Librairie WKYJS

Préambule

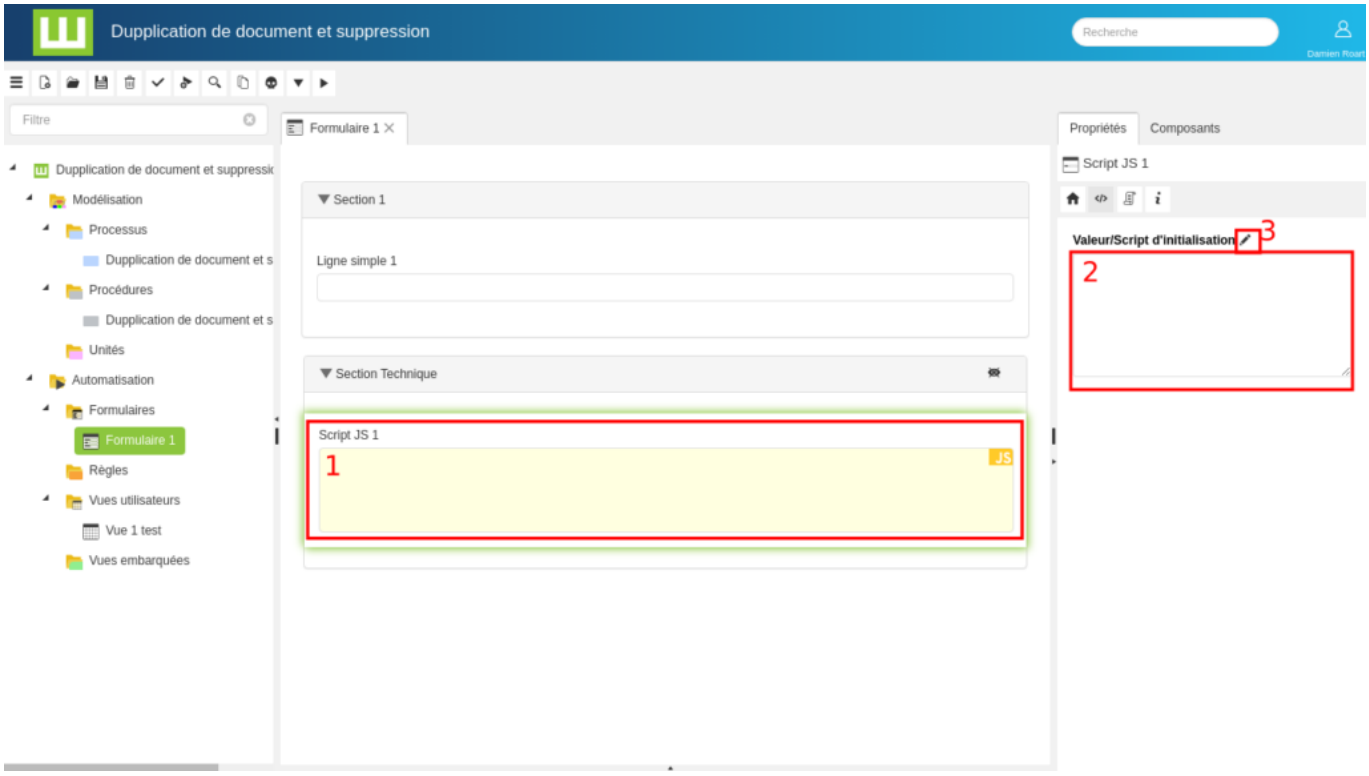
Lors de vos différentes modélisations au sein de Workey Designer, vous êtes souvent amenés à améliorer vos projets à l'aide de Javascript. Le but de ce document est donc de détailler les méthodes mises à votre disposition pour y parvenir. Elles sont fournies par la librairie WKYJS qui est disponible avec l'application Workey. Elles sont donc toutes préfixées avec le terme *WKYJS.* Elles permettent d'effectuer des opérations lors de l'affichage des documents dans votre navigateur. Il y a donc des méthodes pour le document, les sections et les champs. Dans un second temps, nous verrons qu'il est possible d'externaliser le javascript des modélisations à l'aide de fichiers.

Insertion de code Javascript dans un formulaire

Pour ce faire, lors de la modélisation d'un formulaire :

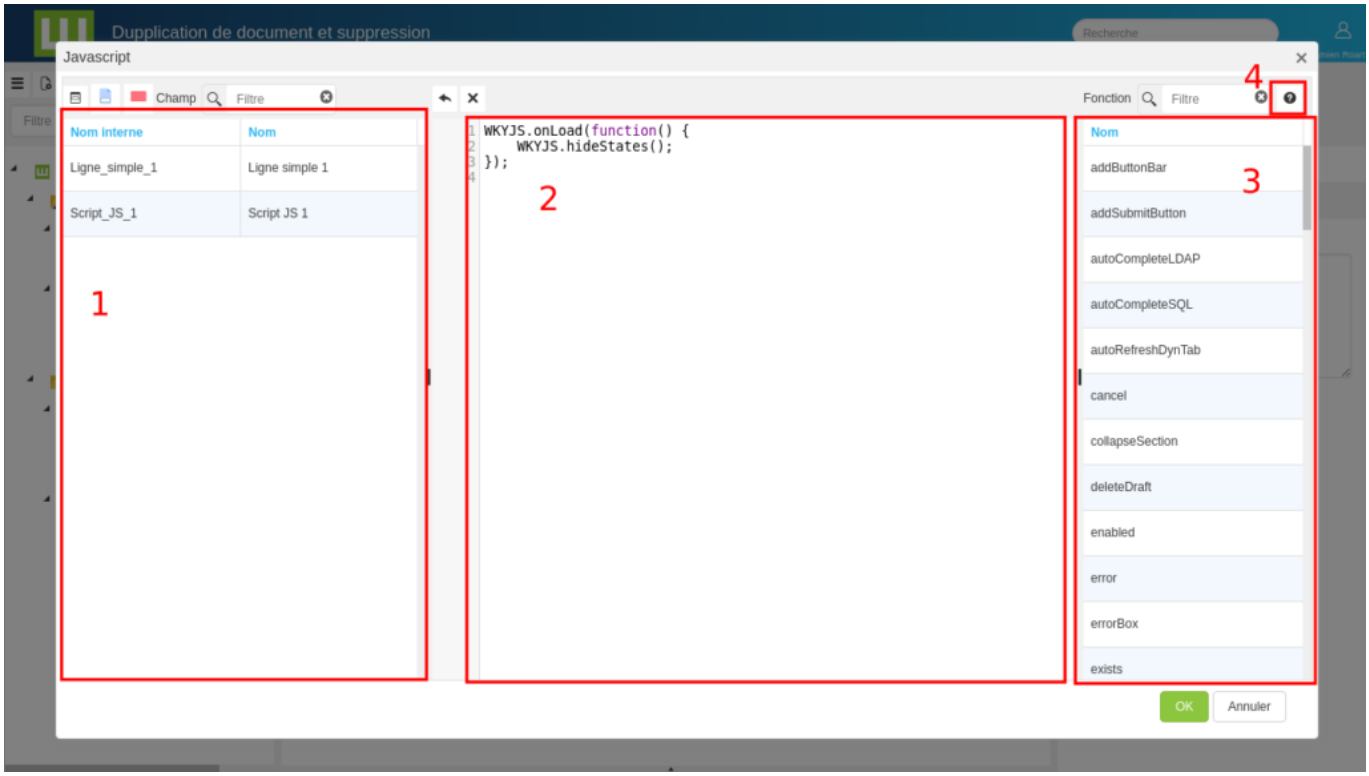
1. Il faut créer un champ de type **Script JS** dans le Designer.
2. Vous pouvez saisir directement le code Javascript directement dans la zone **Valeur/Script d'initialisation**.
3. Vous pouvez cliquez sur l'icône d'édition du code Javascipt qui permet d'ouvrir une boite de dialogue d'assistance à la saisie du scipt (image ci-dessous).

