DRAFT

JavaScript Quickstart

In diesem Wiki-Eintrag sollen die wichtigsten Aspekte von JavaScript-Einbindung in TIM beschrieben werden.

1. Einbinden einer eigenen Scriptdatei

Im Gegensatz zu herkömmlichen Web-Design MUSS bei TIM Script und HTML Code getrennt werden. HTML Code wird im Prozessmodell hinterlegt, der JavaScript Code wird in der custom.js Datei abgelegt. Wenn man nun sein Code hinterlegen will, muss man in dem „Administrations-client“ in dem „Resources“- Tab die JavaScript-Datei hochladen, mit dem Namen: „custom.js“.
Diese gilt allerdings für den gesammten Mandanten, um JavaScript in einen bestimmten Prozess einzubinden, muss in der Smartform eine „initMethod“ hinterlegt werden, diese muss einzigartig benannt werden. Dies geschieht in der ersten Zeile der Smartform (also des HTML-Codes) wie folgt:
`<form class="example-process" initMethod="REPLACEME" > [...] </form>`

```
<form class="form-quickstart" data-bootstrap="true"
validationMethod="validateQuickstart" initMethod="initQuickstart"> [...]
</form>
```

2. Aufbau einer Scriptdatei

2.1. Init-Methode

In der custom.js Datei muss die Init-Methode wie folgt benannt werden:
`gadget.functions.REPLACEME = function(){ }`

2.1.1. JQuery beziehen

Eine Smartform in TIM ist etwas anders aufgebaut als eine herkömmliche Internetseite, ergibt eine `document.getElementById("ID");` Suche nicht das gewünschte Ergebnis. Um mit diesem Ansatz ein richtiges Ergebnis zu bekommen, muss man mit dem this-Pointer arbeiten, dies kann allerdings schnell zu Verwirrung führen. Um trotzdem z.B. Felder in der Smartform nach der ID suchen zu können, wird JQuery empfohlen. Um dieses zu benutzen müssen wir es allerdings erst einbinden, dies geschieht so:
An den Anfang der Init-Methode muss folgende Zeile:

```
jq = (this.form.ownerDocument.defaultView!=null) ?
```

```
this.form.ownerDocument.defaultView.jQuery :
this.form.ownerDocument.parentWindow.jQuery;
```

2.1.2. Eigenen Code einbauen

Auch hier gibt es Unterschiede zum herkömmlichen Webdesign:
Zum Beispiel werden Funktionen mit dem Präfix `gadget.functions.` angegeben (siehe Beispiel).
Außerdem muss man beachten, dass die Init-Funktion einmal aufgerufen wird, während die Smartform geladen wird. Das macht sie zu einem guten Ort um wiederkehrende Skriptpassagen einzubauen, zum Beispiel kann ein Feld immer wieder mit dem aktuellen Datum befüllt werden. Dafür fügen wir unserer Init-Methode folgenden Code hinzu:

```
//Ein neues Datumsobjekt wird erzeugt
    var d = new Date;

    //Das Textfeld mit der ID "date" wird mit dem wert von d befüllt (!Jedes
mal wenn die Smartform geöffnet wird!)
    jq("#date").val(d);
```

2.2. Validate-Methode

Die Validate-Methode ist die Funktion die aufgerufen wird sobald eine Smartform gespeichert wird, d.h. wenn man auf den Button „Prozess starten“, „Speichern“ oder „Speichern und Aufgabe abschließen“ klickt. Wie der Name bereits sagt, ist diese Methode dazu gedacht, Überprüfungen an den Smartformeingaben durchzuführen bevor diese gespeichert werden. Um eine Validate-Funktion mit der Smartform zu verknüpfen muss sie ähnlich wie die Init-Methode in der Smartform benannt werden, z.B. so:
`<form class="example-process" initMethod="initQuickstart" validationMethod="CHANGEME"> [...] </form>`

```
<form class="form-quickstart" data-bootstrap="true"
validationMethod="validateQuickstart" initMethod="initQuickstart"> [...]
</form>
```

