

Datengewinnung für die Erstellung eines Microsoft Power BI - Berichtes zum Thema: „Wie hat die Covid-19 Pandemie den Aktienmarkt beeinflusst?“

Projektarbeit

im Studiengang
Wirtschaftsinformatik

vorgelegt von

Timo Schwind
Matr.-Nr.: 759823

am 14. Juli 2021
an der Hochschule Esslingen

Erstprüfer: Prof. Dr. Dirk Hesse

Zweitprüfer: Dipl.-Wirt.-Inf. (FH) Matthias Müller

Inhalt

Abbildungsverzeichnis	3
Einleitung	4
1. API	5
1.1 Was ist eine API	5
1.2 Anforderungen an die API	6
1.3 Financial Modeling Prep API	7
2. Datenbankverbindung	8
2.1 Verbinden mit einer PostgreSQL-Datenbank mit Python	8
2.2 Verbinden mit einer PostgreSQL-Datenbank mit R	9
3. Programme zur Verwendung der API	10
3.1 Vorgehen zum Extrahieren der verfügbaren Aktienkürzel mit Python	10
3.2 Abrufen und Speichern der Fundamentaldaten der Unternehmen mit Python	14
3.3 Zugriff und Speicherung der historischen Preisdaten mit Python	17
3.4 Zugriff auf die Sektoren und Länderverteilung von ETF's mit R	22
3.5 Zugriff auf historische Geldwechselkurse mit Python	25
4. PostgreSQL-Datenbank	27
4.1 Aufbau der Datenbank	27
4.2 Datenbanklogik	28
4.2.1 Umrechnung der Währungen	28
5. Datenquellen im PowerQuery-Editor	30
5.1 Erstellen einer umfassenden Datumsdimension	30
5.2 JSON-Index-Daten von Yahoo-Finance	33
5.3 Zugriff auf die Datenbank mit Power Query	37
6. Power BI Report Vorstellung	38
6.1 Überblick	38
6.2 Startseite	39
6.3 Phasenbetrachtung der Markt- und Pandemieentwicklung	41
6.4. Detaillierte Aktien- und Kursverlaufsanalyse	43
7. Gewonnene Erkenntnisse	47
Literaturverzeichnis	48
Ehrenwörtliche Erklärung	49

Abbildungsverzeichnis

Abbildung 1: API-Aufruf	5
Abbildung 2: Aufbau des JSON-Objektes	10
Abbildung 3: API-Zugriff und Weiterverarbeitung der verfügbaren Aktienkürzel	10
Abbildung 4: Aufbau des Arrays nach Konvertierung	12
Abbildung 5: NumPy Array nach Bearbeitung	12
Abbildung 6: Aufbau der JSON-Datei nach dem Zusammenfügen	14
Abbildung 7: Fehleranzeige der JSON-Validierung	15
Abbildung 8: Bearbeitungsschritte der JSON-Datei.....	15
Abbildung 9: Aufbau des Dataframes ohne Formatierung	18
Abbildung 10: Vorgehen beim Auflösen des historical Arrays.....	19
Abbildung 11: resultierender Dataframe.....	19
Abbildung 12: Änderung der Daten durch etf\$symbol	22
Abbildung 13: Änderung des Inhaltes von req1.....	24
Abbildung 14: Unterschied von orient='index' und orient='columns'	25
Abbildung 15: ER-Diagramm Datenbank.....	27
Abbildung 16: Aufbau CurrencyRates	28
Abbildung 17: Vorgehen bei der Währungsumrechnung	29
Abbildung 18: Umwandlung Liste zu Tabelle	30
Abbildung 19: Ergebnis der erzeugten Spalte „Börsenfeiertag“	31
Abbildung 20: Parametereingabe	33
Abbildung 21: Übersicht der Records	34
Abbildung 22: M-Formel der benutzerdefinierten Spalte „Date“	34
Abbildung 23: Endergebnis der Datumstabelle	35
Abbildung 24: Zusammenführen der Abfrage	36
Abbildung 25: Import von „dailyprices“	37
Abbildung 26: Report Startseite	39
Abbildung 27: Zusätzliche Information auf der Startseite	39
Abbildung 28: Phasenbetrachtung des Reports.....	41
Abbildung 29: Kursverlauf des MSCI World	42
Abbildung 30: Erweiterte Informationen durch Mouseover	42
Abbildung 31: Einzelaktienansicht	43
Abbildung 32: Gewinn- und Verlustrechner.....	44
Abbildung 33: ETF Informationsseite.....	45

Einleitung

With a good perspective on history, we can have a better understanding of the past and present, and thus a clear vision of the future." (Slim Helu, Carlos, 2020)

Sobald etwas anfängt schief zu laufen, fangen viele Investoren an, in Panik zu verfallen. Die daraus resultierende Reaktion besteht meist darin, dass sie ihre Investitionen so schnell wie möglich liquidieren. Ein gutes Beispiel für ein solches Verhaltensmuster und dessen Folge für die Kapitalmärkte ist der sogenannte „Corona-Crash“. Dieser lieferte, ausgelöst durch das neuartige Coronavirus Sars-CoV-2, Ende des ersten Quartales 2020 einen historischen Werteverlust vieler Aktien. Historisch gesehen schafften es die Märkte immer wieder, sich von einem derart fallenden Kurs zu erholen. Die Finanzkrise 2008, der Dotcom-Crash 2000 oder auch die große Depression sind nur einige Beispiele hierfür. Genau dies kann man auch in den Monaten unmittelbar nach dem Crash im Jahr 2020 beobachten. Viele Aktien erholten sich sehr schnell wieder von ihren Verlusten und schrieben sogar neue Allzeithochs, eine Entwicklung, die rückblickend viele Chancen geboten hätte.

Diese Projektarbeit setzt ihren Fokus primär darauf, wie genau vorgegangen wird, um die richtigen Daten für die Visualisierung genau dieser historischen Entwicklungen zu besorgen. Darüber hinaus beschreibt sie, wie diese Daten transformiert und in einer PostgreSQL Datenbank abgespeichert werden, sodass auf diese anschließend mit dem Datenvisualisierungstool Power BI zugegriffen werden kann. Die aus der Datenbank geladenen Daten werden durch weitere, unterstützende Tabellen ergänzt, wobei darauf geachtet wird, eine möglichst große Vielfalt von verschiedenen Quellenzugriffsformen zu nutzen. Ebenfalls Thema dieser Arbeit ist die Vorgehensweise der Extraktion dieser Daten über Power BI.

Die Logik und die verschiedenen Techniken des Berichtes sind aus diesem zu entnehmen, jedoch wird der Report mit all seinen Funktionalitäten und Interaktionsmöglichkeiten am Ende des Berichtes erklärt.

Der Bericht soll es dem Anwender ermöglichen, den Aktiencrash 2020 bis ins kleinste Detail zu analysieren. Dies soll das Verständnis für eine solche Krise und deren Auswirkung auf die Kapitalmärkte erhöhen, um wie im obenstehenden Zitat angedeutet, beim nächsten größeren Zwischenfall Fehler zu vermeiden.

1. API

1.1 Was ist eine API

API ist die Abkürzung für „Application Programming Interface“, was im Deutschen als „Programmschnittstelle“ übersetzt wird. Über eine API können Entwickler auf die Funktionen einer Anwendung zugreifen. Sie ermöglicht es also zwei Programmen, miteinander in Echtzeit miteinander zu kommunizieren. APIs werden auch als Verträge mit Dokumenten angesehen, die eine Vereinbarung zwischen zwei Parteien repräsentieren: Wenn eine Partei eine in einer bestimmten Weise strukturierte Remote-Anfrage sendet, antwortet die zweite Partei entsprechend. Eine API ist keine Datenbank. Sie ist ein Zugriffspunkt auf eine App, welche auf eine Datenbank zugreifen kann. In Abbildung 1 sieht man den typischen Ablauf eines API-Aufrufes.

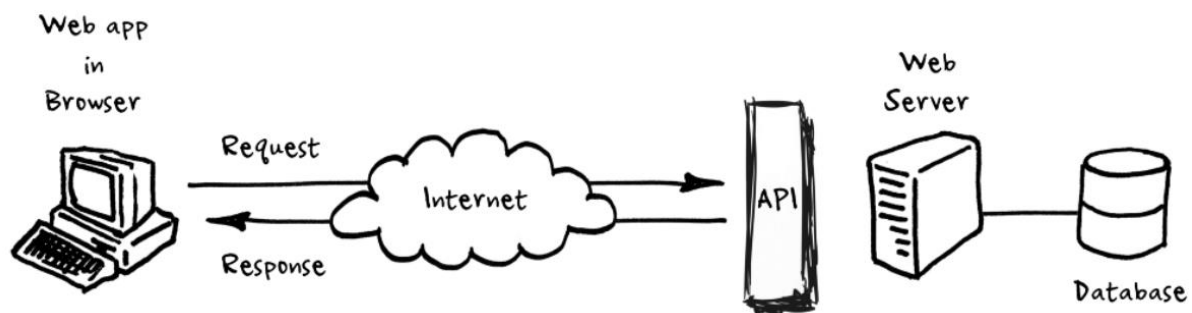


Abbildung 1: API-Aufruf

Oft werden dabei die Dateien im JSON-Format oder im CSV-Format übertragen. Beides sind Syntaxen um Daten zu speichern und zu verschicken. Im täglichen Gebrauch kommen APIs beispielsweise bei Webseiten zum Einsatz. Wer nach einem Urlaub sucht, kann entsprechende Suchmaschine beauftragen, alle Hotels und Flüge zu einem bestimmten Reiseziel und Datum zu finden. Sobald man auf „Suchen“ klickt, fängt die Webseite an, mit den APIs der einzelnen Anbieter zu kommunizieren, um die Preise abzufragen. Für API-Schnittstellen gibt es noch zahlreiche andere Beispiele aus dem täglichen Leben:

- Google Maps (z.B. für die Einbindung des Services auf der eigenen Webseite)
- PayPal (Integration der PayPal-Transaktionen in die eigene Kaufabwicklung)
- DHL (Integration der Sendungsverfolgung in eigene Systeme oder Shops)

1.2 Anforderungen an die API

Um später im Power BI Report mit Aktiendaten arbeiten zu können, muss zunächst klar sein, welche Daten benötigt werden, um vernünftige Informationen gewinnen zu können. Der Anspruch liegt darin, die Entwicklung der Wertpapiere innerhalb des Zeitraums der Corona-Pandemie zu untersuchen. Die wichtigsten Informationen für den Bericht sind folglich die historischen Aktienkurse seit Anfang des Jahres 2020. Die Kennzahl, aus der später der Verlauf der Wertpapiere über das Jahr verteilt ersichtlich werden soll, ist der tägliche, angepasste Schlusskurs. Dieser stellt den letzten Kurs eines Wertpapiers vor Börsenschluss nach Berücksichtigung etwaiger Kapitalmaßnahmen dar. Zusätzlich sind weitere Informationen wie der Tageshöchst- und Tagestiefpreis oder der Eröffnungspreis nützlich. Zur weiteren Darstellung der Unternehmen und auch des Unternehmenszweckes bzw. dessen Sektor benötigt die API auch fundamentale Daten. Um die Abbildung einzelner Indize zu ermöglichen, sollten auch Daten über Exchange Traded Funds (kurz ETF) vorhanden sein. Zusammenfassend kommt man also auf folgende Anforderungen an die Daten, welche die API bereitstellen sollte:

- Historische Preisdaten für Aktien
- Fundamentale Unternehmensdaten
- Historische Preisdaten für ETFs
- Fundamentale Daten über ETFs (Sektorenaufteilung, Landesaufteilung)

1.3 Financial Modeling Prep API

Die Financial Modeling Prep API bietet Echtzeit-Aktienkurse, Fundamentaldaten, wichtige Indexpreise, historische Aktiendaten, Forex-Echtzeitkurse und Daten zu Kryptowährungen. Die API hat 50 verschiedene Endpunkte, die die Daten im JSON-Format bereitstellt. Einige der Endpunkte stellen auch Daten im CSV-Format bereit. Zahlreiche Daten werden dabei über die sogenannten SEC Filings gewonnen. Dies ist ein formales und standardisiertes Dokument, welches amerikanische Unternehmen bei der U.S. Securities and Exchange Commission (SEC) einreichen müssen. Echtzeit Aktienkurse werden direkt von standardisierten Schnittstellen bestimmter Börsen gewonnen. Die API bietet mehr als 25.000 Aktien, unter anderem auch an der deutschen Börse „XETRA“ gehandelte Wertpapiere (vgl. Financial Modeling Prep - FinancialModelingPrep o. D.). Die Mehrländerbörse „Euronext“ mit Sitz in Amsterdam ist ebenfalls vertreten. Die API bietet eine gründliche Dokumentation auf deren Webseite, welche dem Nutzer Informationen über jeden einzelnen Endpunkt zur Verfügung stellt. Diese beinhaltet unter anderem den API-Link und ein Beispiel. Die Endpunkte unterteilen sich wie folgt, wobei die für den Power BI Bericht ausgewählten Endpunkte fett markiert sind:

- Unternehmensbewertung (**Profil**, Schlüsselkennzahlen, finanzielles Wachstum, ...)
- Weiterführende Daten (Quartalsweiser Ergebnisbericht, Daten wichtiger Vorkommnisse, ...)
- Interner Handel
- Kalender (IPO Daten, Dividendendaten, Veröffentlichung der Geschäftszahlen, ...)
- Institutionelle Fonds (**ETF Sektorengewichtung**, **ETF Ländergewichtung**, ...)
- Aktien Zeitreihen (Echtzeitpreise, **historische Preise**, historische Dividenden, ...)
- Technische Indikatoren (Tägliche Indikatoren, historische Indikatoren, ...)
- Markt Indizes (Liste der S&P500 Aktiengesellschaften, Informationen über die größten Indizes, ...)
- ...

Mit einem Starter-API-Schlüssel bekommt man 300 API Aufrufe pro Sekunde und Zugriff auf alle Endpunkte.

2. Datenbankverbindung

2.1 Verbinden mit einer PostgreSQL-Datenbank mit Python

Zum Verbinden mit der Datenbank wird hier die SQLAlchemy Library importiert. SQLAlchemy ist ein Open-Source-SQL-Toolkit. Unterstützt werden viele relationale Datenbanken, darunter auch DB2- und MySQL-Datenbanken. Zunächst einmal müssen die beiden folgenden Bedingungen erfüllt sein:

```
pip install sqlalchemy
```

```
from sqlalchemy import create_engine
```

Über „pip install“ installiert man das Paket. Anschließend importiert man die „create_engine“ Funktion. Diese erstellt ein Engine-Objekt basierend auf einer URL, die sich folgendermaßen zusammensetzt:

```
dialect+driver://username:password@host:port/database
```

Für die Verbindung mit einer PostgreSQL-Datenbank sieht die URL so aus:

```
postgresql://username:password@host:port/database
```

Die Verbindung zu der erstellten lokalen PostgreSQL-Datenbank stellt man also mit folgendem Code her:

```
db_password = "wkb6"  
engine =  
create_engine('postgresql://postgres:{}@localhost:5432/StockData'.format(db_password))
```

Die Engine-Variable beinhaltet nun die Verbindung zur Datenbank und wird später zum Ausführen von Operationen auf der Datenbank verwendet.