Lorsque c'est possible, il est conseillé de créer une section technique qui regroupe les Javascripts du formulaire. La visibilité de la section doit être cachée pour ne pas être affiché à l'écran.



Dans cette boite de dialogue vous y trouvez :

1. les différents objets à insérer dans votre code Javascript.
2. une zone d'édition du code Javascript.
3. les différentes méthodes disponibles avec la librairie WKYJS.
4. l'icône d'aide qui affiche cette documentation



Les méthodes de document

Les méthodes d’action utilisateur

Ces méthodes simulent, la plupart du temps, un clic sur un bouton qui effectue une action sur le document.

La méthode submit

Cette méthode permet de soumettre le document. Elle simule un clic sur le bouton *Soumettre*.

```
WKYJS.submit();
```

La méthode save

Cette méthode permet d’enregistrer le document sans le soumettre. Elle simule un clic sur le bouton *Enregistrer*.

```
WKYJS.save();
```

La méthode cancel

Cette méthode permet d’annuler la saisie en cours sur le document. Elle simule un clic sur le bouton *Annuler*.

```
WKYJS.cancel();
```

La méthode takeInCharge

Cette méthode permet de prendre en charge le document. Elle simule un clic sur le bouton *Prendre en charge*.

```
WKYJS.takeInCharge();
```

La méthode releaseTakeInCharge

Cette méthode permet de libérer la prise en charge sur le document. Elle simule un clic sur le bouton *Libérer la prise en charge*.

```
WKYJS.releaseTakeInCharge();
```

La méthode deleteDraft

Cette méthode permet de supprimer le brouillon. Elle simule un clic sur le bouton *Supprimer le brouillon*.

```
WKYJS.deleteDraft();
```

La méthode refresh

Elle permet de rafraichir les données du document. Elles sont envoyées au serveur où les connecteurs sont exécutés. Une fois les données traitées, elles sont de nouveaux réaffichées.

Exemple d'un *refresh* avec un bouton :

```
<button class="btn btn-primary" onClick="WKYJS.refresh()">Rafraichir le document</button>
```

La méthode selectStateByOperationNameAndStateName

Cette méthode permet de sélectionner le prochain état du document. Elle simule le clic de l'utilisateur sur une coche qui est en regard de l'état souhaité. Il faut spécifier le nom interne de l'opération en cours et le nom interne du prochain état.

```
WKYJS.selectStateByOperationNameAndStateName("Controler", "Controlee");
```

La méthode selectFirstState

Cette méthode permet de sélectionner le premier état affiché dans le document. Elle simule le clic de l'utilisateur sur la coche qui est en regard du premier état.

```
WKYJS.selectFirstState();
```

Les écouteurs d'action

Ces méthodes permettent d'enregistrer des traitements pour des évènements. Elles acceptent en paramètre une fonction contenant vos traitements et les enregistrent dans le système d'évènements dans l'ordre ou elles apparaissent dans le navigateur. Les traitements sont ensuite exécutés lorsque les évènements sont déclenchés dans le document. Pour la plupart des méthodes, les évènements sont déclenchés juste avant l'action, sauf pour les fonctions *onLoad, onRefresh* et *onChangeState*.

La méthode onSubmit

Elle permet d'exécuter du code Javascript juste avant la soumission du document.

```
WKYJS.onSubmit(function () {
    // votre code ici
});
```

Si une des fonctions, déclenchées par le onSubmit, renvoie la valeur false la

soumission du document est stoppée. Ceci peut permettre d'effectuer des contrôles, juste avant la soumission.

La méthode onSave

Elle permet d'exécuter du code Javascript juste avant la sauvegarde du brouillon.

```
WKYJS.onSave(function () {
    // votre code ici
});
```

Si une des fonctions, déclenchées par le *onSave*, renvoie la valeur *false* la sauvegarde du brouillon est stoppée. Ceci peut permettre d'effectuer des contrôles, juste avant l'enregistrement.

La méthode onCancel

Elle permet d'exécuter du code Javascript juste avant l'annulation de la saisie.

```
WKYJS.onCancel(function () {
    // votre code ici
});
```

Si une des fonctions, déclenchées par le *onCancel*, renvoie la valeur *false* l'annulation du document est stoppée. Ceci peut permettre d'effectuer des contrôles, juste avant l'annulation.

La méthode onTakeInCharge

Elle permet d'exécuter du code Javascript juste avant la prise en charge du document.

```
WKYJS.onTakeInCharge(function () {
    // votre code ici
});
```

Si une des fonctions, déclenchées par le *onTakeInCharge*, renvoie la valeur *false* la prise en charge est stoppée. Ceci peut permettre d'effectuer des contrôles.

La méthode onReleaseTakeInCharge

Elle permet d'exécuter du code Javascript juste avant la libération de la prise en charge du document.

```
WKYJS.onReleaseTakeInCharge(function () {
    // votre code ici
```

```
});
```

Si une des fonctions, déclenchées par le *onReleaseTakeInCharge*, renvoie la valeur *false* la libération de la prise en charge est stoppée. Ceci peut permettre d'effectuer des contrôles.

La méthode onDeleteDraft

Elle permet d'exécuter du code Javascript juste avant la suppression du brouillon.

```
WKYJS.onDeleteDraf(function () {
    // votre code ici
});
```

Si une des fonctions, déclenchées par le *onDeleteDraft*, renvoie la valeur *false* la suppression du brouillon est stoppée. Ceci peut permettre d'effectuer des contrôles.

La méthode onLoad

Elle permet d'exécuter du Javascript une fois que le document est affiché. A ce moment-là, tous les champs du formulaire sont accessibles dans vos développements Javascript.

Exemple :

```
WKYJS.onLoad(function () {
    // votre code ici
});
```

La méthode onRefresh

Cette méthode permet d'exécuter du Javascript une fois que le document a été rafraichi. Attention de ne pas lancer un *refresh* dans cette méthode, car le navigateur ne répondrait plus (boucle infinie de *refresh*).

Exemple :

```
WKYJS.onRefresh(function () {
    // votre code ici
});
```

La méthode onChangeState

Cette méthode permet d'exécuter du Javascript lorsque l'utilisateur change l'état du document. La fonction de callback récupère en paramètre le nom interne de l'état sélectionné.

Exemple :

```
WKYJS.onChangeState(function(stateName) {
    // votre code ici
});
```

Les méthodes de formulaires

Ces méthodes permettent de masquer/afficher des zones dans le document affiché. Elles peuvent aussi rajouter des zones.

La méthode addSubmitButton

Cette méthode ajoute une barre de bouton pour la soumission du document.

```
WKYJS.addSubmitButton();
```

Appelé sans paramètres, vous obtenez la barre suivante.

Vous pouvez customiser les boutons en passant un objet JS composé de paramètres à la méthode *addSubmitButton*.

