Eigene Widgets mit iobroker.vis entwickeln

21. März 2019

Seit vielen Jahre rüste ich, teilweise zum Entsetzen meiner Frau, unser Haus mit Smart-Home Funktionen auf: Lichter, Dimmer, Jalousien, Sensoren mit Wifi, Homematic und Zigbee-Funkstandards.

Ursprünglich habe ich das System mit FHEM betrieben, das wurde mir aber mit der Zeit etwas zu komplex. Vor einem Jahr bin ich daher auf ioBroker umgestiegen, dessen Basistechnologien mit JavaScript/TypeScript und Node.js doch etwas aktueller ist.

Seit einem Jahr habe ich dieses System sehr stabil im Betrieb und wollte nun die „reizarme Umgebung“ meines Dänemark-Urlaubs auch dazu nutzen, eine ansprechende Visualisierung aufzubauen.

Erste Entscheidung: baue ich ein eigenes Frontend mit irgendeinem schicken Framework oder nutze ich VIS, die Visualisierung, die mit ioBroker schon mitkommt? Da man mit VIS doch recht schnell und komfortabel eigene Oberflächen „zusammenklicken“ kann, hab‘ ich mich entschieden, VIS zu verwenden und dort zu erweitern, wo mir etwas fehlt.

VIS wurde vom ioBroker-Guru „Bluefox“ schon vor einigen Jahren entwickelt, ist sehr mächtig – dafür aber auch ziemlich komplex und nicht sehr gut dokumentiert. Auch scheint die Community, die tatsächlich eigene Widgets entwickelt, recht überschaubar. Dennoch: Bluefox hat seine Sache gut gemacht und wenn man sich anhand eigener Entwicklungsbeispiele hinreichend mit dem vorhandenen Code beschäftigt hat, ist das Ganze doch ganz gut erweiterbar.

Als, zugegeben schon etwas betagter, Informatiker komme ich recht schnell in solche Systeme rein, bin aber seit Jahrzehnten schon kein Programmierer mehr, sondern eher lehrend tätig – bleibe also immer etwas an der Oberfläche der Praxis. Ich habe ca. 7 Tage für das gebraucht, was ich hier beschreibe und hab‘ viel weniger hinbekommen als ursprünglich geplant. Tatsächlich ging es mir am Ende nur noch um die grundsätzliche Funktionsweise anhand eines einzigen Use-Cases.

Bei meinem Vorgehen habe ich mir folgende Fragen bzw. Aufgaben gestellt:

1. Welcher Use-Case ist hinreichend komplex, das System „einigermaßen“ zu verstehen, hinreichend einfach, um nicht alles verstehen zu müssen und pragmatisch genug, um ihn auch umsetzen zu können? („Das Projekt“)
2. Welche Werkzeuge sind zur Umsetzung des Projektes nötig und geeignet? („Die Entwicklungsumgebung“)
3. Wie bekomme ich ein Widget als Kopie eines vorhandenen Widgets im ioBroker zum Laufen? („Ein erstes Widget“)
4. Wie bekomme ich ein selbsterstelltes Widget im ioBroker zum Laufen? (Ein erstes eigenes Widget“)
5. Wie gehe ich mit Fehlern um? („Fehlerbehandlung“)
6. Wie programmiere ich meinen Use-Case? („Das Dimmer-Widget“)

(Legal) Disclaimer

Ich beschreibe hier meine eigenen Erfahrungen und übernehme keine Verantwortung dafür, wenn Sie sich etwas zerschießen. Zudem beschreibe ich **meine** Arbeitsschritte. Möglicherweise gibt es geschicktere Wege. Und: Ich habe nur Einblick in einige Teile des Systems gewonnen, so dass außen herum noch vieles „magic“ ist. Wahrscheinlich hab‘ ich sogar einige dieser „magic“-Sachen wegoptimiert.
Dabei hatte ich an vielen Stellen kleinere und größere Probleme und bin im Forum oft fündig geworden. Mit diesem Text möchte ich etwas zurückgeben;-).
... und, natürlich, kann diese Beschreibung das ioBroker-WIKI (github.com/ioBroker/ioBroker/wiki) und das Forum in keiner Weise ersetzen.
... und: das Ganze hat sich als nicht ganz dünnes Brett erwiesen, daher ist der Text auch etwas länglich geworden und braucht für das Verständnis schon ein wenig Programmierkenntnisse.

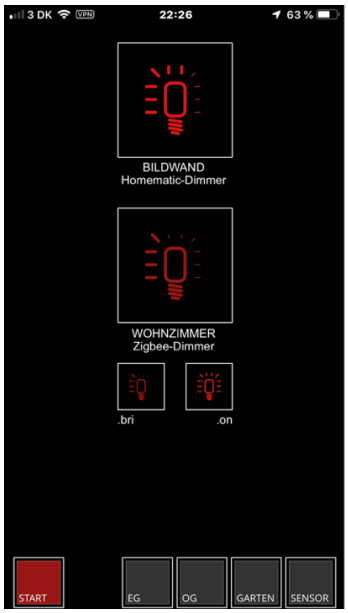
Das „Projekt“

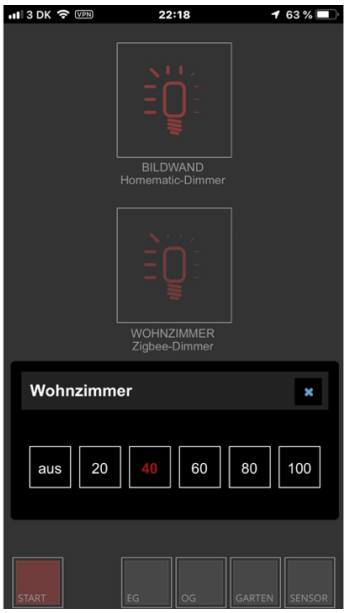
Mein Use-Case, den ich exemplarisch als eigenes Widget umsetzen wollte, ist ein benutzerfreundlicher Button für einen Dimmer.

Dieser soll den aktuellen Dimmzustand anzeigen und beim Antippen den Dimmer entweder auf einen vordefinierbaren Wert (z.B. 60%) ein- oder, falls er schon zumindest ein wenig angeschaltet ist, abschalten. Dabei sollen neben den „üblichen“ Homematic-Komponenten auch Zigbee-Dimmer schaltbar sein.

Zusätzlich soll sich zum Einstellen eines beliebigen Dimmwertes über einen Doppelclick (bzw. ForceTouch) ein Dialog öffnen, in dem ein Dimmwert aus vordefinierbaren Werten auswählbar ist und der, anders als bei „normalen“ Dialogen, schließt, sobald ein Wert ausgewählt wurde.

Das Ganze muss so einfach zu bedienen sein, dass auch ein Nicht-Nerd (z.B. meine Frau) gerne damit umgeht. Und es muss gut aussehen, also keine runden, farbigen, 90erJahre-Schieberegler oder Computer-3D-Knöpfe haben.





Die Entwicklungsumgebung

Wie erstellt man nun ein neues Widget und wie bekommt man es in die laufende ioBroker-Umgebung?

Ich hab‘ mich für webStorm (www.jetbrains.com/webstorm/) als IDE entschieden. WebStorm ist sehr mächtig, für Privatprojekte (meines Wissens) kostenfrei und für Web-Projekte mittlerweile eigentlich „Standard“. Wir setzen es daher auch für die Lehre ein.

Obwohl ich TypeScript präferiere und TypeScript im ioBroker auch an einigen Stellen verwendet wird, soll das Ganze mit JavaScript realisiert werden, da VIS auch mit „Vanilla“-JS umgesetzt wurde. Innerhalb der Widgets wird zudem EJS (ejs.co) als Template-Generator verwendet.

Zum Installieren wird der Paketmanager von node (www.npmjs.com) eingesetzt. Zum Einspielen in den ioBroker verwende ich die ioBroker-Admin-Oberfläche, hab‘ aber teilweise auch mit den shell-Kommandos gearbeitet (GitHub.com/ioBroker/ioBroker/wiki/Console-commands).

