GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Philosophische Fakultät
Seminar für Allgemeine Sprachwissenschaft
Wintersemester 2018/2019
„Python & SQLite“
Dozenten: Prof. Dr. Eggers

Hausarbeit

GreekWebApp
Eine Python & SQLite Webanwendung für das Altgriechische

vorgelegt von:
Antonio Masotti
Matrikelnummer: 21568383
Allgemeine Sprachwissenschaft & Slavische Philologie
7. Fachsemester
E-mail: toniomasotti@gmail.com
am:
17. April 2019

1 Einführung

Das Altgriechische ist sicherlich eine der am meisten studierten bzw. untersuchten alten indogermanischen Sprachen. Und trotzdem fehlen bis jetzt viele digitale Tools, die das Leben von Studenten und Forschern deutlich vereinfachen könnten. Diese WebApp, entwickelt als Prüfungsleistung für den Kurs *SQLite & Python* von Prof. Dr. Eggerts an der Universität Göttingen (WiSe 2018/2019) versucht eine kleine von vielen großen Lücken zu schließen.

Aufbauend auf dem Internet Portal Perseus, das schon viele unabdingbare Hilfsmittel zum Studium des Altgriechischen anbietet, ergänzt diese App die schon vorhandenen Funktionen.

Perseus bietet z.B. die Möglichkeit einzelne Kapitel oder Abschnitte von Werken (u.a.) altgriechischer Autoren online zu lesen und stellt ein morphologisches Wörterbuch zur Bestimmung von einzelnen Wortformen zur Verfügung.

Was allerdings fehlt ist die Option ganze Texte auf einer Seite abzurufen, sie herunterzuladen, sowie die Möglichkeit alle Instanzen einer grammatischen Kategorie beispielweise *Aorist* oder *Substantiv im Genitiv* in einem Text abzufragen.

Der Download von ganzen Texten sind zusammen mit der Abfrage von morphologischen Formen in einem Text die wichtigsten Funktionen dieser Anwendung.

Die technischen Informationen werden in den nächsten Abschnitten etwas ausführlicher diskutiert. Hier reicht es zu sagen, dass die gesamte App das Potential von Python als eine der vollständigsten und flexibelsten Programmierungssprachen ausbeutet. Mit Python lassen sich nämlich nicht nur Desktop-Anwendungen und Skripts entwickeln, sondern auch sehr elegante und schnell angefertigte Webanwendungen. Die deutlichen Vorteile einer Webanwendung gegenüber traditionellen Desktop-Anwendungen haben zu einer Explosion der ersten geführt. Dies betrifft nicht nur die Anzahl der Webanwendungen, sondern auch die Tools und Frameworks, die in den letzten Jahren spezifisch dafür entwickelt wurden.

Unter diesen Vorteilen könnte man mindestens die folgenden auflisten:

- **Aktualisierung:** Desktopanwendungen müssen oft aktualisiert werden, was nicht selten eine neue Installation bedeutet (Daten sichern, alte Version löschen, eine neue ausführbare Datei herunterladen, neue Version installieren usw...). Dass in einem dieser Schritte etwas schief gehen kann, weiß jemand oft aus eigener Erfahrung. Bei Webanwendungen wird das Problem grundsätzlich gelöst und damit viel Mühe und Zeit gespart; jede Aktualisierung findet auf der Server-Seite statt und der Benutzter kann die App ununterbrochen verwenden. Bei jedem neuen Start wird die App neu geladen und ist damit immer auf dem aktuellsten Stand.

- **Kompatibilität:** Webanwendungen sind i.d.R. per default kompatibel mit den meisten Betriebssystemen und verlangen viel weniger Ressourcen als die traditionelle Anwendungen.

- **GUI (Graphic User Interface):** bei einer Desktop-Anwendung muss diese separat programmiert werden und nicht selten (obwohl nicht bei allen Programmierungssprachen) entstehen Kompatibilitätsprobleme. Bei Webanwendungen bieten die Browser schon einen Container für den Inhalt und es muss nur der eigene Stil der App programmiert werden. Dieses geschieht in der Regel mit sehr wenigen und sehr kurzen CSS Befehlen oder sogar mit einem einzigen Import aus Stilbibliotheken wie Bootstrap (dies ist auch der Fall bei der hier beschriebenen GreekWebApp).