Les paramètres par défaut sont les suivants :

```
{
    "vertical":"bottom",
    "horizontal":"right",
    "text":wui.localize("Submit"),
    "cancelable":true,
    "savable":true
}
```

Liste des propriétés de paramétrage :

vertical : La propriété positionne verticalement les boutons dans le formulaire, la valeur par défaut est *bottom*. Les boutons son donc en bas de formulaire. Cette propriété peut prendre deux valeurs : *bottom* ou *top*.

horizontal : La propriété positionne horizontalement les boutons dans le formulaire, la valeur par défaut est *right*. Les boutons son donc alignés à droite du formulaire. Cette propriété peut prendre deux valeurs : *right* ou *left*.

text : La propriété contient le texte du bouton de soumission qui est affiché. Elle peut être aussi localisée (voir localisation d'un libellé)

cancelable: La propriété indique si la barre doit contenir ou non le bouton *Annuler*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

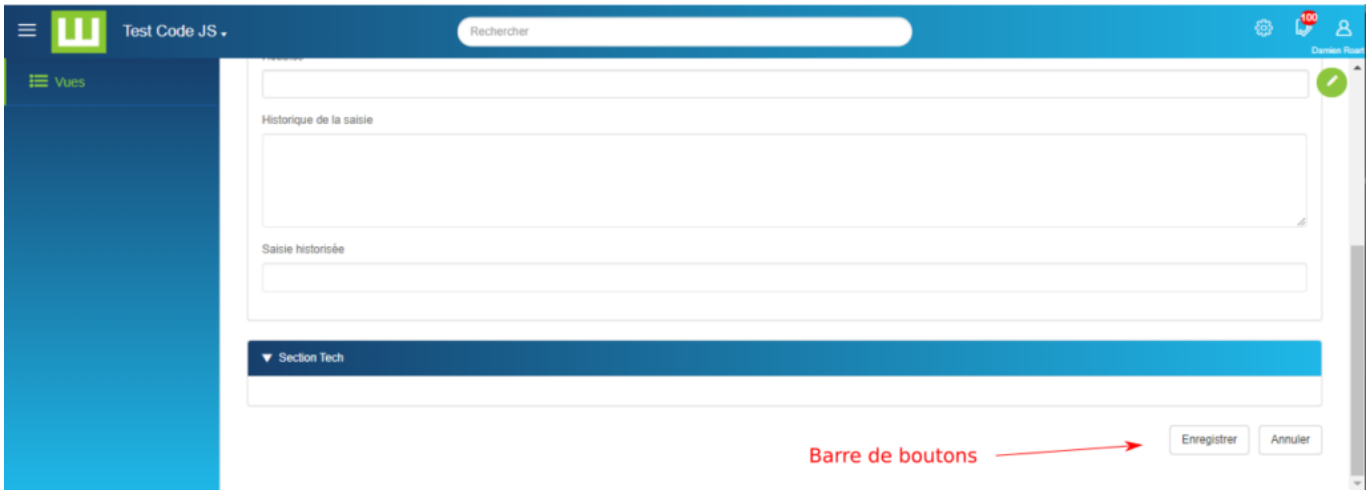
savable: La propriété indique si la barre doit contenir ou non le bouton *Sauvegarder*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

La méthode addButtonBar

A la différence de *addSubmitButton*, cette méthode ajoute une barre de boutons sans le bouton de soumission. C'est à vous de spécifier un ou des boutons d'actions. Cette méthode est plus complète que addSubmitButton, il est donc préférable de l'utiliser lors que c'est possible.

```
WKYJS.addButtonBar();
```

Appelé sans paramètres, vous obtenez la barre suivante.



Vous pouvez remarquer qu'il n'y aucun bouton d'action. Il va falloir les spécifier à l'aide d'un objet JS contenant les paramètres adéquates.

Les paramètres par défaut sont les suivants :

```
{
    "vertical":"bottom",
```

```
    "horizontal":"right",
    "buttons":[],
    "cancelable":true,
    "savable":true,
    "takeInChargable":false,
    "releasable":false,
    "discardable ":false
}
```

Liste des propriétés de paramétrage :

vertical : La propriété positionne verticalement les boutons dans le formulaire, la valeur par défaut est *bottom*. Les boutons son donc en bas de formulaire. Cette propriété peut prendre deux valeurs : *bottom* ou *top*.

horizontal : La propriété positionne horizontalement les boutons dans le formulaire, la valeur par défaut est *right*. Les boutons son donc alignés à droite du formulaire. Cette propriété peut prendre deux valeurs : *right* ou *left*.

buttons : La propriété contient une liste de boutons d'actions. C'est un tableau JS qui contient un ou des objets JS pour chaque bouton.

Un bouton est donc spécifié avec différentes propriétés, il y en a trois :

- ***label*** : cette propriété contient le texte qui va être affiché dans le bouton. Elle peut être aussi localisée (voir localisation d'un libellé)
- ***type*** : cette propriété contient le type de bouton, elle peut prendre différentes valeurs : *default*, *primary, secondary, danger, warning, success ou info*. Un bouton de type *primary* est affiché avec une couleur dominante (action principale par défaut). Avec comme type *default*, le bouton a une couleur plus discrète.
- ***action*** : cette propriété contient la fonction JS à exécuter lorsque l'utilisateur clique sur le bouton. Dans la plupart des cas, la fonction JS spécifiée contient généralement la sélection d'un état et la soumission du document.
- ***bgColor*** : cette propriété permet de spécifier une couleur de bouton. Elle accepte les noms de couleur HTML ou les code HEXA de couleur HTML (par exemple pour rouge : *red* ou *#CC0000*). Elle prend la main sur la propriété *type*.
- ***txtColor*** : cette propriété permet de spécifier la couleur du texte dans le bouton. Elle accepte les noms de couleur HTML ou les code HEXA de couleur HTML (par exemple pour noir: *black* ou *#000000*). Elle prend la main sur la propriété *type*.

cancelable: La propriété indique si la barre doit contenir ou non le bouton *Annuler*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

savable: La propriété indique si la barre doit contenir ou non le bouton *Sauvegarder*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

takeInChargable: La propriété indique si la barre doit contenir ou non le bouton *Prendre en charge*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

releasable: La propriété indique si la barre doit contenir ou non le bouton *Libérer la prise en charge*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

discardable: La propriété indique si la barre doit contenir ou non le bouton *Supprimer le brouillon*. Cette propriété peut prendre deux valeurs : *true* ou *false*.

L'exemple si dessous affiche une barre de boutons avec deux boutons d'action. Le bouton *Sauvegarder* n'est pas affiché.

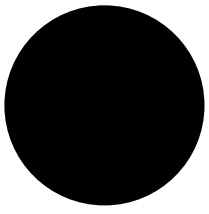
```
WKYJS.addButtonBar({
    "buttons":[
    {
        "label":{"fr-FR":"Accepter", "en-US":"Accept"},
        "type":"primary",
        "action":function() {
            WKYJS.selectStateByOperationNameAndStateName("Traiter",
"Accepte");
            WKYJS.submitDoc();
        }
    },
    {
        "label":{"fr-FR":"Rejeter", "en-US":"Reject"},
        "type":"normal",
        "action":function() {
            WKYJS.selectStateByOperationNameAndStateName("Traiter","Rejete");
            WKYJS.submitDoc();
        }
    }],
    "savable":false
});
```

Vous obtenez la barre de bouton suivante.

L'exemple si dessous affiche une barre de boutons de différentes couleurs.

```
WKYJS.addButtonBar({
    buttons: [
        {
            "label": "Primary",
            "type": "primary"
        },
```

```
        {
            "label": "Secondary",
            "type": "secondary"
        },
        {
            "label": "Warning",
            "type": "warning"
        },
        {
            "label": "Danger",
            "type": "danger"
        },
        {
            "label": "Success",
            "type": "success"
        },
        {
            "label": "Info",
            "type": "info"
        },
        {
            "label": "I'm darkorchid !",
            "type": "primary",
            "bgColor": "darkorchid"
        },
        {
            "label": "Orange avec texte noir",
            "type": "secondary", //sera surchargé par les deux propriétés
suivantes
            "bgColor": "darkorange",
            "txtColor": "#000000"
        }
    ]
});
```



La méthode showButtonInButtonBar

Cette méthode permet d’afficher un bouton de la barre de boutons ajoutée à l’aide la méthode *addButtonBar*. Elle prend en paramètre le rang du bouton dans la barre. La numérotation du rang au sein de la barre commence à 0.

```
// affiche le 3ème bouton de la barre
WKYJS.showButtonInButtonBar(2);
```

La méthode hideButtonInButtonBar

Cette méthode permet de cacher un bouton de la barre de boutons ajoutée à l'aide la méthode *addButtonBar*. Elle prend en paramètre le rang du bouton dans la barre. La numérotation du rang au sein de la barre commence à 0.

```
// cache le 1er bouton de la barre
WKYJS.hideButtonInButtonBar(0);
```

La méthode showStates

Cette méthode permet de montrer la section *Prochain état*. Elle est très rarement usitée.