Als Versionsverwaltung verwende ich, wie ioBroker selbst auch, GitHub. Anleitungen dazu gibt es zuhauf im www, zudem wird Git durch die webStorm-IDE unterstützt.

Anders als im ioBroker-wiki (github.com/ioBroker/ioBroker/wiki) beschrieben, hab‘ ich den ioBroker nicht lokal auf meinem Mac installiert, sondern teste „am offenen Herzen“ in meiner ioBroker-Installation auf dem RasPi. Im Nachhinein hätte ich mir die Mühe mit der Installation auf meinem Mac doch machen sollen, da ich seeeehr viel zu debuggen hatte und das in webStrom sicher komfortabler als in der Chrome-Entwicklungsumgebung ist ... sei’s d’rum – direkt in der Laufzeitumgebung zu arbeiten hat auch Vorteile: da weiß man, wenn es tut, dass es tut.

Wie sieht also meine Arbeitsumgebung aus:

1. Ein webStorm-Fenster
2. Ein Chrome-Browser mit folgenden Tabs.
    a. GitHub zum Nachschauen im eigenen Projekt und in den ioBroker-Projekten
    b. ioBroker-Oberfläche: Adapter zum Installieren und Updaten des eigenen Widgets
    c. ioBroker-Oberfläche: Instanzen zum Löschen der Adapter-instanz und Starten der VIS-Oberfläche
3. Optional: Ein Terminal-Fenster zum schnelleren Aufruf der ioBroker Kommandos.

Ein erstes Widget

Als Vorlage für mein Projekt habe ich mich für iobroker.vis-jqui-mfd (ebenfalls von Bluefox) entschieden. Dieses Widget baut auf dem bereits eingebauten VIS-Widgets auf (konkret auf: „jqui“) und liefern bereits viele der Funktionalitäten, die ich brauche (konkret: das „dimmer – jquery Dialog- Widget). Alternativ bietet das vis-metro Widgets ebenfalls das, was ich brauche und ist noch vielseitiger zu konfigurieren. Insgesamt war mir das für den Anfang aber dann fast zu vielseitig und damit zu komplex.

Ich habe mir also *vis-jqui-mfd* aus dem original GitHub-Projekt in eine eigenes neu erstelltes GitHub-Projekt kopiert. Die Option zum Kopieren eines anderen Projektes wird beim Anlegen eines neuen Projektes in GitHub angeboten. Das Ganze dauert etwas (ca. 5 Minuten), man bekommt aber eine mail, sobald es fertig ist.

Wie bekommt man nun ein erstes Widget in den laufenden ioBroker?
Im Forum (forum.iobroker.net/topic/19929/getting-started-with-visualisations/3) findet man eine schöne Anleitung, wie man direkt im Live-System Änderungen vornimmt. Ich entwickle aber gerne außerhalb des Live-Systems in meiner gewohnten IDE und spiele dann das fertige „Widget mit den üblichen ioBroker-Bordmitteln in das Live-System – so wird's mit fertigen Widgets dann ja auch gemacht. Der Nachteil: das Ganze dauert ein wenig – aber ich nutze die Zeit der Installationen und Updates immer schon, um den nächsten Implementierungsschritt vorzubereiten.
Da das Aufspielen eines Widgets während der Entwicklung ständig vorkommt, beschreibe ich es hier etwas ausführlicher:

1. Wenn man Veränderungen vornehmen möchte (bei der ersten Kopie ist das erstmal nicht nötig), kann man das in webStorm tun, muss diese Veränderungen in GIT „commiten" und dann auf GitHub „pushen".
   Dabei nicht das Hochzählen der Versionsnummern vergessen (-> später mehr)
2. Falls das Widgets bereits installiert ist und als Adapter auf dem ioBroker bereits läuft, muss man dieses erst löschen, bevor man das Neue verwenden kann. Dasselbe gilt für jede neue Version des neu zu entwickelnden Widgets.
   Dazu sucht man sich im ioBroker/Instanzen-Fenster den Adapter und löscht ihn über das Papierkorb-Icon.
3. Installation des neuen Widgets aus GitHub
   Im ioBroker/Adapter-Fenster den Expertenmodus anschalten und über das „GitHub-Katzen-Icon" eine eigene Installationsdatei verwenden (Installationsart: „BELIEBIG" – nicht: „VON GITHUB").
   Als link verwendet man die URL des eigenen, neuen Projektes aus GitHub – den kann aus dem GitHub-Projekt herauskopieren (im GitHub-Projekt „Clone or download").
   Entfernt man die Option „Schließen wenn fertig", kann man die Version überprüfen, die von npm installiert wurde (die kommt aus *package.json*). Auch sollte man die installierte Versionsnummer aus ioBroker-Sicht überprüfen, die im ioBroker/Adpater-fenster angezeigt wird (die kommt aus *io-package.json*").
4. Upload des VIS-Adapters
   Im ioBroker/Adapter-Fenster den Upload-Button drücken (Im Expertenmodus: der Pfeil nach oben, hinter dem vis-Adapter).
   Manchmal bekommt VIS die Änderung eines Widgets nicht richtig mit, z.B. immer dann, wenn man Änderungen in der Adapter-Konfiguration vornimmt. Das liegt wohl auch am Caching-Mechanismus des VIS-Adapters. Man denkt dann, mit einer neuen Version des eigenen Widgets zu arbeiten, arbeitet aber noch mit der Alten. Um sicherzustellen, dass der VIS-Adapter alle Änderungen mitbekommt, sollte man den ebenfalls nochmals uploaden. Ich hab's vorsichtshalber immer gemacht – es dauert ca. 45 Sekunden.

Manchmal reicht auch das nicht. Dann muss man die VIS-Adapter Instanz löschen und neu anlegen (Vorsicht: dazu braucht man den VIS-Lizenz-Schlüssel).
In github.com/ioBroker/ioBroker/wiki/How-to-debug-vis-and-to-write-own-widget-set ist beschrieben, wie man das Caching ausschaltet. Das hab' ich gemacht – zum Editieren der json-Konfigurationsdatei im Termin „*sudo*" nicht vergessen ;-)
... mit dieser Einstellung konnte ich vermeiden, die VIS-Adapter-Instanz ständig neu löschen und neu anlegen zu müssen

5. Start des VIS-Editors:
   z.B. über den "Pfeil-rechts" neben dem VIS-Adapter.
6. Starten der VIS-Runtime:
   Pfeil/Menü rechts oben im VIS-Editor.

Alternativ kann man das Ganze auch direkt im Terminalfenster des ioBroker-Servers im Verzeichnis */opt/iobroker* machen: z.B. Installation mit: *npm install GitHub.com/xyz/projekt/ tarball /master* oder alternativ *.../iobroker url https://GitHub.com/projekt.git*. Auch alle anderen Kommandos, wie z.B. das updaten geht mit ioBroker Shell-Kommandos. Nach einer manuellen Installation eines Adapters muss dieser noch „ge-uploaded" werden.
Aber Vorsicht: bei dieser Art der Installation wird der eigenen Server-User verwendet, so dass der iobroker aus der ioBroker-Oberfläche dort manches nicht mehr installieren kann. Damit die ioBroker-Oberfläche wieder schreibenden Zugriff bekommt, muss man den User bzw. die Gruppe wieder auf „iobroker" stellen (mit *sudo chown -R iobroker:iobroker projekt*). Wenn man zu viel im Terminal gearbeitet hat, „verbiegt" man ganz gerne die Schreibberechtigungen. Diese sollten beim ioBroker liegen, was man zur Not mit *sudo chown -R iobroker:iobroker /opt/iobroker* erzwingen kann (aber nur, wenn man genau weiß, was man da tut ;-)

Ein erstes eigenes Widget

... jetzt hätten wir also eine Kopie eines vorhandenen Widgets aufgespielt – wie bekommen wir nun einen neues?

