Entwicklung einer Administrationsoberfläche für Datenbanken in Python-Webanwendungen

Bachelorarbeit

von

Alexej Elrich

aus

Lenki

vorgelegt am

Lehrstuhl für Rechnernetze
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

Juni 2020

Betreuer:
Markus Brenneis

Zusammenfassung

Im Rahmen dieser Abschlussarbeit sollte ich eine Anwendung entwickeln, die das dialogbasierte Argumentationssystem der Heinrich-Heine-Universität um eine Administrationsoberfläche erweitert. Dabei sollte die Administrationsoberfläche auch mit weiteren Python-Webanwendungen kompatibel sein, die auf *Pyramid* und *SQLAlchemy* basieren. Mithilfe der Anwendung sollte es möglich sein, aus einem Webbrowser heraus Datenbanktabellen zu bearbeiten. Mein persönliches Ziel war es dabei eine Anwendung zu entwickeln, die möglichst einfach installiert und verwendet werden kann.

Ich habe mich gegen die Entwicklung einer gewöhnlichen Webanwendung entschieden. Stattdessen habe ich eine Erweiterung für *Pyramid*-Webanwendungen entwickelt. Der Unterschied zwischen einer gewöhnlichen Anwendung und einer Erweiterung ist, dass eine Erweiterung in eine Hauptanwendung integriert werden muss. Dabei kann die Erweiterung nicht ohne die Hauptanwendung gestartet werden. Für die Kommunikation mit der Datenbank verwendet die Erweiterung *SQLAlchemy*. So konnte ich die Kompatibilität zwischen meiner Anwendung *GADI* und *Pyramid*-Webanwendungen, die *SQLAlchemy* verwenden, gewährleisten. Bei der Entwicklung musste ich besonderes darauf achten, nicht zu viele Informationen vom Anwender zu verlangen. Dies war nötig, um die Installation und Verwendung der Anwendung so einfach wie möglich zu gestalten.

Dieser Ansatz hat zu erschwerten Entwicklungsbedingungen geführt. Die größten Probleme entstanden bei der Kommunikation zwischen *GADI* und der Datenbank der Hauptanwendung. *GADI* konnte durch den eingeschränkten Zugriff auf die Daten der Hauptanwendung nicht auf die Datenbankmodelle zugreifen. Somit mussten jegliche Informationen über die einzelnen Tabellen aus den Metadaten der Datenbank ausgelesen werden. Dies führte zum Metadaten-Problem. Dieses konnte jedoch gelöst werden.

Das Ziel war es eine mit *Pyramid* und *SQLAlchemy* kompatible Administrationsoberfläche zu entwickeln. Die Umsetzung dieses Ziels war erfolgreich. *GADI* kann mit sämtlichen *Pyramid*-Webanwendungen verwendet werden. Dabei ist sowohl die Installation, als auch die Verwendung von *GADI* einfach gestaltet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Kontext der Arbeit

Im Rahmen dieser Abschlussarbeit soll eine Anwendung entwickelt werden, die das dialogbasierte Argumentationssystem der Heinrich-Heine-Universität um eine Administrationsoberfläche erweitert. Bei dem dialogbasierten Argumentationssystem, kurz *D-BAS*, handelt es sich um eine Webanwendung. Diese Webanwendung soll dabei helfen, Online Argumentationen zu vereinfachen. Gängige Online Argumentationsverfahren, wie Pro-und-Contra-Listen oder Argumentationskarten, skalieren entweder schlecht mit hohen Teilnehmerzahlen oder sind für neue Benutzer unzugänglich. *D-BAS* löst viele dieser Probleme, eignet sich jedoch nicht für alle Argumentationen. Die Teilnehmer von *D-BAS* stehen in einem zeitversetzten Dialog. So werden sie Schritt für Schritt mit verschiedenen Argumenten konfrontiert und können dazu Stellung beziehen. Anhand der Reaktionen der Teilnehmer auf die verschiedenen Argumente werden Argumente ausgewählt, mit denen zukünftige Benutzer konfrontiert werden. So entsteht ein übersichtliches und gut skalierendes Argumentationsverfahren [Met].

Die Entwicklung an *D-BAS* ist weit fortgeschritten und es wurden bereits erste Online Argumentationen durchgeführt. Der Anwendung fehlt es jedoch noch an einigen Funktionen. So fehlt *D-BAS* eine Administrationsoberfläche. Mithilfe einer Administrationsoberfläche können Administratoren direkt aus der Anwendung heraus mit der Datenbank interagieren. Somit lassen sich Daten bequem und schnell hinzufügen, bearbeiten und löschen. Das Fehlen einer Administrationsoberfläche führt dazu, dass jeder einzelne Administrator sich ein exter-

nes Programm zur Interaktion mit der Datenbank herunterladen muss. Dies ist umständlich und zeitaufwändig. Dementsprechend wäre eine Administrationsoberfläche eine sinnvolle Erweiterung für *D-BAS*.

1.2 Vorgaben an die Administrationsoberfläche

1.2.1 Kompatibilität

Die Administrationsoberfläche muss möglichst wiederverwendbar sein. Das heißt, dass die Anwendung nicht nur mit *D-BAS* kompatibel sein soll, sondern auch mit allen Webanwendungen die ähnliche Komponenten verwenden. Darunter fallen die Programmiersprache, das Webframework und das SQL-Toolkit.

Ein Webframework ist eine Software, welche die Entwicklung von Webanwendungen vereinfacht. Häufig benötigte Funktionen, wie das Routing, werden von einem Webframework übernommen. Einige Webframeworks bieten sogar fertige Administrationsoberflächen an. Die Wahl eines Webframeworks hängt jedoch häufig von der Wahl der Programmiersprache ab, da nicht jedes Webframework mit jeder Programmiersprache kompatibel ist. *D-BAS* wurde in Python geschrieben und verwendet als Webframework *Pyramid*. *Pyramid* skaliert gut mit stetig wachsenden Anwendungen, bietet jedoch keine Administrationsoberfläche [Rya].

Eine weitere Software, die *D-BAS* verwendet, ist *SQLAlchemy*. Bei *SQLAlchemy* handelt es sich um ein SQL-Toolkit und um einen ORM-Mapper. Ein SQL-Toolkit ist dabei für den Aufbau der Verbindung zur Datenbank zuständig. Jegliche Kommunikation zwischen der Anwendung und der Datenbank laufen über das Toolkit. Ein ORM-Mapper bildet Datenbanktabellen auf Python Klassen ab. Somit ist es möglich Python Objekte zu verwenden um auf die entsprechenden Daten in der Datenbank zuzugreifen [Entc].

Um die Kompatibilitätsanforderungen zu erfüllen, muss die Administrationsoberfläche selbstverständlich mit *D-BAS* kompatibel sein. Außerdem muss die Administrationsoberfläche mit allen Python-Webanwendungen kompatibel sein, die *Pyramid* und *SQLAlchemy* verwenden.

1.2.2 Funktionen

Die Administrationsoberfläche soll es dem Benutzer ermöglichen, aus dem Webbrowser heraus mit der Datenbank der Hauptanwendung zu interagieren. Zu dieser Interaktion zählt das Erstellen, Bearbeiten und Löschen von Datenbanktabelleneinträgen. Außerdem soll es möglich sein, neue Datenbanktabellen anzulegen und bestehende Tabellen zu löschen. Durch das Durchsuchen und Sortieren von Tabellen soll es möglich sein diese zu filtern. Bevor der Benutzer Zugriff auf die Administrationsoberfläche bekommt, soll dieser sich einloggen. Diese Funktionen sollen den Kern der Anwendung bilden.