La méthode hideStates

Cette méthode permet de cacher la section *Prochain état*. Elle est souvent utilisée avec les méthodes *addSubmitButton* et *addButtonBar*.

La méthode hideSaveButton

Cette méthode permet de cacher le bouton *Sauvegarder* qui est situé dans la section *Prochain état*.

La méthode hideCancelButton

Cette méthode permet de cacher le bouton *Annuler* qui est situé dans la section *Prochain état*.

Les méthodes de contexte

Ces méthodes permettent de récupérer des données sur le document hors formulaire. Elles renseignent sur le contexte du document affiché.

La méthode getWorkeyProcessType

Cette méthode permet d'accéder au nom interne du processus associé au document affiché.

La méthode getWorkeyProcessVersion

Cette méthode permet d'accéder au numéro de version processus associé au document affiché.

La méthode getWorkeyDocumentType

Cette méthode permet d'accéder au nom interne du document affiché.

La méthode getWorkeyForm

Cette méthode permet d'accéder au nom interne du formulaire affiché.

La méthode getWorkeyDocumentId

Cette méthode permet d’accéder à l’id du document affiché.

La méthode getWorkeyRole

Cette méthode permet d’accéder au nom interne du rôle qui a ouvert le document affiché.

La méthode getWorkeyCurrentState

Cette méthode permet d’accéder au nom interne de l’état du document affiché.

La méthode getWorkeyNextStates

Cette méthode permet de renvoyer un tableau contenant les noms internes des prochains états disponibles.

La méthode getWorkeyUserLanguage

Cette méthode permet d’accéder à la langue utilisée pour l’affichage document.

Les méthodes de contrôles

La méthode isEmailValid

Cette fonction renvoie *true* si l’adresse email passé en paramètre est valide, sinon retourne *false*. Elle possède aussi un second paramètre optionnel qui permet de passer une chaine de caractères représentant une expression régulière. Elle sera utilisée si celle par défaut ne correspond pas à vos attentes.

```
WKYJS.setValue("Email", "john\@company.com");
// valorise le champ Email avec 'john\@company.com'

WKYJS.isEmailValid(WKYJS.getValue("Email"));
// renvoie true
```

La méthode isPhoneNumberValid

Cette fonction renvoie *true* si le numéro de téléphone passé en paramètre est valide, sinon retourne *false*. Elle possède aussi un second paramètre optionnel qui permet de passer une chaine de caractères représentant une expression régulière. Elle sera utilisée si celle par défaut ne correspond pas à vos attentes.

```
WKYJS.setValue("Tel", "01 55 85 11 95");
// valorise le champ Tel avec '01 55 85 11 95'

WKYJS.isPhoneNumberValid(WKYJS.getValue("Tel"));
```

```
// renvoie true
```

La méthode isFieldValueValid

Cette fonction renvoie *true* si la valeur du champ passé en paramètre est valide, sinon retourne *false*. Elle a deux paramètres, le premier est le nom interne du champ concerné, le second est une chaine de caractères représentant une expression régulière.

```
WKYJS.setValue("Code", "12abc99");
// valorise le champ Code avec '12abc99'

WKYJS.isFieldValueValid("Code", "abc");
// renvoie true, le champ Code contient la valeur abc
```

Les méthodes utiles

La méthode msg

Cette méthode permet d’afficher un message éphémère à l’écran.

```
WKYJS.msg("Hello les gens !");
// affiche 'Hello les gens !' comme message à l'écran
```

La méthode informationBox

Cette méthode permet d’afficher une boite de dialogue avec un message informatif.

```
WKYJS.informationBox("Hello les gens !");
// affiche 'Hello les gens !' dans une boite de dialogue d'information
```

La méthode errorBox

Cette méthode permet d’afficher une boite de dialogue avec un message d’erreur.

```
WKYJS.errorBox("Nom obligatoire");
// affiche 'Nom obligatoire' dans une boite de dialogue d'erreur
```

La méthode reopenDocAfterCreation

Cette méthode permet de rouvrir un document, après sa création, à l’étape suivante. En d’autres termes, lorsque l’on soumet un document pour création, il est directement réouvert pour renseigner le formulaire à l’étape suivante.

Cette méthode est surtout utilisée lors des dérivations. Elle permet juste après la création du document père d’enchainer avec la création des documents

fils. On évite ainsi le passage par la vue *A Faire*.

A noter que Le document est réouvert par le même utilisateur avec le même rôle utilisé à la création.

```
WKYJS.onLoad(function () {
    WKYJS.reopenDocAfterCreation();
});
```

La méthode sqlRequest

```
WKYJS.sqlRequest({
    "sqlQuery":"...",
    "dataSource":"...",
    "parameters":[ /* les paramètres ici */ ],
    "done":function(data) {
        // Votre code ici
    }
});
```

Cette méthode permet de requêter une source de connée et de récupérer les données.

Pour ce faire, il faut alimenter la méthode avec un objet javascript composé de plusieurs attributs :

sqlQuery : cet attribut contient la requête SQL. Elle doit contenir les placeholders *?* qui sont substitués avec les valeurs contenus dans le tableau *parameters*.

dataSource : cet attribut spécifie la source de données, c'est-à-dire la base de données à utiliser.

parameters : cet attribut spécifie les valeurs à utiliser pour substituer les placeholders dans la requête (*sqlQuery*). Il est composé d'un tableau de valeurs.

done : cet attribut permet de spécifier la fonction de callback qui est utilisée lors de la récupération les données issues de la requête. La fonction comporte un seul paramètre (*data)* correspondant aux données, c'est un tableau d'objets JSON.

Ci-dessous un exemple qui va chercher les lignes d'une commande.

```
WKYJS.sqlRequest({
    "sqlQuery":"select PRODUIT as Produit, QUANTITE as Quantite, PRIX as
Prix_unitaire, TOTAL as Total from  TEST_COMMANDES where NUMERO_COMMANDE =
'*?*'",
    "dataSource":"Workey_RW_DS",
    "parameters":[WKYJS.getField("Numero_de_commande").value()],
```

```
    "done":function(data) {
        WKYJS.getField("Lignes").value(data);
    }
});
```

La fonction de callback *done*, récupère un tableau composé d'objet javascript. Les objets sont composés d'attributs qui ont pour nom les colonnes de la requête. Dans l'exemple la requête contient des alias, évidemment ils ne sont pas obligatoires, mais nécessaire si l'on veut forcer le nom d'un attribut dans le résultat.

