Projekt 1

Stocksearch

Lösungsdokumentation

Studiengang:	Informatik
Autor:	Fedor Gamper, Gian-Luca Frei
Betreuer:	Stefan Cotting
Experten:	Christoph Schaller
Datum:	29.05.2018

Management Summary

Es wurde eine Applikation erstellt welches einem erlaub nach einer eigens entworfenen Abfragesprache SSQL Aktiendaten zu suchen. Dazu wurde eine Python-Programm entwickelt welches SQL-Statements generiert und diese auf einer Datenbank aufrührt. Ebenfalls wurde ein Webinterface erstellt welches neben einem fortgeschrittenen Modus, auch einen einfacheren geführten Modus bietet. Dieses Dokument zeigt erklärt die wichtigsten technischen Konzepte und soll es ermöglichen den Source-Code zu verstehen und weiterzuentwickeln. Ebenfalls zeigt dieses Dokument wie man StockSearch auf einer Linux-Maschine installiert und betreibt.

Inhaltsverzeichnis

1.1 Systemarchitektur

1.1.1 Überblick

Im Folgenden ein kurzer Überblick über die Architektur der gesamten Applikation:

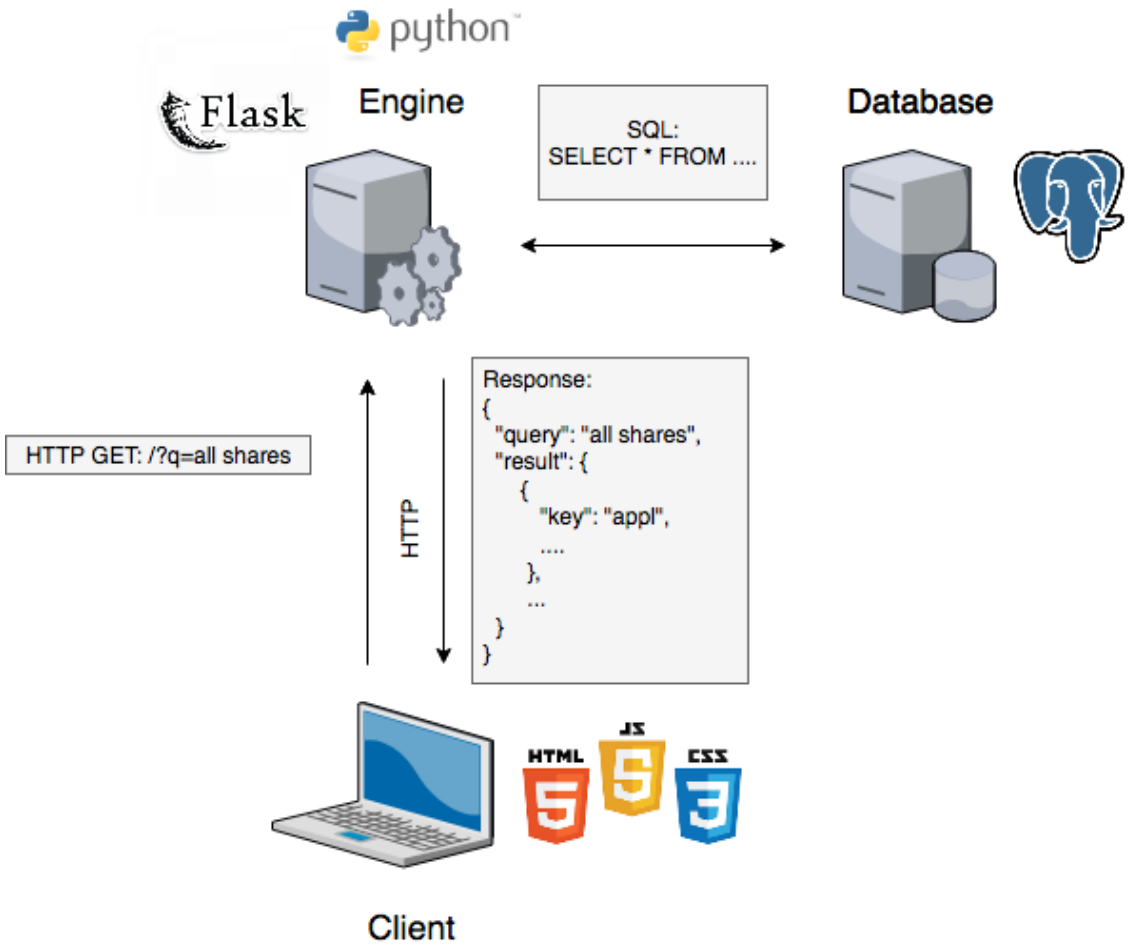


Abbildung 1 Systemarchitektur

Im Wesentlichen besteht die Applikation aus drei grossen Teilen, namentlich die Engine, der Client sowie die Datenbank. Der Client ist das Programm, welches vom Benutzer bedient wird. Es steht dem Benutzer eine graphische Oberfläche zu Verfügung und sendet im Hintergrund die Abfragen an die Engine. Die Engine nimmt die Abfragen entgegen und verarbeitet diese in dem sie mit der Datenbank interagiert.

Die Kommunikation zwischen dem Client und der Engine läuft wie HTTP, genauer über eine REST-API. Die Antworten werden von der Api im Json-Format zurückgesendet.

1.1.2 Engine

Die Engine ist sozusagen das Herzstück des Systems. Sie verarbeitet Abfragen in dafür entwickelten StockSearch-Query-Language, kurz SSQL. Die Engine wird in Python programmiert, da sich diese Sprache für Serveranwendungen gut eignet, sehr populär ist und zudem einfach zu lesen.

Für die Kommunikation via Http wird Flask verwendet, ein Microframework für Serveranwendungen mit Python. Zudem verwenden wir Modgrammar, eine freie Parsing-Bibliothek sowie Doctest und Pytest.

1.1.3 Client

Der Client bietet dem Benützer eine Oberfläche um die Aktiendaten abzufragen, dazu erstellen wir eine Seite welche im Browser aufgerufen werden kann. Dies hat im Gegensatz zu einem nativen Programm den Vorteil, dass wir die Webtechnologien JavaScript, HTML und CSS verwenden können, welche sehr geeignet sind um graphische Oberflächen zu erstellen. Zudem entspricht es dem allgemeinen Trend, dass immer mehr Programme im Browser bedient werden. Wir verwenden dazu JQuery und Bootstrap.

