Node-RED (Teil 2)

Der erste Teil dieser Reihe ist im PCNEWS Heft 182 zu finden.

Was erwartet Sie in diesem Beitrag?

Schwerpunkt sind Funktionen in Node-RED.

Sie werden...

- ... Experimente mit JavaScript machen
- ... das Schreiben von Funktionen in JavaScript kennen lernen
- ... Funktionen in Node-RED verwenden
- ... Werte in Node-RED speichern
- ... drei Lampen nacheinander einschalten können
- ... die Anwendung von topics in Funktionen verstehen
- ... Funktionen zum Steuern einer Stiegenhausbeleuchtung anwenden

---

Ein ☝ im Text der Kurzfassung weist darauf hin, dass in der Langversion an dieser Stelle mehr Text, ein Programmcode oder eine detaillierte Anleitung zu finden ist. Die Langversion kann über den Link und den QR-Code am Ende des Beitrags abgerufen werden.

JavaScript

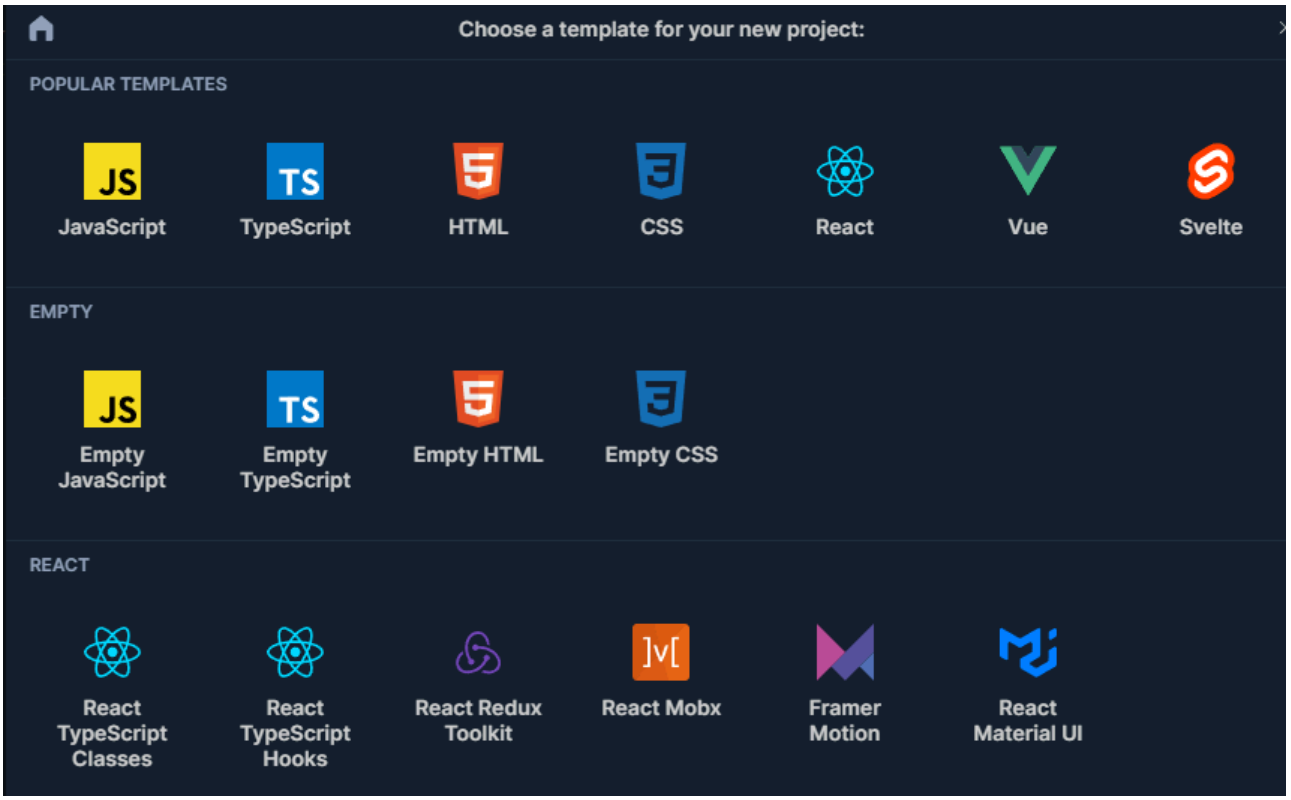
JavaScript wurde 1995 von Netscape für dynamische Webseiten entwickelt. Jeder Webbrowser versteht die JavaScript-Programme, die in Webseiten enthalten sind. Mittlerweile wird JavaScript auch in Webservern oder Microcontrollern verwendet. ECMAScript ist die Spezifikation einer Skriptsprache, die in JavaScript umgesetzt wird. Jedes Jahr gibt es eine neue Version.

TypeScript ist eine Weitererweiterung von JavaScript. Dabei werden Datentypen eingeführt, mit denen Programme schon in der Entwurfsphase leichter auf Fehler überprüft werden können. TypeScript ist eine echte Erweiterung von JavaScript, das heißt, dass jedes JavaScript-Programm direkt auch in TypeScript verwendet werden kann.

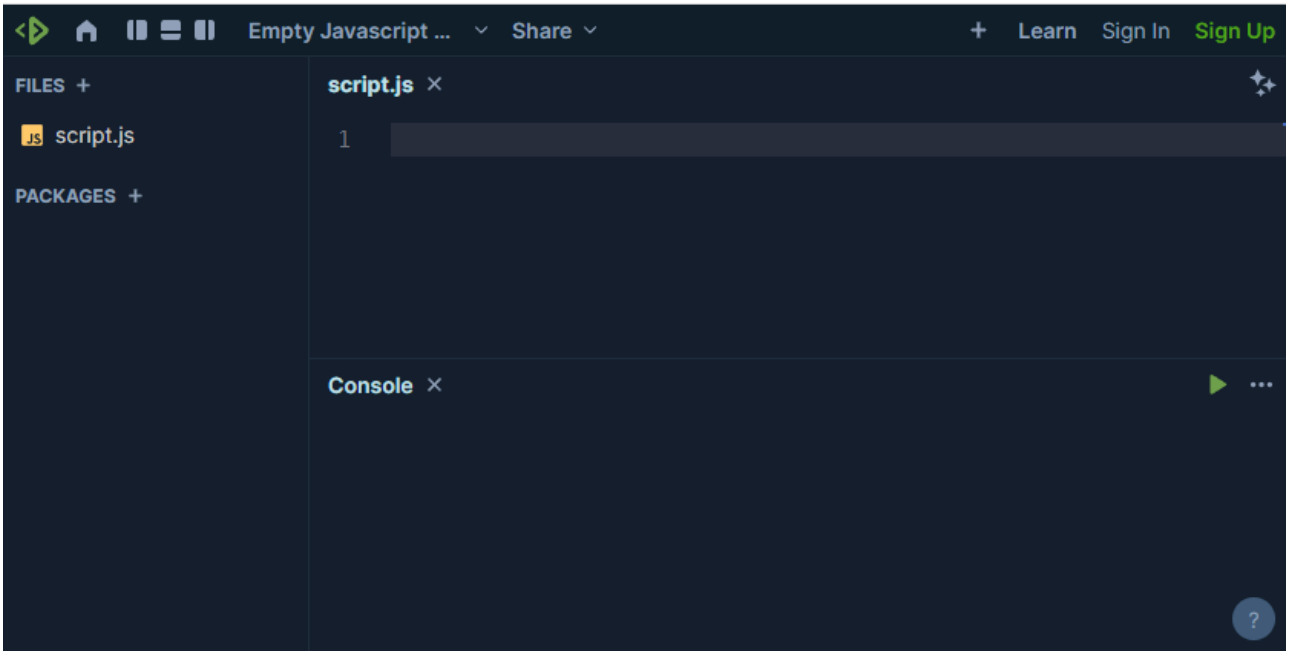
Was hat JavaScript mit Node-RED zu tun? Node-RED und seine Komponenten sind in JavaScript programmiert. Natürlich kann Node-RED auch ohne eine einzige Zeile von JavaScript verwendet werden. Aber oft können mit den Node-RED *functions* besondere Steuerungsideen umgesetzt werden - und dafür brauchen wir JavaScript!

Experimente im Browser

Um JavaScript kennenzulernen und auch um die folgenden Beispiele in Node-RED besser zu verstehen, rufen wir eine Webseite auf, über die Experimente mit JavaScript *online*, also *ohne eine lokale Installation*, durchgeführt werden können. Es gibt viele Angebote, oft auch mit lästigen Werbeeinblendungen. Ich empfehle https://playcode.io/new



Hier gibt es mehrere Angebote, die alle irgendwie mit der Entwicklung von (dynamischen) Webseiten zu tun haben. Für die einfachen Experimente wählen wir `Empty JavaScript`.