Les méthodes de section

La méthode showSection

Cette méthode permet d'afficher une section grâce à son nom interne.

```
WKYJS.showSection("Informations");
// montre la section Informations
```

La méthode hideSection

Cette méthode permet de cacher une section grâce à son nom interne.

```
WKYJS.hideSection("Informations");
// cache la section Informations
```

La méthode collapseSection

Cette permet de replier la section définie par le nom interne

```
WKYJS.collapseSection("Informations");
// replie la section Informations
```

La méthode expandSection

Cette permet de déplier la section définie par le nom interne

```
WKYJS.expandSection("Informations");
// déplie la section Informations
```

La méthode enabled

Cette méthode permet d'activer ou désactiver tous les champs d'une section. Bien sûr lorsque les champs sont désactivés, il n'est plus possible de modifier leurs valeurs. Cependant cette méthode ne permet de récupérer l'état d'activation de la section.

```
WKYJS.enabled("Infos", false);
// Désactive tous les champs de la section Infos

WKYJS.enabled("Infos");
// Renvoie un avertissement dans la console du navigateur
```

Les méthodes de champ

La méthode getFieldValue

Cette méthode permet d'accéder à la valeur du champ grâce à son nom interne.

```
WKYJS.getFieldValue("Nom");
// renvoie la valeur du champ Nom
```

La méthode setFieldValue

Cette méthode permet de modifier le contenu d'un champ grâce à son nom interne.

```
WKYJS.setFieldValue("Nom", "Jean-Pierre");
// affecte la valeur Jean-Pierre au champ Nom

WKYJS.getFieldValue("Nom");
// renvoie la chaine de caractère "Jean-Pierre"
```

Manipuler les valeurs de champs

Il faut bien sur respecter les types de données initiés par les champs.

Dans la même logique si un champ est multivalué, il faut manipuler les données à l'aide de tableau Javascript.

Par exemple pour un champ de type liste, représentant une liste de choix de villes, l'expression suivante nous donnes les choix sélectionnés.

```
WKYJS.getFieldValue("Villes");
// renvoie le tableau ["Paris", "Tokyo"]
```

Vous pouvez aussi utiliser la fonction *setFieldValue* pour valoriser, vider etc.

```
WKYJS.setFieldValue("Villes", ["New-York", "Mumbai"]);
// valorise le champ Villes avec les choix : New-York et Mumbai

WKYJS.getFieldValue("Villes");
// renvoie le tableau ["New-York", "Mumbai"]
```

```
WKYJS.setFieldValue("Villes", []);
// vide la liste des choix sélectionnés

WKYJS.getFieldValue("Villes");
// renvoie un tableau vide : [].
```

Si par mégarde, vous valorisez un champ multivalué avec autre chose qu'un tableau JS, une erreur s'affiche à l'écran.

La méthode showField

Cette méthode permet de montrer un champ grâce à son nom interne.

```
WKYJS.showField("Nom");
// montre le champ Nom
```

La méthode hideField

Cette méthode permet de cacher un champ grâce à son nom interne.

```
WKYJS.hideField("Nom");
// cache le champ Nom
```

La méthode onChange

Cette méthode permet d'exécuter du code javascript lorsque la valeur du champ est modifiée. Le champ est identifié par son nom interne.

```
WKYJS.onChange(designerName, function(designerName) {
    // Votre code ici
});
```

La méthode refreshOnChange

Cette méthode combine la méthode *onChange* d'un champ avec le *refresh* du document. C'est-à-dire que lorsque le champ identifié par son nom interne voit sa valeur modifiée, le document est rafraichi.

```
WKYJS.refreshOnChange (designerName, [cible1, cible2, ... cibleN], scrollTo);
```

Cette méthode accepte deux paramètres optionnels supplémentaires :

Le paramètre *[cible1, cible2, … cibleN]* est un tableau JSON qui contient les nom internes des champs qui seront mise à jour avec les nouvelles valeurs issues du refresh.

Le paramètre *scrollTo* est le nom interne d'un champ du formulaire, sur lequel

le focus sera appliqué après le rafraichissement. Cette fonctionnalité reste expérimentale et peut parfois donner un résultat non satisfaisant dû à l'évolution dynamique du DOM.

Par exemple, pour des listes de choix cascadée, il faut rafraichir le document pour calculer les autres listes descendantes.

```
WKYJS.refreshOnChange ("Choix_de_la_marque");
```

La méthode showOrHideElements

```
WKYJS.showOrHideElements(designerName, values, fieldNameTargets);
```

Cette méthode permet d'afficher au de cacher des champs en fonction des valeurs que peut prend un champ.

En d'autres termes, lorsque le champ identifié par son nom interne (*designerName*) a sa valeur égale à une des valeurs contenues dans *values.* Les champs dont le nom internes est contenu dans *fieldNameTargets* sont affiché, et l'inverse dans le cas contraire.

Par exemple, pour l'expression suivante, si l'utilisateur sélectionne un abonnement expert ou premium, une section d'informations sera affichée :

```
WKYJS.showOrHideElements("Choix_abonnement", ["expert", "premium"],
["section_infos_niv_sup"]);
```

La méthode hideOrShowElements

```
WKYJS.hideOrShowElements(designerName, values, fieldNameTargets);
```

Cette méthode permet de cacher au d'afficher des champs en fonction des valeurs que peut prend un champ.

En d'autres termes, lorsque le champ identifié par son nom interne (*designerName*) a sa valeur égale à une des valeurs contenues dans *values.* Les champs dont le nom internes est contenu dans *fieldNameTargets* sont caché, et l'inverse dans le cas contraire.

Par exemple, pour l'expression suivante, si l'utilisateur sélectionne un abonnement expert ou premium, une section d'informations sera cachée :

```
WKYJS.hideOrShowElements("Choix_abonnement", ["expert", "premium"],
["section_infos_niv_inf"]);
```

La méthode error

Cette méthode permet d'appliquer ou de retirer le style reflétant une erreur sur un champ.

```
WKYJS.error("Email",true);
// Affiche le champ Email en erreur

WKYJS.error("Email",false);
// Réinitialise le champ sans erreur
```

La méthode setPlaceholder

Cette méthode permet d'affiche un placeholder sur les champs de saisie de type texte.

```
WKYJS.setPlaceholder("Veuillez saisir le nom");
// affiche 'Veuillez saisir le nom' à l'intérieur du champ de saisie lorsque celui-ci n'est pas renseigné
```

La méthode autoCompleteSQL

```
WKYJS.autoCompleteSQL({
    "designerName":"...",
    "minLength":2,
    "sqlQuery":"...",
    "dataSource":"...",
    "dataField":"...",
    "select":function(value, dataItem) {

    }
});
```

Cette méthode permet de sélectionner une valeur dans un domaine de valeur externe à Workey. Ce domaine est calculé en fonction du texte saisi par l'utilisateur et est le résultat d'une requête SQL, d'où le nom de la méthode.

Pour implémenter cette fonctionnalité, dans le désigner, il faut l'appliquer à un champ de type Ligne simple monovalué. Le champ sera alors transformé en champ avec complétion automatique.

Pour ce faire, il faut alimenter la méthode avec un objet javascript composé de plusieurs attributs :

designerName : cet attribut contient le nom interne du champ Ligne simple à utiliser dans le formulaire

minLength : cet attribut spécifie le nombre de caractères à saisir avant

l’exécution de la requête SQL qui va peupler la liste

sqlQuery : cet attribut contient la requête SQL. Elle doit contenir le placeholder *VALUE* pour le substituer à la saisie de l’utilisateur.

dataSource : cet attribut spécifie la source de données, c’est-à-dire la base de données à utiliser.

dataField : cet attribut spécifie la colonne à utiliser qui est retournée par la sélection d’une entrée dans la liste.

select : cet attribut permet de spécifier la fonction de callback qui est utilisée lors de la sélection d’une entrée dans la liste. La fonction comporte deux paramètres :

- la valeur sélectionnée (en accord avec l’attribut *dataField*)
- la ligne correspondante à la valeur sélectionnée (permet de peupler une fiche par ex)

Ci-dessous un exemple qui, à l’ouverture du formulaire, construit un champ autoComplete sur le champ RechDossier.

```
WKYJS.onLoad(function () {

    WKYJS.autoCompleteSQL({
        "designerName":"RechDossier",
        "sqlQuery":"select Nom as Toto, Prenom, Age, isAdmin, Description,
Passion from aaa where Nom like '%*VALUE*%' or Prenom like '%*VALUE*%' order
by Nom, Prenom",
        "dataSource":"DB_Clients",
        "dataField":"Toto",
        "select":function(val, dataItem) {
            WKYJS.getField("Nom").value(dataItem.Toto);
            WKYJS.getField("Prenom").value(dataItem.Prenom);
            WKYJS.getField("Age").value(dataItem.Age);
            WKYJS.getField("Description").value(dataItem.Description);
        }
    });

});
```

On peut remarquer que la requête utilise un alias Toto pour la colonne Nom, c’est donc pour cela qu’il est utilisé dans l’attribut dataField et la fonction *select*. Le but d’utiliser des alias sera nécessaire pour les requêtes complexes (jointure, group by…). Dans notre exemple l’alias est présent à but informatif, il n’est pas du tout nécessaire et aurait pu être retiré.