- **Datensicherung:** Alle Daten, die im Umgang mit einer Webanwendung generiert werden, werden typischerweise in einer Datenbank auf dem Server gespeichert und können optional auch lokal heruntergeladen werden. Dies vermindert die Chance, dass Daten verloren gehen können, erheblich.

2 Installation und Start von GreekWebApp

Die Anwendung `GreekWebApp` ist im Grunde ein mittel-kleines Python-Programm, mit Ausnahme von einigen Teilen und Funktionen, die sich anderer Sprachen (Jinja2, HTML, JavaScript) bedienen. Da im Moment die App nicht veröffentlicht wurde, kann sie nur lokal laufen und wird wie ein normales Skript in Python gestartet.

Benötigt werden einige spezielle Python-Pakete, die in der Textdatei `requirements.txt` aufgelistet wurden[1]. Eine schnelle Installation aller benötigten Pakete erfolgt mit dem folgenden Befehl:

(1) `pip install -r requirements.txt`

Möchte man nicht alle Paketen installieren, wird der App alternativ eine eigene Windows `virtual environment` (Verzeichnis `venv`) beigefügt. Die virtuelle Umgebung kann - wie üblich unter Python Benutzern - mit (Ana)Conda oder direkt in einer IDE wie VisualStudio oder PyCharm geladen werden. Die mitgelieferte Version von Python ist die 3.6.6 (64bit Edition).

Nachdem alle *requirements* erfüllt sind bzw. die mitgelieferte virtuelle Umgebung aktiviert wurde, kann man die App einfach starten, indem man eine Konsole (Windows Eingabeaufforderung) im Verzeichnis `quellcode` öffnet und die Hauptdatei (`app.py`) startet. Dieses kann mit zwei - in unserem Fall gleichwertigen Befehlen - erfolgen:

(2) # im Verzeichnis „quellcode“
   a. **Windows:** python app.py
   b. **Linux:** python3 app.py

(3) # im Verzeichnis „quellcode“
   a. set FLASK_app = app.py[2]
   b. flask run

Der Befehl (2) startet die Hauptdatei der App, während der Befehl (3) den Server startet, der dann automatisch den Befehl in(2) abruft. In unserem Fall sind die zwei Befehle identisch, da die App nicht online ist und Server und Endpoint übereinstimmen.

Ab diesem Moment läuft die Anwendung auf der Adresse, die auf der Konsole angezeigt wird:

(4) http://127.0.0.1:5000/

oder gleichwertig:

(5) localhost:5000/

[1] Am wichtigsten sind allerdings nur zwei nicht-standard Python Pakete: SQLAlchemy (für die Verwaltung der Datenbank) und Flask (Framework für die WebApp). Beide können, wie alle andere auch, mit dem gewöhnlichen Befehl (i) installiert werden:

(i) `pip install name_des_pakets(=version)`

Die Installation auf Linux von diesen Pakete weicht von der Installation auf Windows ab (siehe z.B. die Dokumentation auf pythoncentral.io). Wenn man diese App auf einem Linux-Rechner starten möchte, wird eine virtuelle Umgebung nachdrücklich empfohlen.

[2] Auf Linux/MacOS ersetzen Sie „set FLASK_app“ mit „export FLASK_APP“. Mehr Infos auf der Flask-Homepage