Im oberen Teil werden die JavaScript-Anweisungen eingetragen. Die Anweisungen werden sofort ausgeführt, die Ergebnisse im unteren Teil eingetragen.

> **Note**
>
> Natürlich ist ein Programm erst fertig, wenn alles eingetragen ist. Während des Tippen (wenn die Anweisungen also noch unvollständig sind) erscheinen im Consolen-Fenster allerlei rote Fehlermeldungen. Keine Panik, die verschwinden, sobald das JavaScript-Programm komplette und korrekt eingegeben ist. Diese Meldungen sind aber auch zum Ausbessern von Tippfehlern hilfreich.

Funktionen

Allgemein brauchen wir Funktionen,

- ... um wiederholt auftretenden Anweisungen zusammenzufassen und sie dann einfach wiederverwenden (aufrufen) zu können
- ... um große Programme in kleinere Einheiten zu unterteilen, die dann leichter gewartet werden können und das Programm übersichtlicher machen
- ... um Programmteile unabhängig von einander (asynchron) ausführen zu können.

Funktionen werden in Node-RED in eigenen Nodes umgesetzt.

Beispiel 1

Das erste Beispiel ist eine Funktion, die zwei Zahlen addiert.

```
1  function add (a, b {
2    return a+b;
3  };
4
5  let s = add(3, 4);
6  console.log("Summe = ", s);
```

Erklärung im Detail:

Zeile 1: Das *Schlüsselwort* **function** sagt JavaScript, dass das, was dahinter kommt, die Beschreibung (Vereinbarung) einer Funktion ist. JavaScript erwartet danach den Namen der Funktion [hier: **add**], die Parameterliste [hier: **(a, b)**, die runden Klammern gehören zur Parameterliste] und den **Rumpf** der Funktion [beginnend mit **{** in Zeile 1 bis einschließlich **}** in Zeile 3].

> **Note**
>
> Bei anonymen Funktionen entfällt der Name, siehe Beispiel 2.

Parameter sind Platzhalter, die beim Aufruf der Funktion (Zeile 5) durch konkrete Werte ersetzt werden.

Zwischen den Klammern { und } steht das, was die Funktion tun soll. Hier wird die Summe von a und b bestimmt und über **return** an der Stelle (in Zeile 5) eingesetzt, an der die Funktion aufgerufen wurde.

Zeile 5: Mit **let** wird eine *lokale Variable* vereinbart [hier **s**], also eine Variable, die nur in der unmittelbaren Umgebung gültig ist. Dieser Variablen wird jener Wert zugewiesen, den die Funktion `add` errechnet. Mit dem *Aufruf* von `add(3, 4)` wird der (formale) Parameter a durch das Argument 3 ersetzt und b durch 4.

Zeile 6: Wir wollen ja auch ein Ergebnis sehen. `console.log ()` schreibt alle Argumente [hier: **"Summe = s"**], also das Wort *Summe* gefolgt von einem Gleichheitszeichen und das in s gespeicherte Ergebnis der Addition [hier: **7**] in das *Consolen*-Fenster am Bildschirm.

> **Note**
>
> Statt *Parameter* wird auch die Bezeichnung *formaler Parameter*, statt *Argument* die Bezeichnung *aktueller Parameter* verwendet.

Wer die Beispiele selbst ausprobieren will, aber nicht abtippen will, kann den Programm-Code auch kopieren. Abtippen hilft aber beim Verstehen des Programms ...

So sieht das Beispiel im Browser aus:

> **Note**
>
> Das Einrücken (wie im Beispiel zu sehen) ist für die Lesbarkeit des Programms sehr wichtig. Ohne übersichtliches Einrücken werden umfangreiche Programme rasch unverständlich.
>
> JavaScript verlangt in den meisten Fällen keinen Strichpunkt am Ende einer Anweisung, die mit der Zeile endet. Trotzdem empfehle ich die Schreibweise wie in den Beispielprogrammen zu sehen, da auch das die Lesbarkeit verbessert.

Beispiel 2

JavaScript ist eine funktionale Sprache. Ein wichtiges Kriterium dafür ist die Verwendung von Funktionen als Objekte. Eine Funktion kann daher einer Variablen zugewiesen werden.

```
1  let add = function (a,b) {
2    return a+b;
3  };
4
5  let s = add(3, 4);
6  console.log("Summe = ", s);
```

Zeile 1: Mit **let** wird die lokale Variable **add** erzeugt, der dann eine *anonyme Funktion* zugewiesen wird. Anonyme Funktionen sehen genauso aus wie benannte Funktion, aber zwischen **function** und den Parametern steht eben kein Name.

Zeile 5: *add* wird genauso verwendet, wie im Beispiel 1.

> **Important**
>
> Obwohl der Aufruf von *add* in beiden Beispielen gleich aussieht, gibt es einen wichtigen Unterschied. Im ersten Beispiel ist *add* der Name einer benannten Funktion. Im zweiten Beispiel ist *add* eine Variable, der eine anonyme Funktion zugewiesen wird.

Der Rest sieht genau so aus, wie im Beispiel 1, auch im Consolen-Fenster erscheint dasselbe.

Ebenso kann eine benannte Funktion Variablen zugewiesen werden:

```
let add = function neuerName (a,b) {
  return a+b;
};

let s = add(3, 4);
console.log("Summe = ", s);
```

Und wieder ist das Ergebnis dasselbe.

Eine *Konstante* sieht aus wie eine Variable, die mit **const** vereinbart wird und der gleich bei der *Vereinbarung* ein Wert zugewiesen werden *muss*. Damit wird eine nachträgliche unerwünschte Veränderung verhindert. Das geht natürlich auch mit Funktionen und wird dort sehr häufig verwindet: Ferner erleichtern Konstanten die Arbeit des Compilers.

```
const add  = function (a,b) {
  return a+b;
};

let s = add(3, 4);
console.log("Summe = ", s);
```

Funktionen – vor allem anonyme Funktionen – kommen in JavaScript sehr häufig vor. Kein Wunder, dass nach einer besonders einfachen Schreibweise gesucht wurde. Hier ist sie: unsere anonyme Funktion in *Pfeilscheibweise*, auch genannt *fat arrow* (*dickerPfeil*):

```
const add  = (a,b) => {
  return a+b;
};

let s = add(3, 4);
console.log("Summe = ", s);
```

Das Wort *function* verschwindet und der Pfeil wird nach den Parametern eingefügt.

Wenn wie diesem Beispiel der Rumpf der Funktion nur aus einem `return`, gefolgt von dem zurückzugebenden Wert besteht, geht es noch einfacher:

```
const add  = (a,b) => return a+b;

let s = add(3, 4);
console.log("Summe = ", s);
```

Eine Schreibweise wie in Zeile 1 ist in vielen Programmen zu finden.

Pfeilschreibweise

Noch kürzer:

Und wie sieht das Ganze mit einem oder mit keinem Parameter aus?

Funktionen mit einem Parameter

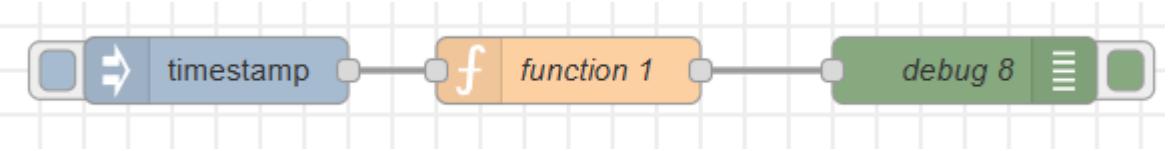
```
script.js ×

1   const quadrat = a => a*a;
2
3   let s = quadrat(3);
4   console.log("Quadrat = ", s);

Console ×

Quadrat =  9
```

Funktionen ohne Parameter (leere Parameterliste)

