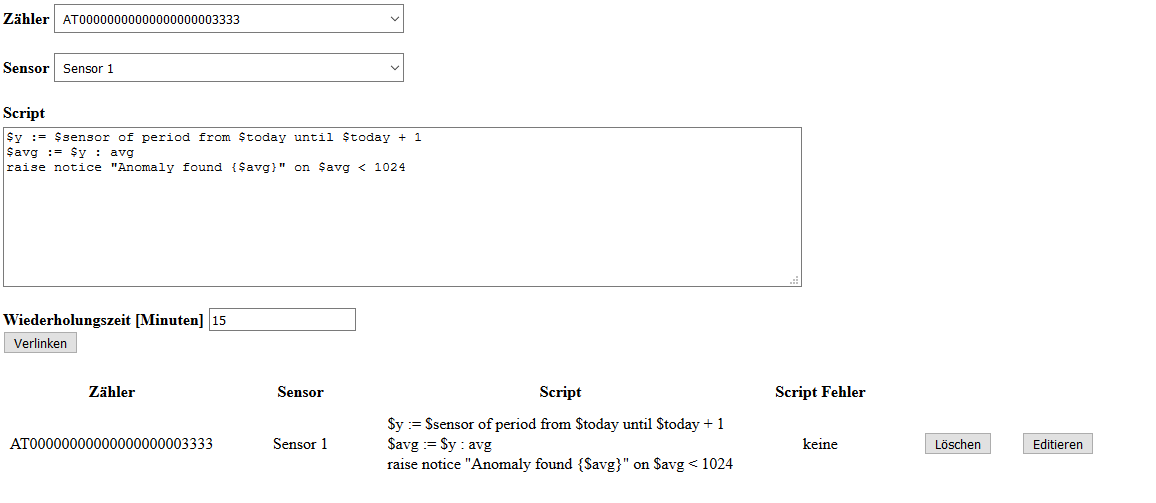
Für die Verwirklichung einer fundierten und benutzerdefinierten Anomalien-Erkennung wurde eigens eine interpretierte Script-Sprache entworfen. Diese Script-Sprache ermöglicht es dem Benutzer Anomalien genau nach dessen Bedürfnissen zu definieren.

Das Konzept der Anomalien-Erkennung besteht im Großen und Ganzen aus drei Bestandteilen nämlich die Script-Sprache selbst, der Interpreter der Script-Sprache und der WebUI, welcher eine graphische Benutzeroberfläche für die graphische Deklaration von Anomalien bereitstellt.

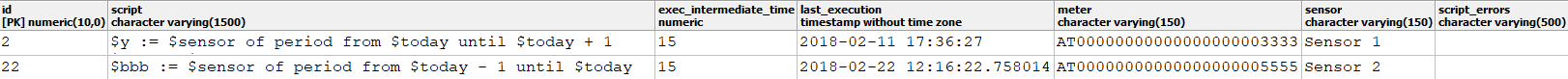
Als erstes wird das Konzept des WebUIs vorgestellt. Wie eingangs schon erwähnt ist der WebUI eine graphische Oberfläche, welche dazu Dient Anomalien zu deklarieren.



Im der oberen Abbildung sind die Bestandteile des WebUIs bemerkbar. Der erste Schritt besteht darin, den **Zähler** und den **Sensor** auszuwählen. Es werden nur jene Sensoren und Zähler angezeigt, welche der/die BenutzerIn auch berechtigt ist zu sehen. Die Daten, über welche der/die BenutzerIn Autorität besitzt, werden über die **AEMS-RestAPI** bereitgestellt. Anschließen gibt man sein **Script** in der AEMS-Scripting-Language ein, welche für die Anomalien-Erkennung konsultiert wird und stellt ganz unten unter **Wiederholungszeit** die Zeit in Minuten ein, nach welcher das Script periodisch ablaufen soll. Das Wiederholungszeiteingabefeld ist mit einem javax.faces.Validator versehen, welcher nur Eingaben erlaubt, welche durch fünfzehn ohne Rest teilbar sind. Im Bild gezeigten Beispiel wird das Script alle fünfzehn Minuten vom Interpreter exekutiert. Die Eingabe wird mit einem Klick auf **Verlinken** bestätigt und in die Datenbank als Anomalie eingetragen.