1.3 Bestehende Lösungen

Es gibt zahlreiche Möglichkeiten eine Webanwendung um eine Administrationsoberfläche zu erweitern. Es existieren Anwendungen, die *Pyramid* um eine Administrationsoberfläche erweitern. Es können auch andere Webframeworks verwendet werden, die bereits eine Administrationsoberfläche bieten. Folgende Beispiele werden verdeutlichen, dass bereits vorhandene Administrationsoberflächen entweder nicht alle Vorgaben erfüllen, veraltet und nicht funktionsfähig sind, oder mit großem Aufwand verbunden sind. Von daher wird eine neu entwickelte Anwendung benötigt.

1.3.1 Django

Django zählt zu den beliebtesten und umfangreichsten Python-Webframeworks. Dabei liefert *Django* sogar einen eigenen ORM-Mapper. Im Gegensatz zu *Pyramid* stellt *Django* auch eine Administrationsoberfläche zur Verfügung. Diese ist sehr umfangreich und bietet viele Features, wie zum Beispiel ein vorgefertigtes Login-System [Tea].

Django ist keine Anwendung, sondern ein Tool zum Entwickeln von Anwendungen. Ohne entsprechenden Aufwand kann man die Funktionen eines Webframeworks nicht in einem anderen Webframework verwenden. Es gibt zwei Optionen, wie man die Administrationsoberfläche von *Django* in *Pyramid* Anwendungen nutzen könnte. Die erste Option wäre es, mithilfe von *Django* eine eigenständige Webanwendung zu entwickeln. Diese Webanwen-

dung würde lediglich aus einer Administrationsoberfläche bestehen. Wichtig bei diesem Ansatz wäre es, die Kompatibilität zwischen der Administrationsoberfläche und der *Pyramid* Anwendung sicherzustellen.

Die zweite Option wäre deutlich aufwendiger. Theoretisch könnte man das Webframework einer bereits vorhanden *Pyramid*-Anwendung gegen *Django* austauschen. Das heißt, dass man die komplette Anwendung mithilfe von *Django* neu entwickeln müsste. Der Aufwand der dabei entsteht ist zu hoch, um die Anwendung um ein einziges Feature zu erweitern. Generell lohnt sich das Portieren einer Anwendung auf ein anderes Webframework nicht.

Wie man sieht, eignen sich beide Optionen nicht. Der Aufwand um Anwendungen um eine Administrationsoberfläche zu erweitern ist zu hoch. Dementsprechend eignet sich *Django* nicht.

1.3.2 Websauna

Wie auch *Django* ist *Websauna* ein Webframework. *Websauna* bietet, genau wie *Django*, eine Administrationsoberfläche mit entsprechendem Login-System. Im Gegensatz zu *Django* baut *Websauna* auf einem bereits vorhandenem Webframework auf. Dieses Webframework ist *Pyramid* [Entd].

Doch auch *Websauna* eignet sich nicht als Administrationsoberflächenerweiterung. Trotz der Nähe zu *Pyramid* bestehen die selben Probleme wie bei *Django*. Es ist nicht möglich, die Administrationsoberfläche von *Websauna* ohne jeglichen Aufwand in *Pyramid* zu verwenden.

Das Team hinter *Websauna* ist deutlich kleiner als das von *Pyramid*. Dadurch erscheinen Updates unregelmäßig und Fehler werden nur sehr langsam behoben. Beispielsweise war die Dokumentationsseite von *Websauna* im Zeitraum dieser Abschlussarbeit häufig nicht erreichbar. Außerdem ist die Dokumentation von *Pyramid* deutlich umfangreicher und es existieren mehr Anleitungen und Beispiele. Ein Wechsel von *Websauna* auf *Pyramid* würde sich dementsprechend nicht anbieten [Entd].

1.3.3 Pyramid Sacrud

Im Gegensatz zu *Django* und *Websauna* ist *pyramid_sacrud* kein riesiges Webframework. Bei *pyramid_sacrud* handelt es sich um eine Anwendung. Das Ziel dieser Anwendung ist es, *Pyramid*-Anwendungen um eine Administrationsoberfläche zu erweitern. Die Administrationsoberfläche von *pyramid_sacrud* orientiert sich dabei stark an der von *Django*. Das heißt, dass das Erstellen, Bearbeiten und Löschen von Datenbanktabelleneinträgen möglich ist. Die Anwendung verfügt auch über ein Login-System [Entb].

Doch auch bei *pyramid_sacrud* handelt sich um keine geeignete Administrationsoberfläche. Die Dokumentation der Anwendung ist ungenau, da diese hauptsächlich aus Installationsanleitungen und Beispielen besteht. Einige der Komponenten der Anwendung werden zwar in der Dokumentation erwähnt, es wird jedoch nicht erklärt, wofür diese Komponenten zuständig sind. Es scheint so, als würde sich die Anwendung nicht mehr in aktiver Entwicklung befinden, da der letzte Git-Commit bereits drei Jahre alt ist [Entb].

Das Hauptproblem der Anwendung ist, dass die Anwendung nur indirekt *SQLAlchemy* verwendet. Statt *SQLAlchemy* wird nämlich *ps_alchemy* verwendet. Bei *ps_alchemy* handelt es sich um eine Erweiterung für *pyramid_sacrud*. Diese Erweiterung stellt *SQLAlchemy* Modelle für *pyramid_sacrud* zur Verfügung. Auch *ps_alchemy* wurde von denselben Entwicklern wie *pyramid_sacrud* entwickelt. Beide Anwendungen sind schlecht dokumentiert und veraltet. Der letzte Git-Commit ist bei *ps_alchemy* sogar bereits vier Jahre alt. Der letzte Travis Build von *ps_alchemy* schlug fehl. Das heißt, dass die Anwendung nicht ohne Fehler gebaut werden konnte. Es kann also davon ausgegangen werden, dass die Anwendung nicht korrekt funktioniert [Enta].

Dementsprechend ist die Installation von *pyramid_sacrud* nicht so unkompliziert wie es der Entwickler verspricht. Aufgrund der Fehler von *ps_alchemy* und der fehlenden Updates, kann die Anwendung ohne Modifikationen nicht gestartet werden. Jegliche Fehler, die bei der Installation und der Verwendung auftreten, müssen selbständig behoben werden. Ohne die Hilfe einer vernünftigen Dokumentation ist der Aufwand zu hoch, um *pyramid_sacrud* in *D-BAS* zu integrieren.

1.3.4 Fazit

Die vorgestellten Optionen zeigen, dass es keine simple und funktionierende Lösung gibt, um *Pyramid*-Anwendungen um eine Administrationsoberfläche zu erweitern. Webframeworks sind nicht dazu gedacht, um als einfache Administrationsoberflächenerweiterung verwendet zu werden. Gerade das Portieren einer gesamten Webanwendung auf ein anderes Webframework ist eine schlechte Idee.

Ein bessere Idee wäre es, bereits vorhandene Anwendungen zu verwenden. Es gibt zahlreiche Anwendungen, die *Pyramid*-Anwendungen um eine Administrationsoberfläche erweitern. Alle diese Anwendungen sind jedoch entweder veraltet oder funktionieren nicht. Mögliche Fehler müssen selbständig behoben werden. Dies wird durch häufig schlechte Dokumentationen erschwert.

