Hochschule Darmstadt

- Fachbereich Informatik -

Neukonzeptionierung und prototypische Umsetzung der Software euSDB unter Einhaltung der REST-Prinzipien

Abschlussarbeit zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

vorgelegt von

Marius Alexander Paus genannt Dickmann

Referent:	Prof. Dr. Akelbein
Korreferent:	Prof. Dr. Horsch

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 29.01.2014

Abstract

Im Chemikalienhandel sind alle Teilnehmer der Lieferkette, also Hersteller, Spediteur und Endabnehmer, dazu verpflichtet, zu jeder weitergegebenen Chemikalie ein dazugehöriges Sicherheitsdatenblatt (SDB)[1] zu übergeben. Dieses muss von allen Beteiligten für mindestens zehn Jahre aufgehoben werden und auf Verlangen zur Verfügung gestellt werden. Ein SDB beinhaltet Informationen des Herstellers einer Chemikalie zur Nutzung, Lagerung und (Umwelt-)Gefährdungen und sichert den Hersteller rechtlich bei falschem Einsatz der Chemikalie ab.

Zu Unterstützung dieses Zwecks wird eine Onlineplattform, *euSDB* , bereitgestellt, die den Austausch und die Archivierung von SDB ermöglicht. Dabei spielt die Realisierung einer Representational State Transfer (REST)-konformen Schnittstelle eine zentrale Rolle. Diese gewährleistet eine einfache und intuitive Nutzung der Plattform durch die geeignete Definition der Ressourcen und deren Repräsentationen, Vergabe von eindeutigen Uniform Resource Identifier (URI) an diese, lose Kopplung von Server und Client. Außerdem ist die statuslose Kommunikation zwischen Server und Client, die Verknüpfung der Repräsentationen der Ressourcen mittels Hypermedia bzw. Hypertext, und die Nutzung der Hypertext Transfer Protokoll (HTTP)-Standardmethoden PUT, GET, usw. möglich.

Eine Volltextsuche nach SDB wurde mit dem auf *Apache Lucene* basierenden Volltextsuchsystem *elasticsearch* umgesetzt.

Die Plattform wird mit dem *Django* -Framework [DJSOFO-2013] realisiert. Dabei handelt es sich um ein in der Programmiersprache *Python* geschriebenes Webentwicklungsframework, welches der schnellen Webentwicklung auf Basis des Model-View-Controller (MVC)-Designmusters dient.

Das Projekt gliederte sich in folgende Phasen: Analyse eines bestehenden Systems, Erstellen eines Lasten- und Pflichtenheftes, Konzeption und Realisierung der Plattform *euSDB* . Die Realisierung wurde nach einem iterativen Vorgehensmodell umgesetzt.

Das Ergebnis des Projektes ist eine Plattform, die sehr flexibel, erweiterbar und portierbar ist. Sie wird der Aufgabenstellung in fast allen Punkten gerecht und verfügt über einen Funktionsumfang, der den der alten Plattform übersteigt.

[1] In Anhang B, Glossar, finden sich Informationen zu Fachbegriffen und Abkürzungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

1.1. Fachliche Umgebung

Nationale Gesetze als auch internationale Richtlinien zwingen die Produzenten von Chemikalien, zu jeder produzierten Chemikalie ein SDB zu erstellen. Ein SDB hat alle Informationen zu enthalten, die für den Umweltschutz, die Anlagensicherheit, die Transportsicherheit und den Arbeitsschutz eine zentrale Bedeutung haben [VCI-2008] (Siehe auch Tabelle 1.1). Wird eine Chemikalie weitergegeben, beispielsweise beim Verkauf oder Transport, muss das SDB zu der Chemikalie mit übergeben werden.
Alle am Handel mit Chemikalien teilnehmenden Betriebe, ganz gleich ob Hersteller, Spediteur oder Endabnehmer, sind dazu verpflichtet, die von ihm erstellten oder von anderen erhaltenen SDB für mindestens zehn Jahre aufzubewahren und verfügbar zu halten. Die Informationen zu Gefahren und die dazugehörigen Sicherheitsanweisungen werden in standardisierten Sätzen übergeben. Ebenfalls sind die Gefahrenpiktogramme nach alter und neuer Norm enthalten.
Da jeder Herrsteller verschiedene Versionen seiner SDB zur Verfügung stellt, z.B. bei unterschiedlichen Verwendungszwecken der Chemikalie, müssen diese ebenfalls archiviert werden.

1.2. Motivation und Ziel des Projektes

Heute ist es weit verbreitet, die SDB in Papierform weiterzugeben und diese in gedruckter Form zu archivieren. Auf lange Sicht ist dies teuer und ineffizient. Auf Grund von Umstrukturierungen innerhalb von Betrieben, oder bei Verkauf einer Firma, kommt es häufig vor, dass SDB abhanden kommen bzw. nicht mehr auffindbar sind. Um dies zu verhindern, betreiben viele große Unternehmen inzwischen firmeninterne Datenbanken zur Verwaltung und Archivierung von SDB. Leider er-

Bezeichnung	Bedeutung
GHS-Kennzeichnung	Die Einstufungsrichtlinien nach Globally Harmonized System of Classification, Labelling and Packaging of Chemicals (GHS)-Standard wurde von den Vereinten Nationen entwickelt und von der EU verfeinert. Zur Zeit gibt es neun GHS-Gefahreneinstufungen, GHS01 bis GHS09. Dabei wird zum Beispiel zwischen brennbar, umweltgefährdend oder ätzend unterschieden.
EU-Kennzeichnung	Die EU-Kennzeichnung ist das Vorgängersystem der GHS-Kennzeichnung zur Einstufung von Chemikalien (bis 1. Juni 2007). Die EU-Kennzeichnung unterscheidet zehn verschiedene Gefahreneinstufungen. Auf Grund der Verwendung über Jahrzehnte hinweg, sind die zu diesen Einstufungen gehörenden Piktogramme sehr weit verbreitet und daher noch immer von großer Bedeutung und sollten nicht in den SDB weggelassen werden.
Hazard-Kennzeichnung	Bei der Hazard-Kennzeichnung handelt es sich ebenfalls um Piktogramme. Sie warnen nicht nur vor chemischen, sondern auch vor elektrischen oder pysikalischen Risiken und Gefahren. Die Hazard-Kennzeichen sind ebenfalls von der UN entwickelt worden und finden auch Verwendung in der GHS-Kennzeichnung.
H-Sätze	Die H-Sätze wurden zeitgleich mit der GHS-Kennzeichnung in der EU eingeführt. Die H-Sätze beschreiben Risiken und Gefährdungen, die von Chemikalien ausgehen können. Das H steht für Hazard, deutsch Gefahr oder Risiko.
P-Sätze	Die P-Sätze gehören zu den H-Sätzen. Sie geben Sicherheitshinweise zum Umgang mit Chemikalien. In der Regel gibt es zu jedem H-Satz einen P-Satz. Auf diese Weise wird jede Gefährdung mit einer Sicherheitsanweisung verknüpft. Das P steht für Precaution, deutsch Vorsicht.
R-Sätze	Die R-Sätze sind der Vorgänger der H-Sätze (bis 1. Juni 2007). Sie beschreiben ebenfalls Gefährdungen. R steht für Risiko.
S-Sätze	Die S-Sätze sind die Vorgänger der P-Sätze. Sie geben Sicherheitshinweise im Umgang mit Chemikalien (Ähnlich wie bei den P- und H-Sätzen). S steht für Sicherheit.

Tabelle 1.1.: Wichtige Informationen und deren Bedeutung in einem Sicherheitsdatenblatt

lauben diese Großunternehmen ihren Abnehmern und Handelspartnern nicht, oder nur gegen Zahlung großer Geldbeträge, auf diese Datenbanken zuzugreifen. Somit werden kleine und mittelständische Betriebe gezwungen entweder die Informationen aus SDB händisch abzutippen oder in Papierform zu verwalten.
Der Betreiber der Plattform *euSDB* hat in seiner Freizeit durch jahrelange Heimarbeit inzwischen 440 000 Datensätze zu SDB digitalisiert und archiviert. Diese Daten bietet er über seine Onlineplattform *euSDB* im Internet käuflich an. Allerdings standen die SDB bisher nur als Portable Document Format (PDF)-Datei zum Herunterladen zur Verfügung. Zur automatischen Verarbeitung ist es jedoch sehr sinnvoll, die Daten nicht nur als PDF-Datei, sondern auch als maschinenlesbare Datenpakete bereitzustellen. Auf diese Weise müssen die Informationen nicht mehr von Hand abgetippt werden, sondern können automatisiert verarbeitet werden. *euSDB* ermöglicht es seinen Kunden langfristig Kosten zu sparen, indem es die Daten einmalig und zentral aus den SDB extrahiert. Je mehr Kunden *euSDB* hat, desto günstiger wird der Zugriff für jeden Einzelnen auf die SDB.

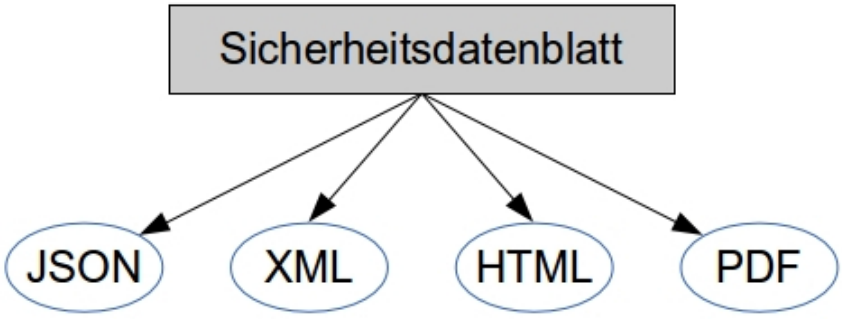

Abbildung 1.1.: Ein Sicherheitsdatenblatt bzw. die daraus extrahierten Informationen können in den folgenden Datenformaten ausgegeben werden: JSON, XML, HTML, PDF.

Die Plattform *euSDB* der Firma *euSDB* ist langsam und über mehrere Jahre gewachsen. Der Entwickler ist kein Informatiker, was sich in verschiedenen Details der alten Plattform *euSDB* wiederspiegelt, z.B. die prototypische REST-konforme application programming interface (API) ist nicht REST-konform. Es ist geplant, die Plattform professionell, kommerziell und gewinnbringend zu betreiben. Daher bedarf es einer grundsätzlichen Überholung und Überarbeitung bzw. Neuentwicklung. Dabei legt der Betreiber auch großen Wert darauf, die Plattform wegen geringerer Fehleranfälligkeit und leichterer Wartbarkeit in einer anderen als

der bisher verwendeten Programmiersprache zu implementieren. Ebenfalls gilt es, die Plattform um eine REST-konforme API zu erweitern. Diese API soll neben Hypertext Markup Language (HTML) und PDF zukünftig auch die Ausgabeformate JavaScript Object Notation (JSON) und Extensible Markup Language (XML) unterstützen (siehe Abbildung 1.1).
Aus diesen Überlegungen ergibt sich die nachfolgende Aufgabenstellung.

1.3. Aufgabenstellung

Es ist eine Online-Plattform in der Programmiersprache *Python* mit Hilfe des *Django* -Frameworks zu programmieren. Diese soll neben der gesamten Funktionalität der bestehenden Plattform *euSDB* auch eine Suchfunktion, eine REST-konforme API und eine Administrationsoberfläche enthalten.
Die Funktionalität der bestehenden Online-Plattform umfasst eine Historisierung, eine Suchfunktion und eine Zugriffskontrolle (siehe Abbildung 1.2). Als erstes ist dieser Funktionsumfang nachzuprogrammieren. Dabei ist darauf zu achten, dass der Quellcode auch für Nichtinformatiker übersichtlich und nachvollziehbar bleibt. Ebenfalls soll die Plattform leicht wartbar sein. Das setzt eine schnelle und effiziente Fehlerbehebung, eine einfache Veränderung von Performanz- und anderen Attributen sowie eine schnelle Reaktionsfähigkeit auf äußere Einflüsse, z.B. Gesetzesänderungen, voraus.

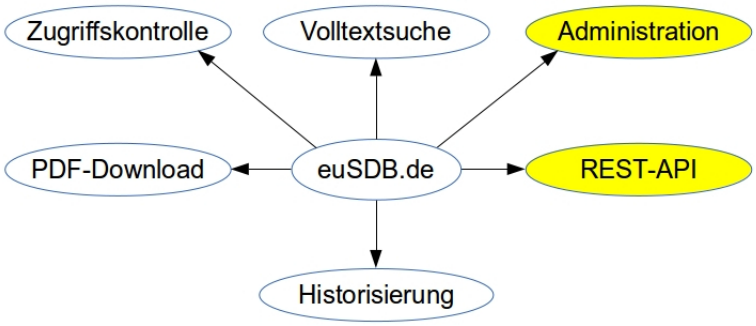

Abbildung 1.2.: Übersicht über den Funktionsumfang der bestehenden Plattform euSDB und die neu hinzugefügte REST-API und Administrationsoberfläche (gelb)

Die Neukonzeptionierung einer REST-konformen API ist ebenfalls durchzuführen.

Sie soll es ermöglichen, die in SDB enthaltenen Informationen zur automatischen Weiterverarbeitung bereitzustellen, für ausgewählte Nutzer eine Möglichkeit bieten, SDB der Datenbank hinzuzufügen oder bestehende Datensätze zu aktualisieren.
Die Administrationsoberfläche stellt ebenfalls eine Möglichkeit dar, Daten zu bearbeiten. Hier können im Gegensatz zur REST-konformen API nicht nur die SDB-Datensätze bearbeitet werden, sondern auch die Nutzerdaten der Plattformbenutzer, z.B. Anschrift der Hersteller, Mailadressen der Hersteller.

1.4. Einordnung in die Themenfelder der Informatik

Bei der Entwicklung der *euSDB* handelt es sich um Webentwicklung. Dabei werden die aus dem Bachelor-Studium der Informatik bekannten Themen wie Datenbanken, Internetprotokolle, Architektur- und Designmuster und Webseiten-Design genutzt. Die Themenfelder wurden um einige Themen, wie Volltextsuche und REST, erweitert. Die neuen Thematiken sind in den Kapiteln Grundlagen und Umsetzung dargelegt.
Prinzipiell geht es bei der Speicherung, Archivierung und Historisierung von SDB um das Thema des „Data-Warehousing“. Dabei gilt es Daten und Informationen unterschiedlicher Quellen in einheitlicher Form in einer Datenbank zu verwalten.
Das „Data-Warehouse“ wurde als Webanwendung umgesetzt. Webanwendungen haben das Ziel, im Internet oder einem Intranet zur Verfügung zu stehen und dort für möglichst viele Nutzer Daten und Informationen bereitzustellen.
In diesem Zusammenhang wurde auf die Einhaltung von Programmierparadigmen, z.B. Objektorientierung und lose Kopplung, geachtet. Sie sind auch Inhalt des Bachelor - Studiums in Informatik.

2. Grundlagen

Einfachheit von Programmstruktur, Datenverarbeitung, Wartung und Nutzung werden durch lose Kopplung, den Einsatz des *Django* -Frameworks, die Einhaltung der REST-Prinzipien und die Nutzung von *elasticsearch* angestrebt.