Im unteren Teil der Abbildung ist die Anzeige der bereits deklarierten Anomalien zu sehen. Dabei werden auch die vom Interpreter erkannten Script-Fehler, welche während der Exekution aufgetreten waren angezeigt. Rechts neben der jeweiligen Anomalien sind zwei Schaltknöpfe vorzufinden. Der erste ist mit „Löschen“ betitelt und der zweite mit „Editieren“. Der „Löschen“ Schaltknopf ist zum Entfernen der Anomalie aus der Datenbank, was mitunter auch zur Folge hat, dass das Anomalie-Script nicht mehr vom Interpreter der AEMS-Scripting-Language exekutiert wird. Der „Editieren“ Schaltknopf ist zum Editieren der jeweiligen Anomalie. Sobald auf den Schaltknopf gedrückt wird, werden alle Parameter in die oberen Felder eingetragen. Anschließen kann man diese Bearbeiten und mit einem erneuten Klick auf Verlinken werden die Änderungen in die Datenbank übernommen.

Wie zuvor schon mehrmals erwähnt, werden die Anomalien, welche mit Hilfe des WebUIs deklariert wurden in die Datenbank gespeichert. In nachfolgendem Bild ist ein Ausschnitt der Tabelle ***Anomalies*** dargestellt:



Die Tabelle besitzt alle notwendigen Spalten, um eine Anomalie zu beschreiben.  
Spalten:

* script: Das Skript, welches vom Interpreter ausgeführt wird
* exec\_intermediate\_time: Die Zeit in Minuten, bis wann das Skript erneut ausgeführt wird
* last\_execution: Diese Spalte wird für den Interpreter benötigt um festzustellen, ob es schon wieder an der Zeit ist, das Skript auszuführen. Dieser Wert wird nach jeder Exekution durch den Interpreter auf die Exekutionszeit gesetzt.
* meter: Eine Referenz auf die ***Meters*** Tabelle. Diese Spalte beschreibt, welcher Zähler an der Anomalien-Erkennung beteiligt ist.
* sensor: Eine Referenz auf die ***Meters*** Tabelle. Diese Spalte beschreibt, welcher Sensor an der Anomalien-Erkennung beteiligt ist.
* script\_errors: Diese Spalte ist sehr essentiell, da es für den/die EndbenutzerIn die einzige Möglichkeit der Fehlererkennung darstellt. Ohne diese Information könnte man keine Einsicht auf Skript-Fehler erhalten. Diese Spalte wird vom Interpreter befüllt, sobald Skript-Fehler während der Ausführung auftauchen.

Nachdem die Anomalien vom WebUI deklariert worden sind und in die Datenbank eingefügt worden sind, kann der Interpreter sich die Anomalien über die **AEMS-RestAPI** holen. Doch bevor im Detail auf die Verarbeitung der Anomalien eingegangen wird, wird zuvor noch die grundlegende Funktionsweise des ***AEMS-Scripting-Language-Interpreters*** erklärt. Der Interpreter ist ein Java8-Programm, welches auf dem Server läuft. Alle fünfzehn Minuten werden die Anomalien neu von der Datenbank akquiriert und deren Skripts exekutiert, falls die *Wiederholungszeit* oder exec\_*intermediate*\_time das zulässt. Der Grund dafür, dass der Interpreter nur alle fünfzehn Minuten die Anomalien aktualisiert ist, da die kleineste Zeitscheibe, welche die Stromzähler, Gaszähler, etc. darstellen fünfzehn Minuten ist. Das heißt nur alle fünfzehn Minuten kommen neue Zählerwerte in der Datenbank an. Somit ist es nur logisch, die kleinste Einheit der Exekution auf fünfzehn Minuten zu fixieren.

So nun da das Grundkonzept des Interpreters erläutert wurde, geht es nun ein bisschen in die Tiefe. In nachfolgendem Bild ist ein Teil der Interpreters zu sehen, welcher dafür zuständig ist, die Anomalien aus der Datenbank zu holen und diese dann in Anomaly-Objekte zu verwandeln. Diese Anomaly-Objekte sind eine interne Repräsentation, welche zur einfacheren Verarbeitung der Anomalien dient. Die Daten werden mithilfe der **AEMS-RestAPI** akquiriert und sind **AES** verschlüsselt. Bevor der Datenaustausch stattfinden kann, muss ein Schlüsselaustausch passieren.