Es zeigt sich also, dass es keine einfache Möglichkeit gibt *Pyramid*-Anwendungen um eine Administrationsoberfläche zu erweitern. Um vorhandene Lösungen zu verwenden ist enorm viel Aufwand nötig. Deshalb soll im Rahmen dieser Abschlussarbeit eine einfache und funktionierende Administrationsoberfläche entwickelt werden.

Kapitel 2

Architektur der Anwendung

2.1 Designentscheidungen

Das Design des von mir entwickelten *Generic-Admin-Database-Interface*, kurz *GADI*, lässt sich auf eine Designentscheidung zurückführen. Die Administrationsoberfläche sollte für den Anwender so zugänglich wie möglich gestaltet werden. Dies gilt sowohl für die Installation und Integration, als auch für die Verwendung der Anwendung. Diese Designentscheidung bildet den Kern des Designs und wurde nach der Recherche über bereits vorhandene Administrationsoberflächen getroffen. Viele der vorhandenen Lösungen sind veraltet und schlecht dokumentiert. Außerdem treten bei der Benutzung dieser Anwendungen Fehler auf. Diese Fehler muss der Anwender selbständig beheben. Die Hauptpriorität bei der Entwicklung war es also eine Plug and Play Lösung zu schaffen.

Bei der Wahl der Programmiersprache, des Webframeworks und des SQL-Toolkits gab es viele Möglichkeiten, um die gestellten Vorgaben an die Administrationsoberfläche zu erfüllen. Um die Installation von *GADI* so einfach wie möglich zu gestalten und um die Designphilosophie einzuhalten, gab es nur eine mögliche Wahl der Entwicklungstools. Laut Vorgaben soll die Administrationsoberfläche mit Webanwendungen kompatibel sein, die *Pyramid* und *SQLAlchemy* verwenden. Man kann also davon ausgehen, dass der Anwender diese Entwicklungstools bereits installiert hat. Um den Aufwand der Installation von *GADI* zu minimieren, muss also auch *GADI* mithilfe dieser Komponenten entwickelt werden. Die Kompatibilität zu anderen *Pyramid* Anwendungen ist gegeben, da *GADI* lediglich eine Erweiterung für *Pyramid*-Anwendungen ist und auf den selben Entwicklungstools basiert.

Es gibt zwei verschiedene Möglichkeiten mithilfe von *Pyramid* eine Administrationsoberfläche zu entwickeln. Die erste Möglichkeit wäre es, die Administrationsoberfläche als eigenständige *Pyramid*-Webanwendung zu entwickeln. Bei diesem Ansatz wäre die einzige Schnittstelle zwischen der Hauptanwendung und der Administrationsoberfläche die Datenbank. Somit würden beide Anwendungen auf verschiedenen Webservern laufen und dementsprechend über verschiedene Domains erreichbar sein. Es ist möglich, dass sowohl Administrationsoberfläche als auch Hauptanwendung unter derselben Domain erreichbar sind. Diese Konfiguration ist jedoch aufwendig und muss vom Anwender selbst vorgenommen werden.

Die zweite Möglichkeit wäre es, die Administrationsoberfläche in Form einer Erweiterung für *Pyramid*-Webanwendungen zu entwickeln. Bei diesem Ansatz müsste die Administrationsoberfläche in eine bereits vorhandene *Pyramid*-Webanwendung integriert werden. Es wäre dementsprechend nicht möglich die Administrationsoberfläche ohne eine Hauptanwendung auszuführen. Wie auch bei der ersten Möglichkeit würde die Administrationsoberfläche bei dieser Lösung eine eigenständige Verbindung zur Datenbank aufbauen. Neben der Verbindung zur Datenbank würde die Anwendung aber auch eine Schnittstelle mit der Hauptanwendung haben. Diese Schnittstelle wäre die *Configurator* Klasse von *Pyramid*. In dieser Klasse werden alle Einstellungen der Webanwendung vorgenommen. In *Pyramid* haben eingebundene Erweiterungen in der Regel kein eigenes Objekt der *Configurator* Klasse. Um Konfigurationen vorzunehmen verwenden eingebundene Erweiterung das *Configurator* Objekt der Hauptanwendung. Durch das Einbinden der Administrationsoberfläche in eine Hauptanwendung laufen beide Webanwendungen auf demselben Webserver. Dies führt dazu, dass ohne weitere Konfigurationen beide Anwendungen unter der selben Domain erreichbar sind.

Die Entscheidung, welche der beiden Möglichkeiten verwendet werden soll, wurde ebenfalls von der Designphilosophie beeinflusst. Um dem Anwender mögliche Webserver Konfigurationen zu ersparen, wurde *GADI* als *Pyramid*-Erweiterung entwickelt.

Aus einer einzigen Entscheidung entstand das Design für die gesamte Anwendung. Dieses Design erschwerte die Entwicklung von *GADI* und führte zu einigen Problemen. Das Ziel, eine möglichst leicht zu verwendende Anwendung zu entwickeln, konnte dennoch erreicht werden.

2.2 Architektur

Viele moderne Webframeworks verwenden dieselbe Architektur. Das Framework wird in die drei Teile *Model*, *View* und *Controller* unterteilt. Dabei ist die *View* für die visuelle Darstellung der Daten verantwortlich. Das *Model* beinhaltet alle benötigten Daten und in manchen Fällen auch die Geschäftslogik. Der *Controller* übermittelt die Eingaben des Benutzers an *Model* und *View*. Dieses Architekturmuster wird *MVC* genannt. Durch die Verwendung von *MVC* entsteht eine flexible Anwendung. Es können im Nachhinein neue Komponenten hinzugefügt werden und bestehende Komponenten ausgetauscht werden [Rya].

Die Autoren von *Pyramid* finden die Unterteilung von *MVC* zu ungenau. Die Grenzen der einzelnen Unterteilungen sind nicht klar definiert. Deshalb verwendet *Pyramid* eine eigene Terminologie. Die Architektur von *Pyramid*-Anwendungen wird in die zwei Teile *resources* und *views* geteilt. In den *views* findet man die Logik der Anwendung. Die *resources* hingegen sind die Daten der Anwendung. Eine *view* präsentiert *resources*. Die Templates, die für die visuelle Darstellung benötigt werden, gehören zu den entsprechenden *views*. Auf *Controller* wird komplett verzichtet. *Pyramid* Anwendungen werden also lediglich in Logik und Daten unterteilt [McD20, p. 23-24].

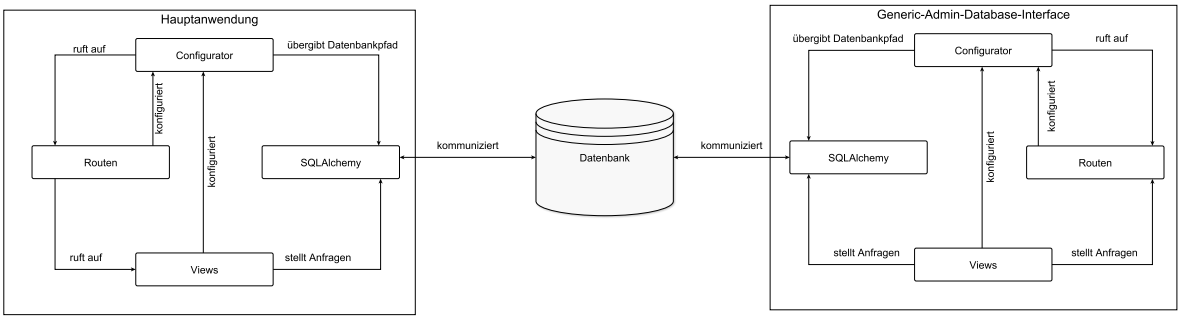

Abbildung 2.1: Zusammenspiel von Hauptanwendung und Erweiterung

Pyramid-Anwendungen können also in drei Python Module unterteilt werden. Ein Modul ist für die *views* zuständig und ein weiteres für die *resources*. Das dritte Modul ist für die Konfiguration der Anwendung zuständig. Meine Anwendung *GADI* ist zwar eine Erweiterung für bestehende *Pyramid*-Anwendungen, folgt jedoch einem ähnlichen Muster. In *GADI* ist ein Modul für die Verbindung zur Datenbank zuständig. Die Daten in der Datenbank gehört zu den *resources*. Zu der Kategorie der *views* gehören mehrere Module. In jedem Modul befindet sich Logik. Durch die Aufteilung in mehrere Module können zusammenhängende Funktionen gekapselt werden. Einen Unterschied zu gewöhnlichen *Pyramid*-Anwendungen