Dafür gibt es im ioBroker einen „Template"-Generator, der JavaScript/TypeScript-Adpater, Widgets und Icon-Set-Templates generieren kann (github.com/ioBroker/ioBroker.template).

Den hab' ich verwendet, um mir ein Verzeichnis mit einer korrekten Widget-Struktur zu erzeugen. Als Name habe ich „vis-jqui-dimmer" verwendet. Alternativ gibt es auch ein web-Interface (adapter-creator.iobroker.in), welches ich aber nicht ausprobiert habe. Das erzeugte Verzeichnis hab' ich mit GIT versioniert und auf GitHub hochgeladen. Wie das funktioniert kann man im WWW nachlesen (;-).

Der Templategenerator erzeugt eine Dateistruktur, wobei ich folgende Dateien angepasst habe:

- *widgets/vis-jqui-dimmer.html*:
  Die HTML-Struktur des widgets in ejs-Format. Oben im File stehen als Kommentar die möglichen Parameter für die Skript-Köpfe der einzelnen widgets.
- *widgets/vis-jqui-dimmer/js/vis-jqui-dimmer.js*
  Die (einige) JavaScript-Funktionen, die in *vis-jqui-dimmer.html* aufgerufen werden,

z.B. eine Funktion zur Anzeige der Version (im Consolen-Fenster der Entwicklungssicht des Chrome-Browsers)

- *widgets/vis-jqui-dimmer/img*
  Bilder, die im Widget verwendet werden, z.B. die Bilder, die im VIS-Editor zur Auswahl des Widgets angezeigt werden sollen oder die Icons für den Dimmer-Zustand.
- *package.json*
  Installationsdatei für den *npm*. Dort wird die Versionnr. eingetragen, die beim Installationsvorgang (Ansicht: „mehr“) angezeigt wird.
- *io-package.json*
  Konfigurationsdatei für den ioBroker. Dort wird die Versionsnr. eingetragen, die im ioBroker angezeigt wird.

Als erstes hab‘ ich aus dem Original (*iobroker.vis-jqui-mfd/widgets/jqui-mfd.html*) ein „passendes“ widget *(<script id="tplMfdLightDialog" …*) in meine HTML-Datei (*widgets/vis-jqui-dimmer.html*) hineinkopiert und den script-header entsprechend der Anleitung im template angepasst:

```
<script id="tplDimmerCtrl"
    type="text/ejs"
    class="vis-tpl"
    data-vis-set="vis-jqui-dimmer"
    data-vis-type="dimmer,ctrl"
    data-vis-prev='<img src="widgets/vis-jqui-dimmer/img/Prev_DimmerCtrl.png"></img>'
    data-vis-name="Dimmer"
    data-vis-
attrs="oid/id;oid_on/id;min/number;max/number;turnOnValue;invert_icon/checkbox;asButton[tru
e]/checkbox;iconColor/color;"
    data-vis-
attrs0="group.colors;iconColor0/color;iconColor1/color;iconColor2/color;iconColor3/color;ic
onColor4/color;iconColor5/color;iconColor6/color;iconColor7/color;iconColor8/color;iconColo
r9/color;iconColor10/color;"
    data-vis-
attrs1="group.dialog;title;count[4];autoclose/slider,0,30000,100;modal/checkbox;dialog_widt
h[470];dialog_height[210];dialog_top;dialog_left;overflowX/nselect,,visible,hidden,scroll,a
uto,initial,inherit;overflowY/nselect,,visible,hidden,scroll,auto,initial,inherit;show_slid
er/checkbox;show_value/checkbox;units;"
>
```

Dort wurde geändert:

- Id
  Name des widgets. Das soll jetzt heißen: *tplDimmerCtrl*
- data-vis-set
  Der „WidgetSatz“ der im VIS-Edit links im Drop-Down angezeigt wird. Damit der angezeigt wird, **muss** VIS geupdated werden, denn VIS generiert seine internen Strukturen selbst und cached die (siehe auch Punkt 6 in „Ein erstes Widget“).
  Bei jeder Änderung im Scripßt-Header muss VIS geupdated werden.
- data-vis-prev
  Ein icon. Das hab‘ ich mit Photoshop neu erstellt

- data-vis-attrsX
  Die durch den Benutzer einstellbaren Parameter für das Widget

Zunächst hab‘ ich – eine alte Angewohnheit – den gesamten Code neu formatiert, einiges ‚was ich auf Anhieb verstanden habe kommentiert und, als erste inhaltliche Änderung, einige häufig vorkommenden Zugriffe als Initialisierung nach vorn gezogen und eigenen Variablen zugeordnet. Das waren insbesondere die im VIS-Editor eingestellten übergebenen Attribute bzw. Optionen. Daneben habe ich einige inline-CSS Style-Definitionen und harten
-Formatierungen entfernt, denn ich bin ein leidenschaftlicher Anhänger der Trennung von Struktur und Formatierung in Textbeschreibungen (alte LaTex-Schule;-)

Damit die Verweise (z.B. Bilddateien) auch noch alle funktionieren hab‘ ich alle *„jqui-mfd“* durch *„jqui-dimmer“* ersetzt. Dabei muss man recht sorgfältig sein, denn der Template-Generator erzeugt teilweise eine etwas andere Struktur als die im Original-Verzeichnis. Manchmal muss man daher auch durch *„vis-jqui-dimmer“* ersetzen, ansonsten bekommt man auch ‘mal kein Bild angezeigt oder, schlimmer, Fehlermeldungen in der Entwickler-Console (siehe „Fehlerbehandlung“).

Im Codes des Widgets hab‘ ich einige *console.log(„Test X“)* eingefügt, um dann zu sehen, ob meine Änderungen auch tatsächlich angekommen sind.

Dann „nur“ noch:

1. die Versionen in widgets/vis-jqui-dimmer/js/vis-jqui-dimmer.js, package.json, io-package.json hochzählen,
2. das Ganze in GIT commiten und auf GitHub pushen
3. und wie in „Ein erstes Widget“ beschrieben auf den ioBroker bringen.
4. Den VIS-Editor starten,
5. das neue Widget auswählen,
6. in einen View ziehen,
7. parametrisieren
8. und dann VIS-Runtime starten (Button rechts oben im VIS-Editor).
9. Im Chrome-Fenster des VIS-Runtime die Entwicklersicht anschalten
10. und die Console öffnen.

Wenn alles gut gegangen ist, sieht man nun die console.log-Ausgaben. Wenn nicht, eine Fehlermeldung.

Fehlerbehandlung

Das erste worauf Sie achten **müssen**: Haben Sie die richtige Version eingespielt? Versionsnummern werden in drei oben schon beschriebenen Dateien verwaltet (*widgets/vis-jqui-dimmer/js/vis-jqui-dimmer.js, package.json* und *io-package.json*). Dort sollten die Versionsnummern unbedingt immer identisch sein. Beim Installieren des Adapters wird die Versions-

nummer vor dem Adapter angezeigt. Die Adapterinstanz muss gelöscht und neu erstellt werden. Schließlich wird die Versionsnummer beim Starten des VIS-Runtime-Fensters in der Entwicklerconsole (ganz oben) ausgegeben. Sind das die Versionsnummern, die Sie erwarten?

Wenn diese Nummern nicht wie erwartet sind:

1. Checken Sie, ob die Versionsnummern in Ihren Quelldateien wirklich hochgezählt und identisch sind.
2. Checken Sie, ob Sie Ihre Dateien commited **und** gepushed haben.
3. Löschen Sie die Instanz Ihres Widgets und instanziieren Sie ihr Widget nochmals.
4. Updaten Sie ihren VIS-Adapter
5. Löschen Sie die Instanz ihres VIS-Adapters und instanzieren Sie den VIS-Adapter neu – dazu brauchen Sie dann ihren VIS-Lizenz-Key.

Wenn ihre gestartete Version die Richtige ist, kann's endlich losgehen.