2.2 Verbinden mit einer PostgreSQL-Datenbank mit R

Zur Verbindung von R mit der Datenbank wird ebenfalls zunächst ein Paket benötigt. Das ‚RPostgres‘-Paket enthält einen Treiber für R zum Zugriff auf ‚PostgreSQL‘-Datenbanksysteme. DBI unterteilt die Konnektivität zum Datenbank Management System in ein "Frontend" und ein "Backend". Anwendungen verwenden nur die Front-End-API. Die Back-End-Funktionen, die mit bestimmten DBMS (SQLite, MySQL, PostgreSQL, usw.) kommunizieren, werden von Treibern (anderen Paketen) bereitgestellt. Die folgenden Bedingungen müssen also erfüllt sein:

```
install.packages('RPostgres')  
library(DBI)
```

Zum Verbinden mit einer Datenbank wird nun, ähnlich wie schon in Python, eine Funktion aufgerufen. Dabei wird die Connection jedoch nicht über eine URL gelöst, sondern die benötigten Informationen werden als Parameter der Funktion übergeben.

```
db <- 'StockData'  
host_db <- 'localhost'  
db_port <- '5432'  
db_user <- 'postgres'  
db_password <- 'wkb6'  
  
con <- dbConnect(RPostgres::Postgres(),  
                 dbname = db,  
                 host=host_db,  
                 port=db_port,  
                 user=db_user,  
                 password=db_password)
```

Mit dem ersten Parameter, welcher der dbConnect-Funktion übergeben wird, wird also das DBI-Treiber-Objekt definiert. Die restlichen Parameter sind Informationen, die benötigt werden, um die Datenbankverbindung herzustellen. Die „con“-Variable beinhaltet nun, ähnlich wie die ‚Engine‘-Variable im Python Programm, die Verbindung zur Datenbank. Diese wird immer mit angegeben, wenn Operationen auf der Datenbank ausgeführt werden.

3. Programme zur Verwendung der API

3.1 Vorgehen zum Extrahieren der verfügbaren Aktienkürzel mit Python

Um an eine möglichst große Anzahl von Daten für verschiedene Aktien zu gelangen, ist der erste Schritt, eine Liste aller verfügbaren Wertpapiere zu finden. Da die Identifikation der Aktien über deren Kürzel funktioniert und man diese auch zur API-Anfrage benutzt, sollten diese auf jeden Fall mitgeliefert werden. Der Endpunkt „Symbols List“ stellt eine Liste mit allen verfügbaren Aktien zur Verfügung. Das JSON-Dokument hat die Attribute Symbol, Name, Price und Exchange. Ein Beispiel für ein solches Objekt sieht man in Abbildung 2.

```
"symbol" : "GME",  
"name" : "GameStop Corp",  
"price" : 150.88,  
"exchange" : "New York Stock Exchange"
```

Abbildung 2: Aufbau des JSON-Objektes

Für den Erhalt einer Liste mit den Symbolen sowie für das Abspeichern der gesamten Daten in der Datenbank, wird wie in Abbildung 3 beschrieben vorgegangen:

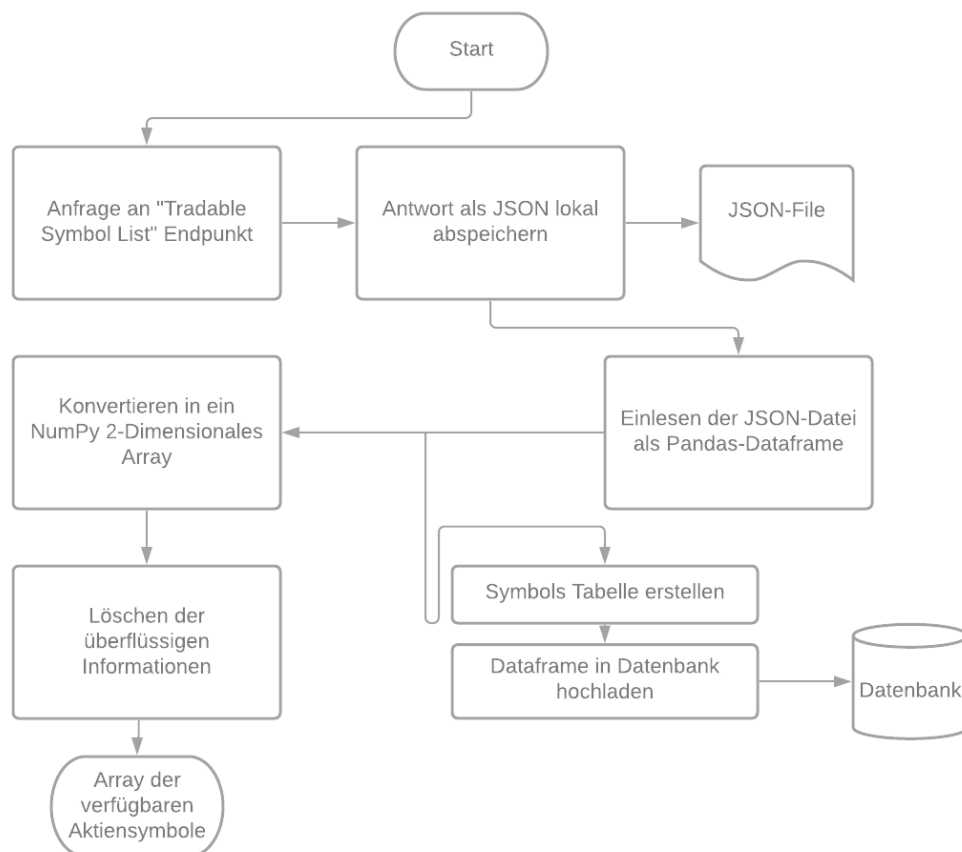


Abbildung 3: API-Zugriff und Weiterverarbeitung der verfügbaren Aktienkürzel

Die konkrete Durchführung des Programms wird im Folgenden anhand der Programmiersprache Python erklärt.

Der API-Zugriff sowie das lokale Abspeichern erfolgen innerhalb der Funktion `get_data()`.

```
def get_data() :
    request = requests.get(
        "https://financialmodelingprep.com/api/v3/stock/list?apikey={}".format(api_
        key) )

    with open(r'C:\Users\timos\Documents\HTML\python-
    eodhistoricaldata\CSV_try\Symbols.json', 'w') as f:
        json.dump(request.json(), f)
```

Die Funktion `get()` des Request-Objektes startet eine Anfrage an die als Argument angegebene URL und speichert deren Ergebnis unter `request` ab.

Im nächsten Schritt wird dann mit dem `with open()` Pattern ein lokales JSON-Dokument erstellt. Angegeben wird hierbei der Speicherort bzw. dessen Pfad. Mit dem `r`-Flag vor dem String wird der String als „Raw-String“ behandelt. Dadurch werden Zeichen wie Backslash (`"\"`) als Literalzeichen interpretiert. Dies bedeutet, dass das Zeichen nicht als Escape-Zeichen verstanden wird. Für Speicherpfade wird dieses Flag immer benötigt. Das Argument „`w`“ öffnet oder erstellt ein Dokument zum Bearbeiten. Dadurch wird der eingerückte Code in dem Dokument als Schreiboperation ausgeführt. Die `json.dump()` Methode konvertiert das Python-Request-Objekt zu einer validen JSON-Datei und speichert diese.

Die Funktion `get_symbols_to_array()` holt sich die JSON-Datei und entfernt die nicht gebrauchten Informationen. Außerdem ruft sie die Funktion `create_symbols_table()` auf, welche die JSON-Datei in die Datenbank schreibt.

```
def get_symbols_to_array() :

    df = pd.read_json(r"C:\Users\timos\Documents\HTML\python-
    eodhistoricaldata\CSV_try\Symbols.json")
    create_symbols_table(df)
    df = df.to_numpy()
    df = df[:, 0]
    return (df)
```

Mit der Pandas Methode `read_json()` wird die JSON-Datei eingelesen. Der Dataframe wird bei dem nachfolgenden Aufruf zum Speichern des Dokumentes direkt an die Funktion übergeben. Danach wird der Dataframe zu einem NumPy-Array konvertiert. Dies lässt sich so beschreiben:

`df[n][m]` = Die n-Dimension steht jeweils für ein Objekt, die m-Dimension steht für die Attribute des Objektes. In Abbildung 4 ist der Aufbau des Arrays dargestellt:

```
[['SPY' 'SPDR S&P 500' 415.27 'New York Stock Exchange Arca']  
 ['CMCSA' 'Comcast Corp' 54.345 'Nasdaq Global Select']  
 ['KMI' 'Kinder Morgan Inc' 16.595 'New York Stock Exchange']  
 ...  
 ['IMPL' 'Impel NeuroPharma, Inc. Common Stock' 0.0 'Nasdaq']  
 ['RAIN' 'Rain Therapeutics Inc. Common Stock' 0.0 'Nasdaq']  
 ['SWIM' 'Latham Group, Inc. Common Stock' 0.0 'Nasdaq']]
```

Abbildung 4: Aufbau des Arrays nach Konvertierung

Das Array wird jetzt durch `df[:,0]` so bearbeitet, dass immer nur das erste Attribut des Objektes bestehen bleibt. Es bleibt also das in Abbildung 5 gezeigte NumPy Array:

```
['SPY' 'CMCSA' 'KMI' ... 'IMPL' 'RAIN' 'SWIM']
```

Abbildung 5: NumPy Array nach Bearbeitung

Momentan hat es eine Länge von 23297, dies hängt natürlich davon ab, wie viele Aktien die API am Zugriffsdatum zur Verfügung stellt.

Die Funktion `create_symbols_table()`, welche dem Dataframe vor Formatierung übergeben wurde, erstellt eine Tabelle, die alle in der JSON-Datei vorhandenen Informationen in der Datenbank abspeichert.

```
def create_symbols_table(df):  
    df = df.to_csv()  
    df = df[['symbol', 'name', 'price', 'exchange']]  
    df = df.fillna(0)  
    df['updated'] = pd.to_datetime('now')  
    df.to_sql('Symbols', engine, if_exists='replace', index=True)
```

Dafür wird der übergebene Dataframe mit den passenden Bezeichnungen versehen. Die nicht vorhandenen Werte werden mit Hilfe der `fillna()` Funktion durch einen NaN-Wert ersetzt und es wird eine Reihe für den Zeitstempel hinzugefügt (der Zeitstempel wurde wegen fehlender Kompatibilität zum SQL Server Management Studio für das Beispielpogramm auskommentiert). Dafür wird mit dem `to_datetime('now')` die aktuelle Zeit verwendet. Dieser Dataframe wird dann mit der `to_sql()` Funktion in die Datenbank übertragen. Das erste Argument ist der Name der Tabelle innerhalb der Datenbank. Die eingerichtete

Datenbankverbindung, die sich hinter dem engine verbirgt, wird bereits in einem vorherigen Abschnitt genauer erklärt. Der Tabelle wird noch ein Index hinzugefügt. Falls eine Tabelle mit demselben Namen schon vorhanden ist, soll diese überschrieben werden.

3.2 Abrufen und Speichern der Fundamentaldaten der Unternehmen mit Python

Da pro API-Aufruf maximal die Fundamentaldaten eines einzigen Unternehmens zurückgegeben werden können, war die Überlegung, durch die angelegte Liste der Unternehmen zu iterieren und dabei jeweils die Request-Antworten in eine einzige JSON-Datei abzuspeichern. Den Anfang macht also eine Variable, welche diejenige Funktion aufruft, die das benötigte Array zurückgibt.

```
symbols = get_symbols_to_array() # Array with all US-Symbols
```

Dieses Array übergibt man dann der Funktion, welche den API-Aufruf tätigt und das Ergebnis in einem JSON-File lokal abspeichert.

```
def get_fund_data_from_api(symbols):  
    for i in symbols:  
        request = requests.get(  
            "https://financialmodelingprep.com/api/v3/profile/{}?apikey=ableb63d17e1f21e847fe71162b752".format(i) )  
  
        with open(csv_path + r"\Fundamentals.json", 'a') as f:  
            json.dump(request.json(), f)
```

Beim Aufruf der Funktion wird eine For-Schleife gestartet, die durch alle Inhalte des übergebenen Arrays iteriert. Zum Ausführen des API-Aufrufes wird nun wieder die Funktion `get()` des `request`-Importes durchgeführt. Die Stelle, an der das Aktiensymbol stehen muss, wird durch `i` ersetzt. `i` nimmt im Laufe der Schleife alle enthaltenen Strings an. Das `open()` Pattern speichert das Ergebnis nun lokal unter dem angegebenen Pfad ab. Das 'a' steht für den append Modus. Dies heißt auf Deutsch so viel wie „anhängen“ und bedeutet, dass die Datei nicht überschrieben werden soll - wie es mit 'w' der Fall wäre - sondern das Ergebnis des `requests` einfach angehängt werden soll. So wird jeder neue Aufruf lediglich an das Ergebnis des vorherigen Aufrufes angehängt. Die JSON-Datei sieht nach dem Durchlauf wie in Abbildung 6 dargestellt aus:

```
[{"symbol": "SPY",  
  ....  
  "isEtf": true,  
  "isActivelyTrading": true}]  
[{"symbol": "CMCSA",  
  "price": 52.78,  
  ....
```

Abbildung 6: Aufbau der JSON-Datei nach dem Zusammenfügen

Sie enthält nach ca. 4 Stunden Durchlaufzeit (abhängig von vielen Faktoren) die Fundamentaldaten zu allen im Array gespeicherten Aktiengesellschaften. Nun sind noch einige Formatierungen zu erledigen, um die Datei in eine valide JSON-Datei zu verwandeln. Ein JSON-Validierungsprogramm gibt bei fehlender Formatierung folgenden Fehler zurück:

```
Error: Parse error on line 36:
...elyTrading": true}][{"symbol": "CMCSA
-----^
Expecting 'EOF', '}', ',', ']', got '['
```

Abbildung 7: Fehleranzeige der JSON-Validierung

Problematisch sind also die "[" und "]" Klammern zwischen den verschiedenen JSON-Objekten. Diese gehören nämlich nur an den Anfang und an das Ende der Datei.

```
with open('{} / {}.json'.format(csv_path, "Fundamentals"), 'r') as infile, \
      open('{} / {}.json'.format(csv_path, "Fundamentals1"), 'w') as out-
file:
    data = infile.read()
1   data = data.replace("[", "")
2   data = data.replace("]", "")
3   data = data.replace("}", ",")
    outfile.write(data)
```

Der Text wird also so verändert:

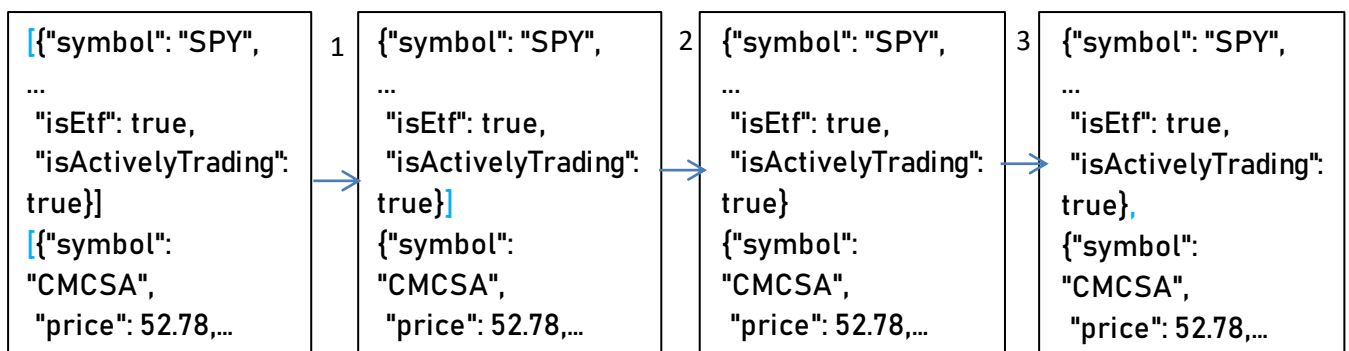


Abbildung 8: Bearbeitungsschritte der JSON-Datei

Da die Klammern am Anfang und Ende der JSON-Datei dadurch gelöscht werden, müssen diese nun wieder hinzugefügt werden. Dies wird wieder über die append-Funktionalität gelöst, indem wir einfach ein "]" an das Ende der Datei schreiben:

```
with open('{} / {}.json'.format(csv_path, "Fundamentals1"), 'a') as the_file:
    the_file.write(']')
```

Für die eckige Klammer am Anfang der Datei lesen wir zunächst die Datei im „r+“-Modus ein. Der „r+“-Modus ermöglicht das Lesen und Schreiben einer Datei. Man speichert die

Datei unter der Variablen `a` und öffnet im nächsten Schritt dieselbe Datei im „w+“-Modus. Um die eckige Klammer am Anfang einzufügen, kombiniert man einfach den "["-String mit dem davor unter der Variablen `a` gespeicherten Text. Dadurch wird an erster Stelle der Datei die eckige Klammer eingefügt.

```
with open('{} / {}.json'.format(csv_path, "Fundamentals1"), "r+") as f:
    a = f.read()
with open('{} / {}.json'.format(csv_path, "Fundamentals1"), "w+") as f:
    f.write "[" + a)
```

Als letzter Schritt wird jetzt noch ein Komma entfernt, welches durch `data.replace("}", ",")` nun zu viel enthalten ist. Das letzte Objekt wird nämlich dadurch, dass jedes „}“ mit „},“ ersetzt wird, auch mit einem Komma versehen. Dieses gehört aber nur zwischen die Objekte und nicht ans Ende, weshalb es mit

```
data = data.replace("},", "}]")
```

entfernt wird.

Am Ende der Funktion werden noch die unnötigen Dateien gelöscht und die bearbeitete JSON-Datei in „Fundamentals1“ umbenannt. Diese ist nun für Pandas auch als solche einlesbar und damit bereit, in die Datenbank geschrieben zu werden. Die Funktion, welche das übernimmt, heißt `create_table()`.

```
def create_table(): # Take JSON -> to CSV -> to DB
    # Import CSV
    df = pd.read_json('{} / {}.json'.format(csv_path, "Fundamentals1"))
    df = df.to_csv()
```

Die Funktion liest zunächst die JSON-Datei in einen Pandas-Dataframe ein und wandelt diese anschließend in eine CSV-Datei um. Danach wird der Dataframe noch formatiert, es werden Attributnamen definiert, die NULL-Werte ersetzt, sowie ein Timestamp unter dem Namen ‚updated‘ hinzugefügt.

Die Pandas `to_sql()` Funktion nimmt einen Dataframe und schreibt ihn in die Datenbank.

```
df.to_sql('FundamentalData', engine, if_exists='replace', index=False)
```

Der erste Parameter gibt an, wie die Tabelle innerhalb der Datenbank benannt werden soll. Engine ist die Variable, die die Datenbankverbindung repräsentiert. Für die `to_sql()` Funktion wird eine Connection vorausgesetzt, die mit SQLAlchemy eingerichtet wurde. Somit werden auch alle Datenbanken unterstützt, die von SQLAlchemy unterstützt werden. Wenn die Tabelle mit dem definierten Namen bereits existiert, soll diese überschrieben werden. Ein Index soll nicht hinzugefügt werden.

3.3 Zugriff und Speicherung der historischen Preisdaten mit Python

Die Strategie zur Heranziehung der historischen Preisdaten der verfügbaren Aktien ist etwas anders als das Vorgehen bei den Fundamentaldaten. Zum einen werden die Informationen nicht mit dem Append-Modus aneinandergehängt, sondern einzeln verarbeitet. Zum anderen werden die Inhalte der einzelnen Dateien zu einem SQL-INSERT-Statement formatiert.

Für den ersten Aufruf des Programmes macht es Sinn, ein statisches Datum einzubauen, ab wann man die Preisinformationen haben möchte. Nachdem die Daten einmal in die Datenbank geladen werden, ist es sinnvoll, in der Datenbank zu prüfen, bis wann dort Preisdaten vorhanden sind. Genau dazu dient die Funktion `get_highest_date()`.

```
def get_highest_date():
    last_date = engine.execute("""Select max(date) from
public.dailyprices""")
    for row in last_date:
        result = row

    datetim = datetime.datetime(int(row[0][0:4]), int(row[0][5:7]),
int(row[0][8:10]))
    datetim += datetime.timedelta(days=1)
    return str(datetim)[0:10]
```

Sie holt sich über einen SQL-Befehl das höchste Datum, welches sich in der Tabelle befindet. Da die nächste Abfrage ab dem darauffolgenden Tag starten soll, wird mithilfe des Datetime-Paketes noch ein Tag auf das Datum aufaddiert. Die Funktion gibt einem also den Tag zurück, der auf den neusten Datumseintrag in der Preistabelle folgt. Genau mit diesem Datum, gespeichert unter der `highest_Date`-Variable wird dann in der nächsten Funktion wieder auf die API zugegriffen.

```
def get_prices(StockSymbol):
    request = requests.get(
        "https://financialmodelingprep.com/api/v3/historical-price-
full/{}?from={}&apikey=ableb63d17e1f21ebe847fe71162b752".format(
        StockSymbol, highest_Date))
```

Dadurch erhält man im JSON-Format verschiedene, tägliche Informationen zum Preisverhalten einer Aktie. Diese Request-Datei wird nun durch die `from_dict()` Funktion von einem Dictionary zu einem Pandas-Dataframe umgewandelt und zurückgegeben.

Um die Datei ordentlich zu einem INSERT-Befehl zu formatieren, war die Idee, zeilenweise mit Hilfe der `iterrows()`-Funktion über die einzelnen Reihen zu iterieren und dabei die Werte abzugreifen. Es stellte sich heraus, dass es Probleme dabei gab, mit dem Array im Objekt zu arbeiten, welches die historischen Preisinformationen beinhaltet.

```

      symbol                                     historical
0      GME  {'date': '2021-04-09', 'open': 169.699997, 'hi...
1      GME  {'date': '2021-04-08', 'open': 185.880005, 'hi...
2      GME  {'date': '2021-04-07', 'open': 183.220001, 'hi...
3      GME  {'date': '2021-04-06', 'open': 185.210007, 'hi...
4      GME  {'date': '2021-04-05', 'open': 171.0, 'high': ...
..      ...                                     ...
315     GME  {'date': '2020-01-08', 'open': 5.49, 'high': 5...
316     GME  {'date': '2020-01-07', 'open': 5.77, 'high': 5...
317     GME  {'date': '2020-01-06', 'open': 5.8, 'high': 5...
318     GME  {'date': '2020-01-03', 'open': 6.21, 'high': 6...
319     GME  {'date': '2020-01-02', 'open': 6.14, 'high': 6...

[320 rows x 2 columns]

```

Abbildung 9: Aufbau des Dataframes ohne Formatierung

In Abbildung 9 sieht man den Aufbau der als Dataframe eingelesenen JSON-Datei. Beim Iterieren über die Reihen werden die Inhalte der „historical“-Zelle als Eins ausgelesen, wodurch ein Error ausgelöst wird.

Zugriff auf die Symbol-Werte:

```
df["symbol"][5]
```

Zugriff auf die anderen Werte:

```
df["historical"][5]["high"]
```

Zum Lösen dieses Problems kann man mit RegEx Anpassungen vornehmen. Um die ganze Formatierungsfunktion wurde ein try/except Block gebaut, da ein paar der Endpunkte leere Dateien liefern und damit zu einem Fehler und folglich dem kompletten Programmabbruch führen. Falls dies passiert, wird der Error angezeigt und die Anzahl der aufgetretenen Fehler gezählt. Bei der Formatierung wird wie folgt vorgegangen:

```

df = df.to_csv() #to_csv don't work as thought therefore we have to
resolve the hist. Data Array in JSON-File

abhier = df.index("{}") # 1
df = df[abhier:] #1
df = "[" + df + "]" #2
df = re.sub("\d{1,3},\w*.{0,1}\w{1,2},", " ", str(df)) #3

df = re.sub('}' '{', '{', df) #4
df = re.sub('"\'s+', '"', df) #5
df = re.sub('""', '"', df) #6
df = re.sub('}' "\s+', '}]']', df) #7

```

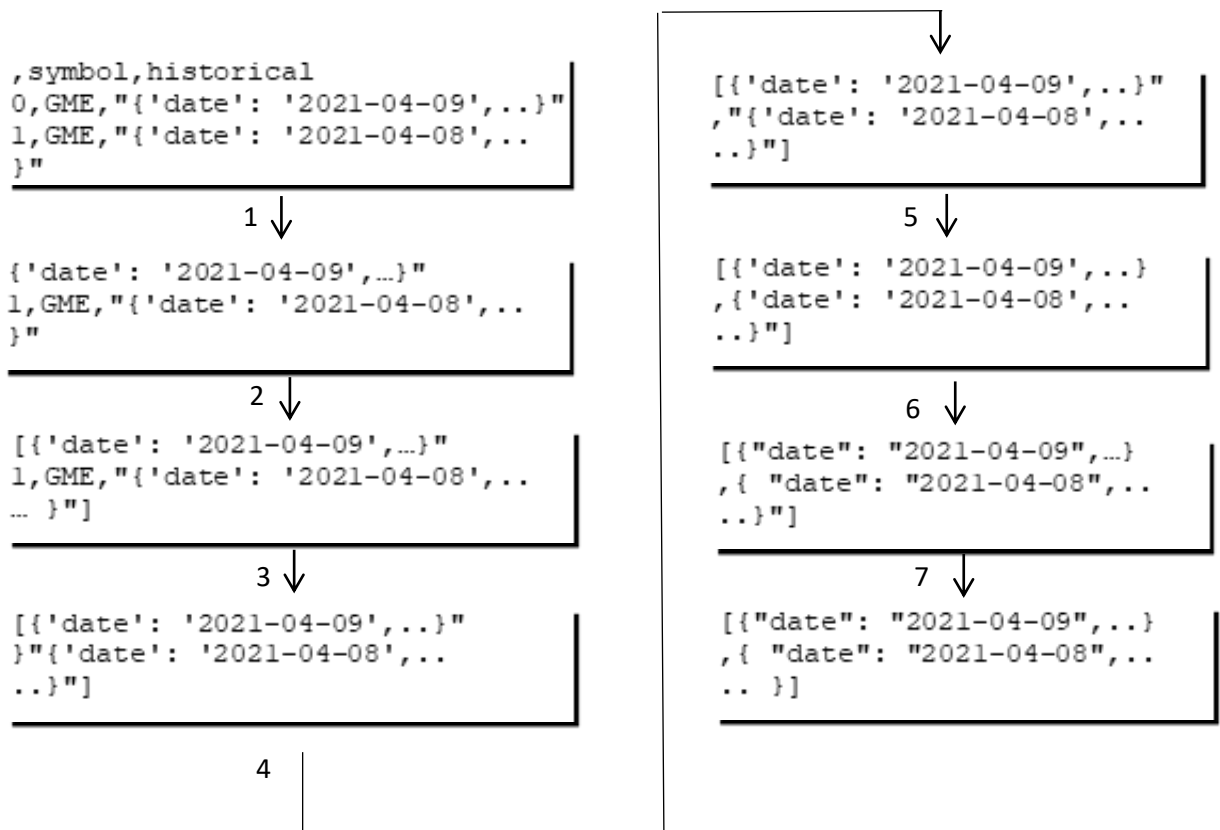


Abbildung 10: Vorgehen beim Auflösen des historical Arrays

Nach der in Abbildung 10 dargestellten Formatierung kann man nun mit Pandas `read_json()`-Funktion den abgeänderten String einlesen und als CSV-Datei konvertieren. Den resultierenden Dataframe sieht man in Abbildung 11. Dieser wird auch von der Funktion zurückgegeben.

	date	open	high	low	close	adjClose	\
0	2021-04-09	169.699997	171.580002	153.000000	158.360001	158.360001	
1	2021-04-08	185.880005	185.880005	164.300003	170.259995	170.259995	
2	2021-04-07	183.220001	184.500000	176.110001	177.970001	177.970001	
3	2021-04-06	185.210007	192.000000	183.559998	184.500000	184.500000	
4	2021-04-05	171.000000	195.000000	164.809998	186.949997	186.949997	
..
315	2020-01-08	5.490000	5.850000	5.410000	5.720000	5.720000	
316	2020-01-07	5.770000	5.830000	5.440000	5.520000	5.520000	
317	2020-01-06	5.800000	5.910000	5.600000	5.850000	5.850000	
318	2020-01-03	6.210000	6.250000	5.840000	5.880000	5.880000	
319	2020-01-02	6.140000	6.470000	6.070000	6.310000	6.310000	

Abbildung 11: resultierender Dataframe

Zum Einpflegen der Daten in die Datenbank werden zwei verschiedene Funktionen benutzt. Die Funktion `create_table()` erstellt eine Datenbank mit dem Name der Variablen 'DatabaseName' und schreibt den ersten resultierenden Dataframe hinein. Das angewandte Prinzip ist dabei genau dasselbe wie bei den Fundamentaldaten.

Alle anderen historischen Preisinformationen werden der Tabelle durch die Funktion `import_csv_into_table()` beigelegt. Der Dataframe wird darin zu einer CSV-Datei konvertiert, mit einer Indexspalte versehen und zu einem SQL-Befehl formatiert, welcher dann auf der Datenbank ausgeführt wird. Um den SQL zu erstellen, wird die Funktion `iterrows()` von Pandas benutzt, wie im unten stehenden Code zu sehen ist. Die Funktion `iterrows()` wird verwendet, um Dataframe-Zeilen als (Index-, Daten-) Paare zu durchlaufen. Die Klammern im String stehen für die Werte. Diese werden durch jede Iteration durch die Reihen des Dataframes eingefügt. Am Anfang wird der benötigte Index angegeben.