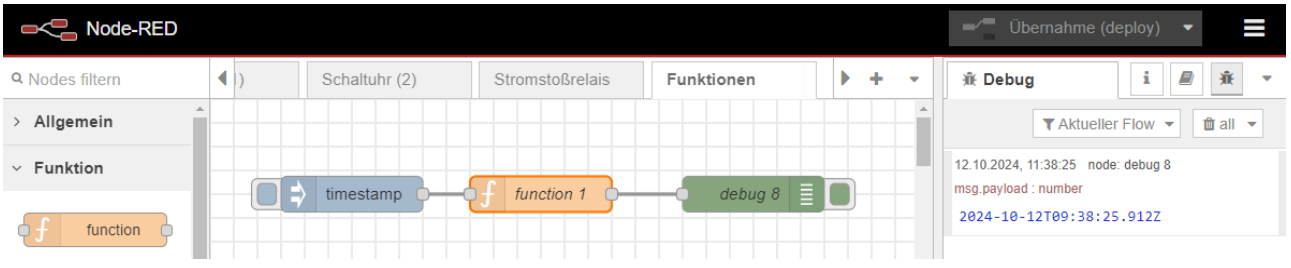
```
script.js ×

1   const jetzt = () => new Date().toUTCString()
2
3   let s = jetzt();
4   console.log("Jetzt = ", s);

Console ×

Jetzt =  Sat, 12 Oct 2024 09:33:05 GMT
```

Funktionen in Node-RED

Wie erwähnt erlauben Funktionen in Node-RED die Umsetzung komplexer Steuerungen. Wir beginnen mit einem einfachen Beispiel:

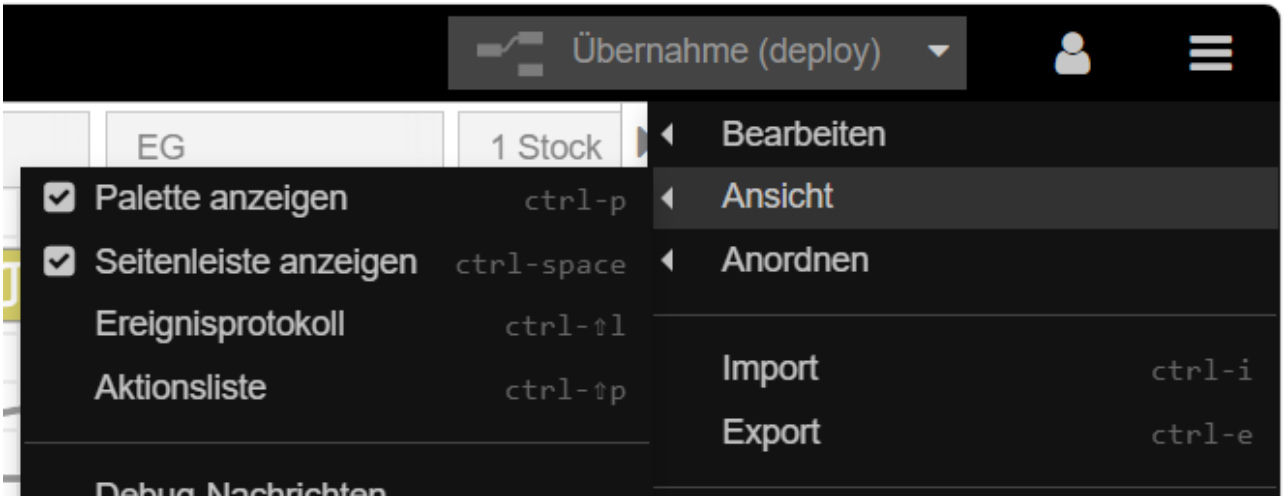


Mehr Details:



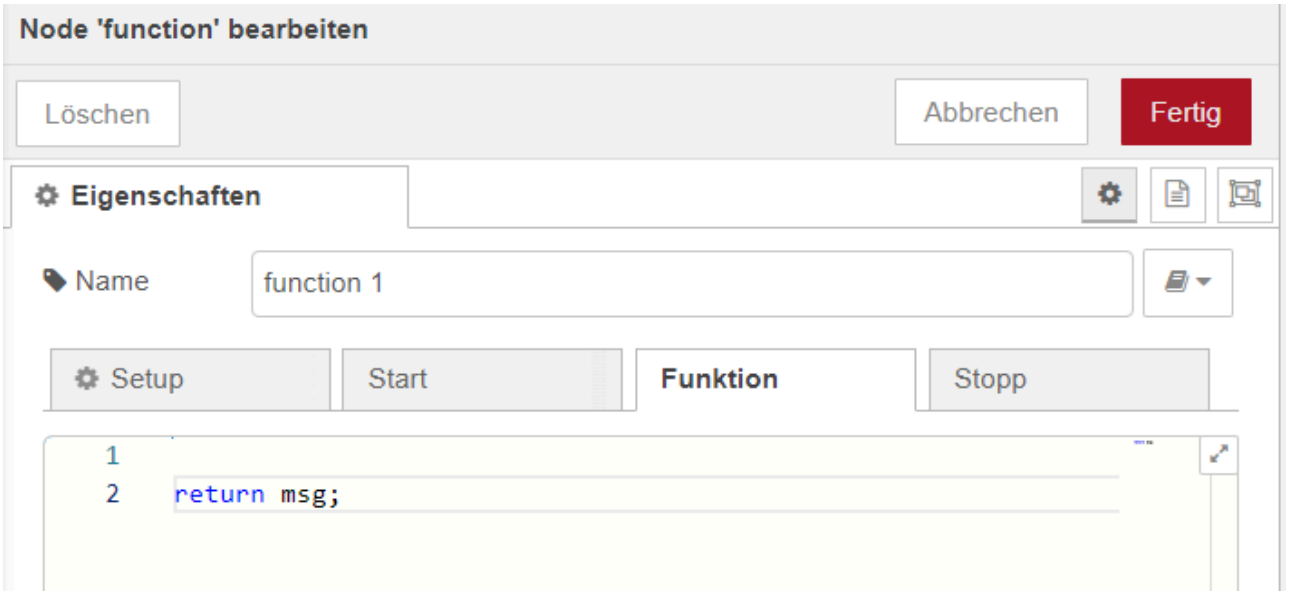
Der *inject*-Node liefert auf Knopfdruck die aktuelle Zeit, *debug* zeigt das Ergebnis im Debug-Fenster.

> **Note**
>
> Zur Erinnerung: wenn das Debug-Fenster nicht zu sehen ist, dann das Hamburger-Menü ☰ aufrufen und über `Ansicht` die `Seitenleiste anzeigen`.



Der Node *function*

Im einfachsten Fall übernimmt die *function* den Wert vom Eingang und gibt ihn ohne Änderung an den Ausgang weiter. Wird die *function* mit einem Doppelklick zur Bearbeitung geöffnet, zeigt sich, dass sie vorerst nur aus einer Zeile (`return msg;`) besteht.



Das ist natürlich keine JavaScript Funktion. Die vollständige Funktion lautet.

```
function (msg) {
  return msg;
}
```

Teile, die immer gleich bleiben, werden weggelassen.

- Die (anonyme) Funktion hat *genau einen Parameter mit dem Namen msg*. Da *msg* ein Objekt ist, kann es sehr viel an Information übertragen.

- Die geschwungenen Klammern um den Rumpf der Funktion werden weggelassen.
- Die Funktion liefert über return ein Ergebnis, üblicherweise wieder ein msg-Objekt mit der Eigenschaft payload. Der Name kann frei gewählt werden.

Beispiel:

```
let m = {
  payload: "Die Antwort ist 42"
}
return m;
```

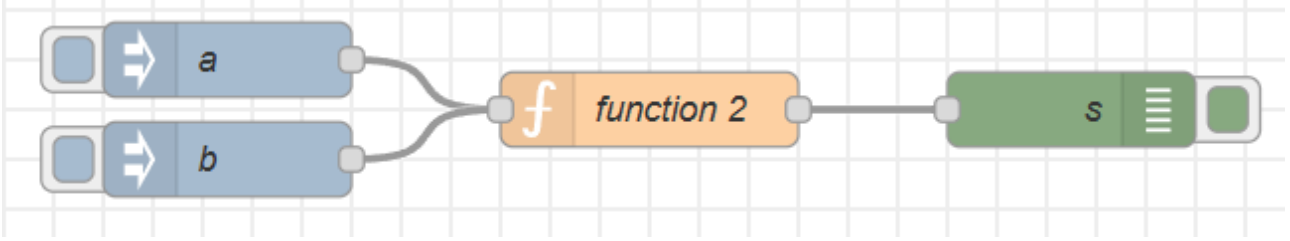
Liefert die Funktion `null` (das ist ein Schlüsselwort von JavaScript), wird kein Signal an den Ausgang gesendet.

> **Note**
>
> Im Setup der Funktion wird die Anzahl der Ausgänge festgelegt. Standard ist 1. Wird über *return* eine Liste ausgegeben, werden die Listenelemente der Reihe nach den Ausgängen zugewiesen.