Ich hab' Stunden verbraucht, um vor allem Syntaxfehler zu beseitigen. Das Problem dabei ist: Ihr HTML-Code wird vom canJS-Framework ((v2.canjs.com) geparsed, welches EJS verwendet. Schließlich wird Ihr Code auch noch von VIS umkopiert. Der Parser des canJS-Frameworks gibt z.B. bei Javascript-Fehlern keine Zeilennummer aus.

Umso wichtiger, dass man in der Entwicklersicht in der Console genau nachließt, wie die Fehlermeldung aussieht. **Genau lesen! Die Meldung hat immer Recht!** Ich hab' mir zudem eine *test.ejs*-Datei angelegt. Dort hinein kopiere ich den Code des Skriptes, um ihn vom EJS-Plugin der webStorm IDE auf Syntaxfehler überprüfen zu lassen.

Wenn man Fehler partout nicht findet, hilft nur oft nur die „binäre Fehlersuche": Den Code durch den letzten noch funktionierenden Stand in den Code ersetzen, Versionen hochzählen, ALLES in den ioBroker laden (siehe „Ein erstes Widget") und auf Fehler überprüfen (Entwickleransicht: Console). Und dann die Hälfte des Codes durch den eigenen ersetzen, hochladen und auf Fehler untersuchen. Und dann ...

Jede dieser Zyklen dauert, wenn man gut ist, ca.2-3 Minuten – ich war bei über 300 Versionen.

Am Anfang macht man (ich) noch viele Syntaxfehler beim EJS. Vor allem vergessene Klammern (runde, geschweifte, eckige, ejs-Klammern, ...), da hilft aber das Umkopieren nach *test.ejs*. Dennoch es hilft ungemein, **sorgfältig** zu arbeiten und **genau** hinzuschauen. Nach vielen Stunden Übung kann man die Fehlermeldung in der Browser-Console ganz gut interpretieren.

... und wenn man mit den Syntaxfehlern umgehen kann, kommen die Programmfehler. Die kann man im Debugger der Entwickleransicht (Reiter: Sources) ganz gut debuggen, also Breakpoints setzen und Werte anzeigen bzw. ändern.

Da es sich um generierten Code handelt, ist das Auffinden der richtigen Stellen nicht ganz intuitiv. Sofern der Code richtig, d.h. ohne Syntaxfehler übersetzt wurde, ist das Widget zu finden unter: *no domain <widgetname>,* also bei mir *tplDimmerCtrl.js*, Meine JavaScript-Funktionen stehen unter *ip:vis/widgets/vis-jqui-dimme/vis-jqui-dimmer.js*. Die Funktionen aus dem Original-Template (z.B. *dialog*) sind in *index.html* zu finden (suche nach „*dialog:*")

Und nun: Syntaxfehler eliminieren, Breakpoints setzen, VIS-Runtime im Browser reloaden. Sobald der erste Breakpoint matched. hat man „gewonnen“. Aber Vorsicht: ioBroker läuft wohl als ein JavaScript-Thread. Sobald man auf einem breakpoint steht, kann man den ioBroker nicht mehr bedienen. Also: Breakpoint weiterlaufen lassen oder das VIS-Runtime-Fenster schließen.

Das Dimmer Widget

Mein neues Widget ist grundsätzlich so strukturiert wie das aus *iobroker.vis-jqui-mfd* kopierte und besteht aus:

- Einem Button, der dynamisch den Zustand des Dimmers anzeigt. Bei mir z.B. den Wert des Datenpunktes hm-rpc.0.<HomematicID>.1.LEVEL
- Einem Dialogfenster, das beim Klick auf den Button geöffnet wird. Dort gibt es:
    - Titelzeile (mit automatisch generiertem Close-Button)
    - Vier Dimmwerte (aus, 25%, 50%, 75%, 100%)
    - Optionaler Slider
    - Optionaler Dimmwert (der in der *iobroker.vis-jqui-mfd* Version noch einen Bug hat: er wird immer angezeigt)
    - Optionale Maßeinheit

Ich will dieses Widget nun so verändern:

- Ein Button, der dynamisch den Zustand des Dimmers anzeigt und bei Klick den Dimmer auf einen vordefinierbaren Wert ein- oder ausschaltet. Als Dimmer sollen auch Zigbee-Komponenten verwendbar sein.
- Ein Dialogfenster, das bei Doppelklick bzw. ForceTouch geöffnet wird. Dort gibt es:
    - Titelzeile (mit automatisch generiertem Close Button) und aktuellem Wert als Keildiagramm.
    - Beliebig konfigurierbare Dimmwerte

... so, dass das Ganze auch noch (in den Augen meiner Frau) gut aussieht,

Die technischen Herausforderungen habe ich in folgender Reihenfolge angegangen:

1. Anpassungen der CSS ...
   ... um einen Überblick über die beteiligten DOM-Strukturen zu bekommen
2. Umgang mit beliebigen Dimmwerten ...
   ... um den DOM etwas anders aufzubauen.
3. Verwendung der eigenen Klasse auch im Dialog ...
   ... um eine sehr überschaubare Codeänderung vorzunehmen und zu testen.
4. Vereinfachung des Codes ...
   ... um weiter in den Code zu kommen.

5. Schließen des Dialog-Fensters mit Klick auf einen Wert (oder des Close-Buttons) ...
   ... um eine wichtige Funktionalität einzubauen und dabei mit dem Event-Handling umzugehen.
6. Berücksichtigung von Zigbee-Komponenten ...
   ... um den Code an mehreren Stellen zu erweitern und dabei noch besser zu verstehen
7. Umgang mit dem Klick und Doppelklick ...
   ... um auch noch die letzte notwendige Funktionalität einzubauen und dabei das Event-Handling noch besser zu verstehen.
8. Automatisches Setzen des zusätzlichen Zigbee-Datenpunktes:
   ... um noch etwas zu haben, an dem ich mir gerade die Zähne ausbeiße ...

Anpassungen der CSS

Für das Verständnis dessen, was das Widget erzeugen und dynamisch verändern soll ist es ganz geschickt, sich die Struktur des generierten HTML-Code anzuschauen. Dieser sieht wie folgt aus:

```
<div id="vis-container>
  <div id="visview_view_1" class="vis-view"> ... </div>
  ...
  <div id="visview_view_n" class="vis-view"> ... </div>
  <div id="visview_view_x" class="vis-view">
    <div id="<widgetId_1>" class="vis-widget"> ... </div>
    ...
    <div id="<widgetId_n>" class="vis-widget"> ... </div>
    <div id="<widgetId_x>" class="vis-widget">
      <div id="<widgetId_x>_body" class="vis-widget-body">
        <svg … />
      </div>
      <div class="popup-helper" <ELEMENT-CALLBACK>> </div>
    </div>
  </div>
</div>
...
<div class="ui-dialog ..." role="dialog>
  <div class="ui-dialog-titlebar>
    <div class="ui-dialog-titlebar> ... </div>
    <span class="ui-dialog-title> ... </span>
    <button class="ui-dialog-titlebar-close ...> ... </button>
  </div>
  <div id="<widgetID>_dialog" class="ui-dialog-content ...">
    <div id="<widgetID>_radio" class="ui-buttonset ...">
      <input> <label>
      <input> <label>
      ...
    </div>
  </div>
</div>
```

Alle sichtbaren Buttons sind also strukturell von den zugehörigen, zunächst noch unsichtbaren Dialogen getrennt. Um das CSS anzupassen, spielt man nun grundsätzlich am besten die Änderungen in der Entwicklersicht des Chrome (Elements) durch.

Damit die Änderungen sich nur auf das eigene Widget beziehen, sollten die CSS-Selektoren so spezifisch wie möglich sein. Dazu sollten man also unbedingt eine eigene Kasse im VIS-Edit setzen (bei mir „pkDialog“), ansonsten verändert man gerne das Aussehen des gesamten VIS-Editors. Diese Klasse wird dann beim DIV des Widgets hinzugefügt.