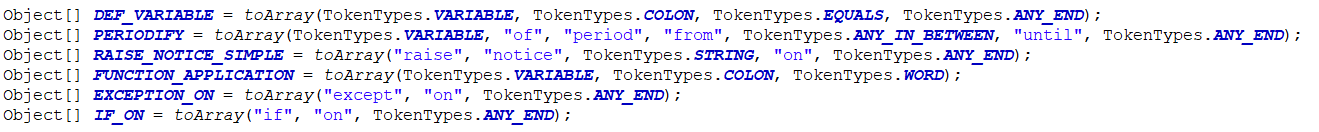
**Schlüsselaustasch**



Im nächsten Bild ist der Algorithmus abgebildet, welcher für die periodische Exekution der Skripts verantwortlich ist. Wie im Bild zu erkennen ist, wird vor jedem Durchgang *fetchAnomalies* aufgerufen, was die Methode ist, welche oben näher beschrieben wurde. Anschließend wird in *startTime* die Startzeit vermerkt, bei welcher der Durchgang gestartet wurde. Dies ist wichtig für später, da das für die Berechnung, der noch zu wartenden Zeit ausschlaggebend ist. Um das mit der *noch zu wartenden Zeit* besser zu verstehen, hier ein kleines Beispiel: Die Exekution aller Skripts benötigt insgesamt sieben Minuten. Damit nun die Skripts in einem fünfzehn Minuten Zyklus ablaufen, muss die Differenz zwischen der Dauer der Exekution und der fünfzehn Minuten errechnet werden. Diese Differenz ist dann die Wartezeit. Nach dieser Wartezeit beginnt der nächste Durchgang. Am Ende jedes Durchgangs wird *updateAnomalies* aufgerufen, welche die Exekutionszeit der jeweiligen Anomalien aktualisiert. Wenn ein Skript-Fehler auftritt, wird die Methode *uploadScriptError* aufgerufen. Diese schreibt dann in die dafür vorgesehene Spalte *script\_errors* von der Anomalie in der Datenbank den aufgetretenen Fehler. In der ganz innersten Schleife des Algorithmus wird dem Parser Zeile für Zeile des Skripts übergeben. Dies ist notwendig, da der Parser nach dem **One-Statement-Per-Line Prinzip** funktioniert. Dies bring Vorteile mit sich zum Beispiel ist kein besonderes Trennzeichen, wie ein Strichpunkt erforderlich, um das Statement abzugrenzen. Des Weiteren sorgt dieses Prinzip für eine bessere Lesbarkeit, da die Statements eher einfach gehalten werden.



Nachdem die Zeile dem Parser übergeben wurde, muss überprüft werden, welches Statement diese Zeile repräsentiert. Dazu wurden die Patterns für die Statements wie folgt definiert:

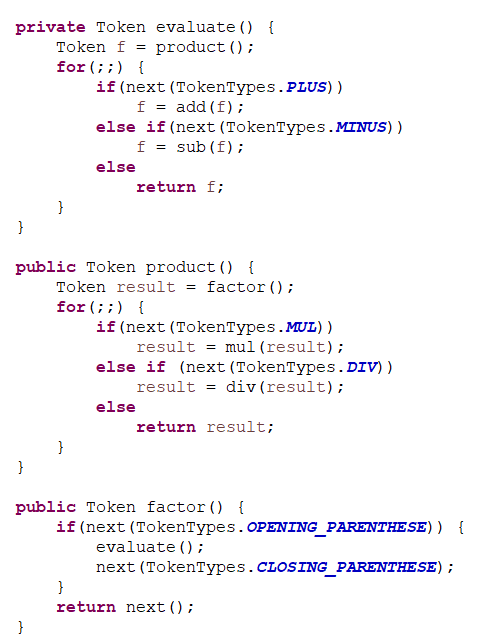


Die Klasse *TokenTypes* beinhaltet jede Art von Token, welche der *Tokenizer* als ein valides Token erkennt. Jedes Statement besteht aus einer Aneinanderreihung von Tokens, welche wie im oberen Bild gezeigt als Object[] definiert werden. Es gibt auch ein paar spezielle Tokens, welche sogenannte Auxilary-Tokens sind, nämlich ANY\_IN\_BETWEEN und ANY\_END. Das ANY\_IN\_BETWEEN Token ignoriert solange alle Tokens dazwischen bis das nächste Token in der Statement-Definition (siehe Abbildung) gefunden wurde. Das ANY\_END Token ignoriert alles was nach diesem Token folgt.