1.2 Datenbank

1.2.1 Datenbankmodel

Überblick über das Datenbankmodel

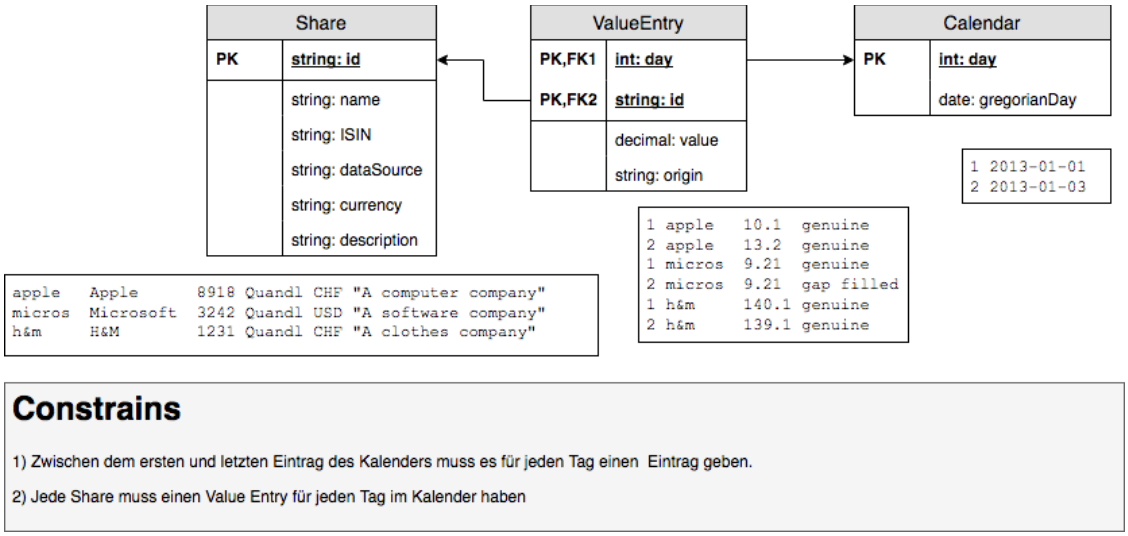


Abbildung 2 Datenbankmodel

Die wichtigsten Tabellen in unserem Datenbankmodel sind Share und ValueEntry. Vorher möchten wir allerdings einige Worte darüberschreiben, wieso wir die Tabelle Calendar benötigen.

Aktienbörsen sind nicht an jedem Tag im Jahr geöffnet. Deshalb liefern die verschiedenen Datenquellen auch keine Daten wenn die Börsen geschlossen sind wie beispielsweise am Wochenende. Deswegen führen wir eine Abstraktion für die den Kalender ein, indem wir die Tage einfach durchnummerieren. Wenn nun beispielsweise der Freitag der Tag Nr. 100 hat, bekommt der darauf folgende Montag die Nr. 101. Somit können wir danach Abfragen bearbeiten wie, 'Was ist der Tag vor Nr. 101' und geben somit das gregorianische Datum des Tag Nr. 100 zurück.

Um die Integrität der Abfragen zu garantieren, führen wir die folgende Bedingung ein: Jede Aktie muss an jedem Tag im Kalender ein Eintrag haben. Dies ist ein pragmatischer Ansatz, denn wenn

wir Lücken zulassen würden, würden Abfragen wie "Change from x to y" keine klaren Antworten haben, wenn beispielsweise an x kein Wert verfügbar ist.

Da allerdings nicht jeder Aktientitel an jedem Tag einen Wert hat (Ausländische Börsen haben andere Feiertage) führen wir eine Gap-Filling-Policy ein. Diese ist so, dass wir immer der letzte verfügbare Wert nehmen. Damit klar ist, wie die Werte aufgefüllt wurden, benötigen wir das Feld origin in ValueEntry. Dort soll gespeichert werden ob ein Feld ein 'echter' Wert ist (genuine) oder gap-filled. Wenn eine Aktie nicht mehr verfügbar ist tragen wir den Wert 0 ein.

Da das Gap-Filling von MariaDB kaum unterstützt wird und deshalb sehr komplex und ineffizient ist, begnügen wir uns damit diese 0 Einträge normal in der Datenbank zu speichern.

1.3 Datenquelle

Wir beziehen die Daten von der Firma Quandl (www.quandl.com). Genauer arbeiten wir mit dem Datensatz 'Swiss Exchange' (https://www.quandl.com/data/SIX-Swiss-Exchange).

Dazu verfügen wir über ein kostenloses Benutzerkonto welches uns täglich 50'000 Abfragen erlaubt. Die Daten beziehen wir über die bereitgestellte JSON-REST API. Da nicht alle 10'000 Aktientitel dieses Datensatzes interessant sind, wird eine Liste der Titel welche in StockSearch importiert werden geführt. Diese Liste liegt im liegt im csv Format vor.

1.3.1 Initialisierung der Datenbank

Folgender Pseudocode zeigt, welche Schritte nötig sind um die Datenbank in einen Zustand zu bringen welcher alle Constraints erfüllt.

Die Schwierigkeit dabei ist, dass man zuerst den Kalender initialisieren muss, bevor die Daten in der Datenbank gespeichert werden können. Um den Kalender zu initialisieren müssen allerdings zuerst alle Daten vorliegen, denn damit kann man den Handelskalender rekonstruieren.

```
titels <= readCsv("titel.csv")
temp  <= createTemporaryTable()
db <= database()

# Download der Daten
for title in titels:
   data <= download(title)
   db.shares.insert(metadata(data))
   temp.insert(data)

# Erstellen des Kalenders
dates <= AllDistictDates(temp)
dates.sort()
counter <= 0
for date in dates:
   db.calendar.insert(counter, date)
   counter <= counter + 1


# Einfügen der Daten in die Datenbank inkl GapFilling
for entry in temp:
     firstDate = db.shares.lookup(entry.id).firstDate
     lastDate  = db.shares.lookup(entry.id).lastDate
     value <= 0
    for day in range(0, counter):
         if date < firstDate or date > lastDate:
            value <= 0
         else if not hasValue(entry)
         else
            value <= entry.value
         db.date.insert(day, entry.id, value)

temp.delete()
```

1.4 REST-Api

Die Engine stellt eine Rest-API zu Verfügung um Abfragen zu senden.

Allgemeines:

- Es gibt keine Autorisierung der einkommenden Anfragen
- Resultate werden jeweils im Json-Format zurückgegeben

Die Api besteht aus nur einem Endpoint, und zwar um Queries durchzuführen.

1.4.1 Query-Endpoint

URL: /query/{ssql}

Methode: GET

Mögliche Antwort Codes:

- 400 Bad Request (Ungültiges SSQL Statement)
- 200 OK
- 500 Internal Error

Im Fall eines Fehlers wird eine Error-Response zurückgegeben. Wenn die Abfrage erfolgreich verarbeitet werden konnte, wird je nach Abfrage entweder eine Table-Response oder Detail-Response zurückgegeben.

1.4.1.1 Table-Response Sample


```
{
  "success": true,
  "type": "table",
  "data": [
    {
      "key": "micro",
      "name": "Microsoft",
      "isin": "300",
      "dataSource": "test",
      "currency": "TST",
      "description": "Bleibt konstant",
      "firstEntry": 1,
      "lastEntry": 4,
      "Value on 2018-01-01": 20,
      "Decimal 15.0": 15
    },
    {
      "key": "usb",
      "name": "UBS AG",
      "isin": "400",
      "dataSource": "test",
      "currency": "TST",
      "description": "Ist teuer",
      "firstEntry": 2,
      "lastEntry": 4,
      "Value on 2018-01-01": 130,
      "Decimal 15.0": 15
    }
  ],
  "sql": [
    "SELECT @var1 := `day` FROM `Calendar` WHERE `date` <= '2018-04-03' ORD
ER BY `date` DESC LIMIT 1",
    "Select * from `Share`, (SELECT `id` AS `key`, `value` AS `Value on @va
r1` FROM `ValueEntry` WHERE `day` = @var1) as f0, (SELECT `key` AS `key`, 1
5.0 AS `Decimal 15.0` FROM `Share`) as f1 where (`Share`.`key` = f0.`key`AN
D `Share`.`key` = f1.`key`) AND (f0.`Value on @var1` > f1.`Decimal 15.0`)"
  ],
  "ssql": "all shares where value > 15"
}
```

