paramètres du widget dans la section *settings*.

Le champ «settings» permet de spécifier une liste de paramètres du widget qui sont disponibles pour édition par l'utilisateur en mode constructeur. Les clés d'objet seront utilisées pour accéder aux paramètres grâce aux helpers décrits ci-dessous. Pour chaque propriété, vous devez spécifier *title*, qui sera vu en mode d'édition du widget, et *datatype*. Deux types de données sont actuellement pris en charge - *string* et *number*.

Le champ «objects» est utilisé pour afficher la liste des objets de fellerLYnk (adresses de groupe) qui peuvent être liés à un widget particulier. Les objets spécifiques doivent être spécifiés par l'utilisateur à l'aide d'un formulaire de création de widget dans le constructeur Touch. Les clés d'objet seront utilisées pour accéder aux valeurs et propriétés d'objet grâce aux helpers décrits ci-dessous. *Title* doit également être spécifié pour chaque objet qui

sera affiché dans le formulaire d'édition de widget ainsi que le numéro de type de données KNX. Une liste de tous les types de données pris en charge est disponible ici http://openrb.com/docs/lua.htm#grp-info. Pour fournir des constantes numériques spécifiques, nous recommandons d'utiliser l'objet *dt* avec une liste de champs définis dans le lien ci-dessus. Si un sous-type KNX spécifique est spécifié comme type de données, l'utilisateur aura l'option de choisir des objets avec ce type seulement. Si un type spécifique est spécifié comme type de données, l'utilisateur aura l'option de choisir des objets avec ce type de données ou ses sous-types.

Le champ «systems» permet de spécifier une liste de systèmes techniques qui seront utilisés dans la navigation Touch. Les widgets d'un type particulier seront affichés sur un écran avec tous les systèmes techniques listés dans un tableau. Voici une liste d'identifiants de systèmes techniques pouvant être utilisés :

- éclairage
- stores
- puissance
- fenêtres
- climatisation
- ventilation
- plancher

4.2 Méthodes auxiliaires de classe Widget (modèle de base)

Voici une liste de méthodes Widget prédéfinies que vous pouvez utiliser dans vos modèles et vues.

4.2.1 Travailler avec les objets fellerLYnk (adresses de groupe) :

getValue(key)

Retourne la valeur actuelle de l'objet.

existsValue(key)

Retourne *true* si l'utilisateur a spécifié un objet spécifique dans le configurateur Touch, ou *false* si l'objet est maintenant spécifié.

getObject(key)

Retourne des informations sur l'objet – adresse du groupe, type de données et nom de l'objet. Exemple de réponse :

```
{
    updatetime:1461825143,
    address : "1/1/1",
    units "",
    value : true,
    name : "alarm",
    datatype: 1005
}
```

write(key, value)

Écris la valeur de l'objet dans le contrôleur/bus de terrain.

4.2.2 Autres méthodes et propriétés

conf(key)

Retourne la valeur du paramètre (décrit dans la section *settings* de la configuration du widget, spécifiée par l'utilisateur

title()

Retourne la valeur du champ «title» qui est spécifié par l'utilisateur lors de la configuration du widget dans le constructeur Touch.

ctitle()

Retourne le nom du widget qui est spécifié dans la section «title» de la configuration du widget.

isPreview

Ce champ a la valeur *true* si le widget est affiché en mode prévisualisation. Note! Le widget est affiché en mode prévisualisation avant les paramètres utilisateur, c'est-à-dire qu'il n'a pas d'adresses de groupe ou d'autres paramètres définis.

4.2.3 Appel de méthodes de classe Widget

Pour accéder aux fonctions de helper ci-dessus, utilisez le contexte correct. Exemple d'appel d'une fonction title() :

- dans les méthodes de modèle *this.title()*
- dans les méthodes de vue *this.model.title()*
- dans les templates *title()*

4.3 Méthodes de helper de classe WidgetView (vues de base)

Liste des méthodes prédéfinies de WidgetView qui peuvent être utilisées dans vos vues.

size(width, height)

Permet de redimensionner le widget après l'initialisation (à la volée). La taille du widget doit être spécifiée en nombre de cellules occupées. La taille d'une cellule est de 110px*110px, mais en tenant compte des champs, la taille réelle du widget peut être déterminée par la formule suivante (110*largeur-10) px * (110*hauteur-10) px

el

Modèle DOM d'un conteneur HTML dans lequel votre widget est inclus. L'utilisation plus détaillée est décrite dans la documentation de la bibliothèque Backbone.js http://backbonejs.org/#View-el. Nous recommandons de l'utiliser chaque fois que vous utilisez les sélecteurs pour accéder au modèle DOM de votre widget pour éviter les collisions avec d'autres widgets. Exemple de recherche sur la page de titre de votre widget :

```
$("div.widget-title", this.el)
```

Les méthodes de classe WidgetView ne peuvent être appelées qu'à partir de votre vue. *this* est utilisé pour y accéder. (Par exemple, *this.size(1,2)*)

4.4 Méthodes de déclenchement

Certaines méthodes décrites dans la structure du fichier JS sont des événements. Ils sont appelés par le noyau du programme dans des cas spécifiques.

changeValue(key, value) pour le modèle

Si vous avez implémenté cette méthode dans votre modèle, elle sera appelée chaque fois qu'une valeur d'objet a été modifiée dans fellerLYnk (adresses de groupe qui sont liées à votre widget). La clé d'objet (telle qu'elle figure dans la configuration) et sa nouvelle valeur sont fournies comme paramètres de la méthode.

init() pour la vue

Si vous avez implémenté cette méthode dans votre vue, elle sera appelée une fois lors de l'initialisation de la vue (avant son rendu). Elle vous permet d'initialiser les paramètres de démarrage de la vue qui peuvent être utilisés dans le template pendant son rendu.

afterRender() pour la vue

Si vous avez implémenté cette méthode dans votre vue, elle sera appelée une fois, juste après le rendu du widget. Elle peut être utilisée pour initialiser des commandes individuelles qui ne peuvent pas être réalisées dans le template (p. ex. bouton de variation).