On peut remarquer que la fonction de callback select, récupère la ligne sélectionnée sous forme d’objet javascript qui permet ensuite de valoriser les champs d’une fiche. On notera que les attributs sont nommés avec le nom

des colonnes de la requête (ou alias si il y a).

La méthode autoCompleteLDAP

```
WKYJS.autoCompleteLDAP({

    "designerName":"RechUser",
    "minLength":2,
    "filterFields":["uid", "sn"],
    "moreFields":[ "description", "mobile"],
    "filterOpe":"contains",
    "dataField":"dn",
    "select":function(val, dataItem) {
        // Votre code ici
    }
});
```

Cette méthode permet de sélectionner une valeur dans un domaine de valeur issu du serveur LDAP utilisé par Workey. Ce domaine est calculé en fonction du texte saisi par l'utilisateur et est le résultat d'une requête LDAP, d'où le nom de la méthode.

Pour implémenter cette fonctionnalité, dans le désigner, il faut l'appliquer à un champ de type Ligne simple monovalué. Le champ sera alors transformé en champ avec complétion automatique.

Pour ce faire, il faut alimenter la méthode avec un objet javascript composé de plusieurs attributs :

designerName : cet attribut contient le nom interne du champ Ligne simple à utiliser dans le formulaire

minLength : cet attribut spécifie le nombre de caractères à saisir avant l'exécution de la requête LDAP qui peuple la liste

filterFields : cet attribut est un tableau JSON qui contient les noms des champs LDAP sur lesquels le filtre de recherche va s'appuyer.

moreFields : cet attribut est un tableau JSON qui contient les noms des champs LDAP à rajouter aux champs déjà récupérés par défaut.

FilterOpe : cet attribut spécifie le type d'action du filtre sur les champs de recherche. Il peut contenir comme valeur :
soit *contains* ou *startsWidth.* Pour *contains,* la valeur saisie doit être présente dans le contenu du champ. Pour startsWidth, la valeur saisie doit correspondre au début du contenu du champ.

dataField : cet attribut spécifie le champ LDAP qui est retournée par la sélection d'une entrée dans la liste.

select : cet attribut permet de spécifier la fonction de callback qui est utilisée lors de la sélection d'une entrée dans la liste. La fonction

comporte deux paramètres :

- la valeur sélectionnée (en accord avec l'attribut *dataField*)
- la ligne correspondante à la valeur sélectionnée (permet de peupler une fiche par ex)

Ci-dessous un exemple qui, à l'ouverture du formulaire, construit un champ autoComplete sur le champ *RechUser*.

```
WKYJS.onLoad(function () {

    WKYJS.autoCompleteLDAP({
        "designerName":"RechUser",
        "filterFields":["uid", "sn"],
        "moreFields":["description", "mobile"],
        "minLength":2,
        "filterOpe":"contains",
        "dataField":"dn",
        "select":function(val, dataItem) {
            WKYJS.getField("Result").value(wky.jstr(dataItem));
        }
    });

});
```

On peut remarquer que la fonction de callback select, récupère l'utilisateur LDAP sélectionné sous forme d'objet javascript comme dans la méthode autoCompleteSQL. Dans notre cas il sera sérialisé et valorisé dans le champ *Result*.

Attention, si vous spécifiez des champs additionnels à récupérer, par l'attribut *moreFields*, ces derniers seront présents dans la réponse dans l'attribut *fields* de la réponse.

Exemple de réponse :

```
{
    "cn": "Jean Sérien",
    "mail": "jserien@efalia.com",
    "type": "PERSON",
    "dn": "jserien",
    "fields": {
        "mobile": "+33 6 11 11 12 12",
        "description": "Utilisateur Jean Sérien"
    }
}
```

La méthode enabled

Cette méthode permet d'activer ou désactiver un champ. Bien sûr lorsque le

champ est désactivé, il n’est plus possible de modifier sa valeur.

```
WKYJS.enabled("Email", true);
// Active le champ Email

WKYJS.enabled("Email", false);
// Désactive le champ Email
```

La méthode required

Cette méthode permet d’ajouter ou de supprimer le caractère obligatoire d’un champ.

```
WKYJS.required("Email", true);
// Active la saisie obligatoire du champ Email

WKYJS.required ("Email", false);
// Désactive la saisie obligatoire du champ Email
```

La méthode hideColumnInDynTab

Cette méthode permet de cacher une colonne d’une table dynamique.

```
WKYJS.hideColumnInDynTab("Table_des_personnes", "Age");
// Cache la colonne Age de la table dynamique Table_des_personnes
```

La méthode showColumnInDynTab

Cette méthode permet d’afficher une colonne d’une table dynamique.

```
WKYJS.showColumnInDynTab("Table_des_personnes", "Age");
// Affiche la colonne Age de la table dynamique Table_des_personnes
```

La méthode getColumnValuesInDynTab

Cette méthode permet de récupérer les valeurs d’une colonne appartenant à une table dynamique.

```
WKYJS.getColumnValuesInDynTab("Table_des_personnes", "Age");
// Retourne les âges de la table dynamique Table_des_personnes
// valeurs retournées sous forme de tableau [25, 33, 40, 18]
```

La méthode setColumnValuesInDynTab

Cette méthode permet de valoriser les cellules d’une colonne appartenant à une table dynamique.

```
WKYJS.setColumnValuesInDynTab("Table_des_personnes", "Age", [40, 22, 55,
19]);
// Dispatch chacune des valeurs du tableau dans les cellules de la colonne
// Attention le nombre de valeurs doit être égale au nombre de lignes de la
table.
```

La méthode addRowInDynTab

Cette méthode permet d’ajouter une ligne à une table dynamique.

```
WKYJS.addRowInDynTab("Table_des_personnes", {"Prenom":"Jean", "Nom":"Durand",
"Age", 33}, true);
// Ajoute une personne à la table Table_des_personnes.
// Attention le deuxième paramètre doit contenir toutes les valeurs de champ
déclarées dans la définition de la table.
// Le troisième paramètre force l'évènement de modificiation.
```

La méthode getFieldLabel

Cette méthode permet de récupérer le libellé d’un champ.

```
WKYJS.getFieldLabel("Nom");
// Récupère le libellé du champ sous forme de chaine de caractères
```

La méthode setFieldLabel

Cette méthode permet de modifier le libellé d’un champ.

```
WKYJS.setFieldLabel("Nom", "Nom de la personne");
// Remplace le libellé du champ par celui passé en paramètre
```

La méthode autoRefreshDynTab

Cette méthode permet de modifier le caractère auto-refresh d’un tableau dynamique.

```
WKYJS.autoRefreshDynTab("Table_Dyn", true);
// Activation de l'auto-refresh pour la table Table_Dyn
```

La méthode previewAttachment

Cette méthode permet de prévisualiser un contenu d’une pièce jointe Workey.

```
WKYJS.previewAttachment("Piece_jointe_1", 0);
// prévisualise le premier contenu de la pièce jointe "Piece_jointe_1"
// le deuxième paramètre est optionnel (0 par défaut), il permet de spécifier
le contenu visé.
```

Exemples de code Javascript

La liste des exemples de code javascript sera complétée au fur et à mesure.