1.4.1.2 Detail-Response Sample


```
{
  "success": true,
  "type": "detail",
  "key": "micro",
  "name": "Microsoft",
  "isin": "300",
  "dataSource": "test",
  "currency": "TST",
  "description": "Bleibt konstant",
  "firstEntry": "2018-01-01",
  "lastEntry": "2018-01-06",
  "data": [
    {
      "value": 20,
      "date": "2018-01-01"
    },
    {
      "value": 20,
      "date": "2018-01-02"
    },
    {
      "value": 20,
      "date": "2018-01-05"
    },
    {
      "value": 20,
      "date": "2018-01-06"
    }
  ],
  "sql": [
    "SELECT v.`value`, c.`date` FROM `ValueEntry` AS v, `Calendar` AS c
WHERE v.`day` = c.`day` AND v.`id`='micro' ORDER BY v.`day` ",
    "SELECT s.`key`, s.`name`, s.`isin`, s.`dataSource`, s.`currency`, s.`d
escription`, f.`date` AS firstEntry, l.`date` AS lastEntry FROM `Share` as
s, `Calendar` as f, `Calendar` as l WHERE s.`firstEntry`=f.`day` AND s.`las
tEntry`=l.`day` AND s.`key`='micro'"
  ],
  "ssql": "get 'micro'"
}
```


1.4.1.3 Error-Response Sample


```
{
  "success": false,
  "type": "error",
  "ssql": "all",
  "error": "[line 1, column 1] Expected 'all shares where ' or 'all shares'
or 'get': Found 'all'"
}
```

1.5 Client-Applikation

1.5.1 Suchmodus

1.5.1.1 Fortgeschrittener Modus

Der fortgeschrittene Modus besteht aus einem einzelnen Inputfeld und einen Knopf welche die Suche startet.

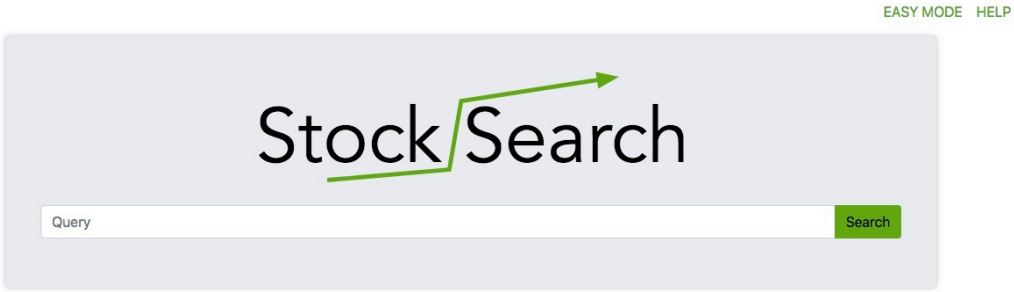


Abbildung 3 Screenshot fortgeschrittener Modus

1.5.1.2 Einfacher Modus

Der Einfache Modus sollte visuell gleich aussehen wie der fortgeschrittene Modus, aber das Statement sollte durch ein Userinterface erstellt werden. Somit lernt der Benutzer beim Verwenden des einfachen Modus sogleich die Sprache für den fortgeschrittenen Modus.

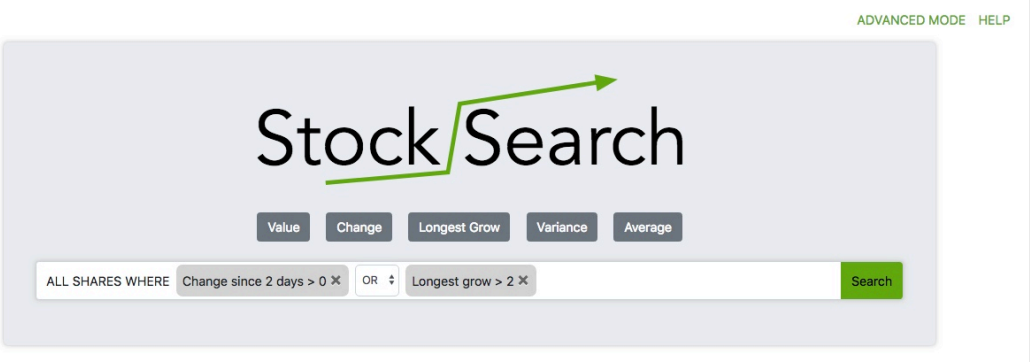


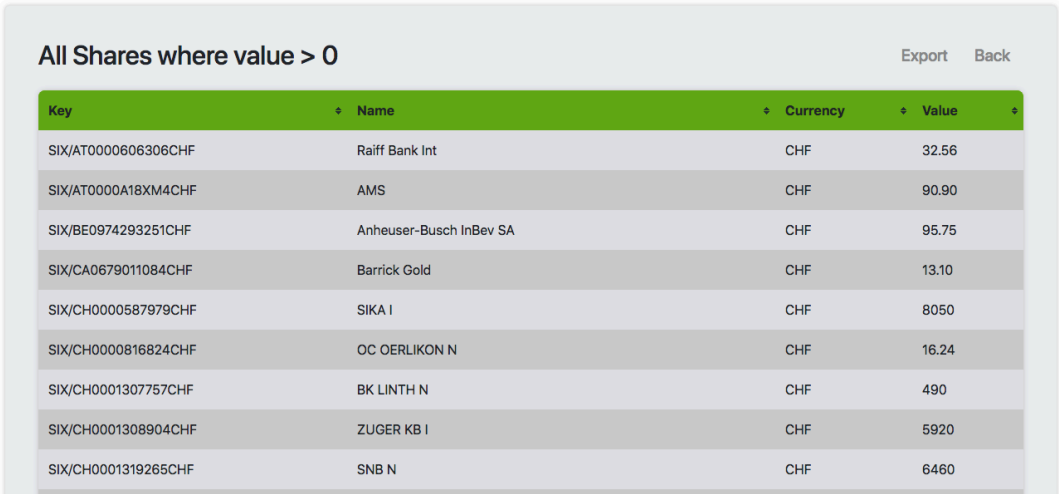
Abbildung 4 Screenshot einfacher Modus

1.5.2 Resultate

Das Resultat wird unter dem Suchmodus angezeigt und besteht aus einer Tabelle, Detail oder als Fehler angezeigt.

1.5.2.1 Tabellen Resultat

Das Resultat der Statements wird in einer Tabellenform angezeigt. Diese Tabelle ist nach Spalte sortierbar und kann per Druck auf den Export Knopfs als Datei gespeichert werden.



Key	Name	Currency	Value
SIX/AT0000606306CHF	Raiff Bank Int	CHF	32.56
SIX/AT0000A18XM4CHF	AMS	CHF	90.90
SIX/BE0974293251CHF	Anheuser-Busch InBev SA	CHF	95.75
SIX/CA0679011084CHF	Barrick Gold	CHF	13.10
SIX/CH0000587979CHF	SIKA I	CHF	8050
SIX/CH0000816824CHF	OC OERLIKON N	CHF	16.24
SIX/CH0001307757CHF	BK LINTH N	CHF	490
SIX/CH0001308904CHF	ZUGER KB I	CHF	5920
SIX/CH0001319265CHF	SNB N	CHF	6460

Abbildung 5 Screenshot Tabellenresultat

1.5.2.2 Detail Resultat

Damit der Benutzer einen Aktientitel genauer untersuchen kann, erhält er eine Übersicht beim Klicken auf den Aktientitel in der Tabelle.

Abbildung 6 Screenshot Detail Resultat

1.5.2.3 Fehler Resultat

Falls der Server nicht antwortend oder die Suchanfrage einen Syntax Fehler enthält oder die Suche keine Resultate enthält, bekommt der Benutzer eine Fehlermeldung.