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [15/Apr/2019 16:56:24] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [15/Apr/2019 16:56:25] "GET /favicon.ico HTTP/1.1" 404 -
```

Mit dem Start der Anwendung wird eine SQLite-Datenbank automatisch erzeugt und im Verzeichnis `common` gespeichert. Es reicht nun ein Browser[3] zu öffnen und eine der Adresse in (4) oder (5) abzurufen, um mit der App interagieren zu können. Die Zahl 5000 nach der Adresse bezeichnet den *Logical Port*, über den Flask aktuell läuft. Diese kann in der Datei `grund_einstellungen.py` geändert werden.

Auf der Konsole können alle Befehle zur SQLite-Datenbank sowie einige Ergebnisse der Python-Funktionen abgelesen werden (diese können auch ausgeschaltet werden, indem man die Variabel debug in `grund_einstellungen.py` auf „False“ setzt.).

Die Anwendung kann jederzeit mit der Tastemkombination `Strg + C` in der Konsole geschlossen werden.

3 Architektur der WebApp

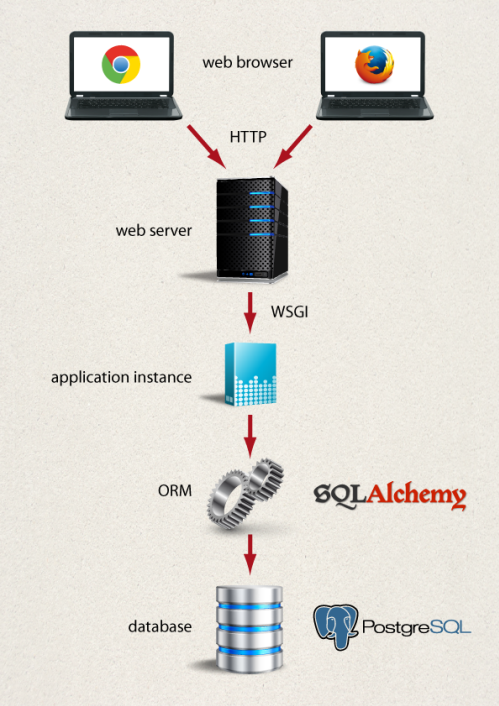

Abbildung 1: Struktur einer Webanwendung. Quelle: vertabelo.com (Abgerufen am 12.04.2019)

Die gesamte Webanwendung besteht aus wenigen Komponenten und wird fast vollständig von Python angetrieben. Diese Bausteine sind - wie in Fig. 1 abgebildet - die folgenden:

1. eine **Datenbank**: GreekWebApp verwendet eine SQLite, aber ist im Prinzip dem Entwickler überlassen welche relationale oder nicht-relationale Datenbank er implementieren möchte.

2. **SQLAlchemy**: das Pythonpaket für die Verwaltung von Datenbanken

[3] Getestet wurde die App mit Chrome, Firefox, Safari, Opera und Vivaldi.

3. **Flask**: das Motor der Webanwendung.

4. **Jinja2** : eine Templatesprache, die uns erlaubt PythonCode, -Variabeln, Schleifen und Statement direkt auf die HTML Seite zu überführen.

5. **WebServer**: im Fall der hier beschriebenen GreekApp wurde dieser Schritt, obwohl einfach, nicht implementiert. Möchte man die App veröffentlichen und allen zugänglich machen, würde man sicherlich einen WebServer wie Cherooke gebrauchen. Auf der Programmierungsseite bedeutet dies nur ein paar Zeilen Code mehr für das *Binding* der `app.py` mit den Servereinstellungen.

6. **Browser**: heutezutage ist mindestens ein Browser wie Chrome, Edge, Firefox u.ä. auf allen Rechnern verfügbar. Der Browser stellt den Container für die graphische Oberfläche der Webanwendung bereit. Direkt vom Browser werden die HTML Dateien, die stylistische CSS Einstellungen und die JS (bzw. JQuery) Scripts geladen. Bei GreekApp wurde auf zusätzliche, rein stilistische JavaScripts verzichtet. Eine Ausnahme dafür bilden die Bootstrap Scripts und zwei sehr kleine JS Funktionen für die Suchmaske und die Visualisierung der HTML Tabellen.

Flask Flask ist zusammen mit Django eines der zwei berühmtesten Microframeworks zur Bereitstellung von Webanwendungen mit Python.

Flask erlaubt dem Programmierer fast ausschließlich mit Python Code und wenigen Zeilen HTML eine vollständige Webanwendung in wenigen klaren Schritten bereitzustellen.

Seine Hauptfunktion ist es, als Bindeglied zwischen HTML-Dateien und Python-Funktionen zu agieren. Die ersten sind alle standardmäßig in dem Verzeichnis `templates` gespeichert, die letzten kommen aus den verschiedenen Modulen, die in der Hauptdatei importiert werden.

Flask nutzt für dieses Zweck eine sehr nützliche und flexible Pythonfunktion: die Dekoratoren. Typische Flask-Befehle sehen wie im folgenden Beispiel aus:

```
@app.route('/URL_einer_Seite_der_Anwendung')
def meine_funktion(parameter):
   # etwas wird gemacht
   return render_template('Datei.html', variabel1 = wert1 ...)
```

`@app.route()` ist der Dekorator, der die URL der abzurufenden Seite definiert. Jeder HTML-Seite wird auf diese Weise eine Python-Funktion zur Erzeugung des Inhalts und eine Adresse (mit der Dekorator) zugewiesen. Der Befehl `render_template` lädt zum Schluss die Seite auf. Innerhalb des `return`-Statement ist es auch möglich, bestimmte Variablen der Seite direkt zuzuschicken.

Auf diese Weise werden die einzelne Seiten gebildet und mit Inhalt befüllt. Eine andere wichtige Flask Methode ist `url_for` mit der eine Verknüpfung mit einer Pythonfunktion hergestellt wird:

```
from flask import url_for

@app.route('/Adresse')
def meine_Funktion():
   link = url_for('meine_zweite_funktion', variabel1 = wert1 ...)
   return render_template('datei.html', button = link)
```

In dem fiktiven Beispiel hier oben, wird eine Variabel `link` innerhalb einer Funktion erzeugt. Diese Variabel ist ein Link zu einer HTML-Seite, die die Funktion `meine_zweite_funktion` verwendet (genau wie die Seite `datei.html` die Funktion `meine_Funktion` im Beispiel.).

SQLAlchemy

> „*SQLAlchemy is a library used to interact with a wide variety of databases. It enables you to create data models and queries in a manner that feels like normal Python classes and statements. Created by Mike Bayer in 2005, SQLAlchemy is used by many companies great and small, and is considered by many to be the de facto way of working with relational databases in Python.*
>
> *It can be used to connect to most common databases such as Postgres, MySQL, SQLite, Oracle, and many others. It also provides a way to add support for other relational databases as well. Amazon Redshift, which uses a custom dialect of PostgreSQL, is a great example of database support added by the community.*“
> (Myers, J., & Copeland, R. (2015). Essential SQLAlchemy: Mapping Python to Databases.S.2)

Bei SQLAlchemy handelt es sich um eine komplexe Python Library, die viele andere - wie z.B. `sqlite3` - enthält.

SQLAlchemy erlaubt dem Entwickler, eine oder mehrere Datenbanken zu verwalten, ohne sich dabei um die Unterschiede zwischen den SQL-Dialekten oder anderen Besonderheiten der gewählten Datenbank kümmern zu müssen.

Alle SQL-Queries und Befehle werden vom Entwickler dem Programm in der Form gewöhnlicher Python Befehle übergeben. SQLAlchemy erledigt dann selbständig die nötigen Übersetzungen und Kommunikationsschritte mit den Datenbanken.

Möchte man z.B. nach dem Autor „Plato“ in der Tabelle der Autoren suchen, wird der Befehl in SQLAlchemy so aussehen:

```
Plato = Autoren.query.filter(name="Plato")
```

Eine unscharfe Suche nach allen Autoren, deren Name mit „P“ anfängt, würde dagegen wie folgt aussehen:

```
P = Autoren.query.filter(Autoren.name.like('P%')).all()
# das Ergebnis ist eine Liste mit allen gefundenen Tabellenzeilen.
```

SQLAlchemy nutzt die Objektorientierte Programmierung in Python. Die Tabellen in der SQLite Datenbank werden als Klassen behandelt und die einzelnen Zeilen als Instanzen dieser Klassen. Diese Strategie wird am besten mit einem Beispiel (die Tabelle „Autoren“ in unserer Datenbank) veranschaulicht. Der folgende Code zeigt die Definition der Tabelle „Autoren“ in der Datenbank:

```
import db # die initialisierte Datenbank muss erstmal importiert werden

# jede Klasse entspricht einer Tabelle in der Datenbank
class Werken(db.Model):

   # Name der Tabelle (optional, wenn nicht angegeben wird den Name der Klasse
      verwendet)
```

```
    __tablename__ = "Werke"

    # Attributen der Klasse (= Spalten in der Datenbank)

    id_werk = db.Column(db.Integer, primary_key=True) # das ID ist das
        PrimaryKey der Tabelle
    # autor wird mit dem id des Autors in der Tabelle "Autoren" befüllt. Dieses
        ID ist in "Werken" ein Foreign Key zur Verbindung der beiden Tabellen.
    autor = db.Column(db.Integer, db.ForeignKey('Autoren.id_autor'))

    # Titel des Werks
    titel = db.Column(db.Text, nullable=False)

    # Link zum Originaltext
    titel_link = db.Column(db.Text)

    # Link zur groben Wortlisten (auf Perseus)
    wortliste_link = db.Column(db.Text)

    # Constraint für die Verbindung der Tabellen "Autoren" mit dr Tabelle
        "Werken"
    verfasser = db.relationship('Autoren', backref="autor", uselist=False)