Die Spalten des Dataframes erkennt man am Spaltennamen und der vorausgehenden Bezeichnung „row“. Da durch Auflösung der vorangegangenen Formatierung des Dataframes die Bezeichnung der Aktie verloren gegangen ist, wird diese hier ebenfalls zum Insert-Befehl hinzugefügt. Es wird außerdem ein Timestamp angefügt.

```
insert_statement = """INSERT INTO {}  
VALUES """.format(DatabaseName)  
  
values = ",".join(["{}'{}'.format(DatabaseName,  
index,  
row.date,  
row.open,  
row.high,  
row.low,  
row.close,  
row.adjClose,  
row.volume,  
row.unadjustedVolume,  
row.change,  
row.changePercent,  
row.vwap,  
row.label,  
row.changeOverTime,  
StockSymbol,  
pd.to_datetime('now')  
) for index, row in df.iterrows()])  
  
query = insert_statement + values
```

Der SQL-Befehl wird dann wieder mit unserer Datenbankverbindung „engine“ und der *execute()*-Funktion aufgeführt.

Die Main-Funktion muss nun die Funktionen so kombinieren, dass die Datenbank einmal erstellt wird und alle nachfolgenden Ergebnisse der API-Aufrufe in diese Datenbank

eingefügt werden. Also unterteilt sich die Ausführung in zwei Abschnitte. Im unteren Code erkennt man, wie die Prozedur mit dem Ergebnis des ersten API-Aufrufs, also mit der Aktie an Position 0 des StockSymbol Arrays, ein einziges Mal durchgeführt wird.

```
get_prices(StockSymbol[0])
create_table(StockSymbol[0], formatting(StockSymbol[0]))
```

Die anderen Elemente des StockSymbol Arrays werden dann mit einer For-Schleife durchgegangen. Diese startet mit dem Index 1 und läuft über die Länge des Arrays. Der einzige Unterschied in der Logik innerhalb der For-Schleife ist, dass statt *create_table()* die Funktion *import_csv_into_db()* benutzt wird. Wie man unten ebenfalls sehen kann, wird auf der Konsole nach jeder hundertsten Iteration eine Zwischenbilanz ausgegeben.

```
for i in range(1, len(StockSymbol)):
    if i % 10 == 0: print(str(round(i/len(StockSymbol), 4)) +
                        "% imported" + ': time elapsed (hh:mm:ss.ms)')
    {}'.format(datetime.now() - start_time)
    )
    get_prices(StockSymbol[i])
    import_csv_into_table(StockSymbol[i], formatting(StockSymbol[i]))
```

3.4 Zugriff auf die Sektoren und Länderverteilung von ETF's mit R

Da jene durch die API in die Datenbank geladenen Daten auch viele Informationen zu sogenannten „Exchange Traded Funds“ enthalten, für deren Darstellung und Verständnis die Sektoren und Länderverteilung essenziell ist, wird im Nachfolgenden erklärt, wie man an diese Daten kommt. Benutzt wird hierzu die Programmiersprache R, welche ähnlich wie Python im Data Science Umfeld eine große Beliebtheit aufweist. Grob zusammengefasst holt sich das Programm mit einem SQL alle Kürzel, welche im Feld „isETF“ die Angabe true haben, macht mit diesen einen API-Aufruf und lädt das Ergebnis in die Datenbank. Dazu wird die im Teil „Verbinden mit einer PostgreSQL Datenbank mit R“ erstellte Datenbankverbindung benutzt. Mit der Funktion `dbGetQuery()` der Bibliothek „DBI“ aus dem Paket „RPostgres“ kann ein SQL ausgeführt werden. Diese nimmt als erstes Argument „con“, also die eingerichtete Datenbankverbindung. Das SQL grenzt auch gleichzeitig durch die inner-Join ETF's aus, für welche keine Informationen in der Preistabelle vorliegen.

```
etf = dbGetQuery(con, 'SELECT distinct "symbol" FROM
public."Fundamental Data" as fd
inner join public."dailyprices" as dp on dp.stock = fd.symbol
where "isEtf" = TRUE')
etf = etf$symbol
```

Die Variable „etf“ ist jetzt ein Array, welches alle Kürzel des Ergebnisses der SQL-Abfrage beinhaltet. Durch das Dollarzeichen und den Spaltennamen wird der Index entfernt und nur das Ergebnis extrahiert. Die Änderung der Ausgabe vor und nach dem „etf\$symbol“ sieht man in Abbildung 12:

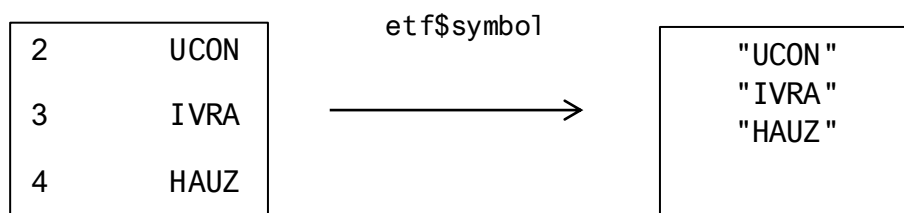


Abbildung 12: Änderung der Daten durch `etf$symbol`

Dieses Array wird nun wieder benutzt, um die API-Aufrufe durchzuführen. Damit aber die `rbind()`-Funktion funktioniert, muss die Variable `df1` erst einmal das gleiche Format bekommen, wie das, welches die Variable `df2` durch die API-Aufrufe annimmt. Dafür wird vor der For-Schleife schon ein API-Aufruf durchgeführt, sodass `rbind()` problemlos funktioniert.

```

for(symbols in etf){
  tryCatch({

    url = paste(api_url_for_sectorspt1,symbols,api_key,sep = "")
    req1 = httr::GET(url)
    char1 <- rawToChar(req1$content)
    char1 #1
    df2 <- jsonlite::fromJSON(char1)
    df2 #2
    df2$ETF = symbols
    df2 #3
    df1 <- rbind(df1, df2)
    Sys.sleep(0,01)
  },error=function(e){cat("ERROR :",conditionMessage(e), "\n")})
}

```

Die Variable df2 erhält hier jeweils den Inhalt des API-Aufrufes und wird dann entsprechend transformiert. Der transformierte Inhalt wird dann an die Variable df1 angehängt, sodass diese am Ende alle Informationen beinhaltet. Um leere API-Aufrufe und den damit auftretenden Fehler abzufangen, gibt es hier ebenfalls einen TryCatch-Block. Die Funktion *paste()* fügt die URL zusammen. Die Anfrage geschieht durch „httr:GET“, welcher als Argument die URL übergeben wird. Wenn man das Antwortobjekt (req1) auf der Konsole ausgeben würde, würde dies nicht den kompletten Inhalt zurückgeben, sondern nur einen Ausschnitt daraus, inklusive weiterer Informationen. Das req1-Objekt, welches von der Anfrage mit dem Kürzel etf[4] zurückgegeben wird, ist hier dargestellt.

```

> req1
Response [https://financialmodelingprep.com/api/v3/etf-sector-
weightings/HAUZ?apikey=ab1eb63d17e1f21ebe847fe71162b752]
  Date: 2021-05-01 11:26
  Status: 200
  Content-Type: application/json; charset=UTF-8
  Size: 732 B
[ {
  "sector" : "Commercial REITs",
  "weightPercentage" : "33.68%"
} ...

```

Um den kompletten Inhalt aus der Anfrage zu bekommen, werden die nächsten, in Abbildung 13 sichtbaren Schritte benötigt. Dabei wird der Inhalt von req1 mit der Funktion *rawToChar()* von Bytes zu einem Zeichenvektor auf einzelnen Bytes konvertiert. Dies wird vorgenommen, da „req1\$content“ einen Hexa-Dezimalcode zurückliefert. Der Zeichenvektor wird dann mithilfe der aus dem Paket „jsonlite“ stammenden Funktion *fromJSON()* zu einem R-Objekt konvertiert. Da jetzt noch die Information fehlt, um welchen ETF es sich handelt, wird dies im dritten Schritt noch als Spalte hinzugefügt.

1.	<pre>"[{\n \"sector\":\n \"Commercial REITs\", \n \"weightPercentage\":\n \"33.68%\" \n},\n {\n \"sector\": \"Holding\nCompanies\", \n \"weightPercentage\":\n \"0.43%\" \n}, ...</pre>	2.	<table><tr><th></th><th>sector</th><th>weightPercentage</th></tr><tr><td>1</td><td>Commercial REITs</td><td>33.68%</td></tr><tr><td>2</td><td>Holding Companies</td><td>0.43%</td></tr></table>		sector	weightPercentage	1	Commercial REITs	33.68%	2	Holding Companies	0.43%			
	sector	weightPercentage													
1	Commercial REITs	33.68%													
2	Holding Companies	0.43%													
		3.	<table><tr><th></th><th>sector</th><th>weightPercentage</th><th>ETF</th></tr><tr><td>1</td><td>Commercial REITs</td><td>33.68%</td><td>CLG.TO</td></tr><tr><td>2</td><td>Holding Companies</td><td>0.43%</td><td>CLG.TO</td></tr></table>		sector	weightPercentage	ETF	1	Commercial REITs	33.68%	CLG.TO	2	Holding Companies	0.43%	CLG.TO
	sector	weightPercentage	ETF												
1	Commercial REITs	33.68%	CLG.TO												
2	Holding Companies	0.43%	CLG.TO												

Abbildung 13: Änderung des Inhaltes von req1

Die Funktion `rbind()` hängt den formatierten Inhalt nun an die „df1“. Nach jeder Iteration stoppt das Programm 0,1 Sekunden, da sonst das Aufruflimit von 300 API-Aufrufen pro Minute überschritten wird und keine Daten mehr geliefert werden.

Um die Daten nun noch in eine Tabelle zu schreiben, benutzt man wieder eine Funktion der DBI-Bibliothek.

```
dbWriteTable(con, name='ETD_DataSector',value=df1, overwrite =TRUE)
```

Diese kreiert eine Tabelle mit dem Namen „ETD_DataSector“, falls es diese schon gibt, wird sie überschrieben. Der Inhalt der df1 wird dann übertragen. Dabei wird die in „con“ gespeicherte Datenbankverbindung benutzt.

Das Prinzip des Programmes bleibt für die Länderverteilung genau dasselbe, der einzige Unterschied dabei ist die URL, die für die Anfrage benutzt wird. Die Form der Antwort ist dieselbe, sodass nichts geändert werden muss, außer der Name der Tabelle in der Funktion `dbWriteTable()` und der Wert der URL-Variable.

3.5 Zugriff auf historische Geldwechselkurse mit Python

Für die Geldwechselkurse gibt es bei Python eine passende Bibliothek, die eine Reihe von nützlichen Informationen bereitstellt. Die `forex_python`-Bibliothek gewährt Zugriff auf alle Wechselkurse seit 1999 und liefert auch eine umfangreiche Datenbasis für Kryptowährungen wie zum Beispiel Bitcoin. Sie stellt unter anderem auch die Funktion `get_rates()` zur Verfügung, die den Wert einer bestimmten Basiswährung in 31 andere Währungen umrechnet. Übergeben muss man der Funktion dabei nur die Basiswährung und das Datum. Um die Werte für einen bestimmten Zeitraum zu erhalten, kann man mit einer For-Schleife und Pandas `date_range()`-Funktion durch einen angegebenen Zeitraum iterieren.

```
daterange = pd.date_range(start='1/1/2020', end='5/1/2021')
c = CurrencyRates()

for date in daterange:
    rates = c.get_rates("EUR", date)
    df2 = pd.DataFrame.from_dict(rates, orient='index')
    # df2 = df2.transpose()
    df2.reset_index(level=0, inplace=True)
    df2["Date"] = date
    df2.columns = ["Currency", "Value", "Date"]
    new_rows = {"Currency": "EUR", "Value": 1, "Date": date}
    df2 = df2.append(new_rows, ignore_index=True)
    df = pd.concat([df2, df])
```

Der Zeitraum und das Currency Objekt wird oberhalb der For-Schleife definiert. Die darauffolgende For-Schleife iteriert dann durch alle Daten, die zwischen den beiden angegebenen liegen. Dabei wird die Variable „rates“ durch die `get_rates()`-Funktion mit Daten gefüllt. Diese stehen in einem Dictionary Objekt, weshalb in der darauffolgenden Zeile dieses in ein Dataframe umgewandelt wird. Mit „orient='index'“ gibt man die Orientierung der Daten an. Sollen die Zeilen der Schlüssel sein, so gibt man ‚index‘ an, sollen es die Spalten sein, gibt man ‚column‘ an. Den Unterschied sieht man in Abbildung 14.

<pre>data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']} pd.DataFrame.from_dict(data, orient='columns')</pre>			<pre>data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']} pd.DataFrame.from_dict(data, orient='index')</pre>		
col_1	col_2				
0	3	a			
1	2	b			
2	1	c			
3	0	d			

	0	1	2	3
row_1	3	2	1	0
row_2	a	b	c	d

Abbildung 14: Unterschied von orient='index' und orient='columns'

Mit `reset_index()` setzt man noch den Index des Dataframes zurück und fügt anschließend das Datum hinzu. Dies ist erforderlich, da es von der `get_rates()`-Funktion nicht mit übertragen wird. Mit „df2.columns“ werden die Überschriften korrekt benannt. Die nächste Zeile fügt für jedes Datum eine Spalte hinzu, die für den Wert des Euros 1 angibt. Dies dient dazu, dass bei

der späteren Umrechnung von Euro in Euro einfach der Betrag Mal 1/1 genommen wird und somit gleichbleibt. Eine IF-Bedingung im später erläuterten UPDATE-Statement kann damit umgangen werden. Mit der *append()*-Funktion wird die erstellte Spalte angehängt. Nach jeder Schleife wird das Ergebnis dem Dataframe „df“ angefügt. Wenn die Schleife durchgelaufen ist, wird der fertige Dataframe wieder mithilfe der SQLAlchemy-Bibliothek in der Datenbank abgelegt.

```
df = getCurrency()  
df.to_sql('CurrencyRates', engine, if_exists='replace', index=False)
```