4.5 Templates de widget

Dans la classe de vue du widget, vous devez implémenter deux méthodes :

template(model)

Elle sera appelée par le système lors de l'initialisation du widget (pour rendre le widget dans une liste commune de widgets). Le modèle initialisé est fourni comme paramètre d'entrée.

preview(model)

Elle sera appelée par le système lors de l'initialisation du widget pour son rendu de prévisualisation dans le constructeur Touch. Notez que dans ce cas, le modèle ne contiendra pas d'objets ni de paramètres de widget qui sont définis par l'utilisateur pendant le processus de création du widget.

Pour le template, nous recommandons fortement d'utiliser la méthode _.template de la bibliothèque Underscore.js http://underscorejs.org/#template. Mais vous pouvez aussi utiliser tout autre template ou créer vos objets DOM de widget en JavaScript. Dans ce cas, n'oubliez pas que l'ensemble du widget est placé dans un conteneur auquel vous pouvez accéder dans votre vue par *this.el*.

Tous les autres exemples de cette documentation utilisent la bibliothèque Underscore.js qui est déjà incluse dans le projet.

4.5.1 Structure de widget recommandée

Nous recommandons d'utiliser la structure HTML suivante pour appliquer toutes les classes de base des widgets système.

```
WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <div class=\"widget-title\"> \
                <div class=\"txt\"><%=title() || ctitle()%></div> \
            </div> \
            widget controls
        </div> \
    "),
    ...
})
```

Vous pouvez également ajouter vos propres classes et définir des styles pour elles dans votre style.css ou utiliser une autre structure de widget. Rappelez-vous que toutes les classes CSS créées pour votre widget peuvent affecter l'apparence d'autres éléments de l'application. Nous recommandons d'utiliser des noms uniques pour les nouvelles classes CSS afin d'éviter de telles collisions.

4.5.2 Utilisation des méthodes et propriétés du modèle dans les templates.

Si vous utilisez _.template Underscore.js pour le template du widget, alors, puisque un modèle de widget est passé comme paramètre d'entrée dans votre template, vous pouvez appeler n'importe quelle méthode de modèle et ses propriétés directement dans le template sans spécifier le contexte.

Exemple:

```
WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <div class=\"widget-title\"> \
                <div class=\"txt\"><%=title() || ctitle()%></div> \
            </div> \
                 <% if (existsValue('object')) { %> \
                 Object value <%= getValue('object')%> \
                 <% } %> \
        </div> \
    "),
    ...
})
```

4.5.3 Template de prévisualisation

Vous pouvez implémenter une méthode séparée pour afficher le widget en mode prévisualisation dans le constructeur Touch:

```
 WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <% if (existsValue(\"object\") { %> \
                <%=getValue(\"object\")%> \
            <%  } %> \
        </div> \
    "),
    preview: _.template(" \
        <div class=\"widget\"> \
            Object value \
        </div> \
    ")
    ...
})
```

Si la méthode *preview* n'est pas implémentée, la méthode *template* sera appelée pour faire la prévisualisation pendant la phase d'initialisation du widget. Ainsi, un seul template peut être implémenté pour le widget, mais gardez à l'esprit qu'en mode prévisualisation, il ne contient ni objets ni paramètres et que tous les contrôles nécessaires doivent donc être effectués dans le template.

L'exemple précédent mais en utilisant un seul template pour les deux modes d'affichage du widget :

```
WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    template: _.template(" \
        <div class=\"widget\"> \
            <% if (isPreview) { %> \
                Object value \
            <% } else if (existsValue(\"object\")) { %> \
                <%=getValue(\"object\")%> \
            <%  } %> \
        </div> \
    "),
    ...
})
```

4.6 Échange de données entre le modèle et la vue

Toute propriété du modèle est disponible dans la vue car l'instance de l'objet est son attribut, mais pas l'inverse. Par ailleurs, il arrive souvent que le changement d'attribut du modèle affecte la vue du widget. Dans ce cas, nous recommandons d'utiliser des événements Backbone http://backbonejs.org/#Events.

Les exemples suivants montrent comment afficher la valeur de l'objet modifié sur la page :

```
Widget["your widget ID"] = Widget.extend({
    ...
    changeValue : function(name, value) {
        //initiate event
        if (name=="status") return this.trigger("changedStatus");
        ...
    }
    ...
}

WidgetView["your widget ID"]  = WidgetView.extend({
    ...
    init : function() {
        //subscribe to a specific event model
        this.listenTo(this.model, "changedStatus", this.changedStatus);
    },
    template: _.template(" \
        <div class=\"widget\"> \
            <%=getValue(\"status\")%> \
        </div> \
    "),
    changedStatus : function() {
        //change the value in HTML code
        $(".widget", this.el).text(this.model.getValue("status"))
    },
    ...
})
```

Feller AG | Postfach | CH-8810 Horgen
Téléphone +41 44 728 72 72 | Téléfax +41 44 728 72 99

Feller SA | Caudray 6 | CH-1020 Renens
Téléphone +41 21 653 24 45 | Téléfax +41 21 653 24 51

Ligne de service | Téléphone +41 44 728 74 74 | info@feller.ch | www.feller.ch

by Schneider Electric