Abbildung 7 Screenshot Fehler Resultat

1.5.3 HTML Layout

Die Applikation ist besitzt ein einziges HTML-Dokument, welches in 2 teile unterteilbar ist. Die Sucheingabe- und der Resultat-Teil. Die Elemente der Suche sind alle in dem HTML gespeichert, die Resultate jedoch werden dynamisch mit JavaScript eingefügt.

1.5.4 JavaScript Dateien Übersicht

Die Applikation ist besitzt ein einziges HTML Dokument, welches in 2 teile unterteilbar ist. Die Sucheingabe und der Resultatteil. Die Elemente der Suche sind alle in dem HTML gespeichert, die Resultate jedoch werden dynamisch mit JavaScript eingefügt.

1.5.5 JavaScript Dateien Übersicht

Es hat 3 verschiedene JavaScript Dokumente welche für die Logik und anzeige zuständig sind:

- main.js
  Diese Datei enthält die Haupt Logik der Webseite.
- easyMode.js
  Diese Datei enthält die Logik und die Anzeigeeigenschaften für den Easy Mode.
- results.js
  Diese Datei ist zuständig für die Anzeige der Resultate der verschiedenen Suchen.

1.5.5.1 main.js

Die 3 wichtigsten Methoden dieser Datei sind:

- getResults
  Diese Methode nimmt ein SSQL Suchabfrage als Argument und sendet diese mit Ajax an die API. Die Antwort wird nach Typ sortiert und an die jeweilige Funktion im results.js weitergeleitet.
- changeMode
  Diese Funktion wechselt die Ansicht der Suchabfrage. Dies geschieht indem bei den jeweiligen Elementen das Style Display Attribut angepasst wird.
- goBack

Damit der Benutzer die vorhergegangene Abfrage wiederholen kann, werden die SSQL abfragen im Session-Storage gespeichert. Beim betätigen des Back Link, wird die letzte Abfrage aus dem Speicher gelesen und erneut ausgeführt.

1.5.5.2 easyMode.js

1.5.5.2.1 Filter

Jeder Filter, der ein Benutzer im Easy Mode hinzufügen kann, besteht aus 2 Funktionen.

Wenn der jeweilige Filterknopf in der Easy Mode Ansicht angewählt wird, wird die Methode add[FILTERNAME]Filter aufgerufen.

Diese Funktion erzeugt ein Modal welches den Benutzer für die Argumente des jeweiligen Filters fragt. Beim betätigen des Save Knopf wird die Methode process[FILTERNAME]Filter aufgerufen.

Diese Funktion erzeugt aus den eingaben im jeweiligen Modal den HTML Code für diesen Filter und übergibt ihn der Funktion renderFilter welche den erzeugten code in den HTML DOM einfügt.

1.5.5.2.2 Verarbeitung der Filter

Die vom Benutzer im Easy Mode hinzugefügte Filter, werden nach der Initialisierung im Dom gespeichert. Falls ein Filter hinzugefügt wird, muss geprüft werde, ob es der erste Filter ist.

Falls es der erste Filter ist wird der HTML Code des Filters zusammen mit einem p-Element in einem div in dem SSQL an der richtigen Stelle eingefügt.

```
<div>
    <p> WHERE </p>
    <div class='filter'>
    ...
    </div>
<div>
```

Falls es schon mehrere Filter gibt wird er mit einem Dropdown-Menu in einem div gespeichert. Dieses div wird dann an der richtigen Stelle eingefügt.

```
<div>
    <div id='concat'>
        <select>
            <option>OR</option>
            <option>OR</option>
        </select>
    </div>
    <div class='filter'>
    ...
    </div>
<div>
```

Beim Klicken auf den Search Knopf im Easy Mode wird die Funktion getEasyResults aus dem HTML-DOM die SSQL suche extrahieren und an der Methode getResults, welche sich im main.js Dokument befindet, weitergeben.

Beim Klicken auf das **x** eines Filters im Easy Mode, wird die Funktion deleteFilter aufgerufen. Als Argument wird das äussere div mitgegeben, welche entweder das `div#concat` oder das `p`

Element enthält. Falls es mehr aus 2 filtern in dem SSQL-Statement hat und der erste Filter mit der Where-Klausel gelöscht wird, muss das `div#concat` in das `p` Element geändert werden.

Damit die Auswahl der Select-Optionen bei einer Änderung gespeichert bleibt, wird beim Select-onChange die Methode persistConcatState ausgeführt, welche die Reihenfolge der option Elemente dieses selectes vertauscht.

1.5.5.3 results.js

Es gibt 3 verschiedene Antwortarten:

- table
  Bei einer SSQL suche wird eine Tabelle als key value array mit den Daten der Verschiedene Aktien und deren Attribute zurückgegeben. Diese werden als Tabelle im HTML gespeichert und angezeigt.

- details
  Falls der Benutzer die Daten einer Aktie analysieren möchte, kann er in der Tabelle die jeweilige Zeile anwählen. Die Aktienkurse werden danach mit Hilfe von Chart.js angezeigt.

- error
  Falls eine suche ein Fehler enthält, wird von der API ein Fehler zurückgegeben, welche danach im HTML angezeigt wird.

1.6 SSQL

SSQL ist die Sprache in der Abfragen auf StockSearch gestellt werden.

Im Folgenden ist die gesamte Grammatik der SSQL-Statements in der erweiterten Backus-Naur-Form definiert. Für eine bessere Erklärung einzelnen Statements sehen Sie sich bitte das SSQL-Tutorial an.

1.6.1 Statements

```
Statement = ( AllStatement | AllWhereStatement | Detail );

AllStatement = 'all shares';

AllWhereStatement = 'all shares where ', Condition;

Detail = 'get', String;
```

1.6.2 Conditions

```
Condition = ( CombinedCondition | FunctionCompareCondition |
            ParenthesedCondition | BooleanCondition );

CombinedCondition = ( FunctionCompareCondition | ParenthesedCondition |
                    BooleanCondition ), BoolOperator, Condition;

FunctionCompareCondition = Function, Comperator, Function;

BooleanCondition = ( 'true' | 'false' );

ParenthesedCondition = '(', Condition, ')';
```

1.6.3 Funktionen

```
Function = ( Decimal | Value | Change | Variance | Average | LongestGrow );

Average = 'average', [Timespan];

Decimal = ['-'], ? WORD('0-9') ?, ['.', ? WORD('0-9') ?], ['%'];

LongestGrow = 'longest grow', [Timespan];

Value = 'value', ['on', Date];

Variance = 'variance', [Timespan];

Change = ['absolute'], 'change', Timespan;
```

1.6.4 Datum und Zeit

```
Date = ? WORD('0-9') ?, '-', ? WORD('0-9') ?, '-', ? WORD('0-9') ?;

Timespan = ( SinceDays | SinceDate | FromTo );

FromTo = 'from', Date, 'to', Date;

SinceDate = 'since', Date;

SinceDays = 'since', PosInteger, 'days';
```

1.6.5 Verschidenes

```
BoolOperator = ( 'and' | 'or' );

Comperator = ( '!=' | '>' | '>=' | '=' | '<=' | '<' );

PosInteger = ? WORD('0-9') ?;

String = ( "'", AnythingButQuotation, "'" | '`', AnythingButQuotation, '`'
|
         '"', AnythingButQuotation, '"' );
```

1.7 Funktionsweise Engine

1.7.1 Prinzip

Ein geparstes SSQL-Statement wird im Speicher als Baum von Objekten repräsentiert, wobei jeweils das Elternelement eine Objektreferenz auf das Kinderelement besitzt. Folgend dient zum besseren Verständnis ein mögliches Objektdiagram welches das Statement:

```
ALL shares WHERE (value on 2018-03-05 > 100) OR change from 2016-01-01 to
2016-06-31 >= 0.1
```

repräsentiert.