4. PostgreSQL-Datenbank

4.1 Aufbau der Datenbank

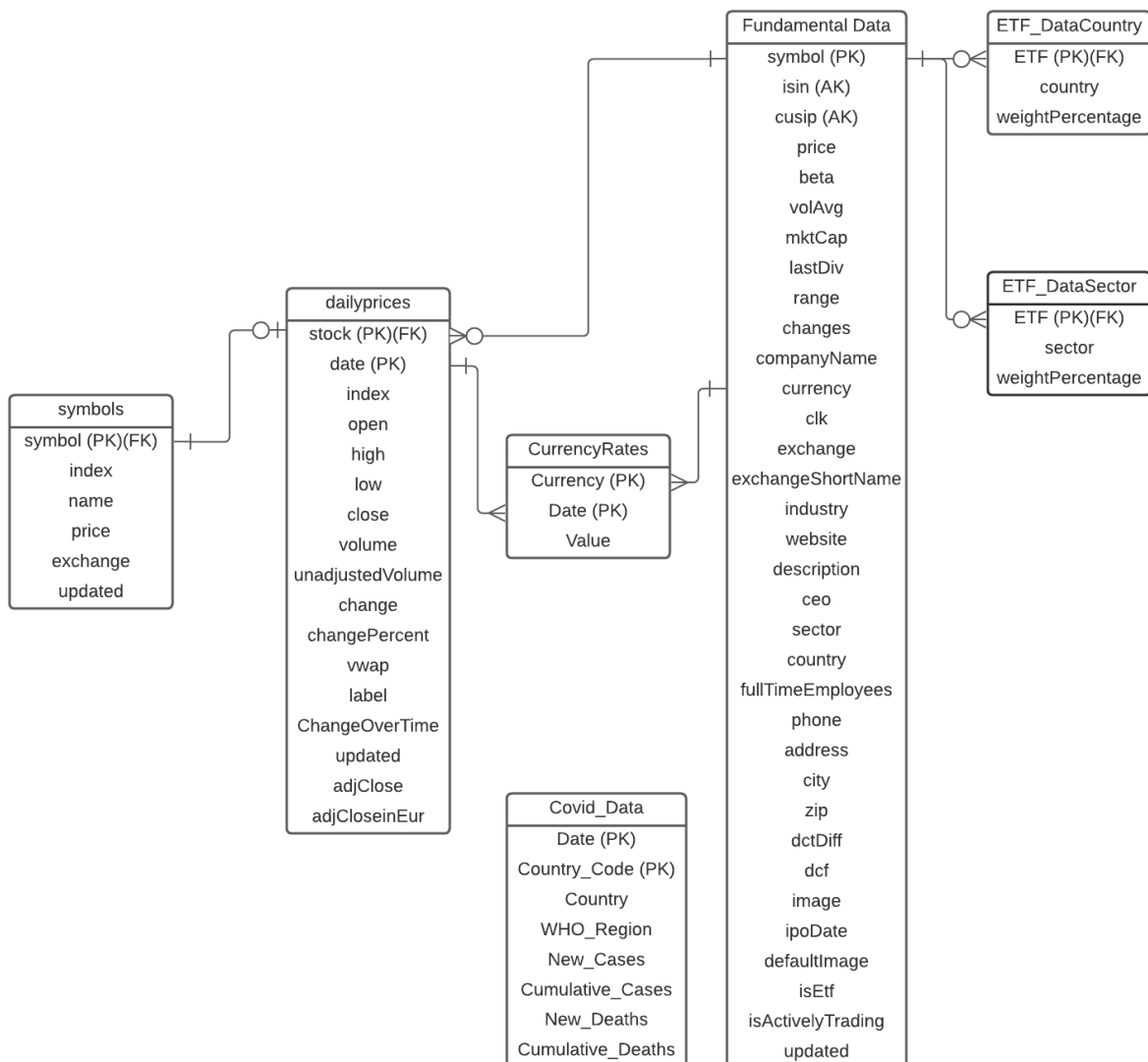


Abbildung 15: ER-Diagramm Datenbank

4.2 Datenbanklogik

4.2.1 Umrechnung der Währungen

Da die verschiedenen Kurse auch in unterschiedlichen Währungen übermittelt werden und der Bericht die Funktionalität bieten soll, alle Währungen in Euro anzuzeigen, ist ein Ziel, diese Umrechnung innerhalb der Datenbank vorzunehmen. Problem hierbei ist, dass die Währungen stark schwanken und dadurch die Umrechnung anhand des jetzigen Umrechnungsfaktors keinen Sinn ergeben würde. Das Python-Programm, welches die täglichen Währungsumrechnungen in die Datenbank schreibt, wurde bereits an anderer Stelle erklärt und wird deshalb hier nicht näher betrachtet. Die Tabelle, welche die Grundlage zur Umrechnung der Werte liefert, hat die in Abbildung 16 dargestellte Form.

CurrencyRates
Currency
Value
Date

Abbildung 16: Aufbau CurrencyRates

Von der Tabelle „CurrencyRates“ soll jetzt der Wert („Value“) benutzt werden, um diesen, abhängig von Datum und Währung, in Euro umzurechnen. Zunächst einmal wurde mit folgendem SQL überprüft, ob es Währungen in der Fundamentaldatentabelle gibt, die es in der Tabelle „CurrencyRates“ nicht gibt.

```
SELECT DISTINCT currency, count(*) FROM public."Fundamental Data"  
WHERE currency not in (  
SELECT "Currency" from public."CurrencyRates")  
GROUP BY currency
```

Nachdem die Abfrage keine Währungen aufzeigt, die in der „CurrencyRates“-Tabelle nicht vorhanden sind, können diesbezügliche Probleme schon einmal ausgeschlossen werden.

In einem ersten Schritt wird die „adjClose“-Spalte der „dailyprices“-Tabelle dupliziert. Dafür fügt man zuerst mit transact-SQL eine Spalte hinzu. Dies zeigt der folgende Code:

```
ALTER TABLE public."dailyprices"  
ADD COLUMN "adjCloseinEur" double precision;
```

Diese befüllt man dann einfach mit den Werten der „adjClose“-Spalte.

```
UPDATE public.dailyprices
SET "adjCloseinEur" = "adjClose"
```

Um jetzt die Spalte entsprechend zu bearbeiten, verwendet man folgendes Statement:

```
UPDATE public.dailyprices as h
SET "adjCloseinEur" =
"adjClose" * (1 / (c."Value"::double precision))
FROM public."CurrencyRates" as c, public."Fundamental Data" as fd
WHERE fd.symbol = h.stock
AND fd.currency = c."Currency"
AND substring(to_char(c."Date", 'YYYY-MM-DD') from 1 for 10) = h.date
```

Man nimmt den Wert, welcher sich in der „adjClose“-Spalte befindet und addiert darauf den im Nenner stehenden Wert „Value“. Dieser sagt aus, wie viel der angegebenen Währung an diesem Tag gleichwertig zu einem Euro war. Im Zähler steht eine 1. Die Logik dahinter ist in folgender Beispielrechnung gut erkennbar:

Angegeben: Wert 1 Euro am 04.05.2021 -> USD 1,2065

EURO(Gleichwertig zu einem USD) =

$1 * 1/1,2065 = 0,8289$ EUR bzw. $1/1,2065 = 0,8289$ EUR

Die Aufgabe besteht nun also noch darin, die Spalte so einzuschränken, dass der richtige Wert (mit richtigem Datum und richtiger Währung) für „Value“ in die Formel eingesetzt wird. Dies erfolgt über die 3 WHERE-Bedingungen. Die erste WHERE-Bedingung wird benötigt, da mehrere Tabellen bei FROM aufgelistet sind, damit wird das JOIN-Kriterium angegeben. Dann wird die Einschränkung der Währung vorgenommen. Um den Wert des richtigen Tages einzuschränken, werden zunächst mit *substring()* die Werte der Zeilen in die gleiche Form gebracht und dann ebenfalls verglichen. Übrig bleibt ein Wert, welcher dann für die Umrechnung genutzt wird. Die Vorgehensweise ist in Abbildung 17 dargestellt.

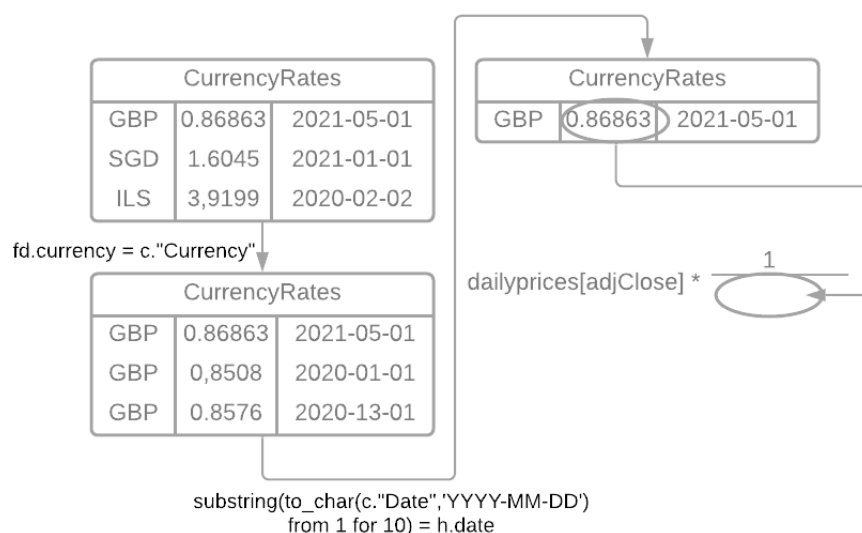


Abbildung 17: Vorgehen bei der Währungsumrechnung

5. Datenquellen im PowerQuery-Editor

5.1 Erstellen einer umfassenden Datumsdimension

Zunächst öffnet man im PowerBI-Fenster den Power Query-Editor und erstellt eine leere Abfrage, um darin später mithilfe von M-Befehlen eine dynamische Datumsdimension erstellen zu können. Nach der Benennung kann man über den Menüpunkt „Parameter verwalten“ einen neuen Parameter hinzufügen. Dort hat man dann die Möglichkeit, einen Parameter zu definieren, auf den man später innerhalb der M-Formeln zugreifen kann. Dadurch ist es möglich, durch späteres Verändern dieses einen Parameters, die gesamte Datumsdimension anzupassen. Mit dem folgenden M-Befehl hat man zunächst einmal die Möglichkeit, alle Daten des zuvor definierten Start-Datums bis zum heutigen Datum darzustellen.

= List.Dates(StartDate, Number.From(DateTime.LocalNow())-Number.From(StartDate),#duration(1,0,0,0))

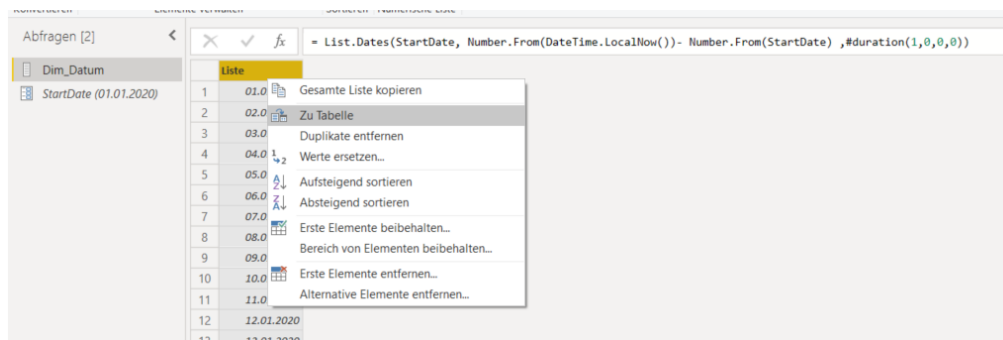


Abbildung 18: Umwandlung Liste zu Tabelle

Durch einen Rechtsklick auf die Tabelle kann man nun die erzeugte Liste in eine Tabelle umwandeln. Dies sieht man in Abbildung 18. Daraufhin sollte man die Spalte umbenennen und diese auch für PowerBI als Datumsspalte kennzeichnen. Das ermöglicht später im Bericht das Anwenden verschiedener Zeitintelligenzfunktionen, da diese nur funktionieren, wenn eine Spalte auch als Datumsspalte gekennzeichnet wurde.

Nun kann man durch die in PowerQuery vorhandenen Datumsfunktionen Jahres-, Quartals-, Monats-, Wochen- und Tagesspalten hinzufügen. Dafür wählt man den Reiter „Spalte hinzufügen“ aus und geht auf „Datum“. Dort findet man alle Optionen zum Auswählen. Die Bezeichnung der Wochentage erhält man durch eine M-Formel die man beim Einfügen einer „Benutzerdefinierten Spalte“ eingibt. Durch den Befehl:

Monat = Date.ToText([Datum], "MMMM", "en-EN")

erhält man die englische Bezeichnung der Monate, durch das Auswechseln von "MMMM" durch "dddd" die der Tage.

Den kompletten M-Code könnte man sich nun im erweiterten Editor kopieren und zu jedem beliebigen Power BI-Dokument hinzufügen und hätte sich somit den Aufwand gespart, die Datumsdimension neu zu erstellen.

Für den spezifischen Bericht des Projektes fehlt nun noch eine Angabe, die aussagt, ob der Tag ein Handelstag an der Stuttgarter Börse war, oder nicht. Diese Information kann man aus den Handelskalendern der jeweiligen Börsen ablesen. Der Handelskalender für das Jahr 2021 ist auf der Webseite der Stuttgarter Börse zu finden. Für den Handelskalender des Jahres 2020 suchte ich einfach im Internetarchiv „Wayback Machine“ nach der Vorjahresversion der URL.

Dafür erstellt man eine neue Web-Quelle in Power BI und greift auf die URL des Handelskalenders für die Jahre 2020 und 2021 zu. Dafür geht man auf „neue Quellen/Web“ und gibt dann die URL ein. Nun durchsucht Power BI die genannte Internetseite nach HTML-Tabellen, zeigt diese an und gibt dem User die Wahl, welche er davon in den Bericht übernehmen will. Da die Werte absolut sind, sich nicht mehr verändern werden und ebenfalls die Gefahr besteht, dass die URL in Zukunft nichtmehr funktionieren wird, kann man die Tabellen durch einen einfachen Rechtsklick von der Aktualisierung ausschließen. Dafür entfernt man einfach den Haken bei „In Berichtsaktualisierung einbeziehen“. Die beiden hinzugefügten Tabellen werden nun im nächsten Schritt zu einer zusammengefügt. Dafür geht man im Reiter „Start“ auf „Abfragen anfügen“ und fügt die beiden Tabellen zusammen. Um nachher alle handelsfreien Tage mit TRUE kennzeichnen zu können, fügt man an die zusammengefügte Abfrage noch eine Spalte mit TRUE-Werten hinzu. Dafür ist es am einfachsten, die „Spalte aus Beispielen“-Funktion im Reiter „Spalte hinzufügen“ zu benutzen. Dabei gibt man einige Spalten manuell ein, bis die KI ein Muster erkennt und dies fortführt. Das Ergebnis sieht man in folgender Abbildung 19.

	Feiertag	Datum	Börsenfeiertag
1	Neujahr	01.01.2020	TRUE
2	Karfreitag	10.04.2020	TRUE
3	Ostermontag	13.04.2020	TRUE
4	Tag der Arbeit	01.06.2020	TRUE
5	Pfingstmontag	01.06.2020	TRUE
6	Heilig Abend	24.12.2020	TRUE
7	1. Weihnachtsfeiertag	25.12.2020	TRUE
8	Silvester	31.12.2020	TRUE
9	Neujahr	01.01.2021	TRUE
10	Karfreitag	02.04.2021	TRUE
11	Ostermontag	05.04.2021	TRUE
12	Pfingstmontag	24.05.2021	TRUE
13	Heilig Abend	24.12.2021	TRUE
14	Silvester	31.12.2021	TRUE

Abbildung 19: Ergebnis der erzeugten Spalte „Börsenfeiertag“

Das Feld „Börsenfeiertag“ fügt man nun einfach über das Datum an unsere Datumsdimension an.