    # Magic function zur Initialisierung der Klasse
    # Hier wird festgelegt, welche Parameter benötigt werden, um eine Instanz
        der Klasse zu bilden.
    # Die Parameter, die von außen kommen (Benutzer oder Python-Funktionen)
        werden als Attribute der Zeilen in der Tabelle gespeichert.
    def __init__(self, autor, titel,titel_link,wortliste_link):
        self.autor = autor
        self.titel = titel
        self.titel_link = titel_link
        self.wortliste_link = wortliste_link

    # Magic function für die Repräsentation der Klasseninstanz.
    # Wenn der Benutzer nach einem Werk fragt, wird dieses so ausgedrückt wie
        in der folgenden Funktion festgelegt.
    # Sie werden alle in der Form z.B. "[0] Republik von Aristoteles" geprintet.
    def __repr__(self):
        return f"[{self.id_werk}] {self.titel} von
            {Autoren.query.get(self.autor).name}"
```

Auf einer sehr ähnlichen Art funktioniert die Herstellung von HTML-Tabellen (siehe Kommentare in den Dateien). In diesem Fall werden wir zwei Klassen haben: eine für die einzelnen Zeilen der Tabelle, mit Informationen darüber, wie die Zellen zu füllen sind und eine andere, allgemeinere Klasse, die diese Zeilen zusammenfügt und einen HTML-Code für die Tabelle generiert.

4 Einige wichtige Dateien

In diesem Abschnitt wird die Struktur der Webanwendung kurz dargestellt und die Funktionalitäten der wichtigsten Seiten erläutert. Es wird hier darauf hingewiesen, dass jede Funktion und jeder Befehl innerhalb der einzelnen Dateien ebenso ausführlich kommentiert wird.

4.1 Strukturbaum der Dateien