bietet das Konfigurationsmodul. Üblicherweise werden in diesem alle Einstellungen der *Pyramid Configurator* Klasse übergeben. Dies geschieht auch in *GADI*. *GADI* hat jedoch keine eigene Instanz des *Configurator* Objektes und verwendet das *Configurator* Objekt der Hauptanwendung.

Abbildung 2.1 zeigt das Zusammenspiel zwischen *GADI* und einer Hauptanwendung. In der Hauptanwendung werden die verschiedenen Routen und *Views* in dem *Configurator* gespeichert. Der *Configurator* übergibt außerdem den Pfad zur Datenbank an die zuständigen *SQLAlchemy* Funktionen, damit diese die Verbindung zur Datenbank aufbauen können. Bei einer Anfrage wird zunächst die entsprechende Route aufgerufen. Sobald eine Anfrage zu einer bestehenden Route eintrifft, wird die dazugehörende Methode in dem *View* Modul ausgeführt. Die Methoden im *View* Modul kümmern sich um die Logik der einzelnen Webseiten und wählen das passende Template zur Darstellung der Seite aus. Die Kommunikation zwischen *View* Modul und Datenbank geschieht über die hergestellte Verbindung des *SQLAlchemy* Moduls. Der Ablauf von *GADI* ist größtenteils identisch. Einzig im *Configurator* unterscheiden sich beide Anwendungen. Bei *GADI* wird der *Configurator* der Hauptanwendung verwendet.

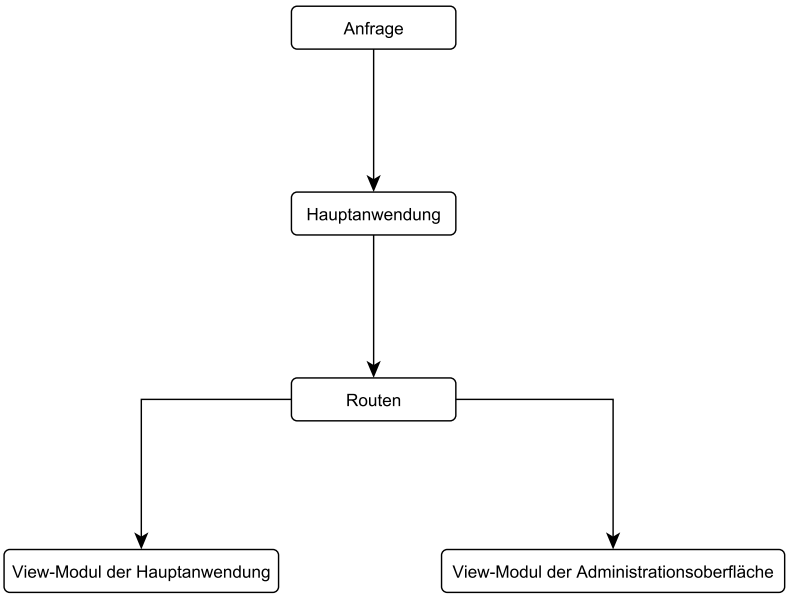

Abbildung 2.2: Ablauf einer Anfrage

In Abbildung 2.2 wird der Ablauf einer Anfrage bei einer Anwendung dargestellt, die *GADI* integriert hat. Zunächst stellt der Anwender eine Anfrage an den Webserver. Der Anwender

möchte auf eine spezifische Webseite zugreifen. Die Anwendung überprüft, ob die Route für die gesuchte Webseite registriert ist. Wenn eine vorhandene Route existiert, wird die entsprechende Methode im *View* Modul aufgerufen. Dabei wird zwischen dem *View* Modul der Hauptanwendung und dem *View* Modul der Administrationsoberfläche entschieden. Die entsprechende Methode führt dann die Logik aus, die benötigt wird, um die Seite darzustellen.

2.3 Aufbau der Anwendung

Die Anwendung besteht aus sechs Modulen und sechs Templates. Jedes dieser Templates definiert die entsprechende visuelle Darstellung der verschiedenen Routen. Die Module haben hingegen alle eine eigene Aufgabe. Das *routes* Modul definiert zum Beispiel die verschiedenen Routen für die verschiedenen Webseiten der Anwendung. Da es sich um eine Webanwendung handelt sind mit Routen die verschiedenen URLs gemeint.

Das *setup* Modul ist für die Installation der Anwendung zuständig. In diesem Modul werden alle Ressourcen deklariert, die für die Installation der Anwendung benötigt werden. Auch wird in diesem Modul der Name und die Versionsnummer der Anwendung definiert.

Das *__init__* Modul wird beim Start der Anwendung ausgeführt. Jegliche Einstellungen die in *GADI* vorgenommen wurden, werden in diesem Modul der Hauptanwendung übergeben. Außerdem liest *__init__* den Pfad zur Datenbank aus der Hauptanwendung aus.

Das *views* Modul ist das Hauptmodul der Anwendung. Die gesamte Logik der einzelnen Routen befindet sich in diesem Modul. Das Modul besteht dabei aus mehreren Methoden. Eine Methode ist üblicherweise für eine Route zuständig. Methoden bearbeiten Anfragen und senden die Resultate dieser an die entsprechenden Templates. Anschließend werden die Resultate von den Templates visuell dargestellt. Bei der Bearbeitung der Anfragen können die Methoden Anfragen an die Datenbank stellen. Die Zuweisung der Templates zu den entsprechenden Routen geschieht ebenfalls in diesem Modul.

In dem Modul *database* wird die Verbindung zur Datenbank aufgebaut. Außerdem werden in dem Modul die Metadaten der Datenbank ausgelesen. Mithilfe dieser werden die Tabellen und die Beziehungen der Tabellen aus der Datenbank in *GADI* abgebildet. Auch eine sogenannte Datenbanksession wird in diesem Modul erstellt. Mithilfe der Session können die

einzelnen Module mit der Datenbank interagieren.

Sämtliche Methoden die für das Login benötigt werden befinden sich im *security* Modul. Das Modul ermöglicht das Ver- und Entschlüsseln von Benutzerpasswörtern und das Erstellen eines neuen Benutzeraccounts.

Neben den sechs Modulen und sechs Templates gibt es auch einige statische Elemente. Bei den statischen Elementen handelt es sich hauptsächlich um *css* und JavaScript Dateien. Einen großen Teil dieser bilden Dateien die für *Bootstrap* benötigt werden. *Bootstrap* ist ein Framework, welches die Gestaltung von Oberflächen vereinfacht. Den restlichen Teil der statischen Elemente bilden Daten, die für *DataTables* benötigt werden. *DataTables* ist ein Plugin für *jQuery* und wird in *GADI* für das Sortieren und Durchsuchen der Tabellen verwendet.