In diesem Beispiel soll geprüft werden, ob die beiden Email-Adressen gleich sind, wenn sie es nicht sind, soll der User darüber informiert werden und die Aufgabe soll nicht abgeschlossen werden.
Um dies zu bewerkstelligen, werden zuerst die Werte beider Felder verglichen, falls diese nicht übereinstimmen, wird ein „Alert“ ausgegeben und anschließend „false“ zurückgegeben. Denn wenn die Validate-Methode „false“ zurückgegeben wird, nimmt TIM an, die Überprüfung hat Fehler ergeben und speichert die Änderungen nicht/lässt den User die Aufgabe nicht abschließen.
Die Validate-Methode muss einen Boolean-Wert als „return“ liefern.

```
gadget.functions.validateQuickstart = function(){
```

```
    //Wenn die 2 Adressen nicht gleich sind, wird der User per "Alert"
informiert, anschließend "returned" die Methode "false", somit werden die
Änderungen nicht gespeichert/die Aufgabe nciht abgeschlossen
    if(jq("#email_1").val() != jq("#email_2").val()){
        alert("The emails don't match!");
        return false;
    }else {
        //Hier wird "true" zurückgegeben, somit nimmt TIM an, die Smartform
wurde überprüft und ist Fehlerfrei
        return true;
    }
}
```

3. Prozessvariablen

TIM pflegt eigene Variablen, sogenannte Prozessvariablen. Diese sind an die Prozessinstanz, in der sie erstellt wurden gebunden und können per JavaScript bezogen und geändert werden. Um Unklarheiten mit dem this-Pointer zu vermeiden wird empfohlen, in der Init-Methode eine globale Variable (tp) zu erstellen auf die der this-Pointer gelegt wird (siehe Beispiel). Im Beispiel wird in der Init-Methode eine Funktion an den Button mit der Beschriftung „Safe the adress to the database“ angehängt. Sobald der Button gedrückt wird, wird der Inhalt eines Textfeldes, „email_1“, in eine Prozessvariable abgelegt. Wichtig ist die Zeile `tp.entity.mergeLocal(true);`. Sie kommt einem „commit“-Befehl zugleich, ist also unabdingbar, wenn die Änderungen in das Entity übernommen werden. Die Änderungen im Entity werden beim Speichern (z.B. Ableßen einer Aufgabe) in die Datenbank übernommen. Es reicht, einen Merge-Befehl an das Ende von mehreren Variablen-Änderungen zu stellen, da ein Mal mergen alle Änderungen miteinbezieht.

```
    //Auf den Button mit der ID "compare_adress" wird eine Funktion gelegt,
die bei Klick ausgelöst wird
    jq("#compare_adress").on("click", function () {
        //Eine Prozessvariable wird gesetzt: Der Name ist "email" und als
Inhalt hat sie den Inhalt von dem Feld mit der ID "email_1"
        tp.entity.setValue("email", (jq("#email_1").val()));
        //Die Änderungen werden "commited"
        tp.entity.mergeLocal(true);
    })
```

Die Prozess-Variable „email“ kann nun z.B. für einen Actionhandler verwendet werden.
Will man eine Prozessvariable auslesen, so muss man folgenden Befehl verwenden:
`tp.entity.getValue("CHANGEME");` (Siehe 4.)

4. Methodenaufrufe

Will man nun Methoden selbst schreiben, die während der Init-Methode benutzt werden sollten (z.B. damit sie „on click“ auf einen Button gelegt werden), muss man auch wie bei der Init-Methode das Präfix `gadget.functions.` an den Funktionsnamen und den Funktionsaufruf anhängen. (Siehe Beispiel)

In diesem Beispiel soll das auslesen von Prozessvariablen in eine eigene Methode ausgelagert werden. Die Parameterübergabe funktioniert hier wie üblich.

```
   //Eine Funktion wird "on click" auf den Button mit der ID
"get_process_variable" gelegt
    jq("#get_process_variable").on("click", function () {
        gadget.functions.getVar(jq("#param").val());
    })
```

```
//Diese Funktion liest eine Prozessvariable aus
gadget.functions.getVar = function (param) {
    alert(tp.entity.getValue(param));
}
```