Ungeschickterweise wird diese Klasse zurzeit nicht für das DIV des Dialogfensters (*class=“ui-dialog“*) verwendet. Das ist ein kleiner VIS-„Issue“, den ich behoben habe (siehe Unterkapitel: „Verwendung der eigenen Klasse auch im Dialog“). Alternativ kann man noch einen Attribut-Selektor auf *role=“dialog“* setzen. Das wirkt sich aber auch auf Dialoge im VIS-Editor aus und ist langsam.

In meinem CSS (für das Projekt) wurden, im Wesentlichen folgende Selektoren verwendet:

```
/*--- common style for pkDialog widgets ---*/
.pkDialog { ... }
.pkDialog .ui-corner-all { ... }
/*--- the dialog elements (from outer to inner) ---*/
.pkDialog .ui-dialog-content {... }
.pkDialog .ui-dialog-titlebar {...}
.pkDialog .ui-dialog-titlebar .ui-dialog-title {...}
.pkDialog .ui-dialog-titlebar-close {... }
.pkDialog .ui-buttonset { ...}
.pkDialog .ui-button { ...}
.pkDialog .ui-button-text { ... } ...}
/*--- remove resize-handles ---*/
.pkDialog .ui-resizable-handle { display:none; }
```

Umgang mit beliebigen Dimmwerten

Das Originalwidget gibt fünf Dimmwerte vor (aus, 25%, 50%, 75%,100%). Ich möchte diese Werte aber als komma-separated List angeben, jeweils mit Wert und dessen Darstellung. Also z.B:

```
0,20,40,60,100
aus,20,40,60,an
```

Dazu müssen die neuen Widget-Parameter in das Widget-Skript eingebaut werden.

```
<script id="tplDimmer"
    type           = "text/ejs"
    class          = "vis-tpl"
    data-vis-set   = "vis-jqui-dimmer"
    data-vis-type  = "dimmer,ctrl,dialog"
    data-vis-prev  = '<img src="widgets/vis-jqui-dimmer/img/Prev_Dimmer.png"></img>'
    data-vis-name  = "Dimmer"
    data-vis-attrs =
"oid/id;oid_on/id/setOnId;min/number;max/number;turnOnValue;invert_icon/checkbox;asButton[true]/checkbox;iconColor/color;"
    data-vis-attrs0 =
"group.colors;iconColor0/color;iconColor1/color;iconColor2/color;iconColor3/color;iconColor4/color;iconColor5/color;iconColor6/color;iconColor7/color;iconColor8/color;iconColor9/color;iconColor10/color;"
    data-vis-attrs1 =
"group.dialog;title;valuelist[0,20,40,60,80,100];valuetext[aus,20,40,60,80,100];modal/checkbox;dialog_width[470];dialog_height[210];dialog_top;dialog_left;overflowX/nselect,,visible,hidden,scroll,auto,initial,inherit;overflowY/nselect,,visible,hidden,scroll,auto,initial,inherit;"
>
```

Zusätzlich benötigt man noch eine englische und russische Übersetzung, die man unter *$.extend (systemDictionary)* einfügt (Russiche Übersetzung mit Google;-).

Die Ausgabe der Buttons mit ihren Werten erfolgt dann in einer Schleife:

```
<%
let valueList = valuelist.split(',');
let valueText = valuetext.split(',');
for (let i=0; i < valueList.length; i++) {
    let id      = WidId + '_radio' + i;
%>
    <input type="radio" id="<%= id %>" name="<%= WidId %>_radio" value="<%= valueList[i] %>" checked="checked"/>
    <label for="<%= id %>"> <%= valueText[i] !== undefined ? valueText[i] : 'no text' %> </label>
<% } %>
```

Verwendung der eigenen Klasse auch im Dialog

Eigentlich wäre *pkDialog.ui-dialog* als Selektor wesentlich spezifischer. Dazu muss man aber die Hilfsfunktion bearbeiten, die aufgerufen wird, um den Dialog zu erzeugen. Diese Hilfsfunktion wird in *vis-jqui-dimmer.html* mit einem, für mich zunächst unbekannt EJS-Konstrukt, aufgerufen:

```
<div class="popup-helper" <%= (el) -> vis.binds.jqueryui.dialog(el, data, true) %> />
```

Mit *<%= (el) -> Funktion%>* wird ein ”Element Callback“ (//www.javascriptmvc.com/docs/can.EJS.html#section_ElementCallbacks) hinzugefügt, der die entsprechende Funktion aufruft. Element Callbacks sind ein Konzept von canJS (bzw. EJS) und verwendet die „arrow“-Notation. Diese Notation ist zwischenzeitlich veraltet und wurde durch die „fat-arrow“-Notation ersetzt. Ich habe das entsprechend geändert (also „->“ durch „=>“ ersetzt).

Hier ist dies eine Funktion des VIS-Adapter (*vis.binds.jqueryui.dialog*), die in *ioBroker.vis/www/widgets/jqui.html* zu finden ist. Die kopiere ich in das *vis.binds.jqui-dimmer*-Objekt und, damit der gesamte Code nicht zu sperrig wird, hab‘ ich den Javascript-Teil aus *vis-jqui-dimmer.html* nach *widgets/vis-jqui-dimmer/js/vis-jqui-dimmer.js* kopiert.

Jetzt wird das Ganze eingebunden mit:

```
<div class="popup-helper" <%= (el) -> vis. binds.jquidimmer.dialog(el,data,true) %> />
```

Damit nun die benutzerdefinierte Klasse auch im Dialog eingetragen wird, hab‘ ich in den event-Handler (in *dialog*):

```
$(el).parent().find('div.vis-widget-body').on('click touchend', function (event) {
```

zusätzlich noch eine Zeile eingefügt:

```
$dlg.parent().addClass(options.class);
```

… die Option *class* wurde zuvor als Widget-Parameter hinzugefügt und wird *dialog* damit automatisch mit übergeben.

Kleiner Ausflug:

Zum Anpassen des Codes aus dem „original“ VIS-Adapter habe ich die angepasste Stelle in mein Projekt hineinkopiert, dort bearbeitet und dann statt des Originals aufgerufen. Dieses Vorgehen hab’ ich für alle benötigten Funktionen verwendet. Ich habe also die Original-Varianten aus dem vis.binds.jquerui-Objekt aus ioBroker.vis/www/widgets/jqui.html in widgets/vis-jqui-dimmer/js/vis-jqui-dimmer.js in das vis.binds,jqui-dimmer-Objekt kopiert und dort bearbeitet. Das waren *dialog*, *toggle* und *radio*. Um auch noch die letzten Anhängigkeiten zu *iobroker.vis/www/widgets/jqui.html* zu eliminieren, habe ich auch noch *classes*, *setSvgColor*, *active* und *activeHelper* in das *vis.binds,jqui-dimmer*-Objekt kopiert aber funktional nicht verändert. Die Autoclose-Funktionalität (*dialogAutoClose)* hab‘ ich vollständig entfernt.

Nach dem Kopieren muss man noch unbedingt darauf achten, dass die ursprüngliche Verwendung des *vis.binds.jquerui* Objektes durch das *vis.binds.jqui-dimmer*-Objekt ersetzt werden (also: find and replace). Tut man das nicht, läuft das Programm scheinbar ungerührt im Originalcode und Änderungen an der Kopie werden scheinbar ignoriert. Ich hab‘ Stunden damit verbracht, breakpoints zu setzen, die nie eingetreten sind, weil der unveränderte Originalcode aufgerufen wurde. Also: „*jquerui*“ sollte nun nirgends mehr auftauchen, weder im html- noch im js-Code

Vereinfachung des Codes

Nach den bisherigen Änderungen habe ich den Code „einigermaßen“ verstanden und wollte ihn ein wenig vereinfachen.

Zuvor hatte ich schon alle Programmteile für die Funktionsparameter entfernt, die in keinem Aufruf verwendet wurde – das ging recht formal, also ohne Verständnis des Codes. Dabei wurden u.A. Funktionalität entfernt, die die Dialoge „persistieren“, also (wahrscheinlich) zum Cachen des Dialogs verwendet werden, was den Code dann, für mich nicht spürbar, verlangsamt.