In der oberen Abbildung ist der Algorithmus zum Erkennen der Statements abgebildet. Die Methode erhält als Eingangsparameter zum einen die Statement-Definition und zum anderen die Zeile, welche dem Parser als Input übergeben wurden.

Um numerische Ausdrücke auswerten zu können wurde ein Algorithmus von Jack W. Crenshaw aus Compiler Building Tutorial vom 10. Oktober 2012 konzeptionell aufgegriffen.



In den jeweiligen Methoden wie add, mul, sub und div werden die Datentypen für die jeweiligen mathematischen Operation berücksichtigt. Das heißt, dass Daten anders als numerische Werte addiert, subtrahiert, multipliziert oder dividiert werden.



In der oberen Abbildung ist ein Teil des Algorithmus für die Evaluation von booleschen Ausdrücken abgebildet. Anders als der numerische Parser ist dieser Parser zu hundert Prozent Eigenbau. Da der AND Operator stärker bindet als der OR Operator muss zuerst der Token-Input beim OR Operator gesplitet werden und dann beim AND Operator. Nachdem bei einem AND gesplitet wurde wird nun auch beim Vergleichsoperator gesplitet. Danach wird die linke und die rechte Seite dem numerischen Parser übergeben. Falls das Ergebnis für die linke Seite nicht definiert sein sollte, könnte es sich um eine Liste handeln, falls nicht, wird ein Fehler geworfen. Nachdem die linke und die rechte Seite ausgewertet wurden und einen Wert haben, werden über ein switch-Statement die Entsprechenden Funktionen angewandt. Die Auswertung des booleschen Ausdrucks ist erst dann zu Ende, wenn entweder ein Fehler auftritt oder keine ORs und ANDs mehr gefunden werden können.

***AEMS-Scripting-Language*** Syntax

Die Syntax besteht im Grundlegenden aus sechs Statements:

* DEF\_STATMENT
* PERIODIFY
* RAISE\_NOTICE\_SIMPLE
* EXCEPTION\_ON
* IF\_ON
* FUNCTION\_APPLICATION

**DEF\_STATMENT**

Das DEF\_STATEMENT ist das DEFINE-STATMENT. Es ist das Grundstatement, welches eine Variablendeklaration beschreibt. Der Aufbau dieses Statements ist wie folgt:

$<var\_name> := <any>

Auf der linken Seite des Ausdrucks befindet sich die Variable, also **$<var\_name>**, welcher man den Wert auf der rechten Seite zuweisen möchte, also **<any>**, wobei der Variable Name nur aus kleinen und großen Buchstaben von A bis Z bestehen darf. **<any>** stellt ein beliebige Reihen von Tokens dar.

Jedes DEFINE-STATEMENT hat einen Eintrag in den SymbolTable zur Folge, welcher für die Verwaltung von „Symbolen“, also in diesem Fall sind es nur Variablen, verantwortlich ist. Über den SymbolTable kann der aktuelle Wert der Variable abgerufen werden. Variable werden aber als *immutable* angesehen, was heißt, dass sie nur einmal einen Wert zugewiesen bekommen können. Sie sind also Konstanten.