Da Wochenenden ebenfalls keine Handelstage sind, müssen wir auch diese kennzeichnen. Dafür erzeugen wir mit der simplen M-Formel:

```
if([TagText] = "Sunday" or [TagText] = "Saturday") then true else false
```

Eine weitere Spalte, die genau dann TRUE zurückgibt, wenn der Tag Samstag oder Sonntag ist. Um nun eine einheitliche Spalte zu erhalten, die aussagt, ob an dem spezifischen Tag gehandelt wurde, muss man nun noch eine Spalte einfügen, die die beiden Spalten „Börsenfeiertage“ und „Wochenende“ ODER- verknüpft. Dafür fügt man eine neue benutzerdefinierte Spalte mit der folgenden M-Formel hinzu:

```
= if([Börsenfeiertage] = true or [Wochenende] = true) then true else false
```


5.2 JSON-Index-Daten von Yahoo-Finance

1. Parameter mit Aktiensymbolen erstellen

Im PowerQuery-Editor kann man über die „Parameter verwalten“-Schaltfläche neue Parameter hinzufügen. Dort kann man dem Symbol einen Namen geben, seinen Typ definieren (Text) und ihm einen Wert zuweisen. Der Parameter ist dann eine Variable, auf die im gesamten PowerQuery-Editor zugegriffen werden kann. In der nachfolgenden Abbildung 20 sieht man den Parameter für den iShares-MSCI-World-Index-ETF. Auf diesen Parameter greift man nachher beim Zugriff auf die Yahoo-Finance API wieder zu.

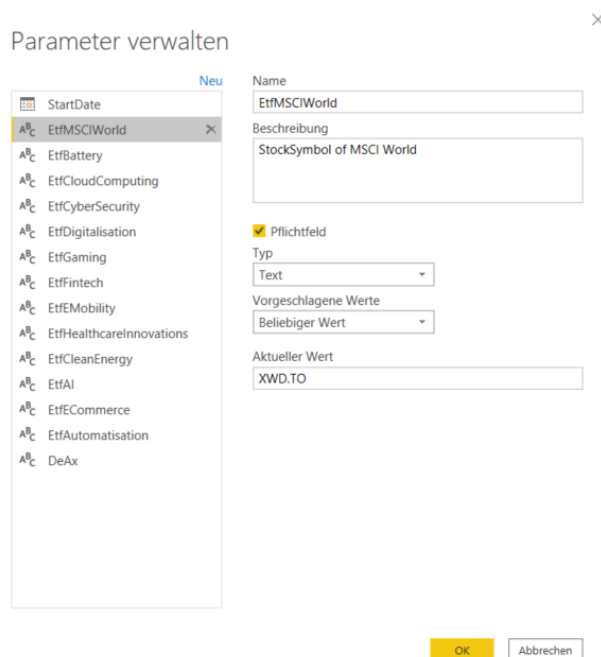


Abbildung 20: Parametereingabe

2. Abfrage erstellen

Als Datenquelle benutzt man wieder die Web-URL. Dazu klickt man auf „neue Quellen“ und wählt „Web“ aus. Im Auswahlmenü wählt man „Weitere“ aus, um die URL in mehrere Teile zu unterteilen.

So kann man später einfach die unterschiedlichen URL-Bestandteile verändern und dadurch auch andere Parameter einfügen. Die URL besteht aus folgenden Teilen:

1. Die Basis-URL die uns zu Yahoo-Finance und den benötigten Daten leitet
<https://query1.finance.yahoo.com/v8/finance/chart/>
2. Unser Aktiensymbol-Parameter (ABC-Symbol zu Parametersymbol ändern)

XWD.TO

3. Parameter, die die Daten noch weiter einschränken (Intervall, Zeitraum)?
range=2y&interval=1d
4. Abfrage bearbeiten

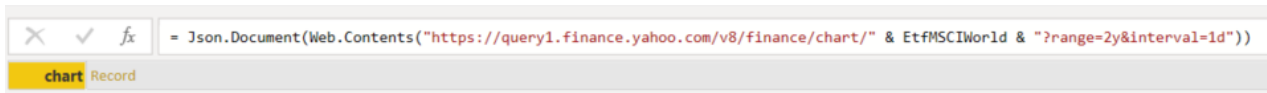


Abbildung 21: Übersicht der Records

Auf der Übersicht in Abbildung 21 kann man jetzt einen Drilldown starten, indem man folgender Hierarchie folgt:

- Chart:Record
- Result:List
- : Record

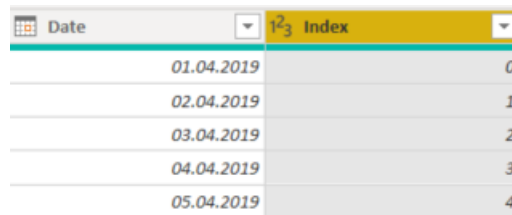
Hier sieht man, dass Datum und der dazugehörige Preis in zwei verschiedenen Listen abgespeichert sind. Die Überlegung zur Lösung dieses Problems ist, den beiden Listen eine Indexspalte hinzuzufügen, über die man sie dann später zusammenfügen kann.

Man fängt an, indem man in angewendete Schritte Fenster links von der Abfrage einen „Schritt einfügen nachher“ einfügt. Damit erstellt man einen neuen Schritt, welchen wir als „StartBranch“ betiteln.

Als erstes holt man sich die Daten aus der Liste. Dafür klickt man auf „timestamp:List“. Man sieht direkt, dass die Liste keine normalen Daten enthält. Die Daten sind im UNIX-Format gespeichert. Das UNIX-Datums-Format gibt die Anzahl der vergangenen Sekunden seit dem ersten Januar 1970 an. Man formatiert also erst einmal die Liste zu einer Tabelle um. Dafür geht man auf den „Transformieren“-Reiter und klickt auf „Zu Tabelle“. Als nächstes fügt man eine benutzerdefinierte Spalte mit folgender, in Abbildung 22 gezeigten, Formel hinzu:

Abbildung 22: M-Formel der benutzerdefinierten Spalte „Date“

Das jetzt erzeugte Datumsformat wird auch vom PowerQuery-Editor erkannt. Dazu ändert man den Typ einfach zu Datum und erhält das Datum im lesbaren DD/MM/YYYY-Format. Die „Column1“ kann man jetzt löschen. Um die Daten später mit den passenden ETF-Kursen verbinden zu können, fügt man jetzt noch die Index-Spalte hinzu. Über „Spalte hinzufügen/Indexspalte/Von 0“ wird nun ein Index eingefügt, der die Zeilen eindeutig identifiziert. Den Schritt nennt man nun noch im „angewendete Schritte“-Fenster in „EndBranchDate“ um. Das Endergebnis der Datumstabelle sieht man in Abbildung 23.



Date	Index
01.04.2019	0
02.04.2019	1
03.04.2019	2
04.04.2019	3
05.04.2019	4

Abbildung 23: Endergebnis der Datumstabelle

Um nun noch die ETF-Kurse aus der Liste zu extrahieren, macht man erneut einen Rechtsklick und fügt einen neuen „Schritt einfügen nachher“ hinzu. Wenn man sich die dazugehörige Formel anschaut, sagt diese „= EndBranchDate“. Das EndBranchDate tauscht man durch StartBranch aus, um wieder die Übersicht über die Listen zu bekommen. Dieses Mal führt man einen anderen Drilldown durch:

- Indicators: Record
- adjclose: List
- Record
- adjclose: List
-

Nun formatiert man die Liste, wie vorhin schon einmal, wieder zu einer Tabelle und fügt die Indexspalte von 0 hinzu. Am Ende nennt man den Schritt im „angewendete Schritte“-Fenster in „EndBranchPrice“ um.

4. Abfragen zusammenführen

Im Ergebnis hat man zwei voneinander unabhängige Tabellen, von denen jeweils nur eine angezeigt werden kann. Man geht auf „Start/Abfragen zusammenführen“ und fügt dort jeweils dieselbe Abfrage über ihre Indexspalte zusammen. Diese Zusammenführung ist auf Abbildung 24 erklärt.

Zusammenführen

Wählen Sie eine Tabelle und übereinstimmende Spalten aus, um eine zusammengeführte Tabelle zu erstellen.

EtfMSCIWorld?range=2y&interval=1d

Column1	Index
50,26730728	0
50,41295242	1
50,56830978	2
50,66540909	3
50,94699478	4

EtfMSCIWorld?range=2y&interval=1...

Column1	Index
50,26730728	0
50,41295242	1
50,56830978	2
50,66540909	3
50,94699478	4

Join-Art
Linker äußerer Join (alle aus erster, übereinstimmende...)

☐ Fuzzyübereinstimmungen zum Zusammenführen verwenden

> Optionen für Fuzzyübereinstimmung

✓ Die Auswahl stimmt mit 504 von 504 Zeilen der ersten Tabelle überein.

OK Abbrechen

Abbildung 24: Zusammenführen der Abfrage

Dies erzeugt die folgende Formel:

```
= Table.NestedJoin(EndBranchPrice, {"Index"}, EndBranchPrice, {"Index"}, "Hinzugefügter Index", JoinKind.LeftOuter)
```

Die Formel für das Zusammenführen der Spalten hat folgende Logik:

```
Table.nestedJoin(Tabelle1, {Key1}, Tabelle2, {Key2}, neuer Spaltenname, JoinTyp)
```

Wenn man sich den Output in der Formel ansieht, erkennt man, dass der Tabellename, falls die Tabelle mit sich selber gejoint wird, die Bearbeitungsschritte anzeigt. Man kann jetzt also einfach die zwei zuvor benannten Bearbeitungsschritte zusammenführen, indem man „EndBranchPrice“ durch „EndBranchDate“ ersetzt.

```
= Table.NestedJoin(EndBranchPrice, {"Index"}, EndBranchPrice, {"Index"}, "Hinzugefügter Index", JoinKind.LeftOuter)
```

```
= Table.NestedJoin(EndBranchPrice, {"Index"}, EndBranchDate, {"Index"}, "Hinzugefügter Index", JoinKind.LeftOuter)
```

Jetzt muss man nur noch auswählen, welche Spalte man von „EndBranchDate“ übernehmen will und hat anschließend die zusammengefügte Tabelle mit Datum und angepasstem Schlusswert. Die jeweiligen Spalten benennt man nun abschließend noch richtig.

5.3 Zugriff auf die Datenbank mit Power Query

Der Zugriff auf eine Datenbank über Power BI ist simpel. Über den „Daten abrufen“-Icon gelangt man direkt in ein Menü, welches eine Auswahl an verschiedenen Datenbanken liefert. Die genutzte PostgreSQL-Datenbank benötigt die beiden Angaben „Server“ und „Datenbank“. Da für das Projekt nur eine lokale Datenbank erstellt wurde, reicht als Serverangabe die Localhost IP-Adresse 127.0.0.1. Als Datenbank gibt man die „StockData“-Datenbank an. Nun hat man noch die Möglichkeit, zwischen Direct Query und dem Datenimport auszuwählen. Direct Query verbindet sich direkt mit den Daten. Das bedeutet, dass die Daten, die schlussendlich im Bericht angezeigt werden, aus Abfragen, die in Echtzeit auf der Datenbank ausgeführt werden, stammen. Die Daten werden nicht in Power BI importiert. Der für den Bericht benutzte Modus ist der Import Modus, wobei die Daten von der Datenbank direkt in Power BI importiert werden. Beim Erstellen oder Interagieren mit einer Visualisierung verwendet Power BI Desktop diese importierten Daten. Über erweiterte Optionen wird nun schlussendlich für jede Tabelle eine einfache „Select *“-Abfrage erstellt, sodass alle verfügbaren Tabellen als eigenständige Abfrage importiert werden. Ein Import am Beispiel der „dailyprices“-Tabelle ist in Abbildung 25 dargestellt.

PostgreSQL-Datenbank

Server
127.0.0.1

Datenbank
StockData

Datenkonnektivitätsmodus ⓘ
☒ Importieren
☐ DirectQuery

▲ Erweiterte Optionen
Befehlstimeout in Minuten (optional)

SQL-Anweisung (optional, erfordert Datenbank)
Select * from public.dailyprices

☒ Beziehungsspalten einbeziehen
☐ Unter Verwendung der vollständigen Hierarchie navigieren

OK Abbrechen

Abbildung 25: Import von „dailyprices“

6. Power BI Report Vorstellung

6.1 Überblick

Der Power BI Bericht soll dem Betrachter einen guten Überblick über die Geschehnisse am Aktienmarkt geben. Bedingt durch die Covid-19-Pandemie gab es vor allem im ersten Quartal 2020 einen deutlichen Abwärtstrend, gefolgt von einer starken Euphorie am Markt, die für viele Allzeithochs sorgte. Der Bericht soll mit Hilfe der Daten die Vorkommnisse zusammenfassend darstellen und dabei dem Betrachter mit Informationen über die Entwicklung der weltweiten Infektionslage sowie pandemiebedingten Schlagzeilen zusätzlich Hintergrundwissen liefern. Besonderer Wert wurde darauf gelegt, durch viele Interaktionsmöglichkeiten Informationen zu liefern, dabei aber nicht die Übersichtlichkeit des Berichtes durch zu viel Inhalt negativ zu beeinflussen. Dies ist auch der Grund, wieso viel mit Mouseover (in Power BI als Quick Info bezeichnet) Optionen sowie aufklappbaren Fenstern gearbeitet wurde. Fast jede der Visualisierungen beinhaltet Optionen zur Detailbetrachtung, sodass dem Verwender bei Interesse zusätzliche, detailliertere Informationen geliefert werden können. Mit Hilfe von Custom Visuals auf Basis von R wurden die von Power BI standardmäßig zur Verfügung gestellten Visualisierungen ergänzt.