Kapitel 3

User Stories

Die verschiedenen Funktionen der Anwendung werden anhand von User Stories erläutert. Eine User Story beschreibt den Ablauf einer Funktion der Anwendung aus der Sicht des Anwenders.

3.1 Administrator erstellen

Diese User Story beschreibt das Verfahren zum Erstellen eines neuen Administrator Accounts.

1. Der Anwender möchte einen Administrator Account anlegen.

2. Vor dem Start der Anwendung führt der Anwender in der Konsole ein Python Skript zum Erstellen eines Administrator Accounts aus.

3. Der Anwender wird in der Konsole darum gebeten einen Namen für den Account einzugeben.

4. Nachdem der Anwender einen Namen eingegeben hat, wird der Anwender nach einem Passwort für den Account gefragt.

5. Nachdem der Anwender ein Passwort übergeben hat, wird ein neuer Account erstellt.

6. Dem Anwender wird in der Konsole mitgeteilt, dass der Account erfolgreich erstellt wurde.

3.2 Login

Das Verfahren zum Login wird in dieser User Story beschrieben.

1. Ein Anwender möchte die Administrationsoberfläche verwenden.
2. Der Anwender ruft die Login-Seite unter *.../gadi_login* auf.
3. Der Anwender gibt auf der Seite seine Login-Daten ein, vertippt sich jedoch.
4. Auf der Seite erscheint eine Fehlermeldung. Diese teilt dem Anwender mit, dass die von ihm eingegebenen Login-Daten inkorrekt sind.
5. Der Anwender korrigiert die von ihm eingegebenen Daten.
6. Die eingegebenen Login-Daten sind korrekt und der Anwender wird auf das Dashboard der Administrationsoberfläche weitergeleitet.

3.3 Tabellen filtern

Diese User Story beschreibt, wie das Sortieren und Durchsuchen von Tabellen funktioniert.

1. Der Anwender möchte nach einer spezifischen Tabelle suchen.
2. Nach dem erfolgreichen Login befindet sich der Anwender auf dem Dashboard.
3. Auf dem Dashboard sieht der Anwender eine Tabelle mit den Namen der vorhandenen Datenbanktabellen.
4. Der Anwender klickt auf den Tabellenkopf.

5. Die Namen der Tabellen werden aufsteigend sortiert.

6. Der Anwender gibt den Namen der gesuchten Tabelle in die Suchleiste ein.

7. Falls eine Tabelle mit diesem Namen existiert, wird diese angezeigt. Ansonsten wird eine Fehlermeldung angezeigt, dass eine Tabelle mit diesem Namen nicht existiert.

3.4 Tabelleneintrag hinzufügen

Der Ablauf beim Erstellen eines neuen Tabelleneintrages wird in dieser User Story beschrieben.

1. Der Anwender möchte zu einer Tabelle einen neuen Eintrag hinzufügen.

2. Nachdem der Anwender auf dem Dashboard auf die passende Tabelle geklickt hat, wird er auf die Tabellenansicht der entsprechenden Tabelle weitergeleitet.

3. Der Anwender sieht eine Tabelle mit den verschiedenen Einträgen der Tabelle.

4. Der Anwender betätigt den *Add Column* Button.

5. Erneut wird der Anwender auf eine andere Seite weitergeleitet.

6. Auf dieser Seite sieht der Anwender ein Eingabefeld für jede Spalte der Tabelle.

7. Der Anwender füllt die Felder aus, gibt in einem Feld jedoch statt einer Zahl einen Buchstaben ein.

8. Nachdem der Anwender seine Eingabe bestätigt hat, erscheint eine Fehlermeldung. Die Fehlermeldung weist darauf hin, dass ein falscher Datentyp eingegeben wurde.

9. Der Anwender korrigiert seine Eingaben und bestätigt die Eingabe erneut.

10. Der Tabelleneintrag wird erstellt.

11. Der Anwender wird zurück auf die Ansicht der Tabelle geleitet.

3.5 Tabelleneintrag löschen

Die folgende User Story beschreibt, wie ein Anwender einen bestehenden Tabelleneintrag löschen kann.

1. Der Anwender möchte einen Tabelleneintrag löschen.
2. Nachdem der Anwender auf dem Dashboard auf die passende Tabelle geklickt hat, wird er auf die Tabellenansicht der entsprechenden Tabelle weitergeleitet.
3. Der Anwender betätigt den *Delete* Button neben der zu löschenden Zeile.
4. Es erscheint eine Meldung. Der Anwender muss bestätigen, dass er sich sicher ist, dass die Zeile gelöscht werden soll.
5. Der Anwender bejaht diese Anfrage.
6. Der Tabelleneintrag wird entfernt.

3.6 Tabelleneintrag bearbeiten

Die folgende User Story beschreibt, wie ein User einen bestehenden Tabelleneintrag bearbeiten kann.

1. Der Anwender möchte einen Tabelleneintrag bearbeiten.
2. Nachdem der Anwender auf dem Dashboard auf die passende Tabelle geklickt hat, wird er auf die Tabellenansicht der entsprechenden Tabelle weitergeleitet.
3. Der Anwender sieht eine Tabelle mit der verschiedenen Einträgen der Tabelle.
4. Der Anwender möchte den ersten Eintrag der Tabelle bearbeiten.
5. Der Anwender betätigt den *Edit* Button neben der Zeile.

6. Erneut wird der Anwender auf eine andere Seite weitergeleitet.

7. Auf dieser Seite sieht der Anwender ein Eingabefeld für jede Spalte der Tabelle. Die Eingabefelder sind mit den Werten des Tabelleneintrages ausgefüllt.

8. Der Anwender bearbeitet die gewünschten Einträge, gibt jedoch in einem Feld einen falschen Wert ein.

9. Der Anwender betätigt den *Confirm* Button.

10. Eine Fehlermeldung erscheint. Sie weist darauf hin, dass der Anwender unpassende Daten eingegeben hat.

11. Die Eingaben werden von dem Anwender korrigiert.

12. Der Anwender betätigt erneut den *Confirm* Button.

13. Der Tabelleneintrag wird bearbeitet.

14. Der Anwender wird zurück auf die Ansicht der Tabelle geleitet.

3.7 Tabelle löschen

Die folgende User Story beschreibt, wie ein User eine Tabelle löschen kann.

1. Der Anwender möchte eine Tabelle löschen.

2. Auf dem Dashboard sucht sich der Anwender die passende Tabelle heraus und klickt auf diese.

3. Der Anwender wird auf die Tabellenansicht weitergeleitet.

4. Um die Tabelle zu löschen, betätigt der Anwender den *Delete Table* Button.

5. Ein Pop-Up erscheint. Der Anwender muss den Löschvorgang bestätigen.

6. Der Anwender bestätigt die Anfrage.

7. Die Tabelle wird gelöscht.

8. Der Anwender wird zurück auf das Dashboard geleitet.

3.8 Tabelle erstellen

Die folgende User Story beschreibt, wie ein User eine neue Tabelle erstellen kann.

1. Der Anwender möchte eine neue Tabelle erstellen.

2. Auf dem Dashboard betätigt der Anwender dafür den *Create Table* Button.

3. Der Anwender wird auf eine Seite zum Erstellen einer Tabelle weitergeleitet.

4. Auf dieser Seite sieht der Anwender mehrere Eingabefelder, Drop-Down Menüs und Checkboxen. Der Anwender kann der Tabelle und den einzelnen Tabellenspalten einen Namen geben. Außerdem kann der Anwender den Datentyp der Spalte wählen und ankreuzen, ob die Spalte ein Primärschlüssel oder ein Fremdschlüssel ist. Falls die Spalte ein Fremdschlüssel ist, kann der Anwender den Primärschlüssel wählen, auf den sich dieser bezieht.