**PERIODIFY**

Das PERIODIFY-STATEMENT ist ein DEFINE-STATMENT. Nachdem ein DEFINE-STATMENT detektiert wurde, wird überprüft, ob es sich nicht um ein PERIODIFY-STATEMENT handelt. Der Aufbau des Statements ist wie folgt:

$<var\_name> := ($meter|$sensor) of period from <from\_time> until <to\_time>

Gewisse Werte lassen sich periodisieren. Diese Werte sind beinhaltet in den zwei Variablen *$meter* und *$sensor*. Diese zwei Variable gehören zu den vordefinierten Variablen der Script-Sprache, auf welche später noch genauer eingegangen wird. Charakteristisch für dieses Statement ist die Periodisierung des Wertes, welcher vor *of period from* steht. Sowohl **<from\_time>** als auch **<to\_time>** sind Platzhalter für Datumswerte oder Ausdrücke, welche in Datumswerte resultieren. In der Script-Sprache gibt es nur eine Möglichkeit Datumswerte zu spezifizieren und zwar in dem man von dem Referenzwert *$today* ausgeht. Dies ist eine weitere vordefinierte Variable, welche das Datum des aktuellen Tages gespeichert hat. Somit wäre ein mögliches Statement:

$values := $meter of period from $today - $week \* 2 until $today

Beim oben gezeigten Statement würde der Interpreter die Zählerwerte innerhalb der zwei Datumsgrenzen und zwar von *$today - $week \* 2* (vom heutigen Datum subtrahiere zwei Wochen) bis heute über die **AEMS-RestAPI** abfragen und diese dann als Liste in *$values* abspeichern. Alle Werte, welche innerhalb dieser Periode liegen befinden sich nun in *$values*. Diese Variable hat nun den Datentyp LIST.

**RAISE\_NOTICE\_SIMPLE**

Dieses Statement ist eines der Kernelement der Scripting-Language. Der Aufbau sieht wie folgt aus:

raise notice “<msg>“ on <any\_conditions>

Wenn die Bedingungen, welche Angegeben wurden sich auf *true* auflösen, dann wird ein Eintrag in der Datenbank getätigt, dass eine Anomalie entdeckt wurde. Die Nachricht, welche in die Datenbank gespeichert wird steht unter Hochkomma. Am nachfolgenden Beispiel ist zu erkennen, dass die Nachricht auch dynamisch zusammengestellt werden kann:

$x := $meter of period from $today – 1 until $today  
$avg := x : avg  
raise notice “Anomaly found! Exceeded average consumption of {$avg}” on $meter > $avg

Mit den geschweiften Klammern ist es möglich Variablenwerte in die Zeichenkette zu integrieren. Die Bedingungen sind Boolesche-Ausdrücke (siehe Boolesche Ausdrücke).

**EXCEPTION\_ON und IF\_ON**

Die EXCEPT-ON und IF-ON Klauseln werden an das RAISE-NOTICE-SIMPLE-STATMENT angehengt. Wenn man möchte, dass eine Notice nur an bestimmten Tagen berücksichtigt wird, kann man dies mit diesen Klauseln tun, zum Beispiel:

raise notice “Anomaly found!” on $meter > $avg if on $monday, $tuesday  
raise notice “Anomaly found!” on $meter > $avg except on $monday, $tuesday

Die IF-ON Klausel bewirkt, dass eine Notice nur an Montagen und Dienstagen erhoben wird, wobei die EXCEPT-ON Klausel bewirkt, das seine Notice nur dann erhoben wird, wenn der aktuelle Tag weder ein Montag noch ein Dienstag ist.

**FUNCTION\_APPLICATION**

Wie der Name schon verrät handelt es sich hier um die Anwendung ein Funktion, genauer gesagt um eine Aggregatsfunktion. Vier Funktionen stehen zur Verfügung, nämlich MIN, MAX, AVG, MEAN.  
MIN gibt aus einer Liste von periodisierten Werten den kleinsten Wert zurück und MAX den Größten. AVG berechnet das arithmetische Mittel, wohingegen MEAN den Median berechnet. In der Anwendung würde das Ganze so aussehen:

$values := $meter of period from $today - $week \* 2 until $today  
$avg := $values : avg  
$mean := $values : mean  
$min := $values : min  
$max := $values : max

Die Syntax des FUNCTION-APPLICATION-STATEMENTS wurde aus dem DEFINE-STATEMENT abgeleitet. Somit ist der Aufbau:

$<var\_name> := $<var\_name> : <function\_name>

**Vordefinierte Variablen**

Wie zuvor schon angekündigt enthält die Script-Sprache unter anderem auch vordefinierte Variablen. Die zwei wichtigsten Variablen im gesamten Script sind $meter und $sensor (beide vom Type NUMBER), wobei $meter den aktuellen Wert des Zählers und $sensor den aktuellen Wert des Sensors beinhaltet. Eine weitere vordefinierte Variable ist $today, diese ist vom Typ DATE und beinhaltet das aktuelle Datum. Weitere Variable sind $monday, $tuesday, $wednesday, $thursday, $friday, $saturday, $sunday sowie $day, $week, $month, $year. Alle diese Variable sind vom Typ NUMBER.