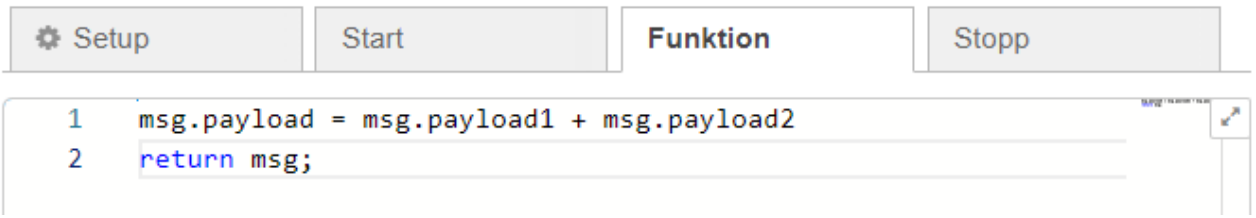
Addition

Wir haben zu Beginn dieses Beitrags mehrere Varianten kennen gelernt, wie in JavaScript eine Funktion zwei Zahlen addieren kann. Geht das auch mit der Node-RED Funktion? Jein.

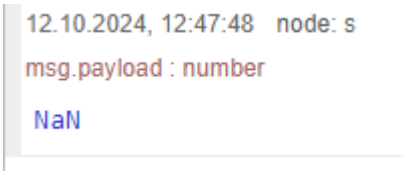
Versuchen wir es mit folgendem Flow:



Da die *function* genau einen Parameter (*msg*) am Eingang erlaubt, müssen bereits die *inject* Nodes auf zwei unterschiedliche *payloads* umgeschrieben werden. Natürlich könnten wir statt payload auch einen anderen Namen verwenden, zum Beispiel a statt payload1 und b statt payload2.



In der Funktion wird addiert. Klicken wir nun auf den *injection*-Node a, erscheint als Ergebnis *NaN*. Das bedeutet *Not a Number*, also sinngemäß *Das ist keine Zahl*. Was ist passiert? Node-RED wird in Smart Homes verwendet. Charakteristisch ist die Steuerung über Ereignisse. Beliebige Sensoren liefern zu unterschiedlichen Zeitpunkten Signale. Diese werden sofort verarbeitet - und dann werden sie vergessen. Daher ist die Summe von *einer Zahl* und *keiner (weiteren) Zahl* wenig überraschend *keine Zahl - Not a Number.*



Um die beiden Zahlen zu addieren, müssen wir sie irgendwie zwischenspeichern.

Speichern von Werten

Zum Speichern von Werten führt Node-RED den context ein. Damit können auch Werte zwischen verschiedenen Nodes ausgetauscht werden. Der Gültigkeitsbereich eines *context* kann eingeschränkt werden:

- *Node*: Werte sind nur in dem jeweiligen Node verfügbar.
  Verwendung: context.set, context.get
- *Flow*: Werte sind nur in dem jeweiligen Flow verfügbar. Ein Flow entspricht dem im Editor geöffneten Fenster.
  Verwendung: flow.set, flow.get
- *Global*: In allen Nodes sichtbar und verwendbar.
  Verwendung: global.set, global-get

> **Note**
>
> *context* steht einerseits *allgemein* für alle alle Varianten zum Speichern von Werten, andererseits *speziell* für das Speichern nur in jeweiligen Node.

Beim Setzen eines Wertes [**set**] werden zwei Parameter übergeben (der Schlüssel (key) und der Wert (value)), beim Auslesen [**get**] nur ein Parameter (der Schlüssel (key)).

Allgemein: die context- Werte werden nur im Arbeitsspeicher abgelegt. Bei einem Neustart oder einem Stromausfall gehen sie verloren. Node-RED kann aber auch context-Werte im lokalen Dateisystem speichern.

Wir ändern unsere *inject*-Nodes so, dass die beiden Eingaben für a und b nicht mittels payload1 und payload2 unterschieden werden, sondern über die *topics* (Themen) . Das wird in Node-RED recht häufig verwendet.

Verschiedene Topics

Die zugehörige Funktion scheint etwas kompliziert:

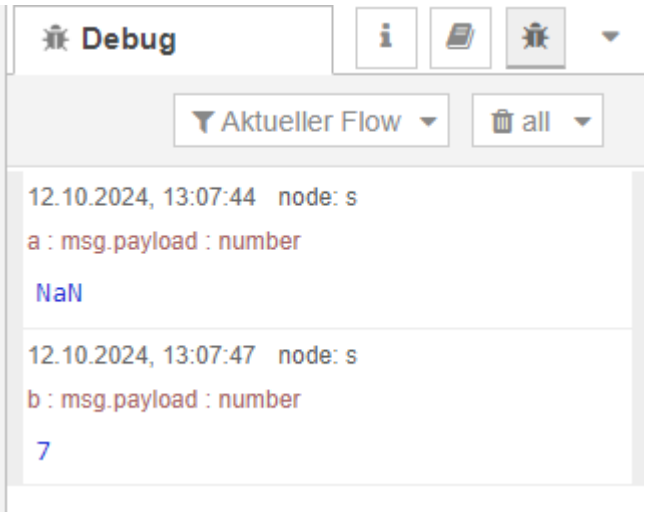
Da alle Eingangswerte über einen einzigen Parameter (*msg*) übergeben werden, wird zuerst über *topic* entschieden, von wo und wohin der Wert kommt (Zeilen 1 bis 3 und 5 bis 7).

Stellen wir uns den context als einen Kasten mit vielen Schubladen vor. Die Laden sind mit a, b usw. beschriftet. In der Zeile 2 wird die Schublade a mit dem Wert gefüllt, der über paylaod an die Funktion übergeben wird. Dasselbe geschieht in Zeile 6 für die Schublade b.

Wir wissen natürlich noch nicht, was und ob überhaupt etwas in den Laden liegt, aber wir probieren es einfach: was in Lade a liegt, wird mit context.get("a") geholt, der Inhalt von Lade b mit context.get("b"). Die Summe wird in payload gespeichert und dann über msg an den *debug*-Node übergeben.

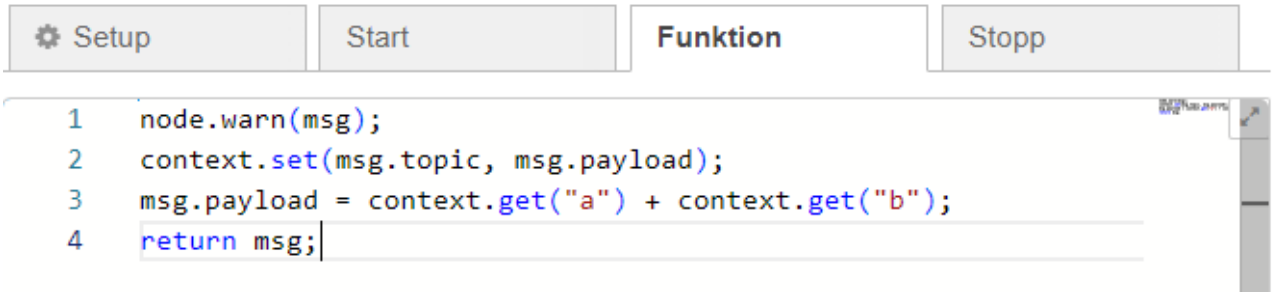
Wird nun die Taste von a angeklickt, wird zwar der Wert 3 über context.set("a", 3) gespeichert und kann gleich danach über context.get("a") wieder ausgelesen werden. Aber die Lade b ist noch leer. context.get("b") liefert keinen Wert, die Summe ist NaN (Not a Number).

Erst der Klick auf die Taste b speichert 4 in dem zweiten context und führt zur erwarteten Summe.

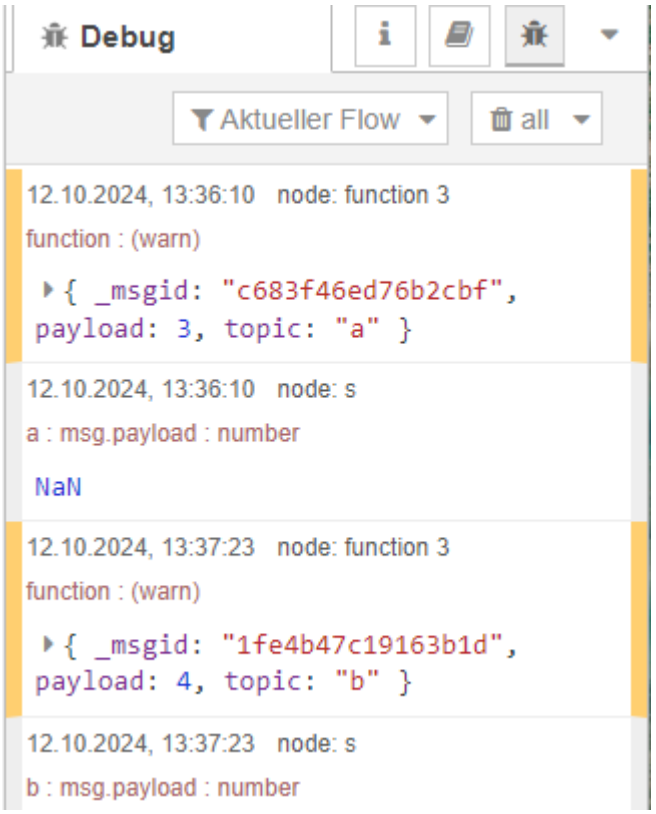


Die Verwendung der topics bei den beiden *injection*-Nodes erlaubt eine radikale Kürzung: aus den Zeilen 1 bis 7 wird die Zeile 2.