2.1. Lose Kopplung

Teilkomponenten einer Software agieren häufig miteinander. Lose Kopplung beschreibt den Zustand, dass die einzelnen Komponenten (Teile) der Software von einander möglichst unabhängig sind: „So wenige Schnittstellen wie möglich, so viele wie nötig.“ Ändert sich eine Komponente, jedoch nicht deren Schnittstelle, müssen die gekoppelten Komponenten nicht geändert werden.
Lose Kopplung erleichtert Portierbarkeit, Skalierbarkeit, Erweiterbarkeit und Wartbarkeit. Als Nachteil stellt sich in einigen Fällen eine schlechtere Performanz ein. Diese ist bei vielen Systemen, dank der hohen Systemleistung, aber vernachlässigbar und wird daher billigend in Kauf genommen.
Bei loser Kopplung sind globale Variablen oder öffentliche Attribute zu vermeiden, damit keine Softwarestati entstehen. Es erlaubt jeder Komponente, sofern sie kommunikationsbereit ist, ohne Vorbedingungen zu kommunizieren.

2.2. Die Programmiersprache Python

Python ist unter anderem eine prozedurale Programmiersprache. Allerdings werden auch andere Programmierparadigmen von ihr vereint. So ist es zum Beispiel auch möglich, objektorientiert [BELAGR-2009][1] oder funktional [RAIGRI-2009]zu pro-

[1] Diese Quelle findet sich auch unter folgendem Link als Webseite: http://openbook.galileocomputing.de/oop/oop_kapitel_01_001.htm#mj311565c5c46a3b13daf9951e408cac80

grammieren. Der vom Entwickler erzeugte Quellcode wird kompiliert. Der daraus entstandene Byte-Code kann dann vom Interpreter einer virtuellen *Python* -Maschine ausgeführt werden. Der Einsatz einer virtuellen Maschine macht Python plattformunabhängig.
Im Lieferumfang von Python ist eine sehr mächtige Standardbibliothek enthalten. Sie ermöglicht es dem Entwickler schnell und einfach strukturierte Anwendungen zu programmieren, die sehr komplexe Aufgaben verrichten und lösen können. Die Programmiersprache wird gerne für die Prototypentwicklung eingesetzt. Auf Grund der Einfachheit, mit der man Software erstellen kann, lassen sich die Prototypen schnell entwickeln. Auf diese Weise kann man schon früh Designfehler der späteren Software entdecken und beheben.
Python wird unter einer Python Software Foundation (PSF)-Lizenz vertrieben. Diese Lizenz ist weniger einschränkend als die General Public License (GPL)-Lizenz und erlaubt es, die *Python* -Software lizenzkostenfrei in große, kommerzielle Projekte einzubinden.

Ein herausragendes Merkmal von *Python* ist seine hohe Flexibilität. Daher wird es nicht nur als Programmiersprache in kleinen und großen Projekten, sondern auch serverseitig im Internet und als Skriptsprache in anderen Programmiersprachen eingesetzt. Die Wichtigkeit von *Python* im wachsenden Markt von Embedded-Systemen nimmt immer mehr zu.

```
1 def sum_1_5{self}:
2     sum = 0
3     for i in range(1,5):
4         sum=sum+i
5         print sum
6     print sum
```

Listing 2.1: Beispielfunktion in Python zur Berechnung der Summe aller Zahlen von 1 bis 5

Python ist mit dem Anspruch entwickelt worden, für den Entwickler möglichst einfach zu sein. Dies spiegelt sich schon in seiner Syntax wieder. Diese ist leicht und schnell zu erlernen. Dabei wird vor allem auf sauber formatierten Quellcode geachtet. Bei vielen Programmiersprachen, z.B. *Java* oder *C/C++* , wird ein Befehl durch eine „ ; “ abgeschlossen, der Inhalt einer Schleife mit „{. . . }“ eingefasst, in *Python* wird dies über Einrückung gesteuert (siehe Listing 2.1).
Der „range()“-Befehl liefert die Zahlen 1 bis 5 in Form einer Liste. Die „for“-Schleife durchläuft mit jedem Iterationsschritt diese Liste und legt die jeweilige

Zahl in die Variable i. Man sieht sofort, dass die einzelnen Befehle der Funktion nicht durch ein Semikolon abgeschlossen werden und es keine einfassenden geschweiften Klammern bei der „for“-Schleife gibt. Der „print“-Aufruf in der Schleife gibt bei jedem Durchlauf den aktuellen Inhalt der Variable „sum“ auf der Konsole aus. Die letzte Zeile Code gibt einmalig das Endergebnis nach dem vollständigen durchlaufen der Schleife aus. Der Compiler erwartet zum Einrücken eines hierarchisch untergeordnetem Quellcodes immer die gleiche Folge von Leerzeichen bzw. Tabulatoren. Ein unter Programmierern üblicher Standard ist das Einrücken mit vier Leerzeichen.
Mit Hilfe der sogenannten *Python* -API ist es möglich, die Standardbibliothek von *Python* flexibel zu erweitern. Auf diese Weise können auch Komponenten in anderen Programmiersprachen an *Python* angebunden werden, z.B. C/C++ Module, die im Gegensatz zu *Python* , sehr hardwarenah sind.
Die Popularität von *Python* nimmt immer mehr zu. Unter anderem setzt die *NASA* diese Programmiersprachen ein. Aber auch Internetnutzer kommen inzwischen tagtäglich mit Produkten auf Basis von *Python* in Kontakt, z.B. mit Youtube oder Google. Der Entwickler der Programmiersprache *Python* , Guido van Rossum, arbeitet heute selbst für *Google* .

2.3. Webentwicklung mit Django

„ *Django* ist ein hochwertiges *Python* Web Framework zur schnellen Entwicklung und für sauberes, praktisches Design.“ Dies ist die freie Übersetzung des Einleitungsatzes der englischsprachigen Internetseite des *Django* -Projektes: „Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design“ [DJSOFO-2013].
Wie man der Abbildung 2.1 entnehmen kann, ist das *Django* -Framework nach dem Model-Template-View (MTV)-Muster (nn der Abbildung 2.1 grau hinterlegt) aufgebaut. Die View verarbeitet den Request. Über das entsprechenden Model werden die Daten aus der Datenbank geholt. Die View verarbeitet dieses Model und dessen Daten mit Hilfe eines Templates und erzeugt eine Response, die dann über den Webserver an den Client weitergeleitet wird. Dieser Aufbau ist dem MVC-Muster nachempfunden.
Die MTV-Architektur, ein zentrales Designmuster der Informatik, entkoppelt Roh-

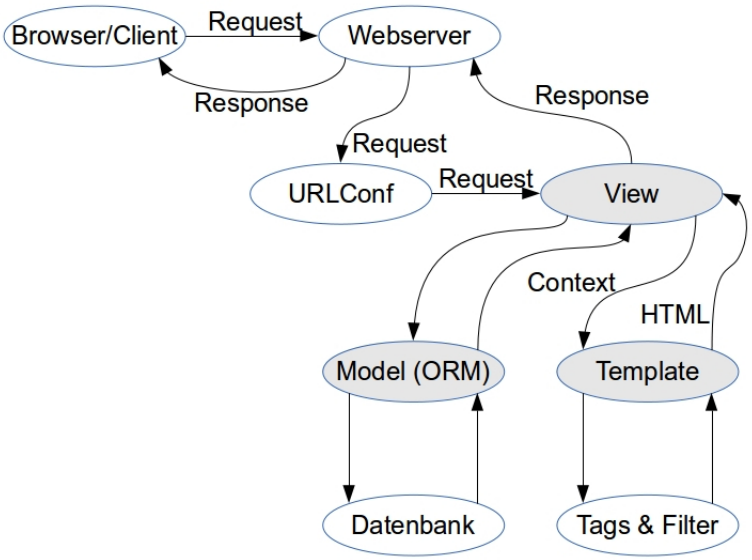

Abbildung 2.1.: Ablauf eines Request vom Client/Browser an den Server und die Response des Servers an den Client/Browser.

daten von deren Darstellung, sodass Datenbank und Darstellung der Daten unabhängig von einander ausgetauscht werden können. Die MTV-Architektur wird von dem *Django* -Framework hervorragend nachgebildet.
Die Komponenten des *Django* -Frameworks sind lose gekoppelt und nach dem Don't repeat yourself (DRY)-Prinzip aufgebaut. Es gilt zu verhindern, gleiche Funktionen mehrfach zu implementieren. Änderungen und Fehlerbehebungen müssen dann an genau einer Stelle im Quellcode durchgeführt werden. Der Code sollte an einer Stelle stehen, an der man ihn auch erwartet. Dies erleichtert die Verständlichkeit und Wartbarkeit des Programms.
Die Entwicklung einer neuen Webanwendung in *Django* erfolgt in einem sogenannten Projekt. Beim Erstellen eines Projektes erhält man das Projektverzeichnis mit dem gleichnamigen Konfigurationsverzeichnis (siehe Abbildung 2.2). Außerdem werden die für das Framework unbedingt notwendigen Dateien erzeugt. Tabelle 2.1 enthält die Kurzbeschreibungen der einzelnen Dateien im Projekt- und Konfigurationsvereichnis.
Das Projekt kann um viele Anwendungen, sogenannten Applications (APP), erweitert werden. Eine APP ist dabei in sich abgeschlossen und wird als eigener Ordner im Projektverzeichnis abgelegt. Hat man beispielsweise eine Webseite mit Gä-

Dateiname	Verzeichnis	Aufgabe
manage.py	Projekt	Das *Django* -Framework enthält eine Datei mit dem Namen „django-admin.py“. Mit dieser Datei werden fast alle administrativen Aufgaben über die Eingabeaufforderung gesteuert. In ihr ist auch ein Webserver zum Test des Projektes enthalten. Die „manage.py“ leitet die Befehle an die „django-admin.py“ weiter und legt vorher das Projekt auf den richtigen Pfad, sodass es korrekt vom Framework verarbeitet werden kann und setzt die Variable „DJANGO_SETTINGS_MODULE“ so, dass sie auf die „settings.py“ im Konfigurationsverzeichnis zeigt.
__inti__.py	Konfiguration	Diese Datei sorgt dafür, dass das Verzeichnis als „package“ (Paket) erkannt wird. Nur Dateien in einem package können in andere Dateien importiert werden.
settings.py	Konfiguration	Hier werden die projektweiten Einstellungen gespeichert. Zu diesen gehören zum Beispiel die Anbindung an eine Datenbank (siehe Anhang C.3, Zeile 15 - 24) oder auch Angaben zu statischen Verzeichnissen (siehe Anhang C.3, Zeile 54 - 60).
urls.py	Konfiguration	In dieser Datei werden die möglichen Uniform Resource Locator (URL) bzw URI mit Hilfe von regulären Ausdrücken abgelegt und mit den entsprechenden Views verknüpft. Dies erlaubt es elegant und leicht, höchst flexibele URI nach eigenen Wünschen und Bedürfnissen zu erzeugen. Es ist auch möglich, dynamische URI zu bauen.
wsgi.py	Konfiguration	Die Datei dient später beim Veröffentlichen des Projektes dazu, dass der Webserver eine Verbindung zu dem Projekt herstellen kann und dieses dann für Clients und Browser über das Internet zugreifbar ist.

Tabelle 2.1.: Beschreibung der Dateien im Projektverzeichnis und dem Konfigurationsverzeichnis und deren Aufgaben.

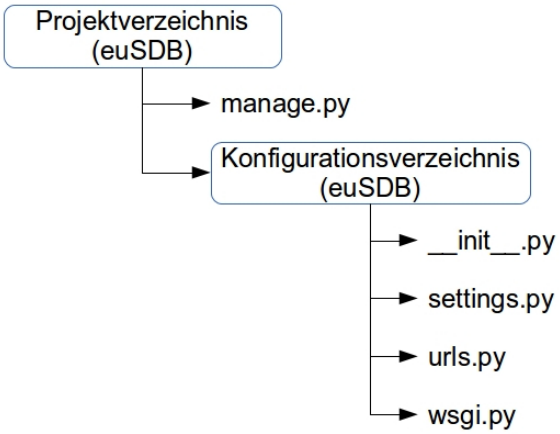

Abbildung 2.2.: Ordnerhierarchie direkt nach dem Erzeugen des Projektes euSDB.

stebuch und Benutzerverwaltung, dann stellt sowohl das Gästebuch als auch die Benutzerverwaltung jeweils eine eigene APP dar. Auf diese Weise kann man das Gästebuch einfach entfernen, ohne die Funktionalität der Benutzerverwaltung zu beeinflussen.
Der Aufbau des Projektes mit APP unterstützt die lose Kopplung der Anwendungen. Eine APP steht dem Projekt erst zur Verfügung, wenn man diese im Konfigurationsverzeichnis in der „settings.py“ des Projektes *euSDB* unter „INSTALLED_APPS“ bekannt gemacht hat (siehe Zeile 91 bis 103, Anhang C.3).
Eine von der „manage.py“ erzeugt APP besteht dabei immer aus folgenden Dateien: __init__.py, models.py, tests.py, views.py. Die Ordnerhierarchie können Sie auch der Abbildung 2.3 entnehmen. Die Rolle einzelner Dateien werden im Kapitel 4 erläutert.
Es kann sinnvoll sein, die entsprechende Struktur zu erweitern, z.B. um eine APP-interne „urls.py“. Diese müssen in der projektweiten „urls.py“ importiert werden. Auf diese Weise wird eine noch losere Kopplung der APP gewährleistet. Bei der Einbindung muss man nur den Import der „urls.py“ (eine Zeile) einfügen und die APP in der „settings.py“ bekannt machen, und nicht jede einzelne URL (mehrere Zeilen) um die App in das Gesamtprojekt einzubinden. Auch ist es mit einer APP-internen „urls.py“ einfacher eine APP in ein anderes Projekt zu portieren. Bei sehr Umfangreichen APP werden weitere Komponenten benutzt, wie z.B. Serialisierer oder Administratoren. Diese werden in APP-eigenen Dateien ausgelagert.

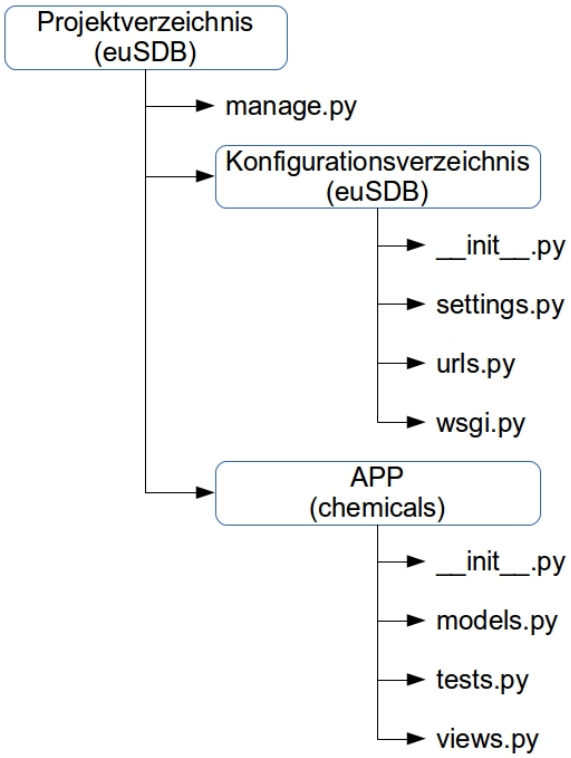

Abbildung 2.3.: Ordnerhierarchie nach dem Erzeugen der ersten APP

2.4. Das REST-Prinzip

REST, Representional State Transfer, ist ein Konzept, mit dem größtmögliche Einfachheit bei Webanwendungen gewährleistet werden soll. Die Forderungen des Konzepts sind seid der privaten und kommerziellen Nutzung des Internets bekannt und ergeben sich aus den Problemen bei der Entwicklung von großen Anwendungslandschaften und der firmenübergreifenden Integration von Daten:

- Lose Kopplung
- Interoperabilität
- Wiederverwendung
- Performance und Skalierbarkeit

Diese Probleme versucht man durch Einführung mehrerer Grundprinzipien bei der Erstellung von Webanwendungen zu lösen bzw. zu umgehen. Man fasst sie unter

dem Begriff REST zusammen:

- Ressourcen und deren unterschiedliche Repräsentationen
- Ressourcen mit eindeutiger Identifikation
- Verknüpfungen und Hypermedia
- Standardmethoden
- Statuslose Kommunikation

In den folgenden Abschnitten wird REST an Hand der Grundprinzipien erklärt.