**Datentypen**

Wie fast jede Script-Sprache hat auch diese Script-Sprache eine Einteilung der Daten in sogenannte Datentypen. Diese Datentypen werden jedoch nur intern für die Evaluation der Statements verwendet und müssen nicht von dem/der BenutzerIn deklariert werden. In dieser Script-Sprache wird der Datentyp anhand des Ausdruckes abgeleitet. Es werden zwischen drei Datentypen unterschieden, nämlich DATE, NUMBER und LIST. Der Datentyp DATE stellt unschwer zu erkennen einen Datumswert dar und der Datentyp NUMBER einen numerischen Wert. Der Datentyp LIST deutet dem Interpreter an, dass diese Variable mehr als nur einen Wert repräsentiert und somit auch Aggregatsfunktionen (MIN, MAX, AVG, MEAN) sowie spezielle List-Operationen vorgenommen werden können. Spezielle List-Operationen sind zum Beispiel das Vergleichen einer Liste mit einem numerischen Wert: $list < 5. In diesem Beispiel ist $list eine Liste von Werten. Der Vergleich mit der numerischen Zahl fünf bedeutet, sobald eine Zahl in der Liste kleiner ist als fünf wird die Bedingung in true resultieren.

**Ausdrücke**

Es gibt zwei Arten von Ausdrücken, welche evaluierbar sind, nämlich numerische Ausdrücke und boolesche Ausdrücke.

**Numerische Ausdrücke**

Wie in den meisten Script-Sprachen ist es auch in dieser Script-Sprache möglich numerische Ausdrücke zu deklarieren, zum Beispiel: 5 + 3 \* 7 + 6. Dies funktioniert genauso wie mit jedem handelsüblichen Taschenrechner, sprich mit Operator-Precedence (Punkt vor Strich). Numerische Ausdrücke können nicht nur Zahlen beinhalten, sondern auch Variable, welche sich in Zahlen auflösen. Es gibt auch eine Ausnahme in Punkto dieser Regel. Die Variable $today kann auch in einem numerischen Ausdruck vorkommen, sie muss jedoch den Start des Ausdrucks bilden, d. h. $today + $week ist ein valider Ausdruck, wobei $week + $today ein ungültiger Ausdruck ist. Das beruht auf dem Konzept des Interpreters, welcher als Regel definiert bekommen hat, dass jener Datentyp, welcher am Beginn eines Ausdruckes steht, auch der Datentyp ist, in welchen der Ausdruck resultiert. Dies ist insbesondere wichtig, da man mit Daten anderes rechnen muss, als mit Zahlen.

**Boolesche Ausdrücke**

Boolesche Ausdrücke sind Ausdrücke, welche eine wahre oder eine falsche Aussage als Ergebnis haben. Ein beispielhafter boolescher Ausdruck ist: $list < 5 + 3 \* 5 and $x > $y or $x < $z. Das Konzept birgt wieder eine Operator-Precedence. Das AND bindet stärker als das OR. Es können so viele ANDs oder ORs verwendet werden, wie man möchte, solange der Ausdruck in eine wahre oder eine falsche Aussage resultieren kann. Bei Variablen vom Datentyp LIST haben Vergleichsoperatoren einen besonderen Effekt. Es ist möglich eine Liste mit dem > oder < Operator mit einer numerischen Konstante oder Variable zu Vergleichen. Dabei wird jeder Wert verglichen und kontrolliert, ob dieser entweder kleiner oder größer als der angeführte numerische Wert ist. Wie man auch an dem angeführten Beispiel erkenn kann, ist der numerische Parser ein Teil des booleschen Parsers. Es können also numerische Ausdrücke in boolesche Ausdrücke geschachtelt werden.