Oft ist es sinnvoll, zum Testen von Programmen Zwischenergebnisse im Debug-Fenster auszugeben, wie hier in Zeile 1.



Nach dem ersten Klick auf a ist im Debug-Fenster wie erwartet NaN zu sehen, nach dem Klick auf b das Ergebnis 7.

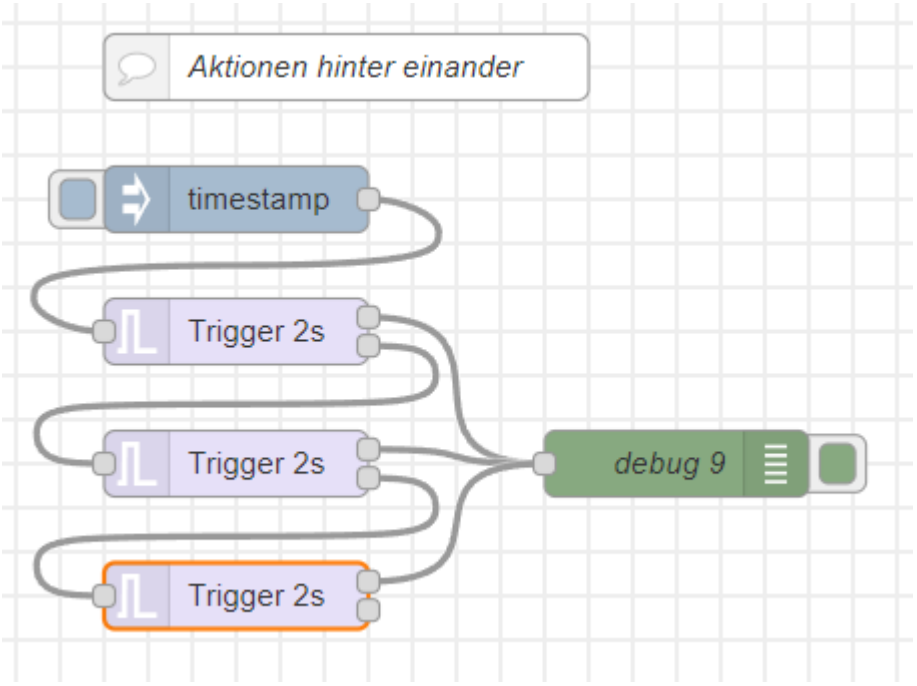


Aktionen nacheinander

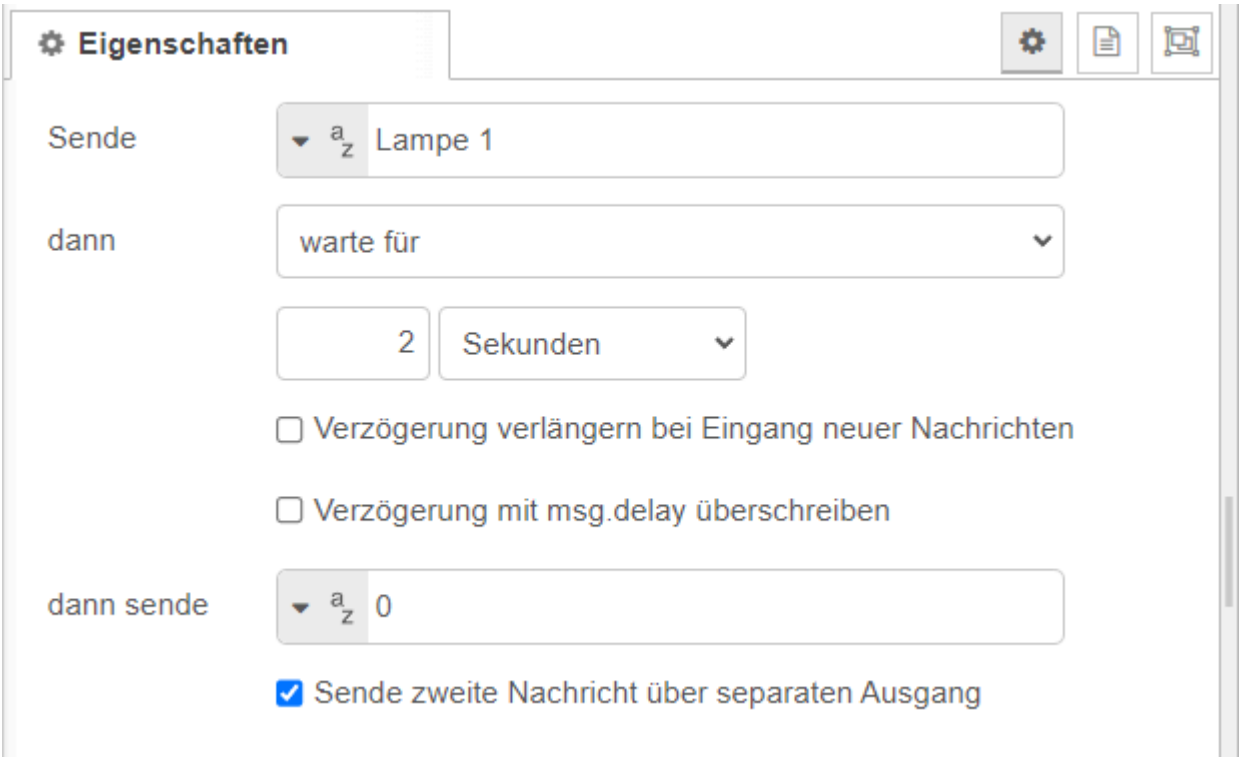
Eine andere Aufgabe: In der Früh sollen nacheinander mehrere Lampen (hier drei Stück) mit einem zeitlichen Abstand (hier zum Testen nach jeweils zwei Sekunden) eingeschaltet werden.

Lösung mit Trigger-node

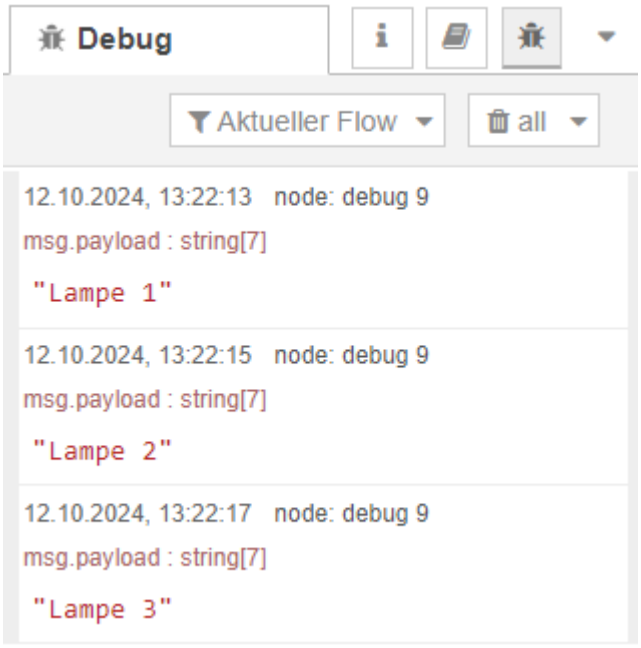
Das ist die einfachste Lösung. Jeder *trigger*-Node kann so eingestellt werden, dass er über einen zweiten Ausgang ein weiteres Signal zeitverzögert ausgibt. Die drei Lampen werden durch *debug*-Nodes simuliert.



So schaut jeder *trigger*-Node aus.