2.4.1. Ressourcen und Repräsentationen

Ressourcen sind die Menge an Daten, die in einer Webanwendung serverseitig verarbeitet und verwaltet werden. Liegen die Ressorcen clientgerecht vor, spricht man von Repräsentationen der Ressourcen. Zum Beispiel gibt es bei einer Benutzerverwaltung für ein Internetkaufhaus die Ressource „User“. Sie wird definiert durch Parameter, im Beispiel durch „Name“ und „E-Mail“ als HTML-Repräsentation der Ressource „User“ (siehe Listing 2.2).

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4        <title>Benutzerverwaltung - Max Mustermann</title>
5      </head>
6      <body>
7          <h1>Max Mustermann</h1>
8          <p>E-Mail: max.mustermann@bsp.de</p>
9      </body>
10 </html>
```

Listing 2.2: HTML-Dokument für die Ausgabe eines Users

Je mehr Datenformate bzw. Repräsentationen beim Abruf von Ressourcen unterstützt werden, desto mehr Entwickler werden ein solches System nutzen, und umso leichter fällt es ihnen. Der Webbrowser ist wohl der weltweit bekannteste Client für Webanwendungen. Er ist in der Lage, HTML-Daten graphisch aufzubereiten, mit Hilfe von Skripten ist er in der Lage auch andere Datenformate darzustellen.
Anhang C.1 zeigt die „User“-Ressource als JSON-Repräsentation, Anhang C.2 als XML-Repräsentation.

2.4.2. Eindeutige Identifikation

Es hat sich als praktisch erwiesen, Ressourcen eindeutige Bezeichner zu geben. Im Web hat sich für die Vergabe solcher Bezeichner ein einheitliches Konzept entwickelt. Die einheitlichen Bezeichner werden als URI bezeichnet. Mit Hilfe einer URI kann eine Ressource weltweit eindeutig identifiziert werden.
Der Vorteil von einem weltweit anerkannten und genutzten Schema zur Vergabe von URI liegt darin, dass man kein eigenes erfinden muss, stattdessen ein bereits sehr gut erprobtes verwenden kann, das sehr gut skaliert und für fast jeden verständlich ist.

```
1 http://example.com/customers/1234
2 http://example.com/orders/2007/10/776654
3 http://example.com/products
```

Listing 2.3: Beispiel URI eines Bestellsystems [STETIL-2009a]

URI enthalten alle wesentlichen Elemente zur eindeutigen Identifizierung einer Ressource oder einer Menge von Ressourcen (Beispiele siehe Listing 2.3).
URL beschreibt einen Speicherort auf einem Server. URI identifizieren Ressourcen weltweit eindeutig. Der Oberbegriff URI beschreibt auch jede URL. Im allgemeinen Sprachgebrauch wird zunehmend URI statt URL verwendet.

2.4.3. Hypermedia

Hypermedia ist die Verallgemeinerung des aus HTML bekannten Hypertext. Das effiziente und sinnvolle Verlinken von Ressourcen und deren Funktionen wird im Listing 2.4 beispielhaft gezeigt und nachfolgend erklärt. Die vorgestellte Repräsentation zeigt einen reduzierten Datensatz eines Herstellers. Zeile 2 gibt den Namen eines Herstellers zurück. Zeile 3 enthält einen Link auf sich selbst. Dieser kann sehr nützlich sein, wenn die Repräsentation maschinell verarbeitet wird, oder der Link an andere Personen oder Funktionen weitergegeben werden muss. Zeile 4 führt dann zu einer Liste mit allen geführten Chemikalien des Herstellers. name": "BASF",
ürl": "http://localhost:8000/producers/01/.json",
chemicals": "http://localhost:8000/producers/basf/chemicals/.json",