Der Bericht lässt sich grob in folgende drei Abschnitte einteilen:

1. Startseite
2. Phasenbetrachtung der Markt- und Pandemieentwicklung
3. Detaillierte Aktien und ETF-Kursverlaufsanalyse

6.2 Startseite

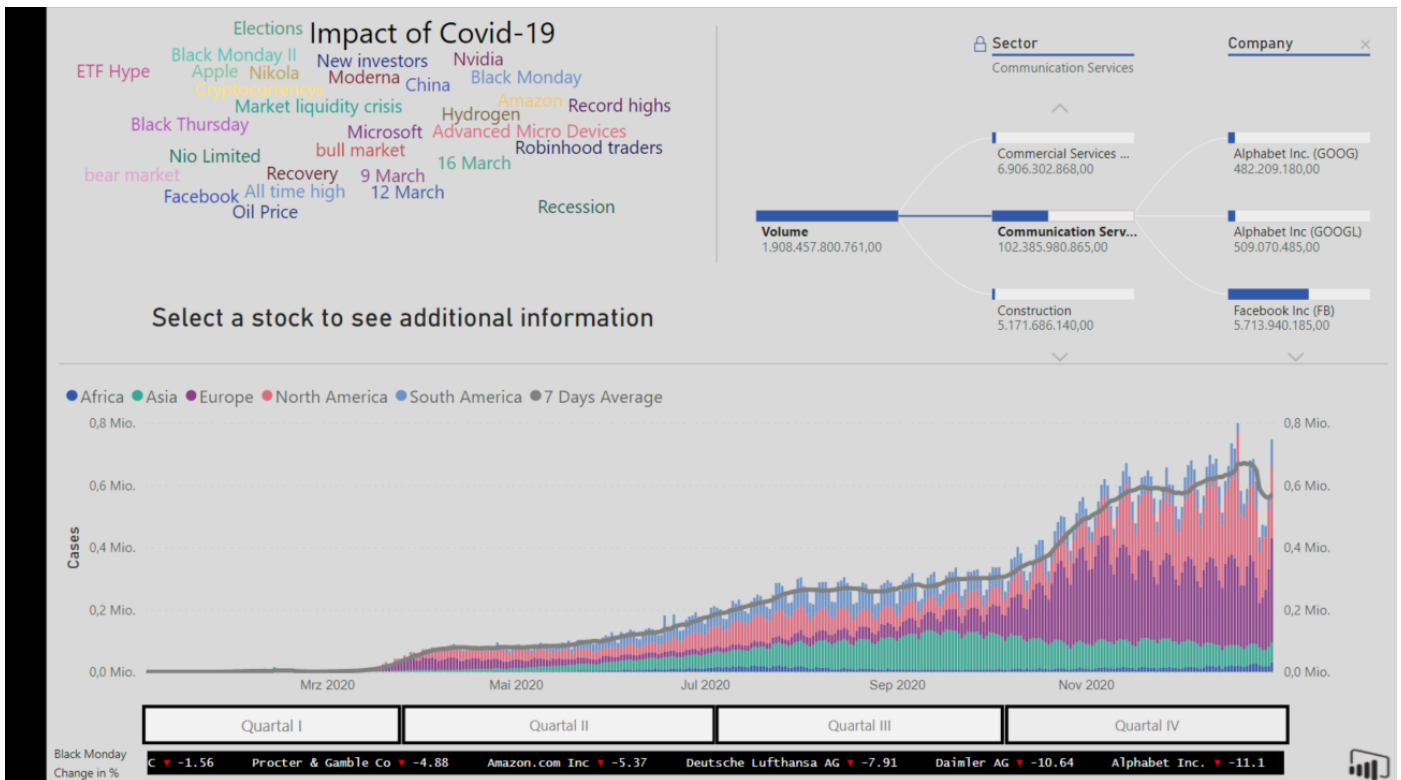


Abbildung 26: Report Startseite

Die Abbildung 25 zeigt die Startseite des Berichtes. Oben links stellt eine Word-Cloud die wichtigsten Vorkommnisse sowie wichtige Aktien dar, vor allem auch sogenannte „Corona Aktien“ oder auch „Stay at home“-Aktien. Diese wurden aus einer Liste der beliebtesten Aktien 2020 entnommen. Viele der Begriffe, welche sich nicht auf bestimmte Einzelaktien beziehen, hatten einen großen Einfluss und tauchen häufig in Artikeln rund um Covid-19 und den Aktienmarkt auf. Rechts davon befindet sich ein sogenannter Analysebaum. Dieser zeigt in der unausgeklappten Form nur das Gesamtvolumen des Jahres 2020 an. Er erlaubt aber das weitere Analysieren des Gesamtvolumens, auf der ersten Ebene aufgeteilt nach Sektoren, auf der zweiten Ebene das Handelsvolumen der zugehörigen Einzelaktien. Ein Mouseover über die Aktien zeigt den Preisverlauf der Aktie im Jahr 2020 an, ein Klick auf die Aktie gibt noch detaillierte Informationen. Der Schriftzug „Select a stock to see additional information“ verschwindet dann, und die in Abbildung 26 gezeigte Tabelle erscheint.

Stock	Close SoY	Lowest Close	Date	Highest Close	Date	Change Year
Facebook Inc (FB)	209,78 USD	146,01 USD	16.03.2020	303,91 USD	26.08.2020	30,21 %

Abbildung 27: Zusätzliche Information auf der Startseite

Das gestapelte Linien- und Säulendiagramm gibt dem Betrachter der Seite einen Überblick über die Infektionslage. Dieser Überblick ist nochmals aufgeteilt auf die verschiedenen Kontinente, welche durch verschiedene Farben abgegrenzt werden. Die Linie stellt den 7-Tages-Durchschnittswert der weltweiten Pandemielage dar. Ein Mouseover über die Kontinente gibt weitere Informationen über die Länder des Kontinents, welche an diesem Tag den höchsten Wert an Neuinfizierten gemeldet haben. Die beiden Y-Achsen sind manuell auf dieselbe Reichweite gestellt worden. Die quartalsweise Unterteilung der Zeiträume ist unter dem Schaubild dargestellt. Diese sind ebenfalls anklickbar, der Benutzer wird dadurch auf die entsprechende Seite des Zeitraums weitergeleitet. Um die Seite abzurunden, befindet sich unten noch eine benutzerdefinierte Visualisierung, welche die Entwicklung einiger Einzelaktien während des sogenannten „Black Monday“ aufzeigt. Dies soll bereits auf der Startseite einen kleinen Einblick in die historischen Verluste während des Corona-Crashes geben.

6.3 Phasenbetrachtung der Markt- und Pandemieentwicklung

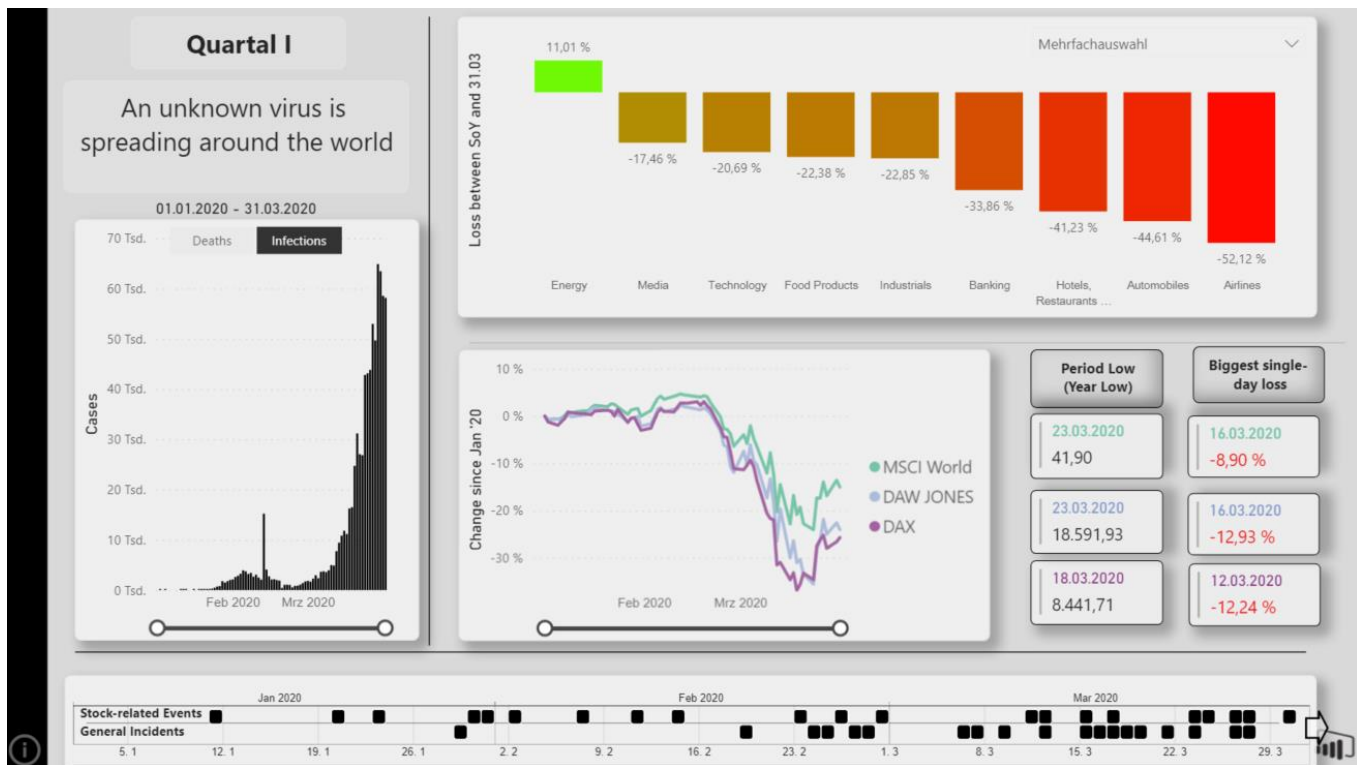


Abbildung 28: Phasenbetrachtung des Reports

Die Phasenbetrachtung des Marktes wurde in die vier Quartale des Jahres 2020 aufgeteilt. Die einzelnen Berichtsseiten sollen dem Betrachter einen guten Überblick über die Vorkommnisse sowie der allgemeinen Marktentwicklung geben. Auf den Seiten sind die Zeiträume noch weiter einschränkbar, so kann beispielsweise der Verlauf des Deutschen Aktienindexes ganz genau betrachtet werden. Links auf der Berichtseite ist eine Übersicht über die Entwicklung der Infektions- bzw. Todeszahlen dargestellt. Das Measure hierfür kann durch den vertikalen Slicer geändert werden, dafür wurde mit dem externen Tool „Tabular Editor“ eine sogenannte „Measure-Switch-Funktionalität“ eingeführt. Die zeitliche Ansicht kann ebenfalls mithilfe des Balkens unter dem Schaubild weiter eingeschränkt werden, sodass genauere Zeiträume untersucht werden können. Die rechte Seite des Berichtes soll Informationen über die verschiedenen Sektoren und deren Kursentwicklungen zwischen dem ersten Handelstag des Jahres und dem Ende des Quartales liefern. Die ausgewählten Sektoren lassen sich auch durch den Filter auf andere Sektoren erweitern, dessen Mehrfachauswahl ist beliebig anpassbar. Ein Mouseover zeigt detaillierte Daten zu den Entwicklungen bestimmter, ausgewählter Einzelaktien. Dort wird auch die prozentuale Kursentwicklung über den angegebenen Zeitraum angezeigt. Das Liniendiagramm in der Mitte der Berichtseite zeigt die Marktentwicklung nochmals, diesmal aber anhand von zwei wichtigen Indexen sowie einem ETF, welcher die Werteentwicklung von fast 1.600 Unternehmen aus 23 Ländern abbildet. Die in Power BI als „mehnteilige Zuordnung“ bezeichneten Flächen rechts des Liniendiagramms zeigen jeweils wichtige Informationen. Ein Mouseover über dem

Quickinfosymbol des Visualheads der „Period low“-Kachel bietet genaue Informationen zu dem Index/ETF sowie einen ganzjährigen Kursverlauf. Sichtbar ist dies in Abbildung 28.



Abbildung 29: Kursverlauf des MSCI World

Ein Klick auf die Textfelder „Period low“ sowie „Biggest single day loss“ ändern die angezeigten Informationen zu „Period High“ sowie „Biggest single day gain“.

Der „Zeitstrahl“ an der Unterseite des Berichtes ist ein Custom Visual, welches ursprünglich als Gant-Diagramm erstellt wurde. Da nach langer Suche keine passende Visualisierung gefunden werden konnte, um den Anforderungen gerecht zu werden, konnte dieses nach einigen Versuchen alle Zwecke erfüllen. Es zeigt auf einer Zeitachse Informationen zu verschiedenen, relevanten Vorkommnissen. Diese sind unterteilt in allgemeine, meist Covid-19 bezogene Zwischenfälle und Nachrichten sowie wichtige Informationen zu Geschehnissen auf dem Aktienmarkt. Ein Mouseover über eines der Kästchen lässt ein Feld erscheinen, welches nähere Informationen zu dem Vorkommnis und dem genauen Datum liefert. In Abbildung 29 ist dies am Beispiel des 14.02.2020 dargestellt.



Abbildung 30: Erweiterte Informationen durch Mouseover

Die anderen drei Quartale haben dieselbe Aufteilung und Logik und werden hier deshalb nicht weiter beschrieben.

6.4. Detaillierte Aktien- und Kursverlaufsanalyse

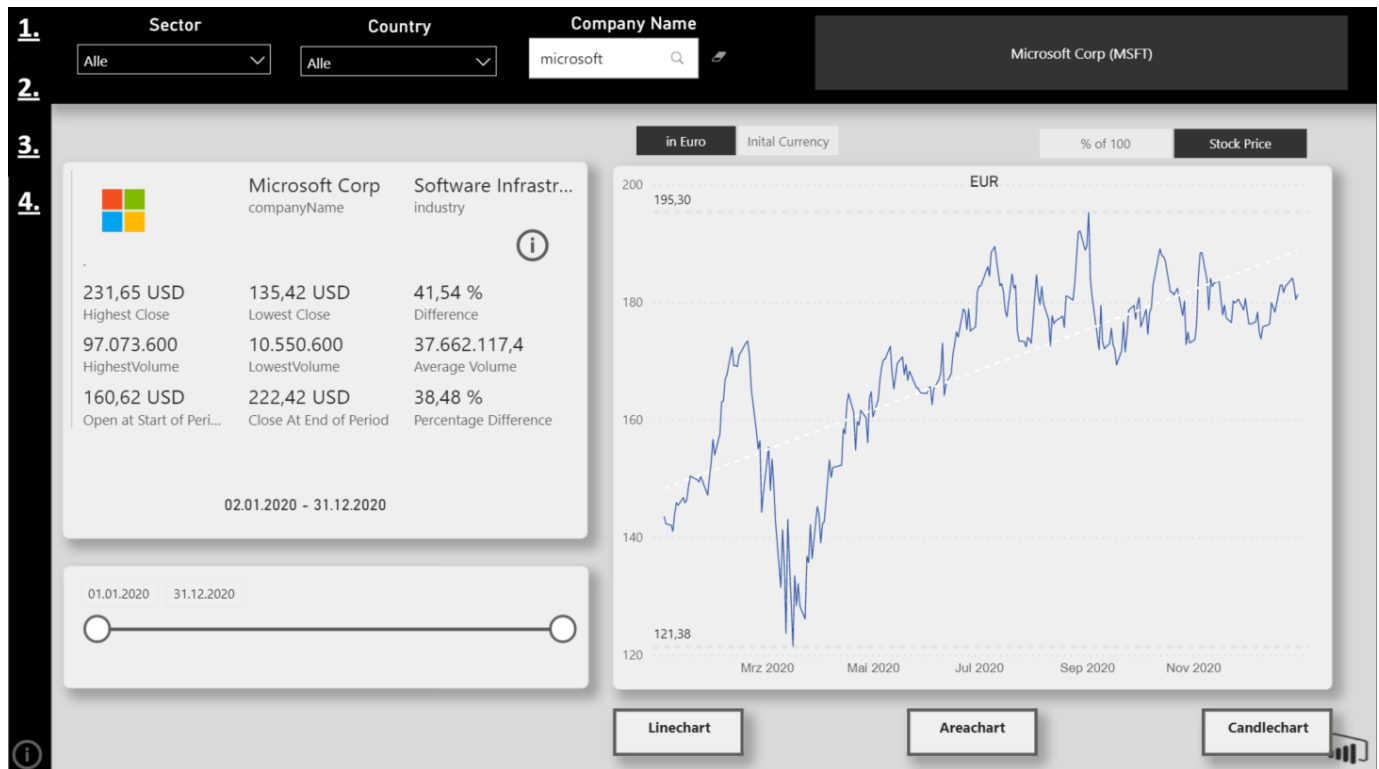


Abbildung 31: Einzelaktienansicht

Für detaillierte Einzelinformationen zu bestimmten Aktien wurde im Bericht eine Seite erstellt, die eine genaue Betrachtung aller verfügbaren Wertpapiere ermöglicht. So kann man genau nachvollziehen, wie die Entwicklung der eigenen Anlagen durch die Pandemie im Jahre 2020 beeinflusst wurde. Oben links befinden sich Textfelder mit darübergelegten Schaltflächen, die die in der Quartalsbetrachtung dargestellten Zeiträume mit einem Klick sichtbar machen. Durch den Sektor- und Landfilter kann man die auswählbaren Aktien einschränken, die konkrete Suche nach bestimmten Wertpapieren wird jedoch mit einem benutzerdefinierten Filter gelöst. Power BI bietet standardmäßig keine Textsuche an. Die verfügbaren Aktien werden dann mit Hilfe eines vertikalen Slicers angezeigt, sodass der Benutzer diese auswählen kann. Bei mehr als vier Übereinstimmungen kann durch Scrollen das richtige Ergebnis gesucht werden. Die Zeit kann entweder durch die direkte Anwahl der Quartale bestimmt werden, oder man benutzt den Datumsslicer. Durch diesen lässt sich der Zeitraum beliebig anpassen, alle verfügbaren Visualisierungen sind dadurch beeinflussbar. Die mehrteilige Zuordnung liefert wichtige Informationen zur ausgewählten Aktie über den ausgewählten Zeitraum. Über die Power BI Bild-URL-Lösung wird als erstes das jeweilige Logo des Unternehmens angezeigt. Die weiteren Informationen, bestehend aus verschiedenen Measures, sollen einen groben Überblick über Beliebtheit und Volatilität der ausgewählten Aktiengesellschaft geben. Alle monetären Kennzahlen werden mit der entsprechenden Währung angezeigt. Eine weitere Funktion, die sich hinter dem Informationsbutton auf der mehrteiligen Zuordnung verbirgt, ist der Gewinn- bzw. Verlustrechner.