5. Der Anwender benötigt mehr als eine Spalte, also betätigt er den *Add Column* Button so oft wie nötig.

6. Die Eingabefelder werden von dem User ausgefüllt. Jedoch vergisst der Anwender eine Spalte zu benennen.

7. Der Anwender betätigt den *Create* Button.

8. Eine Fehlermeldung erscheint, da der Anwender nicht alle Felder ausgefüllt hat.

9. Der Anwender korrigiert seinen Fehler und betätigt den Button erneut.

10. Die Tabelle wird erstellt und der Anwender wird auf das Dashboard zurückgeleitet.

Kapitel 4

Probleme und Entscheidungen

4.1 Das Metadaten-Problem

Durch die getroffenen Designentscheidungen kam es bei der Entwicklung zu einigen Problemen. Um dem Anwender die Integration der Anwendung zu erleichtern, wurden von diesem keinerlei Daten abgefragt. Einzige Ausnahme ist hierbei der Pfad zur Datenbank. *SQLAlchemy* bildet Datenbanktabellen in Python als Klassen ab. Diese Klassen bezeichnet man als Datenbankmodelle. Zu jeder Datenbanktabelle gibt es normalerweise in einer Anwendung ein passendes Datenbankmodell. Die Datenbankmodelle werden in der Anwendung verwendet, um mit den entsprechenden Daten in der Datenbank zu interagieren. Die Entscheidung von dem Anwender diese Datenbankmodelle nicht zu verlangen führte dazu, dass *GADI* keinen Zugriff auf diese Modelle hatte.

Dies war eine Zeit lang kein Problem, da die Datenbank Informationen über die einzelnen Tabellen in den Metadaten der Datenbank speichert. Mithilfe von *SQLAlchemy* ließen sich diese Metadaten auslesen und daraus temporäre Objekte der verschiedenen Tabellen bilden. Mithilfe dieser Objekte konnte man mit der Datenbank interagieren.

Die Probleme traten bei dem Bearbeiten und Löschen von Datenbanktabelleneinträgen auf. Um der Datenbank mitzuteilen, welcher Eintrag verändert werden soll, muss man die *SQLAlchemy* Methode *where* verwenden. Allerdings akzeptiert diese Methode die aus den Metadaten erstellten Tabellenobjekte nicht. Sie benötigt zwingend die Datenbankmodelle. Das Löschen und Bearbeiten von Datenbanktabelleneinträgen sind zwei der wichtigsten Funktion

der Administrationsoberfläche und mit bloßen Metadaten ließen sich diese Funktionen nicht umsetzen. Im Verlauf der Entwicklung entstanden drei Lösungen für dieses Problem.

4.1.1 SQL-Statements

Die *SQLAlchemy where* Methode akzeptiert nicht nur Datenbankmodelle, sondern auch SQL-Statements. SQL-Statements sind Befehle, die von SQL-Datenbanken verstanden werden. Der *where* Teil eines SQL-Statements spezifiziert dabei die Auswahl genauer. So wäre ein simpler *where* Teil eines SQL-Statements *id=1*. Dabei ist *id* der Name einer Spalte und *1* der Wert, den diese Spalte annehmen soll. Ein erster naiver Ansatz war es also, für jede Anfrage diesen *where* Teil des SQL-Statements für die entsprechenden Tabellen zu generieren.

Diese Lösung des Metadaten-Problems war leicht umzusetzen und funktionierte gut, sie war jedoch nicht sonderlich elegant. Dass die Anwendung dadurch eventuell durch SQL-Injections angegriffen werden kann habe ich dabei nicht bedacht. Herr Brenneis hat mich darauf aufmerksam gemacht. Als SQL-Injection bezeichnet man einen Angriff, bei dem versucht wird ungewollte böswillige SQL-Befehle auszuführen. Dies ist häufig in Eingabefeldern oder URLs möglich. Wenn der eingegebene Wert nicht zunächst kontrolliert wird, sondern direkt in ein SQL-Statement eingesetzt wird, kann der Angreifer mit seiner Eingabe das SQL-Statement verändern. Somit kann der Angreifer eigene SQL-Statements aus dem Eingabefeld heraus ausführen. Dadurch lassen sich ganze Tabellen löschen und Passwörter ausgeben.

Nach einigen gründlichen Tests war es mir nicht möglich in *GADI* aus den Eingabefeldern heraus böswillige SQL-Statements auszuführen. Das Risiko wollte ich dennoch nicht eingehen und suchte nach einer anderen Lösung für das Metadaten-Problem.

4.1.2 Wrapper

Der nächste Ansatz war es, die aus den Metadaten ausgelesenen Tabellenobjekte in neue *SQLAlchemy* Tabellenobjekte zu hüllen. *SQLAlchemy* bietet hierfür sogar ein Attribut. Mit dem Attribut *__table__* lässt sich in einem *SQLAlchemy* Tabellenobjekt eine bereits vorhandene Tabelle speichern [Ent20b].

Dadurch hatte die Anwendung Zugriff auf richtige *SQLAlchemy* Tabellenobjekte und nicht nur die Tabellen aus den Metadaten der Datenbank. Mithilfe dieser eingehüllten Tabellen ließen sich Datenbanktabelleneinträge löschen und bearbeiten.

Auch dieser Ansatz führte zu Problemen. Die umhüllten Tabellen lösten die Datenbank-Trigger, genannt *Cascades*, nicht aus. Diese Trigger werden zum Beispiel dazu verwendet, um beim Löschen eines Eintrages einen verwandten Tabelleneintrag mit zu löschen. *SQLAlchemy* bietet zwei Optionen einen *Cascade* zu definieren. Entweder definiert man den *Cascade* an dem Primärschlüssel der zu löschenden Tabelle mit *cascade='delete'*, oder man definiert den *Cascade* mit *ondelete='cascade'* an dem Fremdschlüssel der abhängigen Tabelle. Die erste Option ist dabei die am weitesten verbreitete und genau diese Option funktioniert mit der Wrapper-Lösung nicht. Bei der ersten Option löst der zu löschende Eintrag den Trigger aus. Dieser Trigger reagiert jedoch nicht auf umhüllte Einträge. Das heißt, dass die Wrapper-Lösung zwar Tabelleneinträge löschen kann, jedoch wird dieser Vorgang von den Triggern aufgrund der Umhüllung nicht erkannt.

Die zweite Option der *Cascade* Definition funktioniert hingegen mit der Wrapper-Lösung. Hierbei wird der Trigger nicht von dem zu löschenden Eintrag, sondern von dem abhängigen Eintrag ausgelöst. Sobald ein abhängiger Tabelleneintrag eine Veränderung wahrnimmt, löst dieser den entsprechenden Trigger selbständig aus und löscht sich somit selbst.

In *D-BAS* werden *Cascades* jedoch nach der ersten Option deklariert. Das heißt, dass um die Wrapper-Lösung zu verwenden alle *Cascade* Definitionen in *D-BAS* umgeschrieben werden müssten. Dies würde gegen die Designphilosophie verstoßen und somit ist auch der Wrapper keine geeignete Lösung.