```
1 "name": "BASF",
2 "url": "http://localhost:8000/producers/01/.json",
3 "chemicals": "http://localhost:8000/producers/basf/chemicals/.json",
```

Listing 2.4: Beispiel für ein JSON Dokument eines Herstellers

Diese Art der Verlinkung ermöglicht es, fast intuitiv durch die Webanwendung zu navigieren. Selbst ein Programm, welches diese Daten auswertet, ist sehr leicht zu programmieren. Der Entwickler eines solchen Programms muss keine Kenntnis über das serverseitige System haben, da alle für ihn wichtigen URI durch die Repräsentation selbst zur Verfügung gestellt werden. Er kann alle Daten, den Zusammenhang in dem diese zueinander stehen, den Repräsentationen entnehmen.
Dem Nutzer können auch die Links auf zur Verfügung stehende Funktionen übergeben werden, z.B. Registrieren eines neuen Nutzers. Dadurch werden Zustände ohne den Einsatz von Zustandsvariablen beschrieben. Für den Client können sich jedoch Zustände und Folgezustände völlig unabhängig vom Server ergeben.
Die Repräsentationen sollen so miteinander verknüpft werden, dass ein intuitives und leichtes Navigieren zwischen den Ressourcen möglich ist. Entwickler von Clients können schneller und effizienter programmieren, da sie nicht zwingend das serverseitige System kennen müssen.

2.4.4. Standardmethoden

Das Standardprotokoll im Internet ist HTTP. In der HTTP-Spezifikation sind unter anderem sechs Standardmethoden bekannt: GET, POST, PUT, DELETE, HEAD und OPTIONS. Die Methoden GET, PUT und DELETE sind laut Spezifikation immer idempotent. Das bedeutet, sollte man mit einer URI wiederholt eine idempotente Standardmethode aufrufen, ist der Zustand des Systems unverändert. Wird beispielsweise zweimal ein GET auf eine gleiche URI aufgerufen, bekommt man jeweils das gleiche Ergebnis zurück. Ruft man mehrfach die DELETE-Funktion auf ein bestimmtes Objekt auf, darf der Befehl nur einmal ausgeführt werden. Der Aufruf der DELETE-Funktion auf ein bereits gelöschtes Objekt darf nicht zum Löschen anderer Objekte führen.
POST ist laut der Spezifikation eine nicht idempotente Methode. Das bedeutet es gibt keine Garantien wie das System bei einem mehrmaligen POST auf eine URI reagiert. Die Reaktion des Systems hängt von der Implementierung des Systementwicklers ab.
Die vier Standardmethoden GET, POST, PUT, DELETE sind auch für ein weiteres Grundprinzip der Webentwicklung von großer Bedeutung. Beim Verwalten und Bearbeiten von Ressourcen wird mittels des Create-Read-Update-Delete (CRUD)-Prinzips eine transparente Implementierung vorgeschrieben. Der Nutzer kann sich

im allgemeinen sicher sein, dass die idempotenten Methoden auch so implementiert wurden und ihm die vier bzw. sechs Standardmethoden bereitgestellt werden, wenn er mit einer REST-konformen Anwendung arbeitet.
Eine natürliche und intuitive Handhabung einer Webanwendung mit REST-konformer API setzt voraus, dass alle auf der Website bereitgestellten Funktionen auch in der REST-konformen API zur Verfügung gestellt werden.

2.4.5. Statuslose Kommunikation

Bei der statuslosen Kommunikation wird der Status des Servers vom Client gehalten oder als Ressourcenstatus übergeben. Einen Ressourcenstatus lässt sich anhand eines Onlineshops erklären: Der Status des Kunden wird im Warenkorb gespeichert und zu einer eigenen Ressource. Der Ressourcenstatus gibt dem Kunden die Möglichkeit, Lesezeichen zu setzen oder Links darauf zu verschicken.
Der Verzicht auf einen expliziten Sitzungsstatus hat vor allem Vorteile, wenn es darum geht, eine Webanwendung skalierbar zu machen. Durch die lose Kopplung von Client und Server können aufeinanderfolgende Anfragen eines Clients von verschiedenen Serverinstanzen verarbeitet werden. Während ein Client arbeitet, ist der Status des Servers für ihn nicht relevant, z.B. kann der Server gewartet oder repariert werden. Mittels eines Links werden erst nach erfolgreichem Ausführen der clientseitigen Aufgabe, die Ergebnisse dem Server mitgeteilt. Hat der Server einen Client mit Informationen versorgt und die Kommunikation ist abgeschlossen, geht der Server in seinen Grundzustand zurück und kann andere Anfragen von anderen Clients bearbeiten.
Eine statuslose Kommunikation impliziert ebenfalls eine lose Kopplung. Es gibt eine klar definierte Schnittstelle, über die die Daten ausgestauscht werden. Der Server ist völlig unabhängig vom Client und muss keine clientspezifischen Informationen kennen. Er kann jederzeit mit allen verarbeitbaren Daten umgehen, ohne dafür in einem zusätzlichen Status befindlich zu sein.
Dieses REST-Prinzip wird genutzt, um eine einfache Kommunikation zwischen Server und Client sicherzustellen. Die Trennung der Verantwortlichkeiten für Daten und Operationen sind ein weiteres Ziel. Für den Datenaustausch sollten Standardformate, wie XML, JSON usw., verwendet werden. Die Kommunikation sollte ebenfalls auf Standardprotokollen basieren, z.B. HTTP. Auf Grund dieser Standards ist der Informationsaustausch zwischen Client und Server leicht zu realisieren.

2.5. Volltextsuche mit elasticsearch

elasticsearch [ELASBV-2013] ist ein auf *Apache Lucene* [THAPSO-2013] basierender Suchserver zur Volltextsuche. Er wurde mit dem Anspruch entwickelt, hoch skalierbar und dennoch leicht bedienbar zu sein. Dabei sollte ein Funktionsumfang, wie man ihn von *Apache Lucene* kennt, zur Verfügung gestellt werden, ohne dabei die Komplexität und die schwierige Bedienung von *Apache Lucene* auf *elasticsearch* zu übertragen. Die Entwickler achteten außerdem darauf, dass das System in Form von Modulen aufgebaut wird, sodass es leicht um neue Module erweitert werden kann. Ein im Softwarepaket enthaltenes API-Modul erlaubt es, den Suchserver zu konfigurieren und zu bedienen.
elasticsearch speichert Daten schemalos und dokumentenorientiert. Dabei werden die zu indizierenden Daten als JSON-Dokument an *elasticsearch* übergeben.

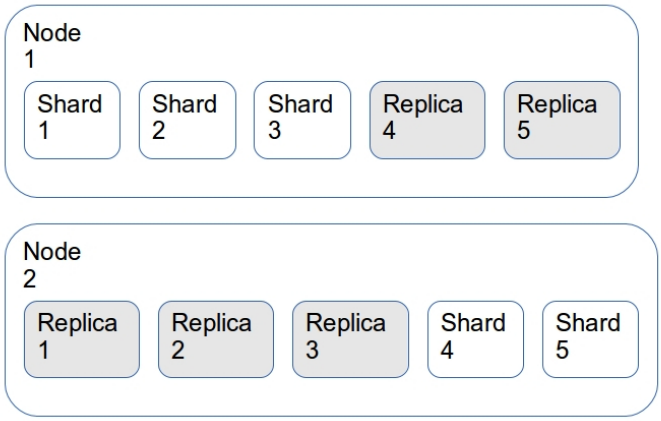

Abbildung 2.4.: Aufbau der Nodes mit Shards und Replicas (grau) in einem System mit zwei Nodes. Die Replicas sind die Kopien der Shards eines anderen Nodes und werden im Falle eines Node-Ausfalls zu Shards.

Wie *elasticsearch* intern und pysikalisch aufgebaut ist, zeigt Abbildung 2.4. Jeder Server auf dem *elasticsearch* läuft, stellt einen sogenannten Node dar. Mehrere Nodes bilden ein Cluster. Jeder Node besteht aus *Apache Lucene* -Instanzen, den Shards. Die Shards dienen der Lastverteilung und der parallelen Verarbeitung, wenn mehrere Anfragen vom Server verarbeitet werden. Bei mehreren Nodes findet eine möglichst gleichmäßige Verteilung der Shards auf diese statt, jedoch werden einzelne Shards nicht auf mehrere Nodes aufgeteilt. Ein logischer Index, die zur Suche bereitgestellten Daten, kann dabei auf mehreren Shards und mehreren Nodes ver-

teilt abgelegt werden.
Um eine höhere Ausfallsicherheit zu gewährleisten, enthält ein Node in einem Verbund von Nodes Replicas. Replicas sind die Kopien der Shards die ein Node nicht enthält. Graphik 2.4 zeigt dies ebenfalls. Fällt nun ein Node aus, werden die Replicas wieder zu Shards und das System steht mit allen Daten und Dokumenten zur Verfügung.
Der Aufbau des Systems durch Nodes, Shards und Replicas dient dazu, höchste Performanz, Effizienz und Ausfallsicherheit zu gewährleisten. Aufgrund der angesprochenen modularen Aufbauweise und der Unterstützung von Clustern, kann *elasticsearch* sowohl in verteilten Systemen als auch in nicht verteilten Systemen effektiv genutzt werden. Entwickler können sehr leicht weitere Module realisieren, die den Funktionsumfang von *elasticsearch* erweitern oder verbessern.

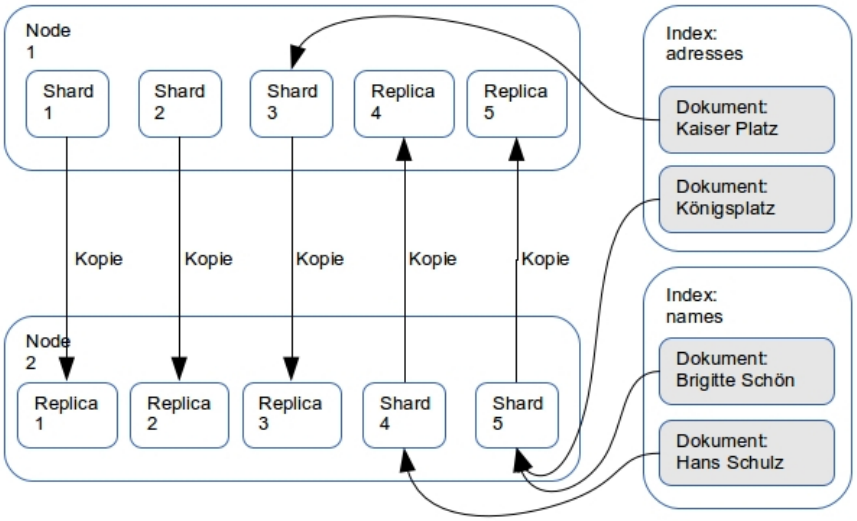

Abbildung 2.5.: Der Index adresses verteilt sich auf Node 1 und Node 2. Eines der Dokumente (grau) des Indexes adresses liegt auf Node 1 in Shard 3 (Kaiser Platz) und auf Node 2 in Shard 5 (Königsplatz). Der Index names liegt auf Node 2 in den Shards 4 (Brigitte Schön) und 5 (Hans Schulz)

Die logische Einteilung ist nicht an die physikalische gebunden. Ein Index kann zum Speichern von Dokumenten mehrere Shards, sogar auf verschiedenen Nodes nutzen, andererseits können auf einem Shard auch Dokumente mehrerer Indizes gespeichert werden (siehe Abbildung 2.5).

Die Logik hinter der Volltextsuche mit *elasticsearch* ist dabei sehr einfach. Vom System wird eine Tabelle angelegt (Beispieltabelle 2.2), die in einer Spalte alle Wörter (Token) enthält, die in den indizierten Dokumenten vorkommen. In einer zweiten Spalte werden die Wörter mit allen Dokumenten, in denen das Wort vorkommt, verknüpft. Sucht man jetzt ein Wort oder einen Wortteil, wird nur diese eine Tabelle durchsucht und man bekommt sehr schnell alle enthaltenden Dokumente. Sucht man nach „Mr“ erhält man nur Dokument 1. Sucht man nach „Müller“ ist das Ergebnis nur Dokument 3. Bei der Verknüpfung von Tokens werden bei der Suchabfrage diese mit der logischen UND-Operation verknüpft. Die Bearbeitung erfolgt serverseitig, genaugenommen bitweise. Das hat zur Folge, dass die Suchabfrage „Mr. Müller“ keinen Treffer ergibt. Die Suchanfrage „Michael Jackson“ liefert als Suchergebnis die Treffer in Dokument 2 und 3. Dokument 1 wird nicht berücksichtigt.

Token	**Dokument(e)**
Mr	1
Jackson	1+2+3
verstarb	1
vor	1
wenigen	1
Jahren	1
Michael	2+3
war	2+3
ein	2+3
berühmter	2
Künstler	2
Müller	3
Fan	3
von	3

Tabelle 2.2.: Tabelle, die den Wörtern (Tokens) Dokumente zuweist. Die Tabelle enthält unter anderem die drei Sätze: „Mr. Jackson verstarb vor wenigen Jahren.“(Dokument 1) „Michael Jackson war ein berühmter Künstler.“(Dokument 2) „Michael Müller war ein Fan von Micheal Jackson.“(Dokument 3)

3. Konzept

3.1. Datenbank

Für die Realisierung von *euSDB* ist der Einsatz von zwei Datenbanksystemen geplant, eines für die gesamte Datenverwaltung (Kapitel 3.1.1), das andere für die schnelle Abwicklung von Suchanfragen an das System (Kapitel 3.1.2). Dies bietet mehrere Vorteile: Eine angemessene Datenverwaltung, die eine übersichtliche Datenstruktur bietet und mit Hilfe von Structured Query Language (SQL)-Befehlen, auch ohne Nutzung des *Django* -Frameworks, gepflegt und administriert werden kann und eine Suchdatenbank, die die modernen Anforderungen nach Datenintegrität, Skalierbarkeit und Erweiterbarkeit in vollem Umfang abdeckt.

3.1.1. Datenbank zur Verwaltung von Sicherheitsdatenblättern

Für die Datenverwaltung und Speicherung kommt ein *MySQL* -Datenbanksystem zum Einsatz. *MySQL* wird von *Oracle* weiterentwickelt und vermarktet. Es gibt eine kostenlose und eine kostenpflichtige Enterpriseversion. [ORACOR-2013]
Neben einer Nutzer- und Rechteverwaltung, wird in der Datenbank auch eine Logtabelle geführt. Diese speichert jede Änderung am Datenbestand. Die Kernaufgabe besteht in der Verwaltung der SDB.
Ein Sicherheitsdatenblatt wird durch die in Kapitel 1.1 vorgestellten Kennzeichnungen (siehe Tabelle 1.1) und Hinweise beschrieben. Diese Kennzeichnungen sind für alle Chemikalien einheitlich und erfahren in absehbarer Zeit keine Änderungen. Die H-, P-, R- und S-Sätze, GHS-, EU- und Hazard-Gefahrenkennungen werden mehrfach verwendet und liegen in jeweils einer eigenen Tabelle in der Datenbank vor (siehe Graphik 3.1).
Wird eine Anweisung oder Gefahrenkennung mit einer Chemikalie verknüpft, ge-

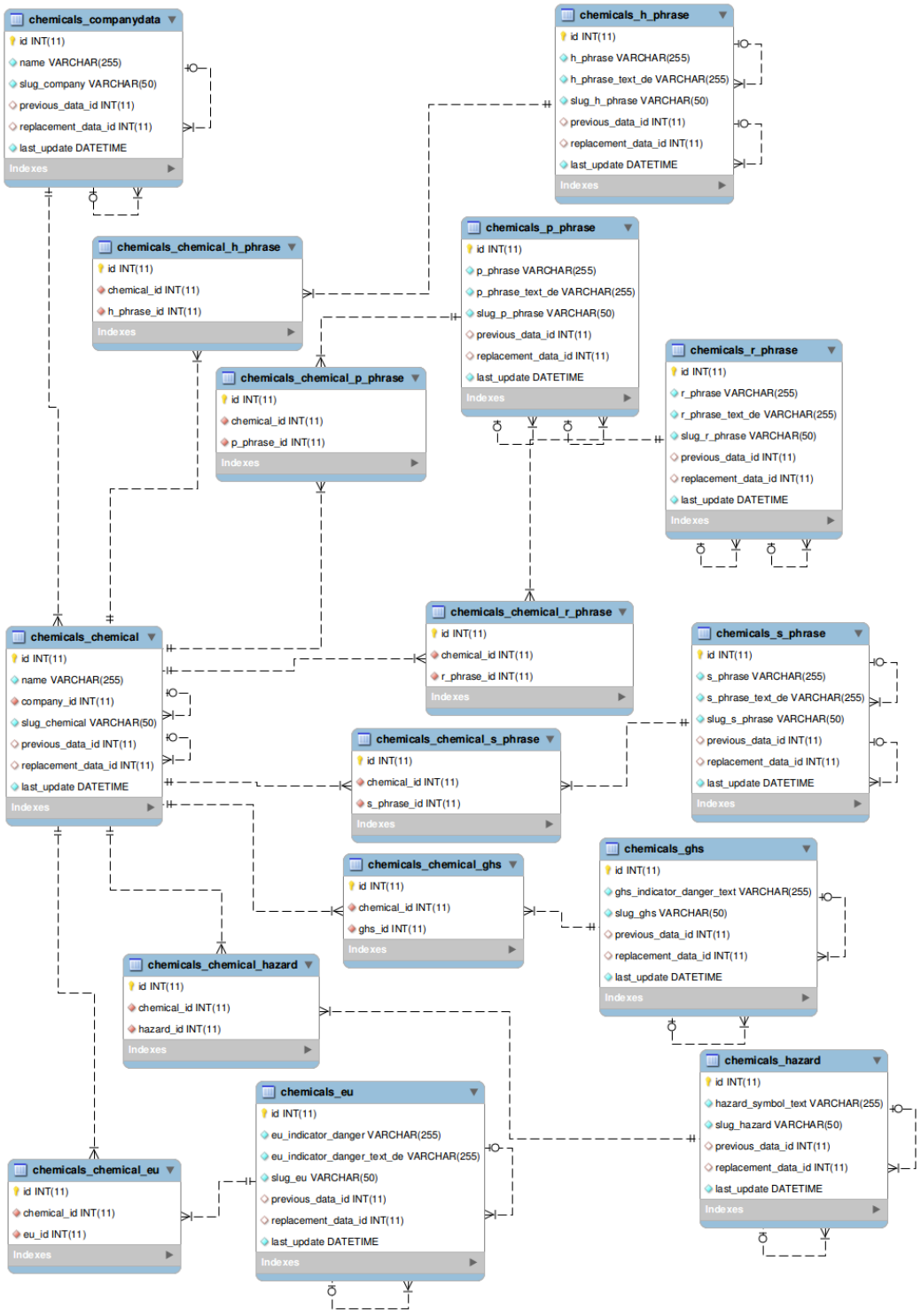

Abbildung 3.1.: Die Graphik zeigt die Datenbankstruktur (Entwurf) von euSDB. Aus Gründen der Übersichtlichkeit zeigen die Tabellen nur wenige Felder. Im Anhang Din A3/2

schieht dies über eine Matchingtabelle. Eine Solche Matchingtabelle enthält dann einen Fremdschlüssel (Foreign-Key) der Chemikalie und eine mit der Chemikalie verknüpfte Anweisung bzw. Gefahreneinstufung. Bei der Verknüpfung von Chemikalien und Anweisungen bzw. Gefahreneinstufung handelt es sich um sogenannte Many-To-Many Beziehungen.
Many-To-Many Beziehungen bedeuten in diesem Fall, dass eine Anweisung mit mehreren Chemikalien verknüpft und eine Chemikalie mit mehreren Anweisungen verknüpft sein kann.
Bei der herstellenden Firma verhält es sich anders. Ein SDB wird immer nur von einer Firma für die von ihr hergestellte Chemikalie herausgegeben. Ein SDB einer Chemikalie hat genau einen Hersteller. In diesem Fall ist es möglich direkt den Fremdschlüssel in der Tabelle der Chemikalie abzulegen. SDB gleicher Chemikalien verschiedener Hersteller sind nicht identisch.
Jeder Hersteller gibt leicht unterschiedliche Daten zu Chemikalien an. So ist es theoretisch möglich, dass sich die Angaben zum Siedepunkt unterscheiden. Aus diesem Grund werden daher gleiche Chemikalien unterschiedlicher Hersteller als unterschiedliche Chemikalien betrachtet. Auch geben die Hersteller gerne unterschiedliche Einsatz- und Verwendungszwecke an. Zur Veranschaulichung ein Beispiel: Ein Farbenhersteller stellt selbst Aceton her. Es wird zur Verdünnung von Farben eingesetzt. Verkauft der Farbenhersteller sein Aceton, wird er im SDB angeben, dass es zur Verdünnung von Farben zu verwenden ist. Ein Reinigungsmittelhersteller wird vermutlich in seinem SDB angeben, dass es sich um ein Reinigungsmittel handelt und dafür einzusetzen ist. Dies macht es ebenfalls sinnvoll, gleiche Chemikalien verschiedener Hersteller als unterschiedliche Chemikalien zu speichern.
Die Datenbank ist in ihrer Struktur sehr einfach gehalten. Das macht das Migrieren und Warten der Daten sehr leicht. Der Einsatz eines Standarddatenbanksystems gewährleistet einen langfristigen Support auf vielen Plattformen. Ein ständiger Umzug von einem Datenbanksystem zu einem anderen ist bei Plattformwechseln nicht nötig.
Eine Chemikalie selbst wird dabei durch ungefähr 60 Parameter (Siedetemperatur, Dichte, molare Masse, ...) gekennzeichnet. Hinzukommen noch die Angaben zur Einstufung der Gefahren und sich aus den Gefahren ergebende Sicherheits- und Arbeitsanweisungen. Jeder dieser Parameter stellt ein eigenes Feld in der Tabelle der Chemikalien dar.

Hinzu kommt noch die Möglichkeit einer Versionierung. Häufig verändern sich im Laufe der Zeit die Angaben auf einem SDB. Jede Veränderung geht mit der Herausgabe einer neuen Version eines SDB einher. Um die Informationen, die in einem SDB aufgeführt sind, lange nachhalten zu können, existiert nicht nur für die Chemikalie selbst eine Versionierung, sondern auch für alle mit ihr verknüpften Komponenten. Sollte sich nun eine Einstufung oder ein Satz ändern, kann man trotzdem nachschlagen, wie ein früheres SDB aufgebaut war.

3.1.2. Datenbank zur Realisierung einer effektiven Volltextsuche

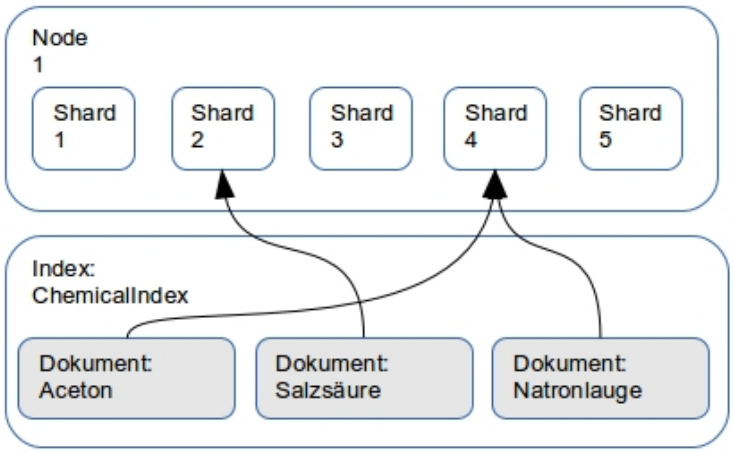

Abbildung 3.2.: Entwurf des Aufbaus von elasticsearch. Es gibt einen Node mit fünf Shards. Der Index ChemicalIndex verteilt sich auf diesen Node und die Shards. Beispielhaft sind hier drei Dokumente auf Shard 2 und 4 verteilt worden

Das System, das zur Suche eingesetzt wird, heißt *elasticsearch* . Das System ist zunächst auf einen Node mit fünf Shards begrenzt. Für jede Chemikalie wird in dem Index „ChemicalIndex“ ein Dokument abgelegt. Dieses enthält die möglichen Identifizierer. Diese sollen nach internationalen Vorgaben möglichst eindeutige Bezeichner (z.B. Name der Chemikalie, internationale Formeln, internationale Identifizierungsnummern) sein. Nur nach solchen zu suchen ist sinnvoll. Wird ein neuer Datensatz einer Chemikalie erstellt oder der einer bestehenden Chemikalie bearbei-

tet, sollen die Änderungen automatisch auch auf die in *elasticsearch* abgelegten Daten übertragen werden. Auf diese Weise hält sich der Verwaltungsaufwand, die Suchfunktion betreffend, im laufenden Betrieb in Grenzen.
Später ist es möglich den Suchindex noch um Einträge zu erweitern oder neue Suchindizes zu erstellen. Auch kann das gesamte Suchsystem zu einem verteilten System umgebaut werden.

3.2. Aufbau des Djangoprojekts

Die Grundstruktur eines Djangoprojektes ist im Kapitel 2.3 Grundlagen beschrieben. Das System setzt sich aus den folgenden Grundfunktionen zusammen: (1) Beschreibung von Chemikalien mit ihren SDB, (2) die Suche nach Chemikalien und SDB und (3) eine Benutzer- und Rechteverwaltung (siehe Abbildung 3.3).

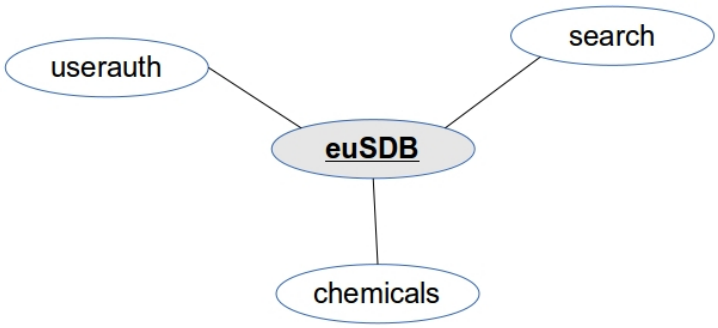

Abbildung 3.3.: Das Projekt euSDB besteht aus den drei Kernmodulen chemicals, search und userauth.

(1) Soll dabei dem Kunden nicht nur die chemischen Daten und SDB zur Verfügung stellen, sondern regelt die Versionierung und langfristige Speicherung von SDB. Dazu nutzt diese Funktion eine REST-konforme API, die nicht nur den menschlichen manuellen, sondern auch den automatisierten Zugriff durch Programme und Maschinen erlaubt. Diese Funktion vermag es, SDB in den Formaten JSON, XML, HTML, PDF auszugeben.
(2) *elasticserach* ermöglicht die gezielte Suche nach einzelnen SDB. Nicht immer führen die angegebenen Werte, nach denen gesucht wird, zu eindeutigen Ergebnissen. Aus diesem Grund wird immer eine Liste mit den möglichen SDB zurückge-

geben. Diese kann kein Element, ein Element oder mehrere Elemente enthalten.
(3) Die Administrationsoberfläche erlaubt es dem Administrator, Rechte an die Benutzer zu vergeben, Benutzer in Gruppen zusammenzufassen und diesen Gruppen Rechte zuzuweisen, damit auf die SDB zugegriffen werden kann.
Durch die hohe Funktionalität von *Python* und den Aufbau des Projektes mit Hilfe des *Django* -Frameworks wird eine langfristige Nutzung der Plattform angestrebt.

3.3. Aufbau der REST-konformen Schnittstelle

Die REST-konforme API soll vier Ausgabeformate haben: JSON, XML, HTML und PDF (siehe Abbildung 1.1). Dabei soll das Ausgabeformat sowohl im HTTP-Header, als auch als Parameter an die URI angehängt werden können.