Masquage de la section Prochain état avec l'ajout d'une barre de boutons pour soumettre le document.

```
// Masquage de la section Prochain Etat
WKYJS.hideStates();

// Ajout de la soumission
WKYJS.addButtonBar({
    "buttons":[
        {
            "label":"Valider",
            "type":"primary",
            "action":function() {
WKYJS.selectStateByOperationNameAndStateName("Traiter","Valide");
                WKYJS.submitDoc();
            }
        }
    ],
    "savable":false
});
```

Ajout d'une zone de prévisualisation d'une pièce jointe

```
WKYJS.onChange("Courrier_numerise" , function (dName) {

    $('[data-iframe-attchment-preview="' + dName + '"]').remove();
    let uuid = WKYJS.getField(dName).value();
    if (uuid != null) {
        WKYJS.getField(dName).element.append('<iframe data-iframe-attchment-
preview="' + dName + '"  width="100%" height="600" src="../api/attachment/' +
uuid + '/inline"></iframe>');
    }
});
```

Ajout d'un bouton de rafraîchissement du document

```
<button class="btn btn-primary" onClick="WKYJS.refresh()">Rafraîchir le
document</button>
```

Externalisation du Javascript

Si vous le souhaitez, en plus du Javascript dans les modélisations, il est possible d'en écrire dans des fichiers. Ces derniers seront chargés au

runtime comme s'ils étaient présents dans vos modélisations. Cette fonctionnalité a des avantages et des défauts :

Le pour :

- Plus confortable si votre projet regorge de Javascript.
- Permet l'utilisation d'un éditeur adéquate au développement.
- Permet de modifier le Javascript sans redéployer une nouvelle version de processus.
- Factorise les javascripts des sections techniques à un seul endroit.

Le contre :

- Il faut pouvoir accéder au système de fichiers des clients où Workey est installé.
- Il faut assurer manuellement les versions de fichiers avec celles des modélisations.
- En cas de maintenance il faut pouvoir récupérer ces fichiers.

Il ne faut pas externaliser systématiquement son Javascript, on peut vite comprendre qu'un projet qui utilise très peu la bibliothèque WKYJS sera plus facile à maintenir sans externaliser.

Localisation des fichiers

Tous les fichiers sont stockés dans des sous-répertoires eux même contenus dans le répertoire */$WORKEY-DATA$/external-js/*.

Les sous répertoires conteneurs respectent une nomenclature/une structure de la forme */$WORKEY-DATA$/external-js/processName/documentType/formName* où *procesName* est le nom interne du processus concerné, *documentType* est le nom interne du document concerné et *formName* le nom interne du formulaire concerné. On peut déjà comprendre que ces sous répertoires vont permettre une granularité d'utilisation du javascript.

Par exemple dans un projet de gestion de contrat, nous pouvons avoir la structure suivante :

/$WORKEY-DATA$/external-js/Gestion_des_contrats/Contrat/F_enregistrement_CTR_SIEGE.

La structure des sous-répertoires se construit automatiquement et au fur et à mesure de l'affichage des formulaires dans l'application.

Granularité d'utilisation

De par le dépôt d'un fichier dans un des sous-répertoires va découler son utilisation. En d'autres termes :

- Si un fichier se situe dans le répertoire *processName*, tous les formulaires associés au processus utiliseront le Javascript du fichier.
- Si un fichier se situe dans le répertoire *documentType*, tous les

formulaires associés au document utiliseront le Javascript du fichier.
- Si un fichier se situe dans le répertoire *formName*, seul le formulaire concerné utilisera le Javascript.

Pour que les fichiers soient lus automatiquement, il faut qu'ils respectent certaines règles de nommage.

Pour qu'un fichier soit utilisé quel que soit le processus, il faut créer et placer le fichier *__all_processes.js* dans le répertoire */$WORKEY-DATA$/external-js/*.

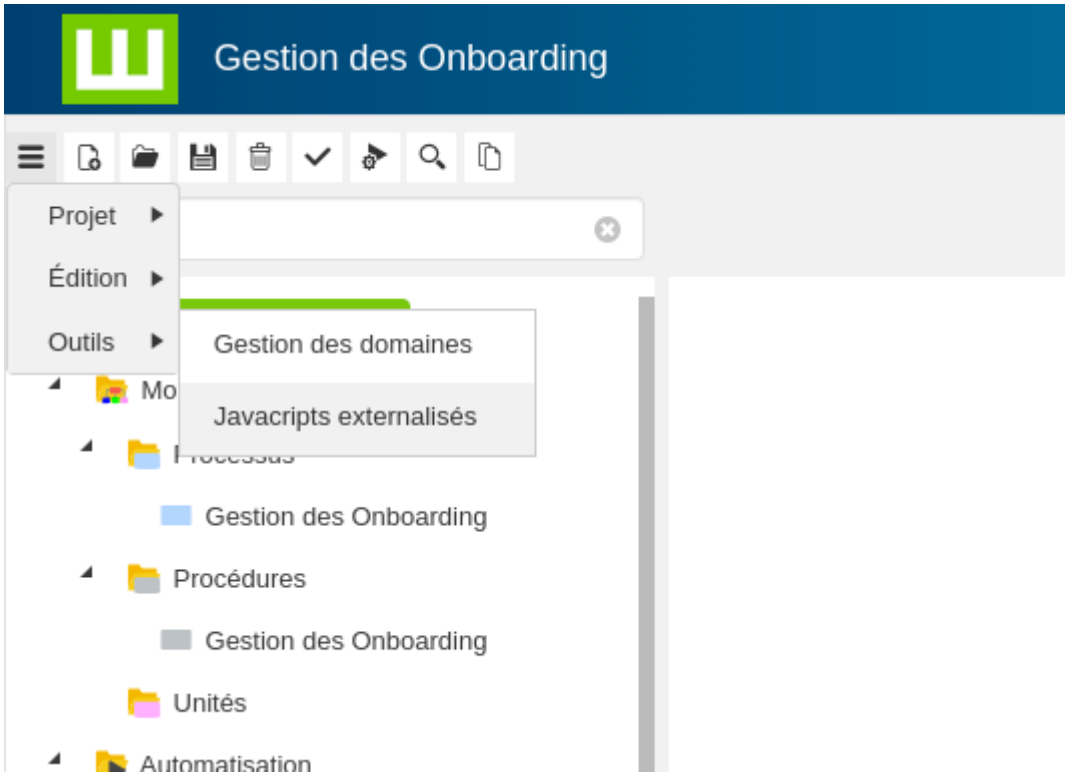
Nommage des fichiers

Le nom des fichiers est normé, il doit être en minuscule, se nommer *process.js* dans le répertoire du processus, se nommer *document.js* dans le répertoire document et pour être original, *form.js* dans le répertoire du formulaire.