Nun wollte ich noch die Teile entfernen, die für einen Dimmer nicht notwendig sind, denn das Widget, auf dem meines basiert, kann auch für ein einfaches Schalten verwendet werden. Beim Schalten muss man nicht mit *min* opder *max* umgehen, hat als Schaltwerte aber keine ganzen Zahlen, sondern *true* oder *false*. Zusätzlich konnte das Widget auch noch mit den strings *‚true‘* und *‚false‘* umgehen. Das hat zu if/then/else-Orgien geführt, die ich alle entfernt habe.

Die neue Logik musste nun **ausführlich getestet werden** und funktioniert nur, wenn *min* und *max* sauber gesetzt werden, was aber für einen Dimmer sowieso notwendig ist. Hier hab‘ ich mir daher eine „ordentliche“ Fehlerbehandlung etwas geschenkt.

Kleiner Ausflug zum grundsätzlichen Verständnis des VIS-Adapters:

Im vorgestellten Code werden die Werte von Datenpunkten explizit gelesen und geschrieben. Das erfolgt über folgende Variable bzw. Funktion.

- vis.states[<objId.val>)]
  Im Objekt *vis.state* stehen die Zustände aller Datenpunkte. Der Wert eines Daten-

punktes lässt sich dann über das Element <*objId.val*> auslesen.
Bsp: `let val = vis.state[oid + '.val']`

- vis.setvalue(<oid>, <val>)
  Mit dieser Methode werden die Werte des Datenpunktes, der in *oid* gegeben wird auf *val* gesetzt.
  Bsp: `vis.setvalue(oid, val)`

Aber Vorsicht: Der benutzte Datenpunkt (hier: oid) ist nur definiert, wenn er im Template-Header angegeben wurde. Möchte man auf weitere Datenpunkte zugreifen so müssen deren Bezeichner mit „oid“ geginnen (z.B. oid-2).

Neben diesem programmierten Lesen und Schreiben der Werte von Datenpunkten gibt es in VIS einen Automatismus, der diese Werte an Variablen übergibt, sobald sich etwas an diesen ändert. Damit kann man dann im DOM Änderungen auch anzeigen. Dieser Mechanismus arbeitet über canJS und setzt eine bestimmte Syntax voraus (v2.canjs.com/guides/EJS.html). Greift man also in einem Template z.B. auf die ObjektId über z.B. *this.data.attr('oid')* zu, so weiß canJS (über „*attr(‚oid‘)*), dass hier das Attribut *oid* gebunden werden soll, es also an jeder Stelle, an der es innerhalb eines EJS-Ausdrucks verwendet wird ändert, sobald sich der Datenpunkt-Wert ändert. Das nennt canJS „live binding“ und tatsächlich **ist** ein Widget ein canJS-Template.

Es gibt noch eine zweite Möglichkeit des live-bindings:
Auch mit der Funktion *vis.states.bind(<objId.val>, <onChange>)* kann ein Datenpunkt gebunden werden,. Dazu gibt man neben der Objekt-Id des Datenpunktes eine Callback-Funktion an, die aufgerufen wird, sobald sich der Wert eines Datenpunktes ändert. In dieser Callback-Funktion können dann die DOM-Elemente angepasst werden, z.B. Icons ausgetauscht.

Beispiel:

```
function onChange(e, newVal) { this.find('input').css('top',10px) }
vis.states.bind(oid + '.val', onChange)
```

Im Objekt *vis* gibt es neben *state* noch (viele) andere Elemente, auf die man zugreifen kann. Ich hab‘ mit den folgenden noch zu tun gehabt:

- *vis.binds* wird verwendet, um für die Widgets Funktionen abzuspeichern – genauer: Referenzen auf Hilfsfunktionen. Die „eingebauten“ Funktion, sind dabei in *vis.bind.jqueryui* abgelegt. Viele Adapter erweitern nun *vis.binds* um eigene Elemente, ich tue das mit *vis.binds.jquidimmer* und lege dort „meine“ Hilfsfunktionen ab. Diese Hilfsfunktionen werden dann gerne in Element-Callbacks (z.B. *<%= (el) -> vis.binds.jquidimmer.dialog(el,data,true) %>*) in HTML-Elemente eingebaut, so dass sie aufgerufen werden, sobald das HTML-Element erzeugt wird.
  Kleiner Ausflug:
  Damit man die eigenen Hilfsfunktionen auch über Callbacks aufrufen kann, die man an die Widget-Parameter hängt, muss das Element unterhalb „bind“ so heißen wie das Widget also in meinem Fall: vis.binds['vis-jqui-dimmer'] (mit *vis.binds.vis-jqui-dimmer* kam canJS nicht klar).

- *vis.editMode* ist ein bool'scher Wert, der angibt, ob man sich gerade im VIS-Editor befindet oder eben nicht, also im VIS Runtime. Die wichtigsten Benutzerinteraktionen sollen im Editor natürlich nicht angesprochen werden können, daher führen die Widgets solche Aktionen meist im *editMode* nicht aus. Der Code des Templates für den Editor und die VIS-Runtime-Umgebung unterscheidet sich also nicht.
- *vis.detectBounce(<jquery-Object>)* überprüft die Zeiten von zwei aufeinander folgenden „lastupdate"-Events eines Devices, die wohl nicht zu eng zusammenliegen sollen – ich hab's aber nicht genauer untersucht.
- *vis.objects* beinhaltet alle Devices, die im ioBroker bekannt sind.

... und noch ein kleiner Ausflug zur asynchronen Programmierung in Javascript:

Vieles in Javascript und damit auch in VIS wird über Callback-Funktionen behandelt. Diese Funktionen werden „irgendwann" aufgerufen, sobald bestimmte Ereignisse eintreten. Der Zeitpunkt des Ereignisses ist also meist unbestimmt und beeinflusst den Kontrollfluss des Programms maßgeblich. Die wichtigsten Ereignisse in VIS sind die Veränderung von Datenpunkten oder der Ablauf eines Timers, der vorher eingestellt wurde. Um dennoch bestimmte Abläufe zu synchronisieren, werden daher manchmal „künstlich" Wartezeiten über Timer erzeugt werden. Beispiel ist das Abarbeiten eines Buttons in einem Dialog, bevor sich das Dialogfenster schließt. In „moderneren" Javascript-Implementierungen würde man so etwas mit „promises" lösen, das habe ich aber auch nicht getan ;-)

Zum Verständnis der Abarbeitung von events ist es noch ganz nützlich zu wissen, wo Events „aufschlagen". Benutzerevents, z.B. ein Mouseclick, treten bei dem DOM-Element auf, mit dem der Benutzer gerade „interagiert" und zusätzlich bei allen DOM-Elementen in die das „interagierte" Elemente verschachtelt ist. Das ist aber oft nicht erwünscht. So möchte man z.B. nicht, dass ein Button-Click auch vom übergeordneten Dialog interpretiert wird. Um das zu verhindern, wird in der Callback-Funktion oft der Befehl *event.stopPropagation()* aufgerufen.

Schließen des Dialog-Fensters mit Klick auf einen Wert

Das Behandeln der Buttons im Dialog erfolgt in der *dialog*-Funktion (genauer: der Funktion die im Objekt *vis.binds.jquidimmer* den key *dialog* hat). Dort gibt es einen Event-Handler, der auf Click-Events des Dialogs lauscht (bzw. „*touchend*"). Die Events der Buttons werden in der *radio*-Funktion verarbeitet – dort aber ohne den Dialog zu schließen.

Der Event-Handler des Dialogs setzt (zunächst) nur die CSS-Werte für den Dialog und den Fokus. Ich habe ihn um einen Aufruf einer Funktion erweitert, die alle Events abmeldet und das Fenster mit einem beliebigen Klick schließt.

```
$dlg.off('click touchstart touchend').on('click touchend', function (event) {
    setTimeout(function () {
        event.stopPropagation();
        $dlg.dialog('close');
        return false;
    }, 200);
});
```

Diese Funktion hab' ich aus der bereits vorhandenen Funktion zum Schließen über den close-Button entnommen, muss sie aber eine wenig warten lassen (über *setTimeout*), da zunächst die inneren Button-Events abgearbeitet werden müssen, bevor das Fenster schließt.