```
1 https://www.euSDB.de/chemicals/?format=json
2 https://www.euSDB.de/chemicals/?format=xml
3 curl -X GET -H "Accept: application/json" https://www.euSDB.de/chemicals/
4 curl -X GET -H "Accept: application/xml" https://www.euSDB.de/chemicals/
```

Listing 3.1: Beispielanfragen, bei denen das Ausgabeformat jeweils header- und parametergesteuert ist

Auf diese Weise soll nach der Eingabe von Zeile 1 des Listing 3.1 im Browser ein JSON-Dokument angezeigt werden, nach Eingabe von Zeile 2 des Listing 3.1 ein XML-Dokument. Ebenfalls soll man das Ausgabeformat über den „Accept“-Wert des HTTP-Headers steuern können. Ziel ist, das Zeile 1 und 3 des Listings 3.1 das gleiche Ergebnis ergeben, genau wie die Zeilen 2 und 4. Zeile 3 und 4 sind Abfragen von einem Unix-System aus dem Programm *curl* .
Zunächst ist es geplant, nur ein Abrufen von SDB mittels der HTTP-GET-Standardmethode bereitzustellen. Dennoch ist bei dem Design der REST-konformen API darauf zu achten, dass sich diese leicht erweitern lässt. Die Antworten des Servers sind in UCS Transformation Format 8 Bit (UTF-8) codiert.
Als Einstiegspunkt soll es zwei Möglichkeiten geben. Zum einen den vordefinierten (API-Wurzel) und zum anderen soll es auch mit Hilfe jedes Suchergebnisses möglich sein, einen Einstieg und somit einen erfolgreichen Zugriff auf die Daten zu erhalten. Mittels Verlinkungen (Hypermedia) der einzelnen Ressourcen untereinander soll es dann möglich sein, auf alle relevanten Daten, die im System zu einem SDB gespeichert sind, zugreifen zu können. Abgesehen von der Wurzel hat jedes Objekt ein Vorgänger-Objekt. Dieses ist ebenfalls verlinkt. Mit diesen Links kann

man durch die gesamte Datenstruktur, die zur Verfügung steht, navigieren. Abbildung 3.4 veranschaulicht, wie die Verlinkungen und das Navigieren funktionieren.

```
1 https://www.euSDB.de/chemicals/?page=2&per_page=50
```

Listing 3.2: Beispielrequest zur Steuerung des Paging (Seite 2 mit 50 Einträgen)

Beim Anzeigen von Listen werden bei mehr als 20 Objekten Seiten ausgegeben (Paging). Dann erhält man, sofern sie verhanden sind, einen Link auf die vorangegangen und die nachfolgenden Seiten. Mittels Parameterübergabe lässt sich steuern, wie viele Objekte pro Seite verlinkt werden sollen, und auch das direkte Springen auf eine Seite ist möglich. Listing 3.2 veranschaulicht dies anhand eines Beispiels. Der generelle Aufbau der URI ist hierarchisch (Abbildung 3.4). Es dient dazu, die Treffermöglichkeiten immer weiter einzuschränken, bis man bei der gewünschten Ressource angekommen ist. Von dort aus sind dann alle dazugehörigen Anweisungen, Hinweise und Symbole schnell und direkt erreichbar.

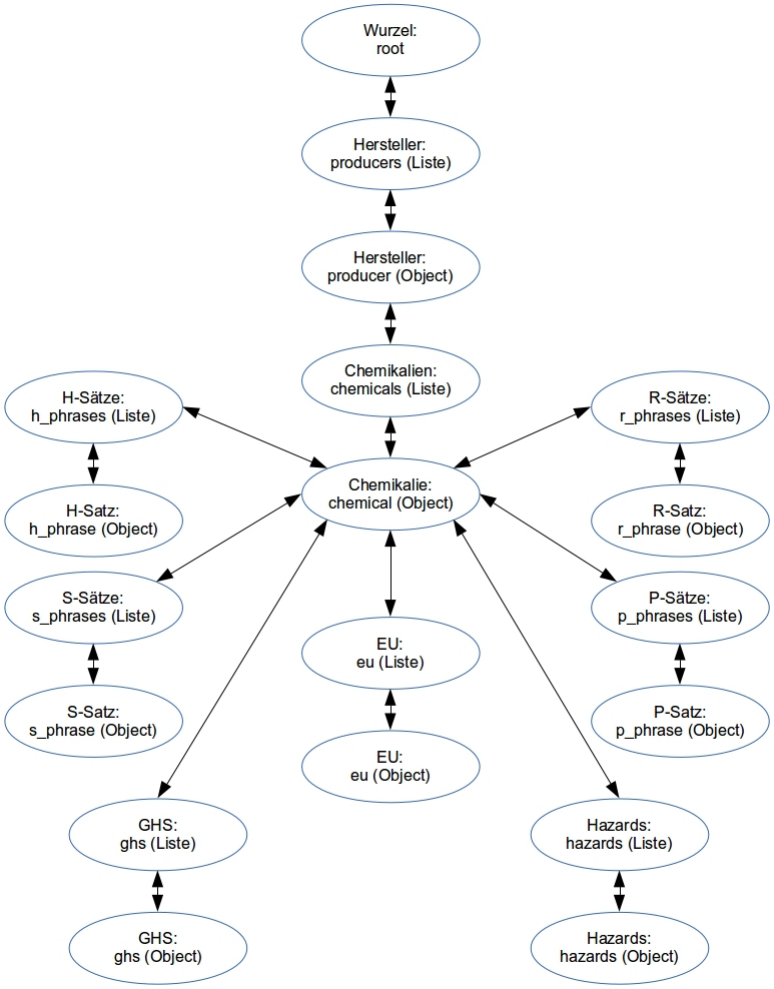

Abbildung 3.4.: Die Darstellung zeigt wie mittels der REST-konforme API durch das System euSDB navigiert werden soll. Dabei spielt es keine Rolle, ob man durch die Suche direkt bei einer Chemikalie startet, oder von der API-Wurzel (root) ausgeht. (Darstellung nicht UML-konform)

4. Realisierung

4.1. Datenbank

Das *Django* -Framework nimmt dem Entwickler einige Aufgaben ab. So kann es unter anderem auch eine Datenbank gemäß den Einstellungen und Klassen erzeugen, die bei der Umsetzung verwendet und eingesetzt werden. Die Einstellungen, die die Kommunikation mit der Datenbank ermöglichen, werden in der „settings.py“ (siehe Anhang C.3) im Konfigurationsverzeichnis vorgenommen. Die Zentrale Einheit zur Verwaltung und Nutzung einer Datenbank ist in jeder *Django* -APP die „models.py“; in Kapitel 2.3 wurde bereits die Ordnerhierarchie und der Aufbau von *Django* -Projekten vorgestellt.

```
1  DATABASES = {
2      'default': {
3          'ENGINE': 'django.db.backends.mysql',
4          'NAME': 'euSDB',
5          'USER': 'eusdb',
6          'PASSWORD': 'XXX',
7          'HOST': '',
8          'PORT': '',
9      }
10 }
```

Listing 4.1: Einstellungen eines Django-Projektes zur Ansteuerung eines MySQL-Datenbanksystems

Die Einträge in der „settings.py“, die hier auch noch mal in Listing 4.1 aufgeführt sind, erlauben den Zugriff auf ein lokales *MySQL* -Datenbanksystem. In Zeile 3 wird der Typ der verwendeten Datenbank und die Komponente ausgewählt, die die Verbindung zur Datenbank aufbauen wird. In diesem Fall ist eine Standardklasse des *Django* -Framework angegeben. In Zeile 4 steht der Name der Datenbank, danach folgen der Benutzername und dessen Passwort in Klartext (hier in Form von „XXX“ dargestellt). "HOST“ und „PORT“ werden je nach Datenbanksystem automatisch eingefügt und können in diesem Fall frei gelassen werden.

```
1  class Chemical(models.Model):
2      """Chemical model."""
3      id = models.AutoField(u'id', primary_key=True, editable=False)
4      name = models.CharField(u'Name', max_length=255, default='')
5      company = models.ForeignKey(CompanyData,
6          verbose_name=u'Hersteller-Firma', related_name='chemicals')
7      hazard = models.ManyToManyField(Hazard, verbose_name=u'Hazard Symbol')
8      ghs = models.ManyToManyField(Ghs, verbose_name=u'GHS Gefahren Kennung')
9      eu = models.ManyToManyField(Eu, verbose_name=u'EU Gefahren Kennung')
10     p_phrase = models.ManyToManyField(P_phrase, verbose_name=u'P-Satz')
11     s_phrase = models.ManyToManyField(S_phrase, verbose_name=u'S-Satz')
12     r_phrase = models.ManyToManyField(R_phrase, verbose_name=u'R-Satz')
13     h_phrase = models.ManyToManyField(H_phrase, verbose_name=u'H-Satz')
14     slug_chemical = models.SlugField(unique=True)
15     previous_data = models.OneToOneField('self', blank=True,
16         null=True, related_name='Vorgaenger_von')
17     replacement_data = models.OneToOneField('self', blank=True,
18         null=True, related_name='Nachfolger_von', editable=False)
19     last_update = models.DateTimeField(editable=False)
```

Listing 4.2: Ausschnitt aus der Datei „models.py“ der chemicals-APP, die in der Datenbank durch die Tabelle „chemicals_chemicals“ und allen dazugehörigen Matchingtabellen dargestellt wird

In der Datei „models.py“ einer jeden APP werden die einzelnen Tabellen in Form von Klassen abgelegt. Die Variablen dieser Klassen findet man in der Datenbank als Felder der entsprechenden Tabellen wieder. Das Listing 4.2 zeigt die Klasse, die die Tabelle „chemicals_chemicals“ mit all ihren Matchingtabellen auf andere Tabellen darstellt. Das erste „chemicals“ steht dabei für die APP, der die Tabelle zuzuordnen ist, das zweite für die eigentliche Komponente. Es existieren noch die Tabellen „chemicals_p_phrase“, „chemicals_ghs“, ... Die Matchingtabellen werden nach der APP, der Hauptkomponente und der zu verknüpfenden Komponente benannt. Es entstehen Tabellennamen wie „chemicals_chemicals_p_phrase“, „chemicals_chemicals_ghs“, ...

Man erkennt an Listing 4.2 sehr gut, wie die Komponenten mit Chemikalien verknüpft werden. Der Hersteller wird durch das Aufnehmen seines Schlüssels als Fremdschlüssel mit der Tabelle „chemicals_chemicals“ verknüpft (siehe Zeilen 5 und 6). Andere Komponenten werden in Form von Many-To-Many Beziehungen verknüpft (siehe Zeilen 7 bis 13).

Dabei existieren im *Django* -Framework schon Basisklassen, für die wichtigsten Datentypen und für das Verknüpfen von Komponenten. Diese sind in der Klasse „Models“ gespeichert und können verwendet werden. Es ist auch möglich, eigene

Feldtypen zu definieren, die das bestehende Angebot an Datentypen erweitern.
Hat man die „Models“ definiert und eine Verbindung zu einem Datenbanksystem erstellt, kann das *Django* -Framework über die Eingabe der Befehle aus Listing 4.3 selbstständig eine Datenbank mit den Tabellen erzeugen. Zeile 1 überprüft zunächst, ob Syntaxfehler vorliegen. Zeile 2 erstellt die SQL-Befehle und speichert sie zwischen. Zeile 3 sendet die SQL-Befehle an das Datenbanksystem und sorgt somit für das direkte Erstellen der Datenbank.

```
1 python manage.py validate
2 python manage.py sqlall chemicals
3 python manage.py syncdb
```

Listing 4.3: Befehlfolge zur Eingabe in ein Linux-Terminal direkt auf dem Server zur Erstellung einer Datenbank in einem Datenbanksystem mit allen Tabellen der APP chemicals

Die Datei mit allen Model-Klassen ist im Anhang C.4 angefügt. Betrachtet man diese Datei, stellt man fest, dass diese eine Subklasse enthält. In dieser wird bestimmt, wie der Name des bzw. der Objekte ist. Ebenfalls gibt es Funktionen, die die Handhabung der Objekte erleichtern sollen. Mit der in den Zeilen 19 und 20 von Listing C.4 definierten Methode „__unicode__“ wird eine Standardtextausgabe eines Objektes erzeugt. Sie kann sowohl im Terminal, als auch auf Webseiten verwendet werden.
Da es manchmal nicht reicht, Objekte einfach in einer Datenbank abzulegen, wird in der Methode „save“ (z.B. Listing C.4, Zeile 22 bis 29) die Speichermethode überladen. Es wird ein Erstellungsdatum gesetzt, wenn das Objekt vom Datenbanksystem noch keine „ID“ zugewiesen bekommen hat. Ist eine „ID“ schon verhanden und wird nur eine Aktualisierung bzw. Veränderung an einem Datensatz durchgeführt, ist es nicht mehr notwendig das Erstellungsdatum zu verändern (z.B. Listing C.4, Zeile 23 und 24).

4.2. REST-Schnittstelle

4.2.1. Aufbau der URI

Der Aufbau der URI wird hierarchisch durchgeführt. Das bedeutet, dass nicht alle Funktionen direkt über die Wurzel der API aufgerufen werden, sondern sich aus gewissen Zusammenhängen zwischen den Ressourcen ergeben.