```
GreekWebapp
├── quellcode
│   ├── __pycache_
│   ├── common
│   │   ├── __pycache_
│   │   ├── Autoren
│   │   │   ├── __init__.py
│   │   │   └── autoren_zeilen.py
│   │   ├── static
│   │   │   ├── assets
│   │   │   ├── css
│   │   │   │   └── meineEinstellungen.css
│   │   │   ├── img
│   │   │   └── tex
│   │   ├── templates
│   │   │   ├── alleautoren.html
│   │   │   ├── allenwerke_table.html
│   │   │   ├── allewerke.html
│   │   │   ├── autorenList.html
│   │   │   ├── basic.html
│   │   │   ├── navbar.html
│   │   │   ├── startseite.html
│   │   │   └── ...
│   │   ├── __init__.py
│   │   ├── formulare.py
│   │   ├── greekData.sqlite
│   │   ├── grund_einstellungen.py
│   │   └── mischmasch.py
│   ├── migrations
│   ├── models
│   └── app.py
├── venv
└── requirements.txt
```

Die Grundstruktur von GreekWebApp weicht nicht vom Standard anderer bekannten Webanwendungen ab.

Das Verzeichnis `venv` enthält eine vollständige Python-Installation (v. 3.6.6) mit allen Paketen, die für die Webanwendung benötigt werden. Dies ist in dem Moment notwendig, in dem man ein Anwendung auf einem Server hochlädt. Zum aktuellen Zeitpunkt ist die virtuelle Umgebung nur eine schnelle Lösung, um nicht alle nötigen Pakete neu installieren zu müssen, wenn die Anwendung auf einen neuen Rechner aufgespielt wird.

Am Wichtigsten ist das Verzeichnis **quellcode**, das dem englischen Standard `source` entspricht. Hierunter sind alle Dateien enthalten, deren sich die Webanwendung bedient, inklusive `app.py`, die die Hauptdatei darstellt.

Das Verzeichnis **common** enthält alle Funktionen, die den Code in der Hauptdatei unterstützen.

Das Verzeichnis **static** enthält alle Dateien, die für die HTML-Seiten eine Rolle spielen. Dort werden die griechischen Texte heruntergeladen und gespeichert (Verzeichnis `assets`), alle Bilder (`img`), TeX-Dateiem (`tex`) und Stileinstellungen (`css`).

Das Verzeichnis **templates** enthält alle HTML-Seiten der Webanwendung. Diese werden vom Browser abgerufen, können alledings auch mit einem Texteditor (wie Notepad oder am besten SublimeText) geöffnet werden. Der Code innerhalb der einzelnen Seite ist kommentiert und die verwendeten Funktionen erläutert.

Das Verzeichnis **migrations** wird aktuell nicht verwendet. Es müsste allerdings angelegt werden, denn hier werden bei einer auf einem Server laufende App alle Backups der Datenbank gespeichert sowie ein Log der Änderungen. Das Verzeichnis `migrations` wird automatisch von den Python-Paketen `alembic` und `Migrate` befüllt.

Das Verzeichnis **models** enthält die Python-Klassen, die die Tabelle der SQLite Datenbank definieren, zusammen mit den Klassen für die Formulare und die HTML-Tabelle zur Visualisierung der gespeicherten Daten.

Zum Schluss enthält die Datei `mischmasch.py`, wie der Name sagt, alle Nebenfunktionen, um den Code der Hauptdateien lesbarer zu machen. Die Datei `grund_einstellungen.py` enthält die Definition der Datenbank und die erste Verknüpfung mit dieser.

4.2 Die wichtigsten Seiten der Anwendung

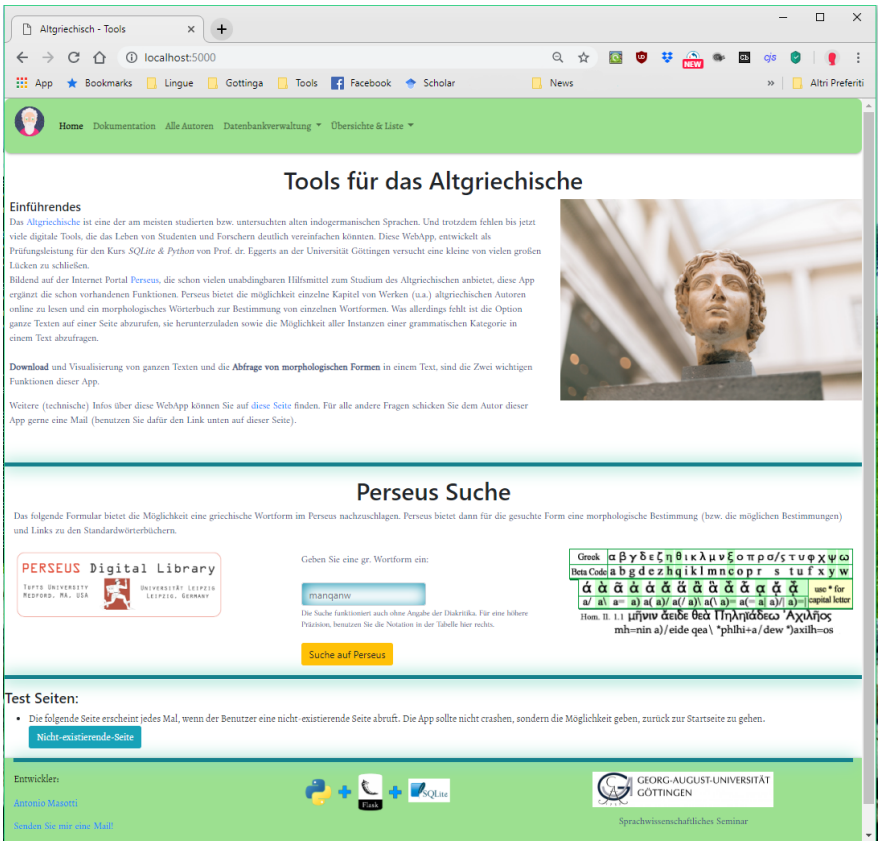

Abbildung 2: Startseite der App auf der Adresse `localhost:5000`

Startseite (http://127.0.0.1:5000/) : auf der Startseite wird die Anwendung präsentiert. Von hier ist es möglich, alle anderen Seiten zu erreichen (dank der Links auf der Navbar), ein Wort/eine Wortform in dem Perseus Wörterbuch nachzuschlagen.

Alle Autoren (http://127.0.0.1:5000/autorenList) : zeigt eine Liste aller auf Perseus verfügbaren griechischen Autoren, sowie die Links zu deren Werken in der Datenbank Perseus under PhiloLogic.