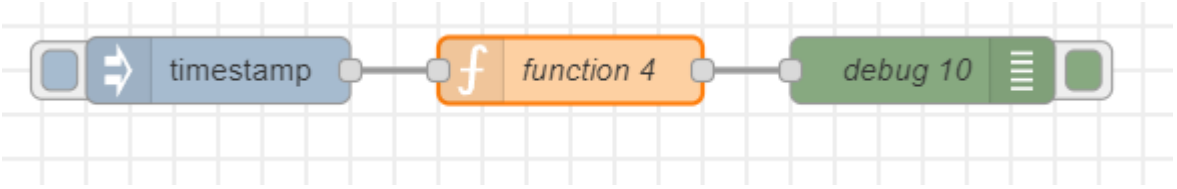
Das ist der Ablauf. Der Zeitstempel im Debug-Fenster zeigt, dass die einzelnen Vorgänge tatsächlich im 2-Sekunden-Takt stattfinden.

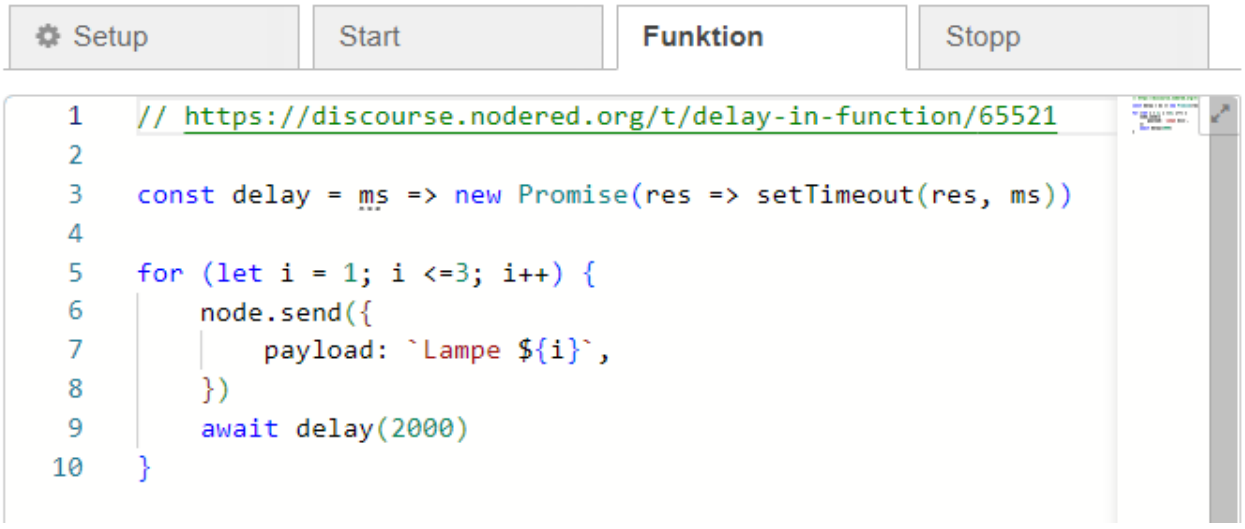


Lösung mit einer *function*

Kann dafür auch eine Funktion verwendet werden?

Ja, aber der Ablauf ist etwas komplizierter. Würde die Steuerung nach dem Einschalten einer Lampe einfach zwei Sekunden warten, könnten Ereignisse, die in der Zwischenzeit eintreten, nicht verarbeitet werden. Die Lösung ist eine asynchrone Steuerung, bei der die Wartezeit (2 Sekunden oder 2000 Millisekunden) in das (asynchrone) Versprechen (Promise) ausgelagert wird, nämlich dass es nach 2000 Millisekunden weitergehen wird.



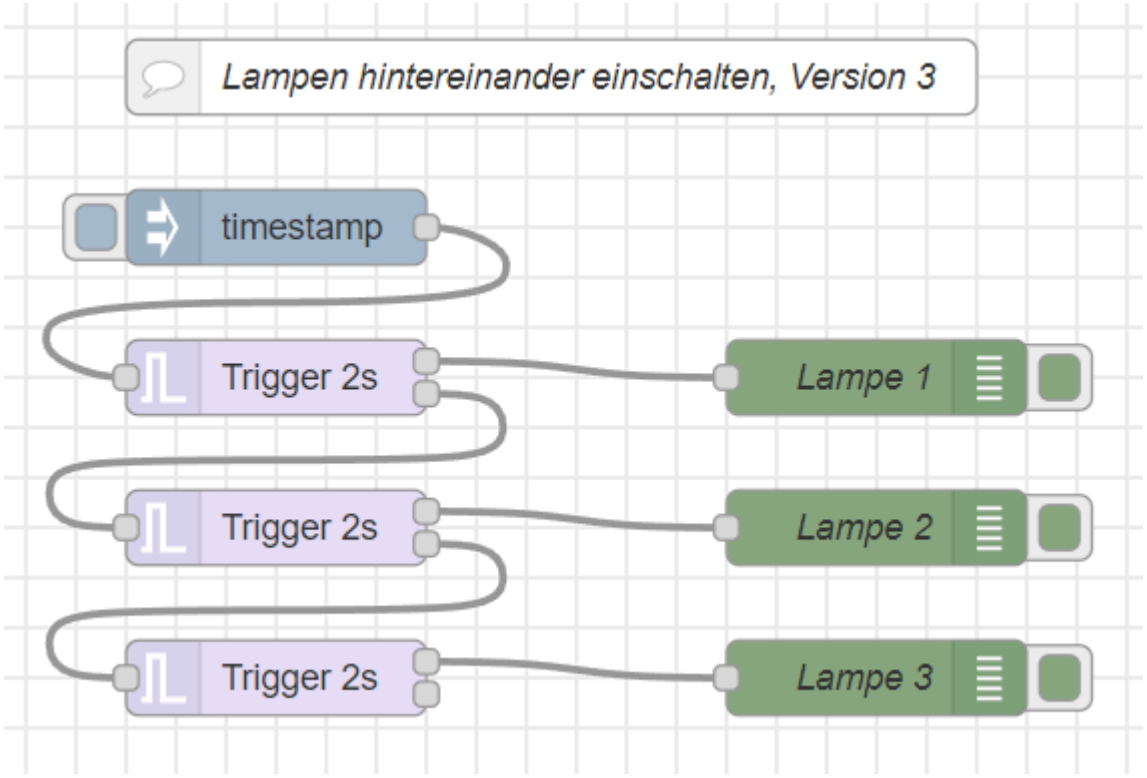


Die Zeichenkette `` `Lampe ${i}` `` ist ein *Template-String*, eine Vorlage. Dieser String ist in *grave accents* oder *backticks* ("`") eingeschlossen und kann sich über mehrere Zeilen erstrecken. Bei `${i}` wird `i` ausgewertet und an die Stelle von `${i}` in den String eingesetzt.

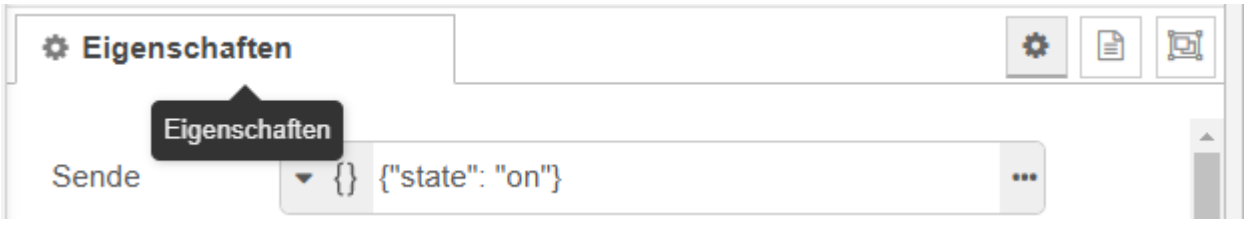
Und warum steht am Eingang `timestamp`? Nun, wir brauchen nur irgendetwas, das ein Signal für den Start ausgibt. Ein *inject*-Node liefert ohne Änderungen standardmäßig die aktuelle Zeit und die reicht zum Starten.