```
1  urlpatterns = patterns('chemicals.views',
2    url(r'^$', 'api_root', name = 'api_root'),
3    url(r'^producers/$', views.ProducerList.as_view(), name = 'producer_list'),
4    url(r'^producers/(?P<id_comp>[-\w]+)/$', views.ProducerDetail.as_view(),
5        name = 'companydata-detail'),
6    url(r'^producers/(?P<id_comp>[-\w]+)/chemicals/$',
7        views.ChemicalList.as_view(), name = 'chemical_list'),
8    url(r'^producers/(?P<id_comp>[-\w]+)/chemicals/(?P<id_chem>[-\w]+)/$',
9        views.ChemicalDetail.as_view(), name = 'chemical-detail'),
10   url(r'^producers/(?P<id_comp>[-\w]+)/chemicals/(?P<id_chem>[-\w]+)
11       /hazards/$',
12       views.HazardList.as_view(), name = 'hazard_list'),
13   url(r'^producers/(?P<id_comp>[-\w]+)/chemicals/(?P<id_chem>[-\w]+)
14       /hazards/(?P<id_haz>[-\w]+)/$', views.HazardDetail.as_view(),
15       name = 'hazard-detail'),
16   url(r'^producers/(?P<id_comp>[-\w]+)/chemicals/(?P<id_chem>[-\w]+)
17       /p_phrases/$',
18       views.P_phraseList.as_view(), name = 'p_phrase_list'),
19   url(r'^producers/(?P<id_comp>[-\w]+)/chemicals/(?P<id_chem>[-\w]+)
20       /p_phrases/(?P<id_p_ph>[-\w]+)/$', views.P_phraseDetail.as_view(),
21       name = 'p_phrase-detail'),
22    )
```

Listing 4.4: Ein Ausschnitt aus der Datei „urls.py“ der APP „chemicals“. Anhand dieser lässt sich der Aufbau der hierarchischen URI nachvollziehen

Die im Listing 4.4 gezeigten URI-Muster realisieren diese Hierarchie. Direkt unter der Wurzel befinden sich die „producer“. Hierarchisch unter einem „producer“ befinden sich die mit ihm verknüpften „chemicals“, unterhalb der „chemicals“ befinden sich dann die weiteren Ressourcen.
Der Aufbau eines Befehls zur Verarbeitung von Anfragen ist dabei fast immer identisch. Zeile 8 bis 10 des Listings 4.4 ist wie folgt aufgebaut: „url(...)“ erzeugt ein neues Muster für eine URI; dann folgt ein regulärer Ausdruck, der den genauen Aufbau der URI definiert. Die Variablen sind daran zu erkennen, dass sie von „<...>“ eingefasst sind. Im Beispiel sind folgende Variablen angegeben: „<id_comp>“ und „<id_chem>“, es sind die ID der entsprechenden Ressourcen.
Nach dem folgenden Komma erfolgt der Aufruf der verarbeitenden „view“. Views können objektorientiert oder funktional programmiert werden. Die im Listing 4.4 in Zeile 2 gezeigte URI zur API-Wurzel ist durch einen Funktionsaufruf, die anderen durch den Aufrufen der objektorientierten „view“-Klasse realisiert worden.
Der Aufruf der Detailansicht einer einzigen Ressource ist über die Verwendung der ID eindeutig geregelt. Es gibt keine Mehrfachverwendung der ID und es ist der Eindeutigkeit im Bezug auf URI genüge getan.

4.2.2. Serialisierung von Ressourcen

```
1  class CompanyData_List_Serializer(serializers.HyperlinkedModelSerializer):
2      url = serializers.HyperlinkedIdentityField(lookup_field='slug_company',
3          view_name='companydata-detail',)
4      class Meta:
5          model = CompanyData
6          fields = ('name', 'url')
7      def restore_object(self, attrs, instance=None):
8          return CompanyData(**attrs)
9
10 class PaginatedCompanyDataListSerializer(PaginationSerializer):
11     class Meta:
12         object_serializer_class = CompanyData_List_Serializer
13
14 class CompanyData_Detail_Serializer(serializers.HyperlinkedModelSerializer):
15     url = serializers.HyperlinkedIdentityField(lookup_field='slug_company',
16         view_name='companydata-detail',)
17     chemicals = serializers.SerializerMethodField('get_producer_chemicals')
18     def get_producer_chemicals(self, obj):
19         return reverse('chemical_list', args=[obj.slug_company,
20             self.context['format']], request=self.context['request'])
21     class Meta:
22         model = CompanyData
23         fields = ('url', 'name', 'chemicals', 'previous_data',
24             'replacement_data')
25     def restore_object(self, attrs, instance=None):
26         return CompanyData(**attrs)
```

Listing 4.5: Serialisierer-Klassen zur Serialisierung von Chemikalien. Sowohl für die Detailansicht, als auch für die Listenansicht (Ressource: Producer)

Ressourcen werden zur Verarbeitung durch Views serialisiert. Sie liegen dann nicht mehr als strukturierter Datensatz vor, sondern in einer sequenziellen Form.
Zur Serialisierung von Ressourcen wird die chemicals-APP um die Datei „serializer.py“ erweitert. Diese Datei enthält alle Serialisierer-Klassen, die zur Serialisierung von Daten notwendig sind. Listing 4.5 zeigt die Serialisierer-Klassen zur Serialisierung der Chemikalien sowohl in Listenform, als auch für die Detailansicht.
An Hand des Beispiels (Listing 4.5) sieht man, dass nur die wichtigsten Informationen einer Ressource zur Darstellung für die Listenübersicht (Zeile 1 bis 8) übergeben werden, Name und URI (Zeile 6).
Erst in der Detailansicht (Zeile 14 bis 26) werden mehr Informationen zur Verfügung gestellt. Die Ausgabe der Informationen kann ebenfalls angegeben werden (Zeile 23 und 24).

4.2.3. Funktionsweise der Views

```
class ProducerList(APIView):
    renderer_classes = (TemplateHTMLRenderer, UnicodeJSONRenderer, XMLRenderer)
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)
    def get(self, request, format=None):
        companys = CompanyData.objects.all().filter(replacement_data=None)
        paginate_by = 3
        paginator = Paginator(companys, paginate_by)
        page = request.QUERY_PARAMS.get('page')
        try:
            obj = paginator.page(page)
        except PageNotAnInteger:
            obj = paginator.page(1)
        except EmptyPage:
            obj = paginator.page(paginator.num_pages)
        serializer = PaginatedCompanyDataListSerializer(obj,
            context ={'request': request, 'format': format})
        if (format == 'html') | (format == None):
            return Response({'object_list': obj},
                template_name='chemicals/producer_list.html')
        return Response(serializer.data)

class ProducerDetail(APIView):
    renderer_classes = (TemplateHTMLRenderer, UnicodeJSONRenderer, XMLRenderer)
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)
    def get_object(self, slug_company):
        try :
            return CompanyData.objects.get(slug_company=slug_company)
        except CompanyData.DoesNotExist:
            raise Http404
    def get(self, request, slug_company, format=None):
        companys = self.get_object(slug_company)
        serializer = CompanyData_Detail_Serializer(companys,
            context ={'request': request, 'format': format})
        if (format == 'html') | (format == None):
            return Response(serializer.data,
                template_name='chemicals/producer_detail.html')
        return Response(serializer.data)
```

Listing 4.6: View-Klassen zur Bearbeitung von Anfragen an das System und zum Erstellen der entsprechenden Antworten (Ressource: Producer)

Die Views erzeugen die Repräsentationen, die der Kunde von *euSDB* auf seinem Client oder Webbrowser zur Verfügung gestellt bekommt. Dabei wird unterschieden, ob es eine Standardausgabe (JSON oder XML) oder eine graphisch aufbereitete Ausgabe (HTML oder PDF) ist.

Im ersten Schritt werden der View Renderer-Klassen zur Verfügung gestellt. Dabei handelt es sich um Standardklassen und Funktionen, die in der Lage sind, Daten in das entsprechende Format zu überführen. Darauf folgen die Authentifizierungs- und Rechtevergabe-Klassen. Das ist notwendig, um zu verhindern, dass Unbefugte Zugriff auf das System bekommen.
Zum Implementieren der Standardmethoden des HTTP sind für Entwickler mehrere Methodennamen („get“, „post“, „head“, . . .) reserviert. In dem Beispiel (Listing 4.6) sind die „get“-Methoden (Zeile 5 bis 21 und 32 bis 39) realisiert. Neben den für die Standardmethoden reservierten Funktionen können auch weitere Funktionen in die Klassen implementiert werden. Zeile 27 bis 31 des Listings 4.6 rufen die Daten aus der Datenbank ab, ist das nicht erfolgreich, erzeugen sie eine Standard-Fehlermeldung.
Die Standardausgaben müssen nicht graphisch aufbereitet werden. Sie haben den Zweck, möglichst kompakt die Daten dem Kunden zur maschinellen Weiterverarbeitung zur Verfügung zu stellen. Dabei ist ein Seitenlayout nicht von Bedeutung und würde bei einer schnellen Verarbeitung stören. Der Datensatz wird sowohl in der Listenübersicht (Listing 4.6 Zeile 1 bis 21), als auch in der Detailansicht (Listing 4.6 Zeile 23 bis 39) jeweils serialisiert. Dabei werden die im vorigen Kapitel 4.2.2 vorgestellten Serialisierer verwendet (Zeilen 16 und 17 bzw. 34 und 35).
Nachdem die Daten serialisiert wurden, werden sie aufbereitet und dem Kunden als Antwort zugeschickt. Dabei werden die Daten in den Formaten JSON und XML direkt an den Kunden versandt. Sollen die Daten jedoch als HTML-Dokument ausgegeben werden, generiert die View mit Hilfe eines Templates das Antwortdokument (z.B. Zeile 18 bis 20).

4.2.4. Hypermedia zwischen den Ressourcen

Die Grundlagen über Hypermedia sind im Kapitel 2.4.3 zu finden. Hier wird beschrieben, wie die Verlinkungen zwischen den Ressourcen realisiert wurden.
Die Models wurden, wie im Abschnitt 4.2.2 beschrieben, serialisiert, sodass sie von den Models verarbeitet und verwendet werden können. Im Beispiel (Listing 4.5) wird in den Zeilen 2 und 3 die Variable „url“ definiert. Diese dient dazu, in der Listenansicht zu jeder Instanz einer Ressource einen Link zu erzeugen.

```
1     serializers.HyperlinkedIdentityField(lookup_field='slug_company',
2         view_name='companydata-detail',)
```

Listing 4.7: Definition der Variablen „url“ aus dem Serialisierer einer Liste

Der Befehl aus Listing 4.7 funktioniert wie folgt: Das „HyperlinkedIdentityField“ erzeugt einen Link auf eine Ressource. Dabei wird als Identifizierungsfeld (lookup_field) ein ID-Feld (id_. . .) der zu verlinkenden Ressource verwendet. Am Schluss wird dann die aufzurufende View angegeben.

```
1     url = serializers.HyperlinkedIdentityField(lookup_field='id_comp',
2         view_name='companydata-detail',)
3     chemicals = serializers.SerializerMethodField('get_producer_chemicals')

5     def get_producer_chemicals(self, obj):
6         return reverse('chemical_list', args=[obj.id_comp,
7             self.context['format']], request=self.context['request'])
```

Listing 4.8: Definition der Variablen „url“ und „chemicals“ aus dem Serialisierer der Detailansicht eines Herstellers

Bei der Vergabe von Links in der Detailansicht gibt es zwei Links. Einer verweist auf sich selbst („url“) und einer, im Beispiel (Listing 4.8) auf die Chemikalien („chemicals“). Der Link auf sich selbet kann clientseitig als Lesezeichen oder zur Weitergabe verwendet werden. Die Variablen werden auch im Serialisierer initialisiert (Listing 4.8, Zeile 1 bis 3). Es ist notwendig, den Link auf eine andere Ressource selbst zu erstellen, da das System eigentlich keine hierarchischen URI unterstützt. Dies übernimmt die Funktion aus den Zeilen 5 bis 7. Hier wird mittels des „reverse“ Befehls eine Link erstellt. Diesem werden das vom Kunden angeforderte Datenformat und der Hersteller an die aufzurufende View („chemical_list“) übergeben. Am Ende entsteht eine absolute URI.
Ohne den genauen Aufbau des Systems zu kennen, kann sich der Kunde über die Verlinkungen und bereits bekannten Links durch das System bewegen.

4.2.5. Paging in den Listenansichten

Das Paging, also das Einteilen von Listen in mehrere Pakete (Seiten), wird sowohl von der View, als auch dem Serialisierer gesteuert. Dabei teilt der Serialisierer den Datensatz auf (Listing 4.9).

```
1 class PaginatedCompanyDataListSerializer(PaginationSerializer):
2     class Meta:
3         object_serializer_class = CompanyData_List_Serializer
```

Listing 4.9: Einteilen der Seiten durch den Serialisierer (Ressource: Producer)

Die von dem „CompanyData_List_Serializer“ gelieferten Daten werden durch den „PaginationSerializer“, einer Standardmethode, in Seiten eingeteilt.

In der View kann man angeben, wie viele Einträge pro Seite zu führen sind, und welche Seite gerade zurückgeliefert werden soll.

```
1         paginate_by = 25
2         paginator = Paginator(companys, paginate_by)
3         page = request.QUERY_PARAMS.get('page')
4         try:
5             obj = paginator.page(page)
6         except PageNotAnInteger:
7             obj = paginator.page(1)
8         except EmptyPage:
9             obj = paginator.page(paginator.num_pages)
```

Listing 4.10: Steuerung und Validierung des Paging in der View (Ressource: Producer)

In Listing 4.10 wird in Zeile 1 ein Standardwert für die Anzahl an Einträgen pro Seite angegeben. Danach werden die Daten von der Datenbank angefordert und unter „page“ gespeichert. In den Zeilen 4 bis 9 wird dann überprüft, ob die Anfrage korrekt ist und beantwortet werden kann; z.B. ob für die Seitenzahl ein Integer-Wert angegeben oder eine leere Seite angefordert wurde. Sollte eine ungültige Anfrage gestellt werden, wird die erste Seite als Antwort bereitgestellt.

4.3. Suchfunktion