Asynchrone Programmierung ist manchmal die Krätze ...

Berücksichtigung von Zigbee-Komponenten

Soweit ich es überblicken kann, berücksichtigt VIS vor allem Homematic-Komponenten. Insbesondere haben Homematic-Dimmer einen Datenpunkt *<device>-working*, der von einem gerade dimmenden Dimmer gesetzt wird. Ich könnte mir denken, dass VIS währenddessen nicht schaltet – hab' das aber nicht getestet. Ich hatte die Option *oid-working* daher entfernt, so dass das Dimmen von Homematic-Komponenten möglicherweise etwas „schwerfälliger" ist – es tut aber dennoch, das hab' ich getestet. Ich hab's aber wieder reingebaut (dazu später mehr)

Die Zigbee-Komponenten, die ich über den *deconz*-Adapter betreibe, haben zum Dimmen zwei Datenpunkten: einen *.on*- und einen *.bri*-Datenpunkt. Will man den Dimmer auf einen neuen Wert stellen, muss dieser eingeschaltet sein (*.on* auf *true*) und dann der neue Helligkeitswert übergeben werden. Beim Ausschalten über einen normalen Schlater bleibt der Helligkeitswert bestehen und nur der *.on*-Wert geht auf *false*.

Damit man das Verhalten von Komponenten, die über den *deconz*-Adapter angesprochenen Komponenten (Zigbee) von dem der *hm-rpc-Adapter* (Homematic) unterscheiden kann, fragt man die Objekt-Id ab, in der als erstes Element der Adapter codiert ist: entweder *hm-rpc* oder *deconz*.

Um nun die Abfrage eines Dimmerwertes vom Dimmertyp (z.Z. Homematic und Zigbee) zu entkoppeln, hab ich, statt der ursprüinglichen vis-Methoden zum lesenden und schreibenden Zugriff auf die Zustände (mit *vis.states()* und *vis.setvalue()*) zwei eigene Methoden geschrieben:

```
getVal: function(oid) {
   //-- get type of device and, in case of Zigbee, its dataPoint device.on and state
   let device = vis.binds.jquidimmer.getDevice(oid);
   let deviceType  = device.deviceType;
   let deviceOnId  = device.deviceOnId;
   let deviceState = device.deviceState;
   //-- get value either from device.on (Zigbee) or device.level (Homematic)
   let val = 0;
   if (deviceType === 'Zigbee') { val = deviceState === false ? 0 : vis.states[oid + '.val']; }
   else                         { val = vis.states[oid + '.val'];                             }
   //-- if device has never been switched its val is undefined -> set it to 0/false
   if (val === undefined) { val = 0; }
   //--- return val
   return val;
},
```

```
setVal: function(oid, val) {
   //-- get type of device and, in case of Zigbee, its data-Point device.on and state
   let device = vis.binds.jquidimmer.getDevice(oid);
   let deviceType  = device.deviceType;
   let deviceOnId  = device.deviceOnId;
   let deviceState = device.deviceState;
   //-- switch device on
   if (val > 0) {
      if (deviceType === 'Zigbee') {
         if (!deviceState) { vis.setValue(deviceOnId, true); } // switch on immediately
         setTimeout( function() { vis.setValue(oid, val); }, 2000 ); // wait
      } else { // Homematic: switch level immediately
         vis.setValue(oid, val);
      }
   }
   //-- switch device off
   else {
      if (deviceType === 'Zigbee') {
         vis.setValue(oid, val);
         setTimeout( function() {
            let device = vis.binds.jquidimmer.getDevice(oid);  // might have changed
            let deviceOnId  = device.deviceOnId;
            let deviceState = device.deviceState;
            if (deviceState) { vis.setValue(deviceOnId, false); } }, 2000 );
      } else {
         vis.setValue(oid, val);  // set level
      }
   }
},
```

Diese Methoden abstrahieren vom zugegriffen Dimmertyp und liefern Werte zwischen einem minimalen und maximalen Dimmerwert, also Werte zwischen meist 0 und 100 (für Homematic) und 0 und 254 (Zigbee).

Beide Methoden brauchen Informationen über das Gerät, auf das Sie zugreifen, das wird mit der Methode *getdevice(<Object-ID>)* umgesetzt:

```
getDevice: function(oid) {
   //-- determine type of device and, in case of Zigbee, its data-Point device.on and state
   let deviceType  = '';
   let deviceOnId  = '';
   let deviceState = false;
   if ((oid.split('.'))[0] === 'hm-rpc') {
      deviceType = 'Homematic';
   } else if ((oid.split('.'))[0] === 'deconz') {
      deviceType = 'Zigbee';
      deviceOnId = (oid.split('.'))[0] + '.' + (oid.split('.'))[1] + '.' + (oid.split('.'))[2] + '.on';
      deviceState = vis.states[deviceOnId + '.val'];
   } else {
      deviceType = '';
   }
   return { deviceType : deviceType, deviceOnId: deviceOnId, deviceState : deviceState };
},
```

Da mein Widget jetzt nur noch mit numerischen Werten umgehen muss, vereinfacht sich auch die HTML-Generierung in *vis-jqui-dimmer.html*:

```
...
//-- get value either from device.on (Zigbee) or device.level (Homematic)
let on  = oid_on !== undefined ? vis.states.attr(oid_on + '.val') : true;
let bri = parseFloat( vis.states.attr(oid + '.val') );
let val = on === false ? 0 : bri;

//-- get image, depending on value
let image = '';
let imageColor = null;
if        (val > min + (max - min) * 0.9 ) {
  image = "light_light_dim_100"; imageColor = this.data.attr('iconColor10');
} else if (val > min + (max - min) * 0.8 ) {
  image = "light_light_dim_90";  imageColor = this.data.attr('iconColor9');
}
...
else {
  image = "light_light_dim_00";  imageColor = this.data.attr('iconColor0');
}
...
```

Diese Abstraktion des Zugriffs auf die Datenpunkte berücksichtigt zurzeit nur Dimmer, daher die Werte zwischen min und max. Werte von Schaltern, also *true* und *false* bzw ‘*true*‘ und ‘*false*‘ werden nicht (mehr) berücksichtigt. Dazu müsste man sich ein entsprechendes Widget für Schalter bauen – oder eben eines der vielen guten Vorhandenen nutzen.

Wie bekommt man nun das Widget dazu, neben dem eigentlichen Datenpunkt *(<device>.bri*) auch noch den Zustand des zweiten Datenpunktes zu „überwachen“ (*<device>.on*)? Für die Antwort bin ich im Forum fündig geworden (forum.iobroker.net/topic/5917/eigenes-vis-widget-daten-werden-im-view-mode-nicht-geladen-erklärung-für-vis-states-gesucht). Dort wird beschrieben, dass alle Widget-Parameter, die mit „*obj*“ beginnen (z.B. *obj_on*), als Datenpunkte angelegt werden, d.h. dann in *vis.states[oid + '.val']* erscheinen und gelesen werden können.

Und damit das Widget auf den Zustand des Datenpunktes *device.on* nicht nur lesend zugreifen kann, sondern auch automatisch auf Änderungen reagiert, muss man den Datenpunkt noch binden (sofern das Device ein Zigbee-Device ist.)

```
//-- bind states and remember bindings (may be two) and it's bind handler
vis.states.bind(oid + '.val', onChange);  // binding for oid (.bri or .level)
let bound = [oid + '.val'];
let device = vis.binds.vis-jqui-dimmer.getDevice(oid);
let deviceType  = device.deviceType;
let deviceOnId  = device.deviceOnId;
if (deviceType === 'Zigbee') {
   vis.states.bind(deviceOnId + '.val', onChange);  // binding for .on
   bound.push(deviceOnId + '.val');
}
$this.parent().parent()
   .data('bound', bound)           // remember up to two bounds
   .data('bindHandler', onChange); // remember bind handler
```

https://forum.iobroker.net/topic/19929/getting-started-with-visualisations/3

... das war alles nicht ganz einfach herauszufinden – und ich versteh‘ immer noch manches nicht wirklich ...