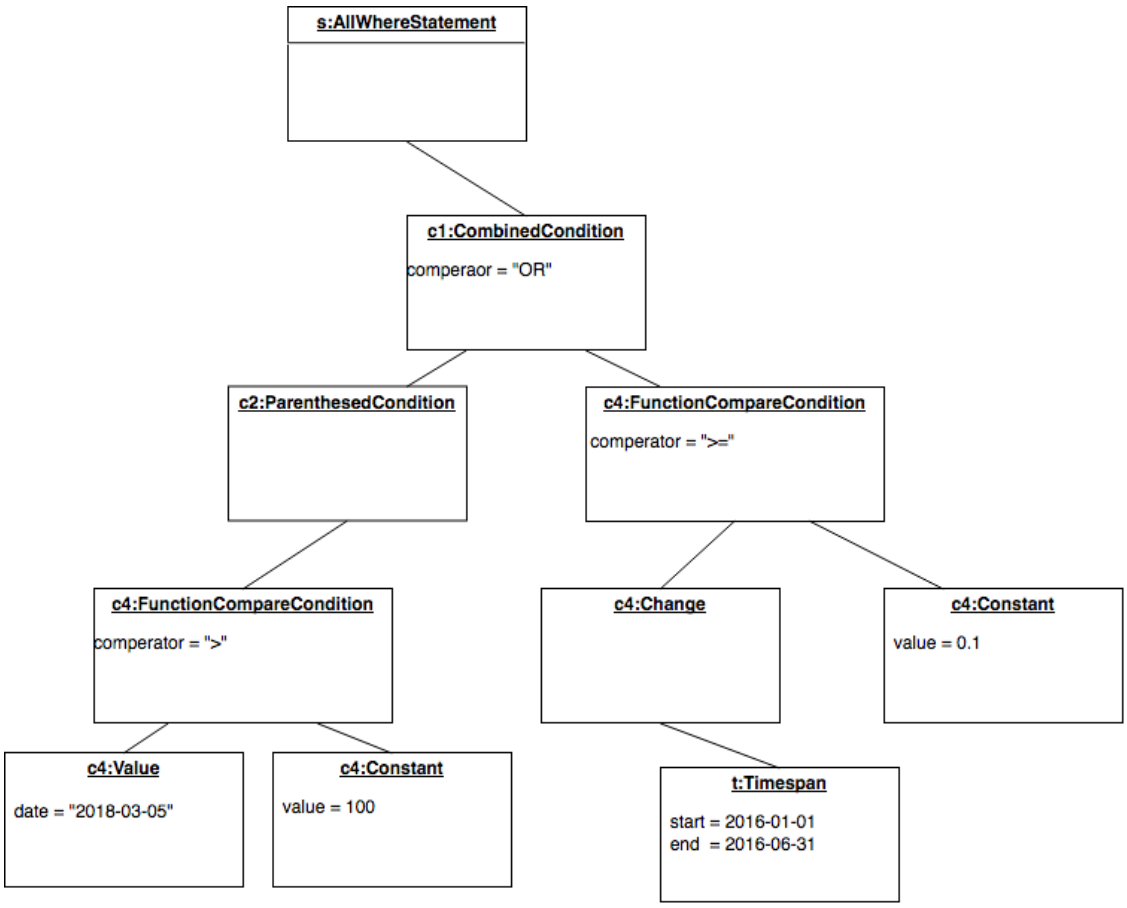


Abbildung 8 Objektdiagram-Engine

Für die Generierung eines SQL-Statement welches von der Datenbank verarbeitet werden kann wird nun rekursiv die Funktion get_sql() aufgerufen und somit das Statement zusammengesetzt. Dabei gelten folgende Bedingungen:

1.7.1.1 Funktionen

Eine Funktion kann als mathematische Funktion angesehen werden, welche jedem Aktientitel (Definitionsmenge) eine reelle Zahl (Bildmenge) zuordnet. Beispielsweise mapt die Funktion Value mit dem Parameter date=2018-01-01 jede Aktie auf ihren Wert zu diesem Datum. Falls dann kein Wert vorhanden sein solle, muss die Funktion entweder null oder 0 zurückgeben. Konstanten werden einfachheitshalber auch als Funktionen dargestellt, diese geben somit einfach die gleiche Zahl für alle Aktientitel zurück.

Funktionen werden später alle Innere-Selects verwendet. Deswegen muss die Funktion get_sql() für alle Funktionsobjekte ein SQL-Statement zurückgeben welches für jede Aktie ein Tuppel aus id und wert zurückgibt. Später wird danach einen Inner-Join auf alle Funktionen über die id berechnet, somit können dann die Funktionswerte pro Aktie verglichen werden.

1.7.1.2 Conditions

Conditions werden mit AND und OR verknüpft. Folglich müssen Conditions einen boolesche Wert für alle Aktien zurückgeben können. In der Engine ist dies folgender massen umgesetzt:

Conditions müssen ein Template und eine Liste von Funktionsobjekten zurückgeben. Die Liste der Funktionen wird erstellt in dem rekursiv auf alle Kinderobjekte dieselbe Operation angewendet wird. Das Template ist ein String, in welchen danach nur noch die Namen der Funktionen eingesetzt werden können.

Beispielsweise gibt die Condition `value on 2018-01-01 > 10` Die beiden Funktionen

```
[(value on 2018-01-01), (constant 10)]
```

sowie das Template

```
{} > {}
```

zurück. Damit kann das Statement Objekt alle benötigten Funktionen joinen, danach in das Template die Namen der in SQL inneren Statements einsetzten, und das Template als Filter im SQL-Statement benützten.

1.7.1.2.1 Beispiel

Das Statement `all shares where value on 2018-01-01 > 10` besteht aus einer Condition und zwei Funktionen.

Die erste Funktion ist `value on 2018-01-01` und liefert das folgende innere SQL-Statement:

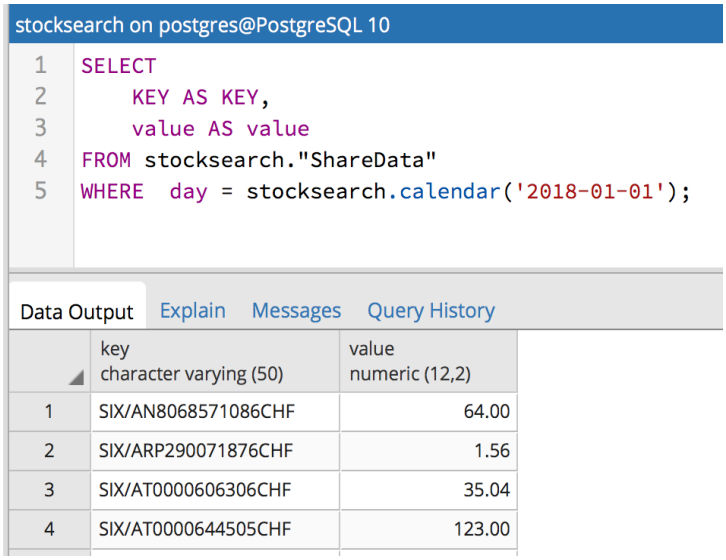


Abbildung 9 SQL Value Funktion

Die einfache Konstante 10 liefert das SQL-Statement:

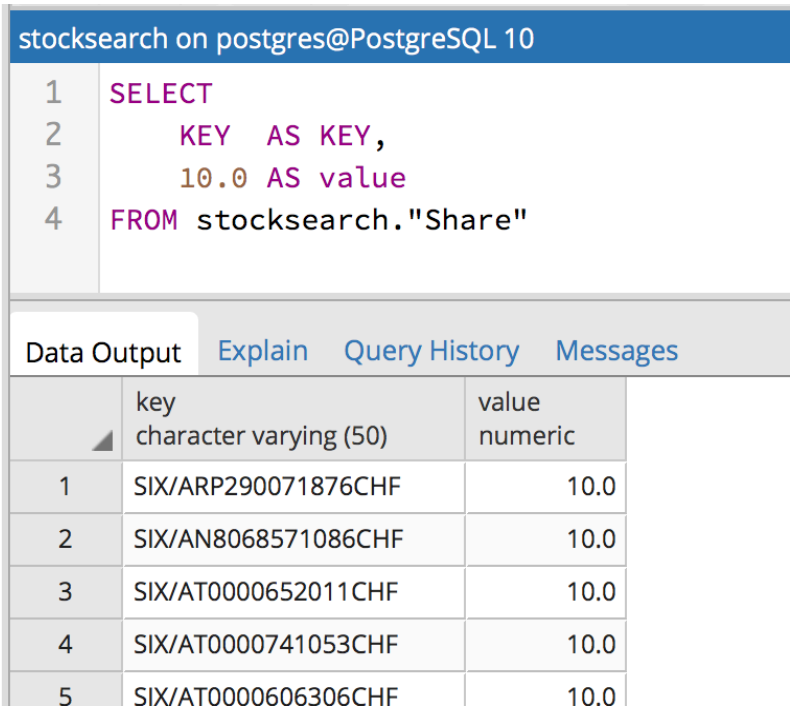


Abbildung 10 SQL Konstante Funktion

Danach wird folgendermassen gejoint:

stocksearch on postgres@PostgreSQL 10

```
1  SELECT share.KEY     AS KEY,
2         f1.value       AS "value on 2018-01-01",
3         f2.value       AS "10"
4  FROM   stocksearch."Share" AS share
5  INNER JOIN (
6                SELECT KEY   AS KEY,
7                       value AS value
8                FROM   stocksearch."ShareData"
9                WHERE  day = stocksearch.calendar('2018-01-01')) AS f1
10 ON     share.KEY=f1.KEY
11 INNER JOIN (
12               SELECT KEY  AS KEY,
13                      10.0 AS value
14               FROM   stocksearch."Share") AS f2
15 ON     share.KEY=f2.key;
```

Data Output | Explain | Query History | Messages

	key character varying (50)	value on 2018-01-01 numeric (12,2)	10 numeric
1	SIX/ARP290071876CHF	1.56	10.0
2	SIX/AN8068571086CHF	64.00	10.0
3	SIX/AT0000652011CHF	46.82	10.0
4	SIX/AT0000741053CHF	20.70	10.0

Abbildung 11 SQL Joins

Schlussendlich wird nur noch das Template als Filter angewendet und noch die zusätzlichen Metadaten im Select erfasst und wir erhalten das folgende SQL-Statement:

```
SELECT share.KEY     AS KEY,
       share.NAME     AS NAME,
       share.currency AS currency,
       f1.value       AS value on 2018-01-01
FROM   Stocksearch."Share" AS share
INNER JOIN (
               SELECT KEY   AS KEY,
                      value AS value
               FROM   stocksearch."ShareData"
               WHERE  day = stocksearch.calendar('2018-01-01')) AS f1
ON     share.KEY=f1.KEY
INNER JOIN (
               SELECT KEY  AS KEY,
                      10.0 AS value
               FROM   stocksearch."Share") AS f2
ON     share.KEY=f2.key
WHERE f1 > f2
```

Man beachte das die Alle Funktionen ausser die Konstanten im Select-Bereich zurückgegeben werden. Umsetzung Mit diesem Prinzip können die Funktionen in SQL beliebig ausgetauscht und erweitert werden. Jedes Funktionsobjekt muss einfach die Bedingung erfüllen das es ein SQL-Statement mit welches als Inner-Statement verwendet werden kann und die Felder "key" und "value" enthält.

Auf die genaue Beschreibung der Umsetzung der anderen Funktionen verzichten wir an dieser stellen, man siehe dazu die Dokumentation im Quellcode der Engine an. Allerdings möchten wir noch auf die Funktion Longest-Grow zu sprechen kommen, da diese auf eine etwas andere Art umgesetzt ist.

1.7.1.3 Longest-Grow

Die Funktion Longest-Grow gibt die Anzahl Tage der längsten Periode jedes Aktientitels zurück, in welcher der Wert jeden Tag gestiegen ist. Da dies mit Standart-SQL sehr schwer umzusetzen ist haben wir dazu eine eigene Aggregatsfunktion in Postgres erstellt.

Die Funktion get_sql von Longest-Grow liefert also folgendes inneres SQL-Statement zurück:

```
/* Longest strike */
SELECT
    key as "key",
    CAST (longest_strike("value" ORDER BY "day") AS numeric ) as "value"
FROM
    stocksearch."ShareData"
WHERE
    day >= calendar_start() AND day <= calendar_end()
GROUP BY key
```

Dazu haben wir die Funktion longest_strike() in PSQL folgendermassen implementiert:

Wir haben eine Left-Folding Reduce-Funktion erstellt.

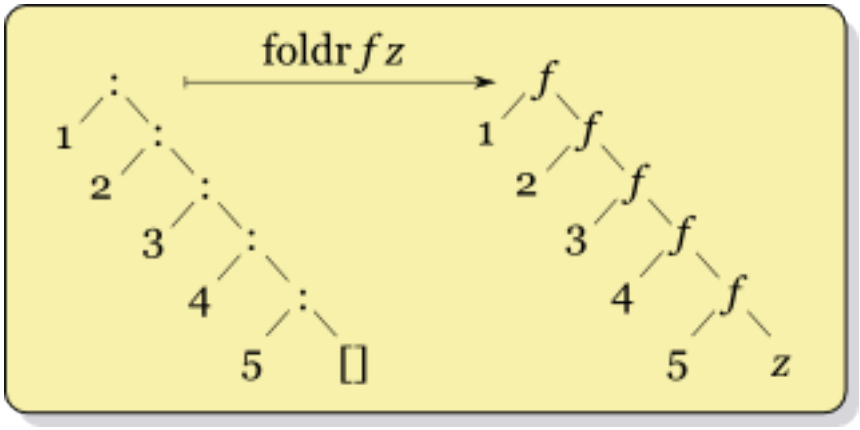


Abbildung 12 Left-Folding

Source Wikipedia: https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Dazu benötigen wird einen State welcher jeweils aktualisiert wird.

```
CREATE TYPE longest_strike_state as (
    currentStrike       integer,
    longestStrike       integer,
    lastValue           decimal
);
```

Die Reduce-Funktion wird nun für alle Dateneinträge jeder Aktie der Reihe nach aufgerufen und aktualisiert jeweils den aktuellen State.

```
/* This function updates the state for each value in the reduce*/
CREATE OR REPLACE FUNCTION longest_strike_reduce(s longest_strike_state, v
decimal) RETURNS longest_strike_state AS $$
    BEGIN
        IF s.lastValue != -1 AND v > s.lastValue THEN
            s.currentStrike = s.currentStrike + 1;
        ELSE
            /* The strike is finished */
            s.currentStrike = 0;
        END IF;

        IF s.currentStrike > s.longestStrike THEN
                s.longestStrike = s.currentStrike;
        END IF;
        s.lastValue = v;
        return s;
    END;
$$ LANGUAGE plpgsql;
```