```
1  class ChemicalIndex(indexes.SearchIndex, indexes.Indexable):
2      text = indexes.EdgeNgramField(document=True, use_template=True)
3      name = indexes.CharField(model_attr='name')
4      cas = indexes.CharField(model_attr='cas')
5      ec_number = indexes.CharField(model_attr='ec_number')
6      ec_index_number = indexes.CharField(model_attr='ec_index_number')
7      hill_formula = indexes.CharField(model_attr='hill_formula')
8      pub_date = indexes.DateTimeField(model_attr='pub_date')
9
10     def get_model(self):
11         return Chemical
12
13     def index_queryset(self, using=None):
14         return self.get_model().objects.
15             filter(pub_date__lte=datetime.datetime.now())
```

Listing 4.11: Die Datei „search_index.py“ verwaltet den Index und die zu indizierenden Daten

Die Suchfunktion ist in einer eigenen APP ausgelagert. Die „search“-APP stellt die Verbindung zu *elasticsearch* her und fügt die Daten automatisch in die Suchdatenbank ein.

Die „search“-APP enthält neben den automatisch von *Django* erstellten Dateien die Datei „search_index.py“ (Listing 4.11).
In den Zeilen 2 bis 8 werden die zu indizierenden Werte einer Chemikalie definiert. Es handelt sich in den meisten Fällen um Textfelder (CharField). Dabei werden Werte verwendet, die es erlauben, Chemikalien möglichst eindeutig zu identifizieren. Auf diese Weise wird die Suchdatenbank schlank und schnell gehalten.
Die Klasse „ChemicalIndex“ hat eine Funktion, die es ihr erlaubt auf Chemikalien zuzugreifen (Zeile 10 und 11). Diese Funktion ist notwendig, damit man schnell Zugriff auf zu aktualisierende Datensätze bekommt. Das Abrufen von Suchergebnissen wird mit Standardmethoden der Bibliothek „haystack“ durchgeführt.
Die „haystack“-Bibliothek ist eine speziell zur Anbindung von *elasticsearch* an *Django* -Projekte entwickelte Programmbibliothek. Sie verfügt über eine Grundfunktionalität, die der Entwickler original nutzen kann, oder sie gegebenenfalls anpassen muss.
Zu der Suchfunktion wird ein Template erstellt. Dieses enthält ein Suchfeld, um die Anfragen einzugeben. In einer Textdatei wird angegeben, in welcher Reihenfolge und welche Felder durchsucht werden. Die View und auch der Aufbau des Formulars muss nicht durch den Entwickler, es sei denn, er möchte ein individuelles Formular haben, definiert werden. Die „haystack“-Bibliothek enthält die notwendigen Views und Formulare und stellt sie dem Entwickler zur Verfügung.

5. Ergebnis

5.1. Vergleich der Datenbanksysteme zum Speichern der Datensätze (Access und MySQL)

Der Wechsel des Datenbanksystems bringt viele Vorteile mit sich. *Mircososft - Access* ist ein Datenbanksystem, dass nur unter dem Betriebssystem *Windows* läuft. Somit sind Nutzer des Datenbanksystems an die Betriebssysteme von *Microsoft* gebunden. Der Erwerb einer Nutzungslizenz von *Mircososft - Access* kostet zur Zeit etwa 360 Euro. Der Umfang an Daten den *Access* verwalten kann ist ebenfalls eingeschränkt: Eine Datenbank, genauso wie einzelne Tabellen, können nicht größer als 2 GB werden. Schnell übersteigen PDF die Größe von mehreren MB. Daher könnte eine Datenbank sehr schnell an ihre Grenzen kommen und dem Datenaufkommen nicht gewachsen sein.
Mit dem Verwenden des Datenbanksystems *MySQL* erhalten die Betreiber von *euSDB* eine plattformunabhängige und professionelle Lösung, die ihn nicht dazu zwingen weitere Produkte eines einzigen Herstellers zu benutzen. Die Möglichkeit das System günstig zu beschaffen und die weitverbreitete Nutzung erlauben einen effizienten Betrieb. Die Grenzen der Datenmengen, die verwaltet werden können, sind wesentlich weiter gesteckt als bei einer *Mircososft - Access* Datenbank. Datenbanken unterliegen bei *MySQL* nur einer physischen Grenze in Form von Festplattengröße bzw. der Schranke, die das verwendete Betriebssystem beim Verwalten des Speichers setzt. Tabellen können bis zu 256 TB groß werden und bieten somit eine vielfach höhere Speicherkapazität als eine *Mircososft - Access* Datenbank.

	Access	**MySQL**
Maximale Datenbankgröße	2 GB	unbegrenzt
Maximale Tabellengröße	2 GB	256 TB
Maximale „Blob“-Größe	1 GB	4 GB

Tabelle 5.1.: Vergleich von zwei Datenbanksystemen

Die Analyse der alten Datenbank ergab ebenfalls, dass diese von Grund auf strukturell neu entworfen werden muss. Grundprinzipien wie Normalisierung und Wiederverwendung von Daten wurden nicht genutzt. Außerdem gab es keine Verknüpfung von Ressourcen, sondern die Gefahren- und Sicherheitssätze waren einfach in einem Textfeld in der Tabelle abgelegt worden.
Die neue Datenstruktur erlaubt ein schnelles Verstehen der Zusammenhänge, auch für nicht fachkundige Betreuer der Software *euSDB* . Durch die Einhaltung strenger Normalisierung werden Redundanzen an Daten vermieden. Dies ermöglicht auch bei großen Datenmengen speicherplatzschonend schnellen Zugriff auf die Daten.
Die Datenstruktur kann problemlos erweitert und verändert werden. Somit lassen sich zum Beispiel Gesetzesänderungen und Auflagen für SDB schnell und ohne großen Aufwand auf die Datenstruktur übertragen.
Der Betreiber ist nicht mehr an ein Betriebssystem gebunden und kann sich frei entscheiden, welche Plattform er als Serverumgebung nutzen möchte. Dies sind nur einige Vorteile, die das Datenbanksystem *MySQL* mit sich bringt.

5.2. Das PHP-Slim-Framework im Vergleich mit dem Python-Django-Framework

Das Vorgängersystem des neu zu entwickelnden Systems ist mit dem *Slim* [1]-Framwork geschrieben. Dieses Framework ist in der Skript-Sprache Hypertext Preprocessor (PHP)[2] geschrieben.
Bei dem *Slim* -Framework handelt es sich um ein sogenanntes „leichtgewichtiges“ Framework. Leichtgewichtige Frameworks dienen dazu, den Entwicklern

[1] Die offizielle Webseite des Slim-Framework ist unter folgender URL zu finden: `http://www.slimframework.com/` zu finden
[2] Die offizielle Webseite von PHP ist unter folgender URL zu finden: `http://php.net/`. Früher stand die Abkürzung PHP für Personal Homepage Tools

einen schnellen Einstieg in eine Materie zu erleichtern. Schnell sind die funktionalen Grenzen eines solchen Frameworks erreicht.
Dies ist auch der Grund, warum der Entwickler und Betreiber von *euSDB* nicht mehr weiter mit dem *Slim* -Framework arbeiten möchte. Die Anforderungen die an die Software *euSDB* gestellt werden, können nicht mehr ausschließlich durch das *Slim* -Framework bewerkstelligt werden. Das Erweitern des *Slim* -Frameworks durch den Entwickler ist sehr zeitintensiv und fordert ein hohes Maß an Fachkunde. Da die Entwicklung und der Betrieb bisher und auch in Zukunft durch einen Nicht-Informatiker durchgeführt werden soll, ist es notwendig ein leicht zu erlernendes, allumfassendes und dadurch mächtiges Framework zu verwenden.
Die Wahl fiel durch den Betreiber auf das *Django* -Framework. Dieses bringt einen sehr großen Funktionsumfang mit sich. Dieser Funktionsumfang lässt sich durch weitere Module und Bibliotheken leicht erweitern. Somit stellt es eine Lösung dar, die leicht zu erlernen ist, und dennoch den Betreiber nicht in der Funktionalität einschränkt. Die Entwickler des *Django* -Framework stellen ebenfalls eine ausführliche Dokumentation zur Verfügung. Diese wird mit jedem Update des Frameworks ebenfalls aktualisiert.
Mit Hilfe der Dokumentation des Frameworks und der Dokumentation der entwickelten Software kann der Betreiber sich einen Überblick verschaffen, die Nutzung des Frameworks erlernen und die Plattform seinen eigenen Bedürfnissen anpassen.

5.3. Suchen mit elasticsearch

elasticsearch ist eine praktische Unterstützung für das effektive Suchen. Der Einsatz ist leicht zu erlernen und das System kann schnell und ohne großen Zeit- oder Arbeitsaufwand an ein *Django* -Projekt angebunden werden. Die Plattform *euSDB* ist ein relativ kleines Web-Projekt, wenn man es z.B. mit Youtube vergleicht. *elasticsearch* ist eine professionelle Lösung, die erheblich mehr Datenaufkommen durchsuchen kann, als es bei *euSDB* momentan der Fall ist.
Die Möglichkeiten des Einsatzes auf einer verteilten Serverarchitektur, als Cluster wird nicht benötigt und nicht genutzt. Auch die damit einhergehende redundante Datensicherung wird nicht genutzt.
Der Einsatz von *elasticsearch* hat keine negativen Auswirkungen auf das Gesamt-

projekt. Es lässt sich festhalten, dass für den momentanen Funktionsumfang und den Aufbau des Projekts *elasticsearch* überdimensioniert ist. Mit *Apache - Lucene* oder dem darauf aufbauenden *Apache - Solr* hätten Lösungen für das Einbinden einer Suchfunktion zur Verfügung gestanden, die wesentlich „leichter“, also mit weniger Funktionsumfang, zur Verfügung gestanden hätten.
Der Einsatz von *elasticsearch* hat einen großen Vorteil. Wenn das System in absehbarer Zeit wachsen würde, oder zu einem auf mehreren Servern verteilten System umgebaut würde, müsste die Datensuche nicht komplett neu aufgebaut und überarbeitet werden, sondern kann dynamisch wachsen. Es müssten lediglich wenige Konfigurationsparameter geändert werden. Das Verteilen von Daten und das Herstellen einer hohen Datensicherheit würde das Programm selbstständig übernehmen.

5.4. Vergleich der Benutzeroberflächen und Funktionalitäten von euSDB im alten und neuentwickelten System

Das alte System *euSDB* umfasst bisher nur eine rudimentär implementierte REST-konforme API. Diese wurde vom Betreiber zu Testzwecken entwickelt und wird nur in wenigen Punkten den Anforderungen einer REST-konformen API gerecht. Die Themen Hypermedia und eindeutige ID sind nicht umgesetzt worden.
Die Vergabe von URI ist nicht REST-konform. Will man alle Chemikalien eines Herstellers aufgelistet bekommen, ist die zugehörige URI nicht REST-konform. Listing 5.1 zeigt die URI des alten Systems in Zeile 1. Die korrekte URI ist in Zeile 3 zu sehen.

```
1 https://localhost:8000/basf/chemicals

3 https://localhost:8000/producer/basf/chemicals
```

Listing 5.1: Aufbau der URI im alten System (Zeile 1) und im neuen System (Zeile 3)

Hypermedia, also die Verlinkung von Ressourcen untereinander, ist im alten System nicht beachtet worden. Folglich wurden die Daten alle in einem Paket dem Kunden angeboten. Das neue System erlaubt dem Kunden die Daten auszuwählen, die er benötigt. Somit werden zwar mehrere Abfragen an das System gestellt, jedoch ist der Kunde flexibler in der Auswahl der Daten, auf die er zugreifen möchte.

Kennt der Kunde die Wurzel der API, ist er in der Lage sich selbständig durch Probieren der angegebenen Links durch das System zu bewegen. Kunden, die sich im Umgang mit Chemikalien und SDB auskennen, werden ohne genaue Kenntnis der API-Spezifikation, schnell die für sie relevanten Daten finden.
Das alte System war einzig zur Ausgabe von SDB spezialisiert. Eine Auflistung von anderen Ressourcen war nicht vorgesehen. Das neu entwickelte System bringt in dieser Hinsicht mehr Flexibilität. Die neue API wird den Ansprüchen nach REST-Konformität gerecht. Sie erlaubt es, sich nach den Richtlinien des Hypermedia durch die Ressourcen zu bewegen, ohne das System genau zu kennen.
Der Kunde kann nun beim Abruf von Daten aus der Datenbank zwischen mindestens drei Formaten unterscheiden (XML, JSON und HTML). Dies gilt für den Zugriff auf sämtliche Ressourcen. Im alten System konnte der Zugriff nur auf den gesamten Datensatz erfolgen. Dabei standen nicht immer die gleichen Variablen zum Abrufen der SDB zur Verfügung. In diesem Bereich ist das System strukturierter und einheitlicher geworden.
Das neue System *euSDB* umfasst nicht alle Funktionen, die die REST-Konformität bedienen würde. Dies hat seine Gründe darin, dass es nicht jedem erlaubt sein soll, beliebig die Datensätze zu manipulieren oder neue hinzuzufügen. Teilweise macht es auch keinen Sinn, gewisse Standardfunktionen zu implementieren. Dennoch wurde bei der Neuentwicklung stets darauf geachtet, dass man fehlende Komponenten schnell und leicht in das bestehende System integrieren kann. Dabei kommt dem Entwickler vor allem der modulare Aufbau entgegen, der es erlaubt, neue Module in das bestehende System zu integrieren.
Das alte System wurde direkt auf dem Server mittels einer dort installierten Software verwaltet. Im neuen System steht dem Betreiber und Administrator eine Weboberfläche zur Verfügung, die er von überall aus dem Web aufrufen kann. Der Administrator ist also nicht mehr gezwungen, direkt am Server zu arbeiten, sondern kann auch von entfernten Arbeitsplätzen, oder aus dem Urlaub die Verwaltung der Plattform durchführen.
Die Ziele des Bachelorprojektes sind im Wesentlichen erreicht. Es sind die Möglichkeiten des *Django* -Frameworks anhand der prototypischen Umsetzung aufgezeigt worden. Es eignet sich, um das Projekt nicht nur prototypisch zu realisieren, sondern auch ein reales Produktivsystem für den globalen Einsatz zu entwickeln.

5.5. Nicht erreichte Ziele

Einige Ziele und Aufgabenstellungen sind auf Grund von Zeitmangel bisher außer Acht gelassen worden. Zum einen handelt es sich dabei um einen Dienst, der den Kunden informiert, sobald sich ein SDB ändert, bzw. wenn es aktualisiert wird. Diese Komponente lässt sich jedoch später entwickeln und in das existierende System einbinden.
Des Weiteren wurden keine Machbarkeitsstudien zur Klärung der Bezahlung der Dienstleistungen erstellt. Dies ist jedoch ein zweitrangiges Problem. Es existiert bereits ein System, dass es ermöglicht, den Kunden verschiedene Leistungen zur Verfügung zu stellen. Dieses Bezahlsystem kann erstmal weitergenutzt werden.
Langfristig soll es auch für die Hersteller von Chemikalien finanziell lukrativ werden, SDB auf *euSDB* zu veröffentlichen. Auf welche Weise man dies sicherstellt, muss ebenfalls geklärt werden und geht in dieselbe Richtung wie die Machbarkeitsstudien zur Abrechnung. Dabei können prozentuale Gewinnbeteiligungen ebenso in Betracht gezogen werden, wie Pauschalenpreise pro Upload gezahlt werden.

6. Fazit

6.1. Ausblick

Die Plattform *euSDB* kann noch um viele Projekte und Funktionen erweitert werden. Zunächst gilt es, die noch offenen Ziele aus Kapitel 5.5 zu erreichen.
Es macht Sinn die REST-konforme API in einer eigenen APP auszulagern. Bisher ist sie in die Kern-APP eingegliedert. Das Auslagern bietet den Vorteil, dass man leicht eine neue Version der API hinzufügen kann, ohne das die bestehende API ungültig wird. Für Nutzer der API ist dies sehr wichtig, denn diese haben nicht immer das Interesse ihren Client neu programmieren, sobald es eine neue Version der API gibt. Damit der Client weiterhin nutzbar bleibt, ist es notwendig, parallel mehrere Versionen der API zu betreiben. Dies kommt auch dem Prinzip der losen Kopplung entgegen und erlaubt schnelles Ein- und Ausbinden der API.
Das wohl größte Projekt wird die Migration der Daten von dem alten auf das neue System. Dabei gilt es viele Probleme zu lösen. Leider sind die Daten nicht alle konsistent und einheitlich abgelegt. Es ist zu vermuten das die ca. 440 000 Datensätze nur teilweise automatisiert migriert und aufbereitet werden können. Bei vielen Details wird man um eine manuelle Bearbeitung und Prüfung nicht herumkommen.
Die Regeln zur Notation von S-Sätzen sind folgende: 1. S-Sätze werden in der Zahlencodierung durch ein Semikolon getrennt. 2. Treten S-Sätze in festen Kombinationen auf, werden sie durch einen Schrägstrich getrennt.