4.1.3 Mapper

Die finale Lösung für das Problem bestand darin die *SQLAlchemy Base* zu tauschen. Die *Base* ist unter anderem für die Erstellung der Metadaten zuständig. Ich habe bis zu diesem Zeitpunkt die *declarative Base* verwendet. Doch *SQLAlchemy* bietet eine weitere *Base*, die *automap Base*. Diese *Base* bildet im Gegensatz zu der *declarative Base* die Beziehungen zwischen den einzelnen Tabellen genauer ab. Außerdem erstellt die *automap Base* die einzelnen Tabellenobjekte automatisch. Dies sorgt dafür, dass mit der *automap Base* sowohl das

Löschen und Bearbeiten von Tabelleneinträgen, als auch die *Cascades* per *cascade='delete'* möglich sind [Ent20a].

Die Verwendung der *automap Base* führt jedoch zu einer weiteren Konfiguration die man vornehmen muss. Durch das exakte Abbilden der Tabellen muss man eine alternative Namensgebung für die Beziehungen der Tabellen definieren. Dies ist erforderlich, da sich ansonsten die Namen der Beziehungen in *GADI* mit den Namen der Hauptanwendung überschneiden.

Auch die *automap Base* hat ein Problem. Tabellen, die nur aus zwei Fremdschlüsseln bestehen, werden von der *automap Base* nicht abgebildet. Diese Tabellen werden *secondary* Tabellen genannt. Sie stellen eine Beziehung zwischen zwei Tabellen dar. Die *automap Base* bildet jedoch nur die Beziehung zwischen den beiden Tabellen ab. Dies hat zur Folge, dass *GADI* keinen Zugriff auf *secondary* Tabellen hat. Die Anzahl an *secondary* Tabellen ist üblicherweise gering und die Beziehungen, die diese abbilden gehen nicht verloren [Ent20a].

4.2 Kritische Entscheidungen

Bei der Entwicklung wurden viele kritische Entscheidungen getroffen. Dabei wurde die Entscheidung mit dem größten Einfluss auf die Entwicklung bereits zu Beginn getroffen. Diese Entscheidung war es, die Anwendung für den Benutzer so einfach wie möglich zu gestalten. Um dies zu ermöglichen, wurden von dem Benutzer möglichst wenig Daten gefordert. Der Anwender sollte, abgesehen von dem Pfad zur Datenbank, keine Daten übergeben müssen. Diese Entscheidung führte zum Metadaten-Problem und erschwerte die Entwicklung der Anwendung enorm.

Eine weitere wichtige Entscheidung war es *DataTables* zu verwenden. *DataTables* ist ein Plugin für *jQuery*. Dieses Plugin erweitert HTML Tabellen um viele verschiedene Funktionen. In *GADI* wird *DataTables* hauptsächlich dafür verwendet, um Tabellen zu sortieren und zu durchsuchen. Zunächst wurde dafür *List.js* verwendet. *List.js* ist eine JavaScript Bibliothek, die hauptsächlich HTML Listen und Tabellen um Such- und Sortierfunktionen erweitert. Die Suchfunktion von *List.js* war langsam und die Anwendung führte zu einigen visuellen Fehlern. Aus diesem Grund wurde *List.js* gegen *DataTables* ausgetauscht. Die Verwendung von *DataTables* erleichterte die Entwicklung, da ich somit die Funktion zum Durchsuchen und Sortieren von HTML Tabellen nicht selber entwickeln musste.

Um sicherzustellen, dass alle Bibliotheken und Frameworks wie *DataTables* und *Bootstrap* zu jeder Zeit verfügbar sind, sind diese in der Anwendung vorhanden. Dadurch ist die Anwendung minimal größer. Es wäre möglich diese Elemente nicht lokal zu speichern, sondern von Online Quellen zu beziehen. Falls die Online Quellen jedoch nicht erreichbar sind oder das Internet ausfällt, könnte man auf diese Bibliotheken und Framworks nicht zugreifen. Dadurch würde ein Großteil der Anwendung nicht funktionieren.

Im Verlauf der Entwicklung wurde die Entscheidung getroffen eine *Form Validation* zu verwenden. Eine *Form Validation* überprüft die Eingaben des Benutzers. Falls ein Benutzer in einem Eingabefeld für Zahlen einen Buchstaben eingegeben hat, wird dieser Fehler von der *Form Validation* erkannt. Dadurch soll verhindert werden, dass falsche Daten in der Datenbank gespeichert werden. In *GADI* wird für die *Form Validation* eine Kombination aus *Colander* und *Deform* verwendet. *Colander* übernimmt die Überprüfung der Eingaben. Dafür muss für jede Datenbanktabelle ein entsprechendes *Colander* Modell erstellt werden. Es ist nicht möglich jeden Datentyp zu überprüfen, deshalb habe ich mich nur auf die wichtigsten Datentypen konzentriert. Nicht erkannte Datentypen werden als Text verarbeitet und an die Datenbank weitergegeben. *Deform* erstellt die eigentlichen Eingabefelder für die *Colander* Modelle auf der Internetseite. Durch die Verwendung einer *Form Validation* wurde das Erstellen von Eingabefeldern vereinfacht, jedoch musste für jede Tabelle ein *Colander* Modell erstellt werden.

Kapitel 5

Installation der Anwendung

5.1 Voraussetzungen

GADI benötigt folgenden Anwendungen:

- Python
- SQLAlchemy
- Pyramid
- bcrypt
- deform
- colander
- pyramid_tm
- pyramid_chameleon
- zope.sqlalchemy

Dabei ist es besonders wichtig, dass Python installiert ist, alle anderen Anwendung kann

GADI während der Installation selbst installieren. Außerdem benötigt man eine Hauptanwendung. Das *GADI* ist eine Erweiterung für *Pyramid*-Webanwendungen und kann dementsprechend nicht ohne eine Hauptanwendung ausgeführt werden.

5.2 Installationsanleitung

Nach dem man *GADI* heruntergeladen hat muss man den heruntergeladenen Ordner öffnen. In dem Ordner navigiert man nun in das Unterverzeichnis *src*. Mithilfe von *pip* kann die Anwendung nun mit dem Befehl *pip install -e .* installiert werden. Nach der Installation muss *GADI* in eine Hauptanwendung eingebunden werden und man muss *GADI* den Pfad zur Datenbank der Hauptanwendung übergeben.

Die Übergabe des Datenbankpfades an die Administrationsoberfläche erfolgt mithilfe der *Pyramid Configurator* Klasse. Zunächst muss man das Konfigurationsmodul der Hauptanwendung öffnen. Dieses Modul erweitert man um die folgende Zeile *config.add_settings ('gadi.sqlalchemy.url': '?')*. Das Fragezeichen ersetzt man dabei mit dem Pfad zur Datenbank.

Anschließend muss man die Administrationsoberfläche nur noch einbinden. Dies geschieht ebenfalls in dem Konfigurationsmodul der Hauptanwendung. Das Modul muss um folgende Zeile erweitert werden *config.include('generic_admin_database_interface')*. Wichtig ist dabei, dass diese Zeile nach der vorherigen Zeile eingefügt wird.

Wenn man nun die Hauptanwendung startet, startet auch die Administrationsoberfläche mit. Unter *.../gadi_login* findet man die Login-Seite. Wichtig ist jedoch, dass man vor dem Start der Anwendung einen Administrator Account anlegt. Um dies zu tun, muss man den Befehl *python -m generic_admin_database_interface.security* vor dem Start der Anwendung ausführen und den Anweisungen folgen.