Nun müssen wir nur noch die Aggregatsfunktion in Postgres erstellen in dem wir einen initialen State angeben, zusammen mit der Reduce-Funktion und einer Funktion welchen den Wert aus dem finalen State selektiert.

```
/* This function pickes the final value out of the state of the reduce func
tion*/
CREATE OR REPLACE FUNCTION longest_strike_finalizer(s longest_strike_state)
RETURNS integer AS $$
    BEGIN
        return s.longestStrike;
    END;
$$ LANGUAGE plpgsql;

/* And this composes everything to one function*/
CREATE AGGREGATE longest_strike(decimal)
(
    sfunc = longest_strike_reduce,
    finalfunc = longest_strike_finalizer,
    stype = longest_strike_state,
    initcond = '(0,0,-1)'
);
```

1.7.2 Klassendiagram

Folgendes Klassendiagram soll zur raschen Übersicht der verschiedenen Klassen in engine.py dienen.

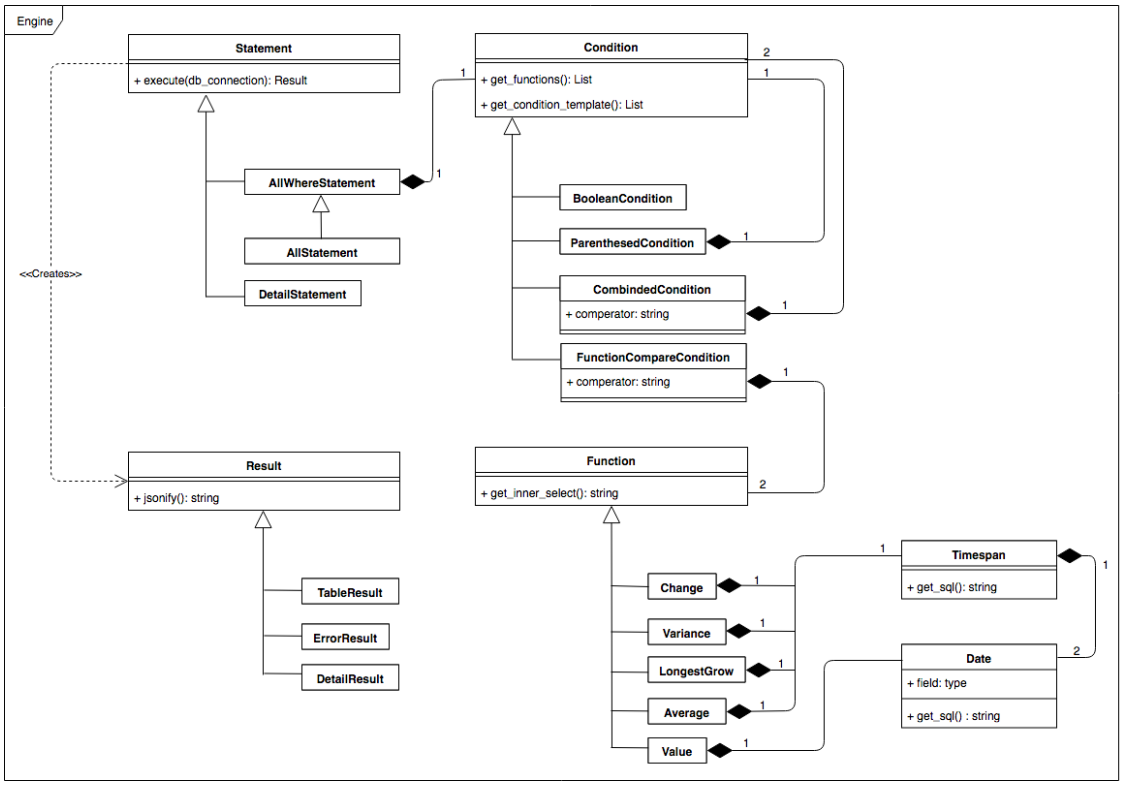


Abbildung 13 Klassendiagram Engine

1.8 Funktionsweise Parser

Für das Parsing der einkommenden SSQL-Statements verwenden wir die Bibliothek Modgrammar, welche ein sehr einfach verständliches Interface bietet.

Modgrammar funktioniert so, dass man die Klasse Grammar erweitert. Im folgenden Beispiel, kopiert aus der offiziellen Dokumentation, ist dies ersichtlich.

```
from modgrammar import *

grammar_whitespace_mode = 'optional'

class MyGrammar (Grammar):
    grammar = (LITERAL("Hello,"), LITERAL("world!"))

myparser = MyGrammar.parser()
result = myparser.parse_text("Hello, world!")
```

Um nun ein Syntaxbaum der Engine zu erstellen benutzen wir das Konzept der Mehrfachvererbung welches Python unterstützt. Dazu erstellten wir eine Unterklasse für jede benötigte Klasse auf dem Engine-Package welche gleichzeitig auch noch von der Klasse Grammar vom Modgrammar-Package erbt. Deutlich wird dies am folgenden Auszug aus dem Source-Code des Parser-Package.

```
class FunctionCompareCondition(Grammar, eg.FunctionCompareCondition)
:

    grammar = (REF("Function"), REF("Comperator"), REF("Function"))

    def grammar_elem_init(self, _):
        self.leftFunction = self[0]
        self.comparator = self[1].string
        self.rightFunction = self[2]
```

Mit der grammar_elem_init() wird nach der Istanzierung der Klasse durch Modgrammar die nötigen Attributgesammtee der Engine-Superklasse gesetzt.

Mit diesem Prinzip wird der gesamte abstrakte Syntaxbaum initialisiert und kann danach von der Engine verarbeitet werden.

1.8.1 Klassendiagram

Folgendes Klassendiagram soll den Zusammenhang zwischen den verschiedenen Modulen veranschaulichen.