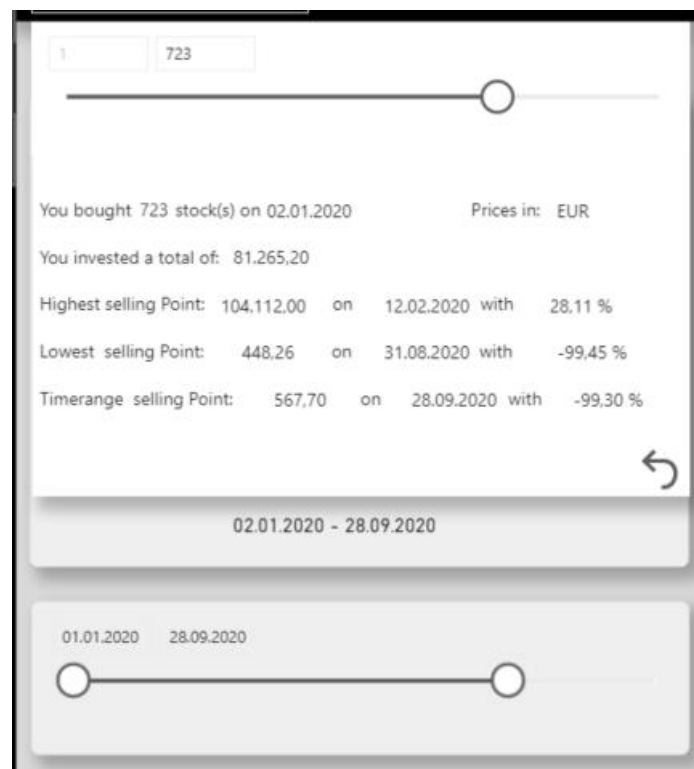


Abbildung 32: Gewinn- und Verlustrechner

Die in Abbildung 31 dargestellte Funktion bietet dem Benutzer die Möglichkeit, nachzuvollziehen, wie eine bestimmte Aktie über einen bestimmten Zeitraum abgeschnitten hat. Hier wird am Beispiel der Wirecard AG gezeigt, was für Chancen beziehungsweise Risiken ein Investment an einem bestimmten Tag geboten hätte. Die Anzahl der Anteile ist zwischen 1 und 999 frei wählbar, alle Angaben passen sich dadurch automatisch an, der Datumsslicer ist auch mit den hier vertretenen Measures synchronisiert. Man bekommt Informationen darüber, zu welchem Gesamtpreis (Tagesschlusskurs) die Transaktion durchgeführt worden wäre, wann der beste Verkaufszeitpunkt gewesen wäre, sowie die dort theoretisch erzielte Rendite und der Gesamtwert der Wertpapiere. Der niedrigste Verkaufszeitpunkt liefert dieselben Informationen, diesmal aber nicht das Best-Case-Szenario, sondern die Konditionen, die man am schlechtmöglichen Verkaufszeitpunkt bekommen hätte. Als letztes werden die Konditionen des An- und Verkaufes über den angegebenen Zeitraum berechnet. Die Gewinn- und Verlustansicht kann über den Zurückpfeil wieder geschlossen werden.

Oberhalb des Liniendiagramms befinden sich zwei Einstellungsmöglichkeiten für dieses. Hier kann zwischen der Darstellung des Aktienkurses in der ursprünglichen Währung und der Darstellung in Euro gewechselt werden. Die Umrechnung wurde auf Datenbankebene durchgeführt, die passende Umrechnung befindet sich also schon als Spalte in der Tabelle. Mithilfe der Measure-Switch-Funktion des externen Tools „Tabulator Editor“ lässt sich der

Wechsel realisieren. Eine weitere Einstellungsmöglichkeit ist der Wechsel zwischen dem monetären Aktienpreis und dem prozentualen Wert. Der prozentuale Wert bezieht sich immer auf den Eröffnungspreis zum Startzeitpunkt des im Datumsslicer ausgewählten Zeitraums. Dieser ist dynamisch anpassbar, sodass die prozentuale Entwicklung des Wertpapiers ab Anfang des ausgewählten Zeitraumes dargestellt wird. Das Liniendiagramm an sich stellt nun die Entwicklung in Abhängigkeit von dem, was der Benutzer für Einstellungen vorgenommen hat, dar. Drei leicht transparente Linien zeigen den Minimal- und Maximalwert sowie einen Trendwert.

Über die unter dem Chart befindlichen Testboxen lässt sich die Darstellung noch weiter verändern. Auswählbar sind dabei das standardmäßig eingestellte Liniendiagramm, ein Flächendiagramm sowie ein sogenanntes Kerzenchart.

Um Informationen zu dem ausgewählten Unternehmen zu erhalten, gibt es zwei verschiedene Lösungen. Zum einen über einen Klick auf das Informationssymbol in der linken unteren Ecke der Seite, zum anderen über die Quickinfoschaltfläche im Visualheader der mehrteiligen Zuordnung. Wichtige Informationen wie zum Beispiel der CEO eines Unternehmens oder die Anzahl der Mitarbeiter erhält man über das Informationssymbol, das Quickinfosymbol liefert eine textuelle Beschreibung des Unternehmens.

Die in Abbildung 33 dargestellte ETF Informationsseite hat größtenteils die gleichen Funktionen wie schon die Einzelaktienseite, enthält jedoch noch einige ETF-spezifische Darstellungen.

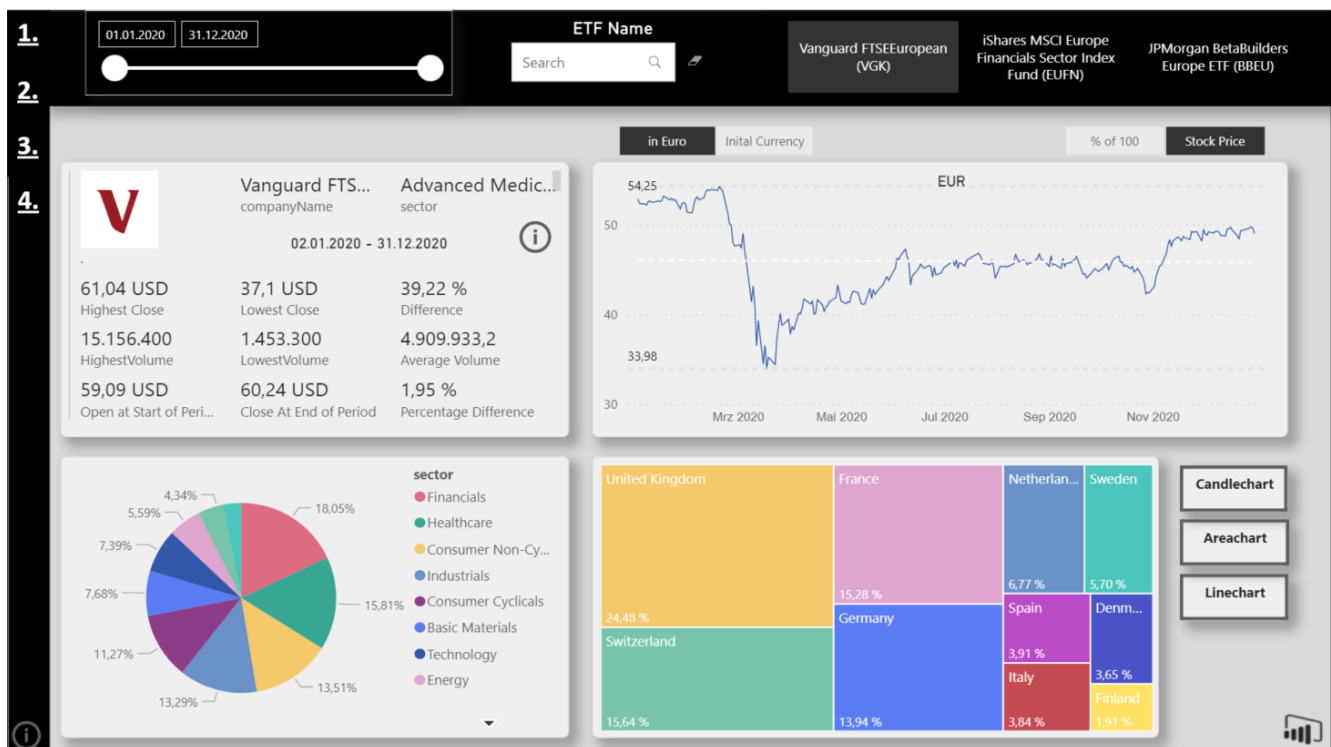


Abbildung 33: ETF Informationsseite

Dies sind zwei Darstellungen, die die Länderaufteilung sowie die Sektorenaufteilung des ausgewählten ETFs anzeigen. Die Länderaufteilung wurde durch eine Treemap-Visualisierung gelöst, welche abhängig von der prozentualen Aufteilung unterschiedliche Größen der Rechtecke anzeigt. In Abbildung 33 sieht man dadurch zum Beispiel, dass Großbritannien das am stärksten vertretene Land im „Vanguard FTSEEuropean“ ETF ist. Der prozentuale Wert wird nochmals gesondert angezeigt. Die Darstellung der Sektorenaufteilung wurde mit einem einfachen Kreisdiagramm gelöst.

Da nicht zu jedem ETF Daten über Sektoren- und Länderaufteilung verfügbar sind, jedoch nicht auf die Darstellung dieser verzichtet werden sollte, bleiben diese Schaubilder in einigen Fällen leer. Alle anderen Angaben sind selbstverständlich vollständig.

7. Gewonnene Erkenntnisse

Im Laufe der Projektarbeit hatte ich die Möglichkeit, mich mit vielen verschiedenen Aktivitäten zu beschäftigen und dabei eine Menge an neuen Erkenntnissen zu gewinnen. Konkret konnte ich mit zwei verschiedenen Programmiersprachen arbeiten und mich dabei auch intensiv mit der Schnittstelle zwischen Datenbank und Applikation beschäftigen.

Dabei habe ich alle Schritte aus dem ETL-Prozess anwenden können. Das Extrahieren von Daten über eine API-Schnittstelle, das Bearbeiten dieser Daten mit den Programmiersprachen Python und R sowie das Laden der Daten in die Datenbank. Die Anwendung der Programmiersprache Python, dessen Syntax ich extra für dieses Projekt erlernte, fiel mir mit der Zeit immer leichter. Die Funktionen und Möglichkeiten, die Python - insbesondere in Kombination mit den Bibliotheken NumPy sowie Pandas - bietet, verstärkten meine Intentionen, in Zukunft mit Daten arbeiten zu wollen. Durch die Bearbeitung der Daten mit RegEx konnte ich auch dort meinen Wissensstand festigen und vertiefen. Schlussendlich konnte ich beim Schreiben von Codes für den Austausch der Daten zwischen Applikation und Datenbank und den dabei verwendeten unterschiedlichen Vorgehensweisen neue Erkenntnisse in diesem Gebiet erlangen.

Während der Erstellung und Konfiguration einer eigenen PostgreSQL-Datenbank konnte ich eine Menge über Datenbanken lernen. Nicht nur die Erstellung an sich war sehr interessant, auch die Auswahl einer Datenbank, die für meine Bedürfnisse passend ist, zeigte für mich bisher unbekannte Kriterien und Spezifikationen.

Vor allem auch bei der Erstellung des Power BI Berichtes konnte ich durch das Anwenden von verschiedenen Tricks und Workarounds meine Fähigkeiten verbessern. Ein Bericht mit einer solch großen Datenmenge und derart vielen, verschiedenen Datenquellen war für mich neu. Das Verwenden der verschiedenen Quellen gab mir nochmals einen besseren Überblick, was alles mit Power BI möglich ist. Während meiner Arbeit an den Report Seiten stieß ich auf viele verschiedene Probleme, die aber alle durch gründliche Recherche und viele Versuche gelöst werden konnten. Meinen Umgang mit Power BI Measures und somit auch mit der Formelsprache DAX, sowie der in Power Query verwendeten Formelsprache M konnte ich deutlich verbessern. Auch der Umgang mit benutzerdefinierten Visualisierungen wird mir nach dem Projekt leichter fallen.

Zu erwähnen ist außerdem noch, dass ich während des Schreibens dieses Dokumentes viel über die Dokumentation eines Programmes und die damit verbundenen Schwierigkeiten gelernt habe.

Literaturverzeichnis

Slim Helu, Carlos (o. D.): The Top 25 Investing Quotes of All Time, Investopedia, [online] <https://www.investopedia.com/financial-edge/0511/the-top-17-investing-quotes-of-all-time.aspx> [abgerufen am 09.07.2021].

Financial Modeling Prep - FinancialModelingPrep (o. D.): Financial Modeling Prep, [online] <https://financialmodelingprep.com/developer> [abgerufen am 09.07.2021].

Ehrenwörtliche Erklärung

Name: Schwind

Vorname: Timo

Matrikel-Nr.:
759823

Studiengang: WKB

Hiermit versichere ich, Timo Schwind, dass ich die vorliegende Studienprojektarbeit mit dem Titel Datengewinnung für die Erstellung eines Microsoft Power BI - Berichtes zum Thema: „Wie hat die Covid-19 Pandemie den Aktienmarkt beeinflusst?“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Esslingen, 14.07.2021

Ort, Datum

Timo Schwind

Unterschrift