5.3 Installationsbeispiel anhand von D-BAS

Um *GADI* in *D-BAS* zu integrieren kann man größtenteils der obigen Anleitung folgen. *D-BAS* wird als Docker-Container verteilt. Von daher sind zwei Modifikationen an obiger Anleitung nötig. Ein Docker-Container ist dabei eine Art virtuelle Maschine. Das *General-Admin-Database-Interface* muss also in den Container eingebunden werden.

Zunächst muss man das *General-Admin-Database-Interface* Verzeichnis in das *D-BAS Master* Verzeichnis kopieren. Hierbei reicht es den *src* Ordner zu kopieren. Anschließend muss man das Dockerfile in dem *D-BAS Master* Verzeichnis bearbeiten. In dem Dockerfile muss nun angegeben werden, dass der *src* Ordner in den Container kopiert wird und dass *General-Admin-Database-Interface* installiert wird. Dazu muss man *COPY . /src* und *RUN cd src && pip3 install -e . && cd ..* dem Dockerfile hinzufügen. Diese müssen jedoch nach *COPY . /dbas/* eingefügt werden. Anschließend muss man der Administrationsoberfläche nur noch den Pfad zur Datenbank übergeben und die Anwendung in *D-BAS* einbinden. Diese Schritte kann man ganz normal wie in obiger Anleitung durchführen.

Kapitel 6

Fazit

6.1 Mögliche Erweiterungen

Die im Rahmen dieser Abschlussarbeit entwickelte Administrationsoberfläche lässt sich noch an vielen Stellen verbessern und erweitern. So kann man die Sicherheit der Anwendung optimieren. Aktuell werden die Zugangsdaten des Administrators verschlüsselt in einer einfachen JSON Datei gespeichert. Dies ist ein Sicherheitsrisiko. Es wäre sinnvoller, der Datei zumindest eingeschränkte Leserechte zu geben. Ein bessere Lösung wäre es die Zugangsdaten in einer Datenbank zu speichern.

Das Erstellen von Tabellen lässt sich an vielen Stellen erweitern. Es ist möglich beim Erstellen einer Tabelle eine Spalte als Fremdschlüssel zu deklarieren. Dieser Fremdschlüssel kann sich jedoch nur auf einen Primärschlüssel beziehen. Ein Bezug auf einen Tupel von Primärschlüsseln ist aktuell nicht möglich. Ein Primärschlüssel einer Datenbanktabelle kann sich aus mehreren Spalten zusammensetzen. Dementsprechend wäre es sinnvoll *GADI* um eine solche Funktion zu erweitern.

Bei dem Setzen eines Fremdschlüssels sollte sich der Datentyp der Spalte automatisch entsprechend des gewählten Primärschlüssels verändern. Dies ist aktuell nicht der Fall. Wenn sich der Datentyp des Primärschlüssels und des Fremdschlüssels unterscheiden, könnte dies zu Fehlern führen. Um dies zu vermeiden, sollte *GADI* um eine solche Funktion erweitert werden.

Außerdem kann man aktuell beim Erstellen von Tabellen nicht angeben, ob Spalten leer sein dürfen oder nicht. Es wird nur vorausgesetzt, dass die Spalte mit dem Primärschlüssel nicht leer ist. Alle anderen Spalten können leer sein. Eine Checkbox mit der man dies beim Erstellen der Tabelle angeben kann wäre sinnvoll.

Die visuelle Darstellung der Anwendung lässt sich ebenfalls erweitern. Aktuell ist die Darstellung nur auf 16:9 FullHD Bildschirme ausgelegt. Es wäre sinnvoll die Darstellung der Anwendung für kleinere Bildschirme zu optimieren.

6.2 Bewertung der Anwendung

Alle im Vorfeld gestellten Voraussetzungen an eine Administrationsoberfläche wurden erfüllt. *GADI* bietet alle geforderten Funktionen, sowie die Kompatibilität zu Webanwendungen, die auf *Pyramid* basieren. Auch die persönliche Designphilosophie wurde eingehalten. Dadurch kann *GADI* bereits mithilfe von wenigen Zeilen installiert und in eine bestehende Anwendung integriert werden. Auch die User Stories konnten ohne Abweichungen umgesetzt werden.

An einigen Stellen kann *GADI* jedoch noch erweitert und optimiert werden. So besteht ein Sicherheitsrisiko beim Abspeichern von Zugangsdaten in JSON Dateien. Auch das Erstellen von Tabellen kann noch verbessert werden. Die visuelle Darstellung von *GADI* kann noch für kleinere Bildschirme optimiert werden. Generell fehlt es der Anwendung an Feinschliff.

Zusammenfassend kann man sagen, dass *GADI* noch einige Makel hat. Im Gegensatz zu bestehenden Administrationsoberflächen macht *GADI* jedoch vieles besser. Der größte Kritikpunkt an bestehenden Administrationsoberflächen für *Pyramid* war es, dass diese entweder veraltet sind und nicht mehr funktionieren, oder schlecht dokumentiert sind. Variablen und Methoden sind in *GADI* aussagekräftig benannt und komplizierte Codepassagen kommentiert. Außerdem wurden im Rahmen dieser Abschlussarbeiten die einzelnen Module der Anwendung erläutert. *GADI* ist mit modernen *Pyramid*-Anwendungen kompatibel. Wenn man also das bestehende Sicherheitsrisiko behebt, eignet sich *GADI* als eine unkomplizierte Administrationsoberflächenerweiterung für *Pyramid*-Webanwendungen.

Literatur

[Enta] PS Alchemy Entwicklerteam. *PS Alchemy Github*. URL: https://github.com/sacrud/ps_alchemy (besucht am 16. 04. 2020).

[Entb] Pyramid Sacrud Entwicklerteam. *Pyramid Sacrud Github*. URL: https://github.com/sacrud/pyramid_sacrud (besucht am 15. 04. 2020).

[Entc] SQLAlchemy Entwicklerteam. *The Python SQL Toolkit and Object Relational Mapper*. URL: https://www.sqlalchemy.org/ (besucht am 20. 05. 2020).

[Entd] Websauna Entwicklerteam. *Websauna Github*. URL: https://github.com/websauna/websauna/ (besucht am 10. 05. 2020).

[Ent20a] SQLAlchemy Entwicklerteam. *SQLAlchemy 1.3 Documentation - Automap*. 2020. URL: https://docs.sqlalchemy.org/en/13/orm/extensions/automap.html (besucht am 29. 05. 2020).

[Ent20b] SQLAlchemy Entwicklerteam. *SQLAlchemy 1.3 Documentation - Table Configuration*. 2020. URL: https://docs.sqlalchemy.org/en/13/orm/extensions/declarative/table_config.html (besucht am 29. 05. 2020).

[McD20] Chris McDonough. *The Pyramid Web Framework*. 2020. URL: https://buildmedia.readthedocs.org/media/pdf/pyramid/1.10-branch/pyramid.pdf (besucht am 20. 05. 2020).

[Met] Christian Meter. *D-BAS – Dialogbasiertes Argumentationssystem*. URL: https://diid.hhu.de/projekte/dbas/ (besucht am 14. 04. 2020).

[Rya] Alexander Ryabtsev. *Web Frameworks: How To Get Started*. URL: https://djangostars.com/blog/what-is-a-web-framework/ (besucht am 20. 05. 2020).

[Tea] Django Stars Team. *Why We Use Django Framework & What Is Django Used For*. URL: https://djangostars.com/blog/why-we-use-django-framework/ (besucht am 15. 04. 2020).

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 11. Juni 2020 Alexej Elrich