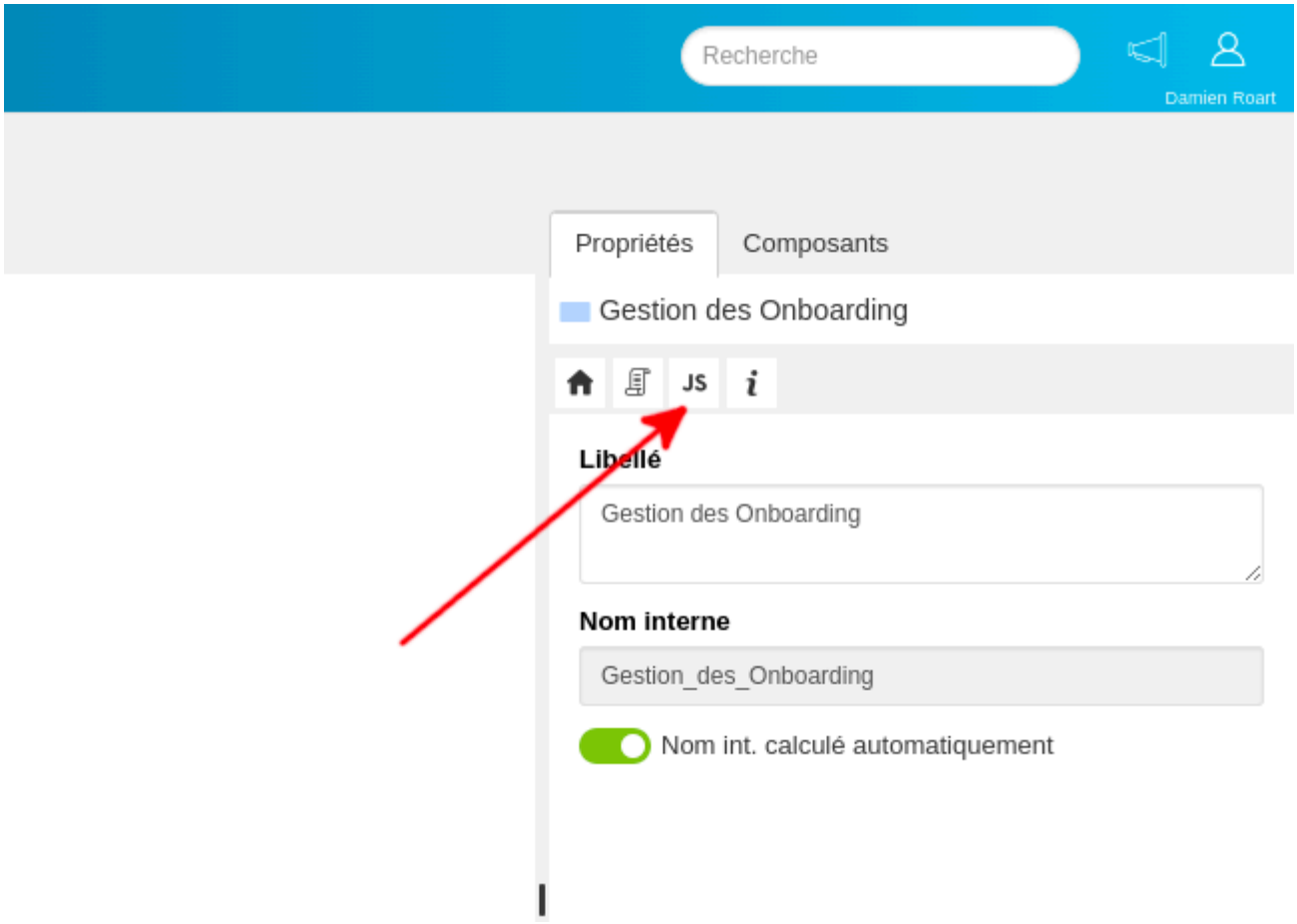
Nommés de telle manière, les fichiers sont chargés quelle que soit la version de processus déployée. Si vous souhaitez avoir un fichier Javascript particulier pour une version de processus donnée, il faut rajouter le numéro de version dans le fichier. Par exemple pour un document de la version 5 d'un processus, il faut nommer le fichier *document.5.js*. A savoir que si un fichier Javascript est versionné, les versions supérieures de processus l'utiliseront jusqu'à ce qu'un nouveau numéro de version apparaisse. Il est évident qu'un fichier versionné avec un numéro de version supérieur au processus en cours sera ignoré.

Edition des fichiers via le Designer

Le Designer propose une interface de gestion des javascripts externalisés. Pour un projet en cours de modélisation, vous pouvez y accéder par le menu déroulant principal.

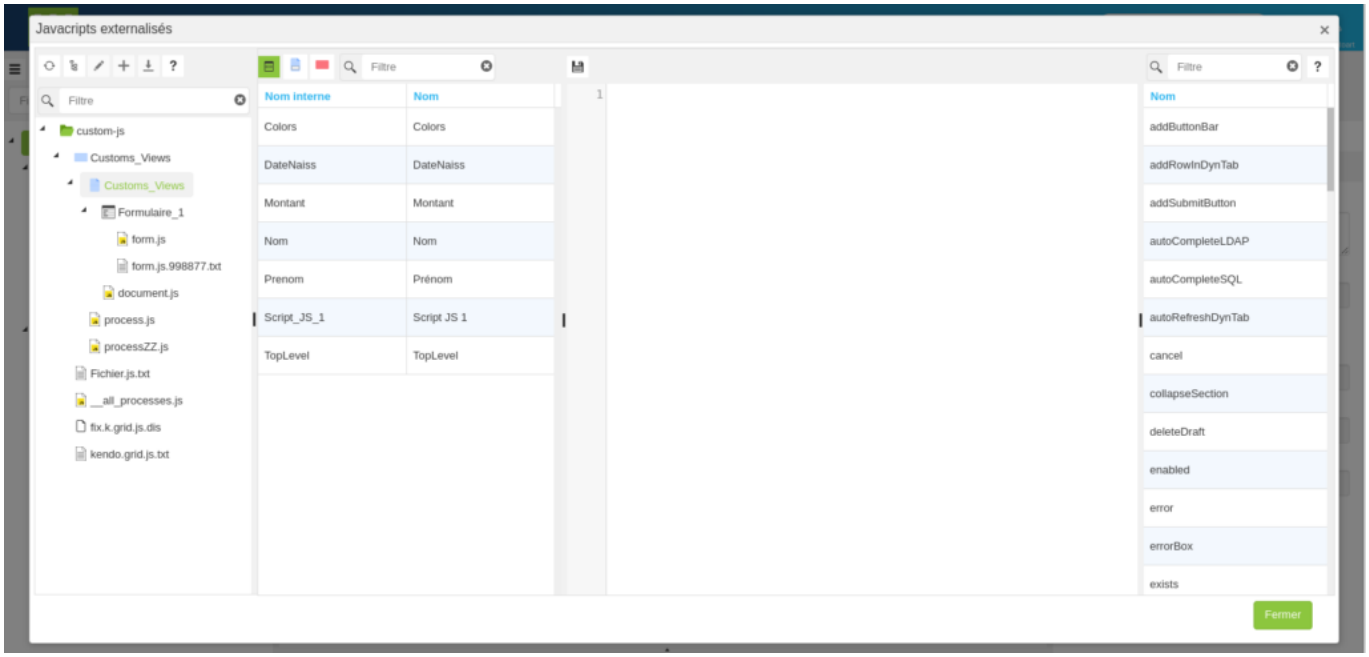


Vous pourrez aussi y accéder depuis le panneau des propriétés des objets Workey qui autorisent les javascripts externes. Il y a trois types d'objets qui permetttent cela, ce sont : les processus, les documents et les formulaires.



La fenêtre des **javascripts externalisés** vous permet de créer la structure sur

le système de fichier du serveur, de créer / modifier des fichiers, de télécharger des fichiers ou tous les fichiers. Pour des raisons de sécurité vous ne pouvez rien supprimer. Pour pouvoir faire cela, il vous faudra avoir accès au serveur Workey et le faire manuellement.



Annexe

Localisation d’un libellé

Il est possible sous forme d’un objet JS de localiser un libellé lorsque la fonctionnalité est prévue. Il suffit de spécifier en clé la langue et en valeur le libellé souhaité correspondant à la langue.

Par exemple pour spécifier le français et l’anglais pour un libellé qui exprime la sauvegarde des changements, il faut définir l’objet JS suivant :

```
{
    "fr-FR":"Sauvegarder les modifications",
    "en-US":"Save changes"
}
```