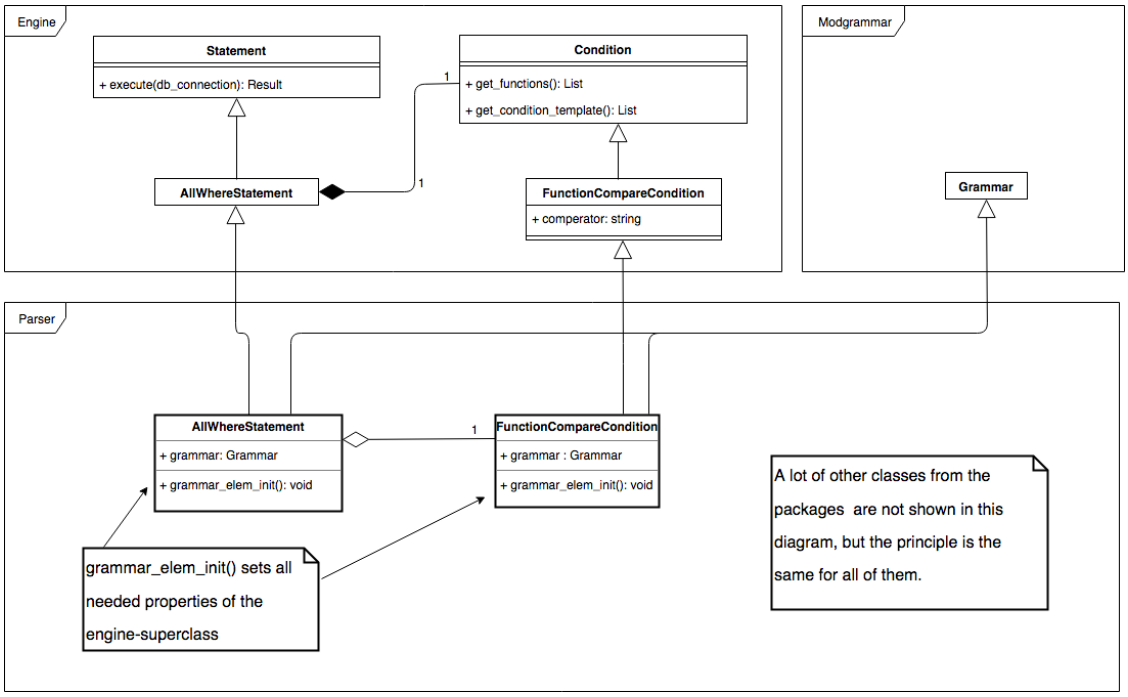


Abbildung 14 Klassendiagram Parser

1.9 Funktionsweise Api

Da das API-Modul eigentlich das REST-Interface zur Verfügung stellt gehen wir nicht darauf weiter ein.

1.10 Tests

Für das Testing haben wir Pytest und Doctest verwendet. Den Fokus haben wir auf diese Teile der Applikation gelegt, welche besonders langlebig sind wie beispielweise die Engine und der Parser. Unittests und Akzeptanztests haben Dateinamen nach dem Schema test_{name}.py

Doctests sind jeweils in den Kommentaren in den verschiedenen Python-Modulen zu finden.

1.11 Verwendete Frameworks / Softwarebibliotheken

Es wurde ausschliesslich Open-Source Software verwendet. Die verwendeten Bibliotheken, Sprachen, Programme und Frameworks sind:

1.11.1.1 Engine:

1.11.1.1.1 Python 3

Programmiersprache

- https://www.python.org/
- Python Software Foundation License

1.11.1.1.2 Flask 0.12.2

Web Microframwork für Python

- http://flask.pocoo.org/
- BSD License

1.11.1.1.3 Modgrammar 0.10

Parsing library

- https://pythonhosted.org/modgrammar/
- New BSD License

1.11.1.1.4 PostgreSQL 10

SQL Datenbanksystem

- https://www.postgresql.org/
- PostgreSQL License

1.11.1.1.5 Pytest 3.4.2

Testframework für Python

- https://www.pytest.org
- MIT licence

1.11.1.1.6 Pg8000 1.11.0

Postgres Datenbank Connector für Python 3

- https://pypi.org/project/pg8000/
- BSD licence

1.11.1.1.7 Pytest-coverage 4.5.1

Code coverage measurement for Python

- https://bitbucket.org/ned/coveragepy
- Apache 2.0 licence

1.11.2 Client

1.11.2.1.1 Bootstrap 4

Front-end component library

- https://getbootstrap.com/
- MIT licence and copyright 2018 Twitter

1.11.2.1.2 JQuery

JavaScript library to simplify HTML modifications

- http://jquery.com/
- MIT licence

1.11.2.2 Verschiedenes

1.11.2.2.1 MkDocs 0.17.3

Static site generator

- http://www.mkdocs.org/
- BSD licence

1.11.2.1.3 Tablesorter

Plugin to create sortable tables

- http://tablesorter.com
- MIT and GLP licence

1.11.2.1.4 Jalc

Plugin for client side caching

- https://github.com/SaneMethod/jquery-ajax-localstorage-cache
- Apache License

2 Setup

Folgende Anleitung zeigt Ihnen Schritt für Schritt wie Sie StockSearch auf Ubuntu 16.04.4 x64 installieren.

2.1.1 Installation Postgres 10

Als erstes müssen wir Postgres 10 auf dem System installieren. Wenn für die Datenbank jedoch ein anderer Rechner benützt werden soll überspringen Sie diesen Schritt.

```
echo 'deb http://apt.postgresql.org/pub/repos/apt/ xenial-pgdg main' >> /et
c/apt/sources.list.d/pgdg.list
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo
apt-key add -
apt-get update
apt-get install postgresql-10
```

Nun müssen wir Postgres konfigurieren. Dafür wählen Sie ein sicheres Passwort für die Datenbank und erstellen zwei neue Datenbanken.

```
sudo su postgres
    psql
        ALTER USER postgres WITH PASSWORD '<DB-PASSWORD>';
        CREATE DATABASE stocksearch;
        CREATE DATABASE stocksearchmock;
        \q
    exit
```

Nun müssen wir Postgres noch anweisen, Benutzername und Passwort als Login zu akzeptieren. Dafür müssen Sie die MD5 Methode im pg_hba.conf file erlauben.

Dafür sollten Sie folgende Zeile in Ihrer pg_hba.conf Datei hinzufügen und Postgres neu starten:

```
# for users connected via local IPv4 or IPv6 connections, always require md
5
host    all   all        127.0.0.1/32          md5
```

```
vim /etc/postgresql/10/main/pg_hba.conf
    # Change file
/etc/init.d/postgresql restart
```

2.1.2 Installation Python 3.6

StockSearch benötigt Python 3.6, ebenfalls ist es zu empfehlen eine virtuelle Umgebung zu benützten. Deshalb installieren wir zusätzlich virtualenv.

```
add-apt-repository ppa:jonathonf/python-3.6
apt update

apt install python3.6
apt-get install virtualenv
```

2.1.3 Installation StockSearch

Am einfachsten clonen Sie das Git-Repository. Dazu benötigen Sie Zugang zum Gitlab-Projekt, allerdings können Sie das Repository auch kopieren.

```
# Download des Repos
git clone https://gitlab.ti.bfh.ch/freig3/project1-stocksanalytics.git
cd project1-stocksanalytics/

# Einrichtung der virtuellen Umgebung
virtualenv --python=/usr/bin/python3.6 env
source env/bin/activate

# Installation aller Dependencies mittels pip
pip install -r requirements.txt
```

2.1.4 Installation Datenbankschema

Um das Schema zu in der Datenbank zu installieren führen Sie die beiden SQL-Scrips aus.

```
psql stocksearch postgres -f sql/stocksearchdump.sql
psql stocksearchmock postgres -f sql/stocksearchmockdump.sql
```

2.1.5 Konfiguration

Die Zugänge zu den Datenbanken befinden sich in JSON-Konfigurationsdateien. Erstellen Sie diese nach folgendem Schema:

```
mkdir secured
echo '
{
    "key": "<QUANDLE-KEY>"
}
' >> secured/postgres.json

echo '{
    "db-host":    "127.0.0.1",
    "db-user":     "postgres",
    "db-password": "<DB-PASSWORD>",
    "db-database": "stocksearchmock"
}' >> secured/postgresmock.json

echo '{
    "db-host":    "127.0.0.1",
    "db-user":     "postgres",
    "db-password": "<DB-PASSWORD>",
    "db-database": "stocksear"
}' >> secured/postgres.json
```

2.1.6 Start

Nun können Sie mittels folgendem Befehl StockSearch ausführen. Wenn Sie dabei die `init` Option verwenden, werden zuerst alle Tests durchgeführt und die Dokumentation erstellt.

```
python run.py --init --host 0.0.0.0 --port 80
```

Um StockSearch im Hintergrund zu betreiben verwenden Sie:

```
python run.py --init --host 0.0.0.0 --port 80 > /dev/null 2>&1 &
```

3 Abbildungsverzeichnis