Damit die Anzeige der Dimmwertes im Button funktioniert, muss in der VIS-Editor-Konfiguration des Widget-Objektes bei Zigbee-Komponenten der *.bri*- und der *.on*-Datenpunkt eingetragen werden. Bei Homematic-Komponenten der *level*-Datenpunkt.

Bei meiner Umsetzung habe ich nur meine Fälle zum Schalten eines Zigbee-Dimmers berücksichtigt, so wie ich sie habe. Ich kann nicht ausschließen, dass es Zigbee-Dimmer mit anderen Datenpunkten gibt – die würden dann nicht funktionieren. Zudem hab‘ ich mit dem Entfernen der „-working“ Option Funktionalität zum Schalten von Homematic-Komponenten ausgebaut. Das tut so bei mir, ich kann aber nicht ausschließen, dass das mit allen Homematic-Komponenten noch tut.

Umgang mit dem Klick und Doppelklick

Das Original-Widget öffnet bei Mouseclick einen Dialog. Das neue Widget soll bei einem einfachen Click nur den Dimmer auf einen vordefinierten Wert stellen, bzw., wenn schon ein Wert größer 0 eingestellt ist, den Dimmer ausschalten. Auch dafür gibt es im Original-Widget eine Vorlage: *tplMfdLightCtrl*. Ich kopiere von dort die *toggle*-Methode aus *ioBroker.vis/www/widgets/jqui.html* und bearbeite sie.

Die Originalversion schaltet immer auf 100% (bzw. true) oder 0% (bzw. false), in Abhängigkeit vom Mittelwert. In meiner viel einfacheren Variante wird auf einen, als Parameter optional übergebenen Einstellwert bzw. 0 geschaltet, je nachdem, ob der Dimmer vorher an (Wert >0) oder aus (Wert = 0) war:

```
// only in runtime-mode: switch on/off, depending on value and default value for "on"
if (oid && !vis.editMode) {
   // get value of state
   let val = vis.binds.jquidimmer.getVal(oid);
   // device off -> switch on
   if ( val === min ) {
      if (turnOnValue !== undefined ) { vis.binds.jquidimmer.setVal(oid, turnOnValue); }
      else                            { vis.binds.jquidimmer.setVal(oid, max );        }
   }
   // device on -> switch off
   else if ( val > 0 ) {
      vis.binds.jquidimmer.setVal(oid, 0);
   }
}
```

Beim Klick auf den Dimmer-Button wird der Dimmer nun geschaltet. Bei einem Doppelklick oder, für ioS-Geräte neuere Bauart, einem ForceTouch, soll nun der Dialog geöffnet werden.

Das Problem dabei: bei einem Doppelklick wird zunächst ein einfaches Klick-Event erzeugt – und das schaltet den Dimmer. Entsprechend wird bei einem ForceTouch zunächst ein einfacher Touch erkannt– das Event dazu heißt *webkitmouseforceup*. Beides, das direkte Schalten und das Öffnen des Dialogs ist natürlich nicht sinnvoll. Das kann man vermeiden, indem man im der Event-Handler zunächst noch auf einen zweiten Event wartet: tritt der ein, handelt es sich

um einen Doppelklick (bzw. ForceTouch), tritt der nicht ein, dann handelt es sich eben nur um das einfachere Event:

```
if ($dlg.timer) {  // Doubleclick: timer still set -> there was an event shortly before
   clearTimeout($dlg.timer);  // clear timer
   $dlg.timer = null;         // reset timer variable
   Code to be done for doubleclick
} else { //--- Singleclick: timer not set -> there was no event shortly before
   $dlg.timer = setTimeout(function () {
      $dlg.timer = null;
      Code to be done for singleclic
   }
}
```

Automatisches Setzen des zusätzlichen Zigbee-Datenpunktes

Betrachtet man das ursprüngliche Widget, so fällt dort der Parameter „In Arbeit Zustand ID“ auf, der zudem automatisch gefüllt wird, nachdem man die Objekt-ID gewählt hat.

Nicht leicht zu finden, wo das passiert. Durch (langes) Steppen im Debugger hab’ ich es schließlich ausfindig gemacht: In *ioBroker.vis/www/js/visEditInspect.js* sind Funktionen implementiert, die aufgerufen werden, sobald die Parameter im VIS-Editor bearbeitet werden. Für Parameter des Typs „ID“ ist das die Funktion hinter *editObjectID*. Dort wird überprüft, ob es ein Parameterfeld mit der ID *inspect_oid-working* gibt. Dieses Parameterfeld gibt es, sobald es im Kopf des Template-Scripts einen Parameter mit dem Namen *oid-working* gibt. Sobald es diesen Parameter gibt, wird die Objekt-ID um den letzten Wert gekürzt (bei Homematic-Geräten: *.LEVEL*) und um den Wert erweitert, der in <adapter>.objects[<objID>].common.workingID steht (bei Homematic-Geräten: *WORKING*).

Auszug aus *ioBroker.vis/www/js/visEditInspect.js* :

```
var parts = newId.split('.');
parts.pop();
parts.push(that.objects[newId].common.workingID);
$('#inspect_oid-working').val(parts.join('.')).trigger('change');
```

Das ist ziemlich hartcodiert und auf Homematic-Geräte zugeschnitten, denn das Attribut *workingID* gibt es nur bei Homematic-Geräten.

Das könnte entsprechend nutzen, um auch für Zigbee-Geräte eine ID abzulegen und automatisiert füllen zu lassen. ... das muss ich aber noch genauer untersuchen ;-)

Erfahrungen und Ausblick

Durch das Projekt habe ich viel über die Funktionsweise von ioBroker-Adaptern im Allgemeinen und des VIS-Adapters im Besonderen gelernt. Wie immer ging das nur, indem man sich darauf eingelassen und es „getan“ hat. Nebenbei kenne ich mich nun auch so richtig gut im VIS-Editor aus und hab‘ auch meine Javascript und CSS-Kenntnisse ein wenig aufgefrischt. Ich hab‘ aber auch gesehen, dass andere viel schlauer sind und teilweise auch viel bessere Ansätze für Dinge aufzeigen, die ich hier erläutere.

Das Projekt will keinen Schönheitspreis in der Konzeptionierung und Umsetzung von Softwarearchitekturen gewinnen – es war nur als erster „quick hack“ gedacht – um „mal reinzukommen“. Ich werde den Code also noch ein wenig aufhübschen und dazu verwenden, meine Visualisierung weiter aufzubauen – dazu werde ich mir auch ‘mal den vis-metro-Adapter anschauen, der doch graphisch etwas anspruchsvoller und funktional reicher ist.

Zudem habe ich bei GitHub einige automatisiert erstelle Issues bekommen – die muss ich dann auch noch angehen, z.B. die Integration in die CI-Umgebung *Travis* und die Unterstützung von Admin3 (forum.iobroker.net/topic/9672/admin3-migration-von-konfigurationsdateien).

Ich habe Änderungen am Code des VIS-Adapters und des vis-jqui-mfd-Adapters vorgenommen und diese zunächst nur lokal in mein Projekt eingebaut. Ich denke aber, dass die ein oder andere Änderung bzw. Erweiterung auch für die Original-Projekte von Interesse sind und werde sie „irgendwie“ dort einkippen.

Ich habe mit meinem Projekt sicher nur einen kleinen Ausschnitt aus dem betrachtet, wozu ioBroker und VIS fähig sind, hoffe aber dennoch einen kleinen Beitrag zu deren Verständnis geleistet zu haben.

Keep Coding ;-)

Ach so, ... der fertige Code? Den muss man nun selbst erstellen, denn nur dabei lernt man.