Getrennte Ausgänge

Genau genommen haben wir bis hierher geschummelt: statt drei Lampen zu verwenden, haben wir alle drei in einer einzigen Debug-Anweisung zusammengefasst. Das lässt sich mit echten Lampen nicht machen. Wir bessern die erste Lösung aus:



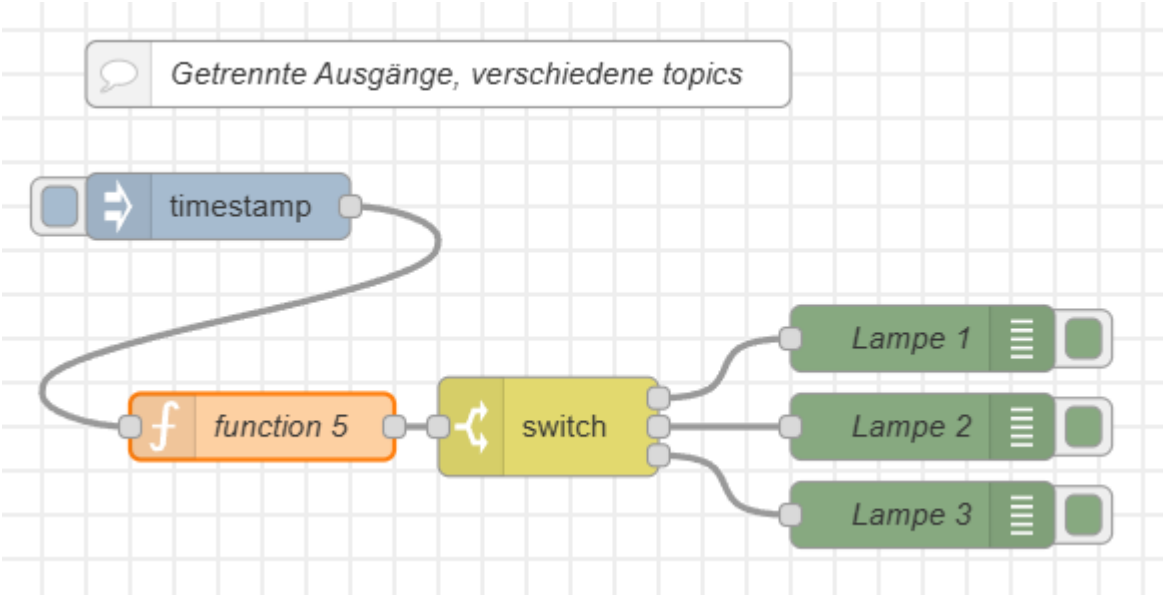
Um eine echte Lampe einzuschalten, wird bei den meisten Lampen oder Schaltern die payload auf `{"state":"on"}` gesetzt.



Die drei *debug*-Nodes zeigen:

Verwendung einer Funktion: verschiedene Topics

Wir können die Funktion anweisen, für jede Lampe ein eigenes topic zu verwenden und dann am Ausgang die topics über einen Switch auflösen:



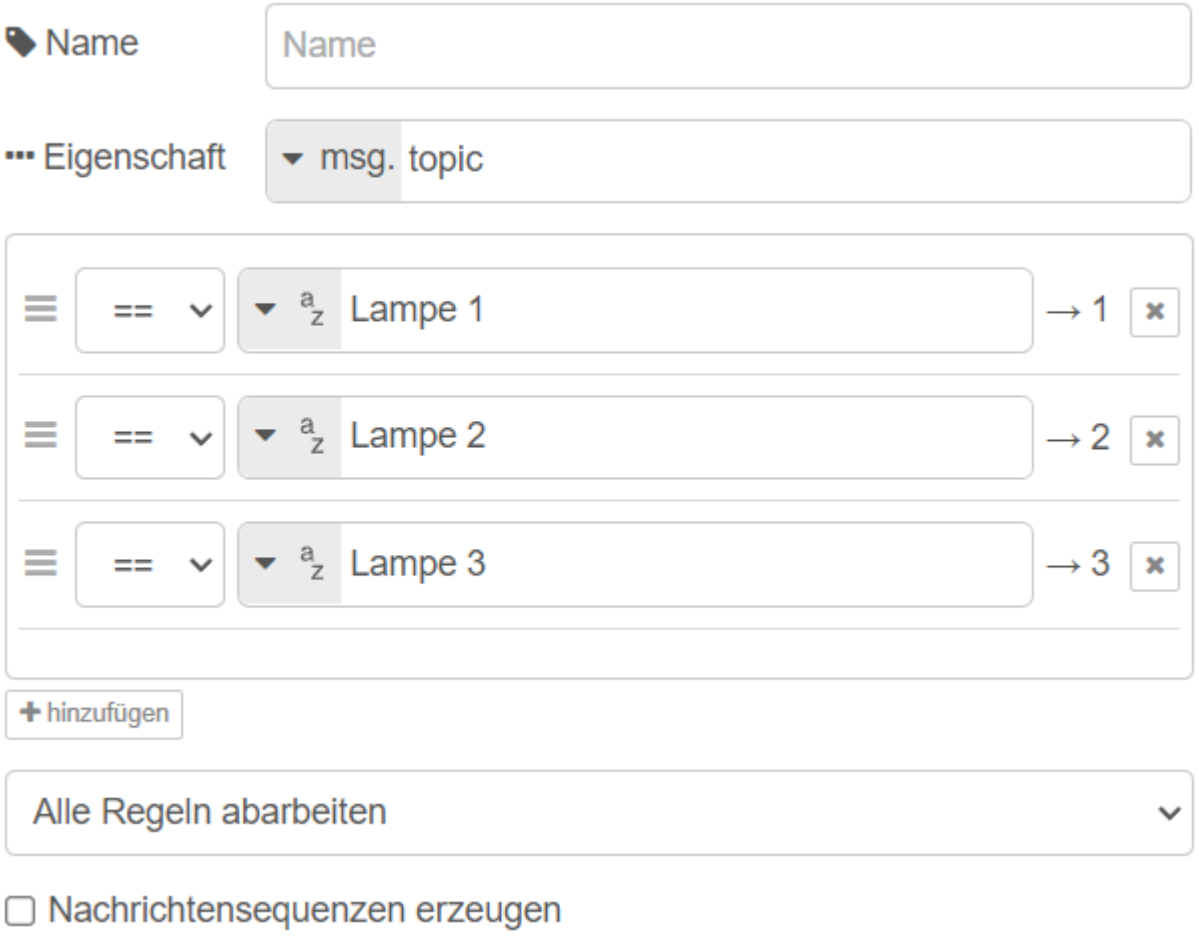
Das ist die geänderte Funktion:

Setup | Start | **Funktion** | Stopp

```
1  // https://discourse.nodered.org/t/delay-in-function/65521
2
3  const delay = ms => new Promise(res => setTimeout(res, ms));
4
5  for (let i = 1; i <=3; i++) {
6      node.send({
7          payload: {
8              "state": "on"
9          },
10         topic: `Lampe ${i}`
11     })
12     await delay(2000);
13 }
```

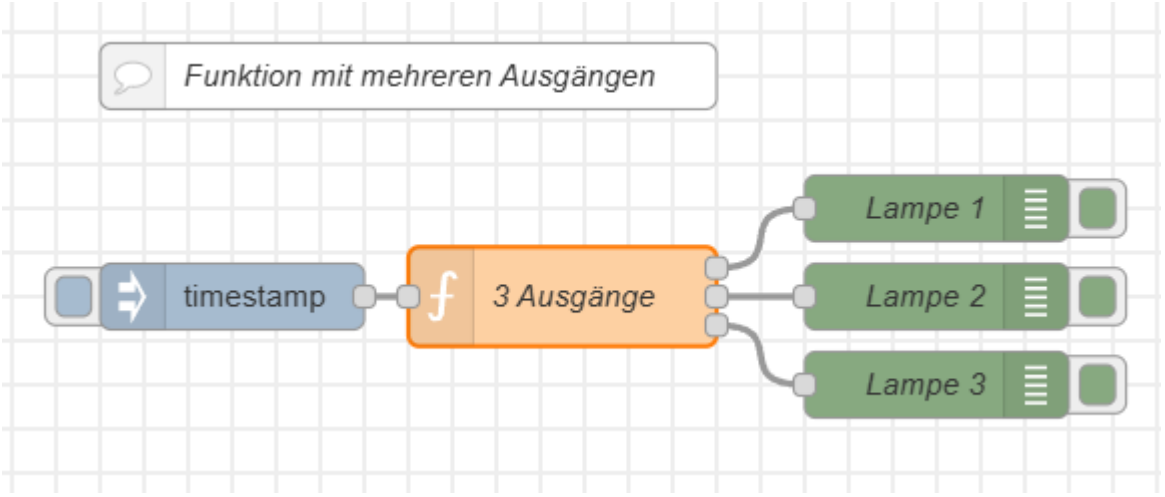
Switch

In einem Switch werden Eigenschaften des Eingangssignals mit vorgegebenen Werten verglichen. Bei Übereinstimmung wird das Eingangssignal an den zur Abfrage gehörenden Ausgang weitergeleitet. Hier wird das topic mit den Zeichenketten "Lampe 1", "Lampe 2" und "Lampe 3" verglichen.