Werksuche (http://127.0.0.1:5000/zeigWerken) : Diese Seite wird von verschiedenen Python-Funktionen abgerufen. Wenn die Seite abgerufen wird und die Variabel `autor_name` nicht leer ist, zeigt die Seite eine Liste der Werke eines bestimmten Autors an. Wenn die Variabel leer ist, wird ein Formular gezeigt, auf dem es möglich ist einen Autor zu wählen. Die Liste der Werke eines Autors enthält auch Links zu den griechischen Texten und Wortliste. Die Texte können heruntergeladen oder im Browser gelesen werden und eine Wortliste kann generiert werden.

Alle Informationen aus externen Webseiten werden mithilfe des Webscraping Python-Paket BeautifulSoup4 abgelesen und verarbeitet. Als HTML-Parser wurde das Python-Paket `lxml` verwendet.

Es sei hier darauf hingewiesen, dass die Links sehr zeitaufwendige Python-Funktionen starten (bis 40-50 Min.). Die einzelnen Kapitel der griechischen Texte sind auf Perseus separat gespeichert. Die Funktion `get_texte` findet sie im Netz und fügt sie alle in eine Textdatei zusammen. Für die Generierung der Wortliste werden mehrere Schleife aktiviert, die jede Wortform in den

Perseus-Wörterbüchern suchen, die nötigen Informationen extrahieren und daraus eine SQLite-Tabelle mit den gewonnenen Informationen befüllen.

Wenn die Funktionen gestartet wurden, wird auf der Konsole den Fortschrittsbericht ausgedruckt.

Suche nach bestimmten morphologischen Formen (http://127.0.0.1:5000/sqlForm) : Die Seite ist im Moment noch sehr einfach. Es ist möglich, ein kleines Suchformular auszufüllen, um gezielt nach morphologischen Kategorien zu suchen. Das Formular startet eine SQL-Query, deren Ergebnisse in einer HTML Tabelle gezeigt werden. Die Seite verfügt darüber hinaus über 3 Buttons, mit denen es möglich ist, alle gespeicherten Wortformen zu visualisieren, neue Werken zu untersuchen oder die Datenbank zu reinigen (Tabelle löschen).

Gespeicherte Daten : Alle Daten in der Datenbank können abgerufen und visualisiert werden. Dafür sind drei getrennte Seiten vorgesehen:

1. http://127.0.0.1:5000/allewerke : visualisiert alle gespeicherten Werke
2. http://127.0.0.1:5000/alleautoren : visualisiert alle gespeicherten Autoren
3. http://127.0.0.1:5000/gespeicherteWortformen : visualisiert alle gespeicherten Wortformen und gibt dem Benutzer die Möglichkeit, die Liste als csv oder als pdf herunterzuladen.

5 To do

Natürlich gibt es kaum eine Obergrenze für die Verbesserungsmöglichkeiten einer Webanwendung. Funktionen zur Berechnung von Statistiken in den griechischen Texten, sowie verbesserte Queries wären zweifellos denkbar. Allerdings muss hier darauf hingewiesen werden, dass es einige Funktionen gibt, die sicherlich eine Verbesserung benötigten. Zwei davon sind besonders auffällig:

- Lesen von Texten im Browser. Das Problem dabei ist, dass die Texte auf Perseus nicht einheitlich sind. Für eine korrekte bzw. angenehme Visualisierung wären spezifische Funktionen (mit spezifischen RegExp) nötig.
- Geschwindigkeit bei dem Herunterladen der Texte bzw. Erzeugung und Verarbeitung von Wortlisten. Aktuell basiert diese Funktion auf einer langen Schleife, die jede Wortform eines Werkes im Internet (im Perseus Wörterbuch) sucht, die gefundene Webseite liest, die nötigen Infos extrahiert, alle Infos formatiert und in der Datenbank speichert. Danach wird eine Tabelle gezeigt. Wenn man bedenkt, dass ein Werk bis zu 100.000 Wortformen enthält, versteht man, dass diese Funktion nicht selten 30-40 min. Zeit benötigt. Das ist natürlich eine kurze Zeit in Vergleich zu den Wochen, die man brauchen würde um manuell eine solche Tabelle zu erstellen, aber es bleibt dennoch eine sehr lange Zeit für die modernen technologischen Standards. Eine Optimierung der Funktion war bis jetzt noch nicht möglich.