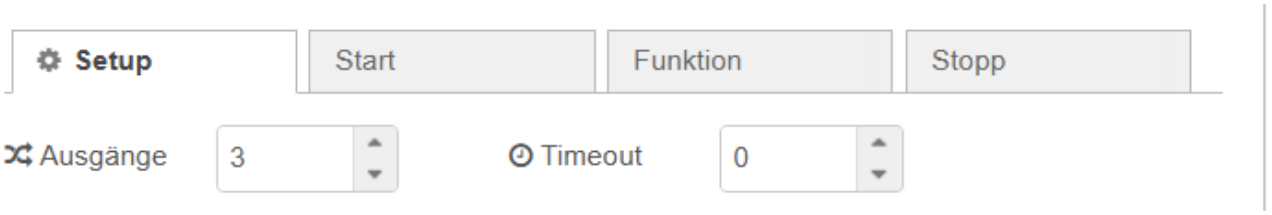


Funktion mit mehreren Ausgängen

Geht das nicht noch einfacher? Funktionen können mehrere Ausgänge ansteuern. So sieht es aus:



Im Setup der Funktion werden drei Ausgänge angefordert.



Das ist die Funktion selbst:

Da sind wohl ein paar Erklärungen fällig.

Zeile 3: Die Funktion *delay* bereitet eine Verzögerung vor. Die Dauer der Verzögerung, gemessen in Millisekunden, wird im Parameter *ms* übergeben.

Zeile 4: hier wird die Variable *lampenliste* vereinbart. In die Liste werden anschließend die Steuerbefehle eingefügt.

Zeile 5: In JavaScript-Listen beginnt die Zählung mit 0. i läuft daher von 0 bis 2.

Zeile 6: Bei drei Ausgängen ist eine Liste mit drei Eintragungen eine gute Wahl. null bedeutet, dass an die zugehörigen Ausgänge nichts geschickt wird - das ist der Anfangswert bei jedem Durchgang.

Zeile 7: jetzt wird ausgewählt, welcher Ausgang dran ist.

Zeilen 8 bis 10: um die Lampe einzuschalten, wird state auf "on" gesetzt.

Zeile 12: Jetzt wird die `lampenliste` an alle Ausgänge geschickt, aber nur ein Ausgang hat das Einschaltsignal `state:"on"` bekommen.

Zeile 13: 2000 Millisekunden warten.

Vielleicht nicht ganz so übersichtlich, aber dafür recht kompakt.
Asynchrone Funktionen sind eine großartige Sache, aber nicht einfach zu verstehen. Zur Erklärung ist ein eigener Beitrag in den PCNEWS erforderlich.

Lampensteuerung

Das Beispiel sollte zeigen, wie der Raum zum Wecken langsam heller gemacht werden kann. Noch wird jede Lampe schlagartig hell. Aber das geht besser.

Bei bestimmten Lampen kann eingestellt werden, dass sie *selbst* langsam von dunkel auf hell schalten. Bei der Lampe TRÅDFRI von IKEA ist im vorherigen Beispiel in Zeile 9 die payload von `{"state":"on"}` auf `{"brightness":80,"transition":20}` zu ändern. Die Helligkeit wird dadurch innerhalb von 20 Sekunden von 0 auf 80 erhöht. *brightness* ist eine Zahl zwischen 0 und 255.

Stiegenhausbeleuchtung

Das Licht im Stiegenhaus soll 20 Minuten vor Sonnenuntergang ein- und um 20 Uhr ausgeschaltet werden, nicht aber wenn die Sonne nach 20:20 Uhr untergeht.

Ferner soll das Licht um 6 Uhr ein- und 20 Minuten nach Sonnenaufgang ausgeschaltet werden, nicht aber wenn der Sonnenaufgang vor 05:40 Uhr ist.

Zuerst ein Nachtrag zum Big Timer

Im Teil 1 (Heft 182) wurde der Big Timer vorgestellt. Hier dazu eine Klarstellung:

Ausgang 1 liefert das msg-Signal,

- sobald sich der Zustand des Timers ändert oder
- einmal pro Minute, wenn *Repeat output* in den Big Timer Eigenschaften angekreuzt ist.

Ausgang 2 liefert *immer* einmal pro Minute die aktuelle Message (msg).

Einstellen des Big Timers

Der Big Timer liefert am Ausgang 2 die notwendigen Daten im msg-Objekt:

- Aktuelle Uhrzeit: msg.now, Sonnenaufgang: msg.sunrise, Sonnenuntergang: msg.sunset.
- Zeiten, die am Timer eingestellt werden:
  - On Time + On Offset → msg.start
  - Off Time + Off Offset → msg.end
  - On Time2 + On Offset2 → msg.start2

- Off Time2 + Off Offset2 → msg.end2

Wann soll das Licht eingeschaltet sein?

Wir setzen

- für den Abend On Time auf Sunset, On Offset auf 20 (Minuten), Off Time auf 20:00 Uhr und
- für den Morgen On Time2 auf 06:00 Uhr, Off Time2 auf Sunrise und Off Offset2 auf 20.

Schaut doch gut aus? Aber wenn die Sonne nach 20:20 Uhr untergeht, wird das Licht *danach* aufgedreht!

So geht es also nicht. Node-RED stellt UND- und OR-Verknüpfungen zur Verfügung - aber das ist recht umständlich. Einfacher geht es mit einer *function*.

Das Licht soll einerseits zwischen *start* und *end* und andererseits zwischen *start2* und *end2* eingeschaltet sein:

- start <= now<= end oder
- start2 <= now <= end2

Daraus wird:

```
msg.payload = ((msg.start <= msg.now) && (msg.now <= msg.end) ||
  (msg.start2 <= msg.now) && (msg.now <= msg.end2)) ? 1 : 0;
return msg;
```

Direkte Steuerung

Der Big Timer hat auch einen einzigen Eingang, mit dem der Zeitablauf übersteuert werden kann, das heißt, Signale von diesem *Eingang* überschreiben den *automatischen* Ablauf.

msg.state zeigt dann (beispielsweise) "On Override" oder "Off Override" an. Wenn also Override in `msg.state` enthalten ist, sollen die mit *msg.start* usw. eingestellten Zeiten ignoriert werden.

Die Zeitsteuerungsfunktion wird geändert und berücksichtigt das:

```
if (msg.state.includes("Override")) {
  return msg;
}

msg.payload = ((msg.start <= msg.now) && (msg.now <= msg.end) ||
  (msg.start2 <= msg.now) && (msg.now <= msg.end2)) ? 1 : 0;
return msg;
```

Sobald *msg.payload* auf 1 gesetzt ist, wird das Licht eingeschaltet, mit 0 ausgeschaltet.

Zusammenfassung

Der Beitrag zeigt (vor allem in der Langfassung), wie mit Funktionen in JavaScript umzugehen ist und wie mit Funktionen in Node-RED gearbeitet wird. Zwei konkrete Beispiele schließen den Beitrag ab.

Im nächsten Teil werden Beispiele gezeigt, wie eine Node-RED Steuerung mit den Nutzern kommunizieren kann: über Telegram, Dashboard 2.0 und User Interface. Für alle drei gibt es in der Palettenverwaltung fertige Nodes.

Die im Heft 182 angekündigten Zustandsdiagramme folgen in einer späteren Ausgabe.

Telegram

Mit ein paar Befehlen schickt die Node-RED-Steuerung über Telegram Nachrichten an den Nutzer, zum Beispiel die Raumtemperatur, den Zustand von Eingangstür oder von Fenstern (offen oder zu), Meldungen des Regensensors oder ein täglicher Bericht über den Wasserverbrauch.

Über Telegram können auch *Befehle an die Steuerung* geschickt werden.

Dashboard 2.0

Ohne eigene Programme zu schreiben, können mit Dashboard sehr einfach optisch ansprechende Benutzeroberflächen entwickelt werden, die auf Tablets, Desktopcomputern oder Smart Phones aufgerufen werden. Die Darstellung passt sich an die Größe des Bildschirms an (*responsive design*).

User Interface

Damit werden ebenfalls Benutzeroberflächen erzeugt, jedoch mit selbst geschriebenen Programmen in unterschiedlichen Programmiersprachen.

Zur Langversion

Die Entwicklung aller Beispiele wird in der Langversion sehr detailliert und mit vielen Zusatzinformationen und Bildern erklärt. Ferner können alle vorgestellten Flows auch unter dieser Adresse geladen werden. Lehrreicher ist es trotzdem, die Beispiele von Hand einzugeben.

Link und QR-Code: http://pcnews.at/markdown/n183/index.html oder https://t.ly/HcaYd

Erstellt mit QR Code Generator - kostenfrei, ohne Login, ohne Konto : https://goqr.me/

Short URL erstellt mit https://t.ly/

Copyright


BY NC SA