```
1 S-Saetze:
2 26/36/37/39
```

Listing 6.1: Beispieldaten, die aus formalen Gründen nicht korrekt sind und einer Überarbeitung bedürfen, bevor sie in das neue System übernommen werden können

Listing 6.1 zeigt die inkonsistente Darstellung der S-Sätze einer Chemikalie in der alten Datenbank. Die S-Sätze 26 und 39 sind einzeln, 36/37 treten als feste Kombi-

nation auf. Die richtige Formatierung der S-Sätze sähe wie folgt aus: S-Sätze: 26; 36/37; 39. Tabelle 6.1 zeigt die ausformulierten S-Sätze. Diese inkonsistenten Daten müssen manuell überarbeitet werden.

S-Satz	**Anweisung**
S 26	Bei Berührung mit den Augen gründlich mit Wasser abspülen und Arzt konsultieren.
S 36	Bei der Arbeit geeignete Schutzkleidung tragen.
S 37	Geeignete Schutzhandschuhe tragen.
S 39	Schutzbrille/Gesichtsschutz tragen.
S 36/37	Bei der Arbeit geeignete Schutzhandschuhe und Schutzkleidung tragen.

Tabelle 6.1.: Mehrere S-Sätze und deren Sicherheitsanweisungen

Diese Fehler sind nicht immer gleichförmig und können daher nicht automatisiert erkannt werden. Das Auftreten von Fehlern ist willkürlich. Man kann keine Systematik identifizieren, was eine automatische Erkennung und Behebung weiter erschwert.

Die Fehler haben ihren Ursprung bei den Herstellern der Chemikalien. Einige geben ihre Daten in Form einer *Excel* -Tabelle an den Betreiber von *euSDB* weiter, damit dieser sie auf der Plattform veröffentlicht. Die Kontrolle der Daten auf Richtigkeit erfolgt dabei händisch und nicht automatisiert. Daher ist es immer wieder passiert, dass Datensätze in der falschen Notation in das System eingegeben wurden.

Die Fehler konnten entstehen, da praktisch alle Daten als Text in einer einzigen Tabelle abgelegt wurden. Zwar gab es Vorgaben zur Notation von Daten, doch diese für jedes Textfeld einzeln zu prüfen, ist aus Zeitgründen nicht immer möglich gewesen.

Zur Zeit stehen die SDB über einen eigenen Link als PDF zum Herunterladen bereit. Schöner wäre es, wenn man die Möglichkeit erhält, sich das SDB über eine Parameterübergabe anzeigen zu lassen. Dabei wird daran gedacht, dass man entweder das Format „pdf“ an die Ressource „chemicals“ übergibt, oder über den HTTP-Header diesen Aufruf auslöst. Dieser Aufruf funktioniert nur bei der Ressource „chemicals“. Hier wird dann das PDF aufgerufen und dem Kunden zum Download bereitgestellt.

Aufgrund des strikt eingehaltenen Entwicklungsstils eines modularen Aufbaus ist

es leicht, die Plattform zu erweitern. Dabei ist es auch nicht zwingend notwendig, das bestehende System in allen Komponenten verstanden zu haben. Versteht man einzelne Komponenten nicht, oder weiß man diese nicht zu nutzen, kann man sie auch in der eigenen Erweiterung selbst neu implementieren. Dies hat die Vorteile, dass man den eigenen Quellcode verstanden hat und von den anderen Entwicklungen unabhängig bleibt.
Folgeprojekte können sich auch aus Nicht-Serverentwicklungen ergeben. Z.B gibt es ein Interesse an einem Client-Dienst, der automatisch Etiketten und Beschriftungen für Aufbewahrungsbehälter von Chemikalien aus den SDB erstellt. Das manuelle Erstellen solcher Etiketten ist sehr zeitintensiv und somit auch teuer. Der Etiketten-Client soll dabei mittels der REST-konformen API auf die Datenbank zugreifen können, die benötigten Daten abrufen und dann ein Etikett zusammenstellen und zum Ausdrucken bereithalten.
Ein weiterer Client der entwickelt werden soll, sieht wie folgt aus: Jedes Unternehmen, das Chemikalien lagert, muss sehr genau buchführen, wo sich die Chemikalien befinden. Ebenfalls muss es in dieser Übersicht alle Gefährdungshinweise und Arbeitsanweisungen aufgelistet sein. Bei letzteren Informationen kommt die Datenbank von *euSDB* ins Spiel. Der zu entwickelnde Client soll zum einen die vom Kunden eingetragenen Lagerungsorte tabellarisch enthalten und diese Datensätze um die Informationen aus der Datenbank erweitern.

6.2. Zusammenfassung

Die neue Plattform *euSDB* stellt den gesamten Funktionsumfang der alten Plattform zur Verfügung. Es existiert eine Suchfunktion, eine Zugriffsverwaltung, ein Downloadbereich für SDB, usw.. Die Umsetzung der Plattform, also die Programmierung, orientiert sich streng an verschiedenen Programmierparadigmen, wie z.B. Objektorientierung und lose Kopplung.
Das Projekt kann auf allen Unix- und Windows-Versionen laufen. Es ist damit relativ betriebssystemunabhängig. Dem Betreiber von *euSDB* wird dadurch freigestellt, auf welchem System er seinen Server laufen lässt.
Dem Auftraggeber steht eine vollwertige Onlineplattform zur Verfügung, die an nur wenigen Punkten einer Überarbeitung bedarf, sodass sie in den Online-Betrieb gehen kann. Dabei geht es vor allem darum, irreguläre Abfragen abzufangen und zu

verhindern, dass das System abstürzt.
Durch die Möglichkeiten der Administration, die nicht mehr ausschließlich an der Datenbank über einen Datenbankmanager durchgeführt werden, wird für eine Bessere Datenintegrität gesorgt. Das System merkt direkt, wenn Fremdschlüssel gelöscht werden, obwohl sie noch gebraucht werden und verhindert in einem solchen Fall das Löschen.
Die Entwicklung der Plattform hat die Mächtigkeit und die vielen Möglichkeiten des *Django* -Framework aufgezeigt. Dabei bleibt sie jedoch nachvollziehbar und leicht zu erlernen. Auch mit nicht so umfangreichen Vorkenntnissen in Webentwicklung und *Python* scheint es effektiv einsetzbar zu sein. Im Web werden viele Erweiterungen und APP zur Verfügung gestellt. Diese lassen sich leicht nachvollziehen und einfach in das bestehende System einbinden.
Einen Nachteil des Frameworks ist es, dass sich Fehler erzeugen lassen, die die *Python* -Umgebung abstürzen lassen. Solche Fehler müssen unbedingt behoben werden. Erst dann kann der Server neugestartet werden. Dies kann zu längeren Ausfallzeiten der Plattform führen, wenn man nicht sofort realisiert hat, dass der Server außer Betrieb ist. Es ist sehr wichtig, das System gründlich zu testen, damit möglichst keine Fehler auftreten.
Der Auftraggeber erhält mit der neuentwickelten Softwareplattform *euSDB* eine ausgiebig getestete, ihn im Funktionsumfang nicht einschränkende und stabile Lösung. Sie gibt ihm die Möglichkeit, sie fast beliebig um neue Funktionen zu erweitern. Auch die Cliententwicklung bietet neue Möglichkeiten. Auf Grund der strikt eingehaltenen losen Kopplung ist sie nicht an eine gewisse Programmiersprach gebunden. *euSDB* stellt insofern eine höchst flexible Plattform dar.
Einige Teile des Projektes sind bisher sehr aufwändig gelöst, da der Programmierer mit *euSDB* das erste Mal ein *Django* -System realisiert hat. Es fehlten ihm Kenntnisse und Erfahrungen in dieser Programmiersprache, sodass einige Komponenten erst während der Realisierung verstanden und nachvollzogen wurden. Die zu diesem Zeitpunkt bereits realisierten Funktionen können ineffizient sein und einer Überarbeitungen bedürfen, z.B. die REST-konforme API ist nicht in einer eigenen APP untergebracht.
Der Programierer und die Firma *innoQ* freuen sich auf weitere Projekte, die der Auftraggeber im Rahmen der Plattform *euSDB* vergibt. Abschließend kann man sagen, dass das Projekt zwar noch nicht vollständig abgeschlossen ist, jedoch das

bis dahin fertiggestellte sowohl aus Sicht des Auftraggebers als auch aus Sicht von *innoQ* als positiv gewertet wird und somit auch von einem erfolgreichen Projektverlauf gesprochen werden kann.

Literaturverzeichnis

[BELAGR-2009] Lahres, Bernhard; Rayman, Gregor: Objektorientierte Programmierung: Das umfassende Handbuch. Galileo Press GmbH, Bonn, 2009.

[DJSOFO-2013] Django Software Foundation: Meet Django. https://www.djangoproject.com/, zuletzt besucht am 10.11.2013.

[ELASBV-2013] Elasticsearch BV: elasticsearch. http://www.elasticsearch.org/, zuletzt besucht am 27.01.2013.

[MAZAGR-2013] Markus Zapke-Gründemann. : Django Workshop. http://www.django-workshop.de, 2012, zuletzt besucht am 10.11.2013.

[OLBFI-2013] Oliver B. Fischer: Search and destroy: Volltextsuche mit ElasticSearch. http://www.heise.de/developer/artikel/Volltextsuche-mit-ElasticSearch-1920454.html, 26.07.2013, zuletzt besucht am 10.11.2013.

[ORACOR-2013] Oracle Corporation: Oracle. http://www.oracle.com/de/index.html, zuletzt besucht am 27.01.2013.

[PEKAER-2008] Kaiser, Peter; Ernesti, Johannes: Python: Das umfassende Handbuch. Galileo Press GmbH, Bonn, 2008.

[PETKAR-2013] Karich, Peter: Immer flexibel: Apache Solr bekommt Konkurrenz: ElasticSearch [Leseprobe] http://www.heise.de/ix/artikel/Immer-flexibel-1440270.html, 03.2013, zuletzt besucht am 25.12.2013.

[RAIGRI-2009] Grimm, Rainer: Funktionale Programmierung (1): Grundzüge. http://www.linux-magazin.de/Online-Artikel/Funktionale-Programmierung-1-Grundzuege, April 2009, zuletzt besucht am 27.01.2014.

[STETIL-2009] Tilkov, Stephan: REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien. dpunkt.verlag GmbH, Heidelberg, 2009.

[STETIL-2009a] Tilkov, Stephan: REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien. dpunkt.verlag GmbH, Heidelberg, 2009. S. 10.

[THAPSO-2013] The Apache Software Foundation: Lucene. http://lucene.apache.org/, zuletzt besucht am 27.01.2013.

[VCI-2008] Leitfaden Sicherheitsdatenblatt: Mit Hinweisen zur Einstufung und Kennzeichnung mit aktuellen Ergänzungen gem. REACH-Verordnung. https://www.vci.de/downloads/123464-lf_sdb.pdf, April 2008, zuletzt besucht am 27.01.2014.

B. Glossar

Apache Lucene Bei Apache Lucene handelt es sich um ein Datenbanksystem zur effizienten Volltextsuche.

API application programming interface: Eine Progammierschnittstelle zur Anbindung von Software an andere Software.

APP Applications: Software die dafür sorgt, dass eine gewisse Funktion sichergestellt ist und wie gewünscht abläuft.

CRUD Create-Read-Update-Delete: Ein Programmierparadigma, dass häufig im Zusammenhang mit REST erwähnt und eingesetzt wird. Die vier Standardfunktionen in HTTP (GET, POST, PUT, DELETE) werden serverseitig durch ein Create, Update, Get und Delete implementiert.

Django-Framework Ein in Python geschriebenes Framework zur schnellen und einfachen Entwicklung von Webanwendungen.

DRY Don't repeat yourself: DRY ist ein Programmierparadigma. Es ermöglicht eine lose Kopplung. Ziel ist es, nach Möglichkeit den Code nur einmal zu erzeugen und ihn dann immer wieder aufzurufen.

euSDB Eine Onlineplattform, die den Austausch und die Archivierung von Sicherheitsdatenblättern ermöglicht.

elasticsearch elasticsearch ist ein auf Apache Lucene aufbauende Datenbank zur Volltextsuche, die mit dem Anspruch einer leichten Handhabung und der Einsatzmöglichkeit in verteilten Systemen entwickelt wurde.

GHS Globally Harmonized System of Classification, Labelling and Packaging of Chemicals: Ein von der UN entwickeltes und von der EU angepasstes Klassifizierungs- und Kennzeichnungssystem für Chemikalien.

GNU GNU's Not Unix: Zu Deutsch: „GNU ist Nicht Unix". Ein rekursives Akronym. GNU[1] ist ein UNIX-ähnliches Betriebssystem. Es wird unter der GNU GPL vertrieben. Wenn von Software unter der GNU-Lizenz gesprochen wird, ist die GNU-GPL gemeint.

GPL General Public License: Dies ist die am weitesten verbreitete Softwarelizenz. Sie erlaubt es die Software zu ändern, zu kopieren, zu nutzen und zu studieren. Man spricht auch von freier Software.

H-Sätze Die H-Sätze beschreiben Risiken und Gefährdungen, die von Chemikalien ausgehen können. Das H steht für Hazard, deutsch Gefahr oder Risiko

HTML Hypertext Markup Language: Bei HTML handelt es sich um eine Auszeichnungssprache. Sie leitet sich von der Sprache XML ab. Sie wird im Internet verwendet, um Websites aufzubauen.

HTTP Hypertext Transfer Protokoll: Hierbei handet es sich um ein Protokoll zur Datenübertragung in Netzwerken. Man kennt es vor allem aus dem Einsatz im Internet.

JSON JavaScript Object Notation: JSON ist ein kompaktes Dateiformat. Die Besonderheit ist, dass es schon selbst ein JAVA-Script darstellt. Es hat sich aufgrund seiner Kompaktheit zu einem immer wichtigeren Daten-Austausch-Format entwickelt und wird von vielen Programmiersprachen unterstützt. (Beispiel Anhang C.1)

[1] Informationen zum GNU-Projekt finden sie hier: http://www.gnu.org/

MTV Model-Template-View: Bei diesem Prinzip baut eine View aus den Daten der Modelle unter zur Hilfenahme von Templates die graphische Ausgabe (z.B. eine Webseite).

MVC Model-View-Controller: MVC ist ein Muster der Softwareentwicklung. Dabei steuert der Controller welche Daten aus den Models von der View zu einer Ausgabe verarbeitet werden.

Node Ein Node stellt einen einzelnen Server in einem Servercluster dar, auf dem eine Instanz von elasticsearch aktiv ist.

P-Sätze Die P-Sätze gehören zu den H-Sätzen. Sie geben Sicherheitshinweise zum Umgang mit Chemikalien. In der Regel gibt es zu jedem H-Satz einen P-Satz. Auf diese Weise wird jede Gefährdung mit einer Sicherheitsanweisung verknüpft. Das P steht für Precaution, deutsch Vorsicht.

PDF Portable Document Format: Ist ein plattformunabhängiges Dateiformat zum Austausch von Dokumenten. Es wurde von der Firma Adobe Systems[2] entwickelt und 1993 veröffentlicht.

PHP Hypertext Preprocessor: Ist eine Skriptsprache die zur Erstellung dynamischer Webseiten dient. Es prägte über viele Jahre das Web maßgeblich und und ist ein weitverbreiteter Standard.

PSF Python Software Foundation: Die Python Software Foundation[3] ist die offizielle Entwicklergruppe, die das Python-Projekt noch heute weiterentwickelt.

R-Sätze Die R-Sätze sind der Vorgänger der H-Sätze. Sie beschreiben ebenfalls Gefährdungen. R steht für Risiko.

[2] Mehr Informationen zu Adobe Systems finden sich unter: http://www.adobe.com/
[3] Weiter Informationen zur Python Software Foundation finden sich hier: http://www.python.org/psf/

Replica Kopie eines Shards auf einem anderen Node.

Repräsentationen Repräsentationen sind die für den Client graphisch aufbereiteten Rohdaten einer Ressource

Ressourcen Eine Ressource enthält ausschließlich Rohdaten gleichen Typs, z.B. Adressen.

REST Representational State Transfer: Ist ein Progammierparadigma der Webprogammierung. Dabei soll jede URL/URI auch nur eine Seite zurückgeben.

S-Sätze Die S-Sätze sind die Vorgänger der P-Sätze. Sie geben Sicherheitshinweise im Umgang mit Chemikalien (Ähnlich wie bei den P- und H-Sätzen). S steht für Sicherheit.

SDB Sicherheitsdatenblatt: Jedes chemikalienherstellende Unternehmen muss zu jeder Chemikalie und jedem Chemikaliengemisch ein Sicherheitsdatenblatt erstellen. In diesem sind alle Gefahren und empfohlene Schutzmaßnahmen zur Verwendung der Chemikalie aufgeführt.

Serialisierung Die Serialisierung wird verwendet, um die Rohdaten eines Models aufzubereiten und einer View zur Verfügung zu stellen.

Shard Ein Shard ist eine Apache Lucene Instanz auf einem elasticsearch-Node

SQL Structured Query Language: SQL ist eine Datenbanksprache. Mit dieser können Befehle zum Ändern der Daten oder der Datenbankstruktur, aber auch Abfragen an Datenbanken übermittelt werden.

URI Uniform Resource Identifier: URI sind eindeutige ID, die im Internet verwendet werden, um auf Ressourcen und/oder Services zu verweisen (identifikation).

URL	Uniform Resource Locator: Zu deutsch: einheitlicher Quellenanzeiger. Im Internet nutzt man URL zum Aufruf von Webseiten, z.B www.google.de (geographisch).
UTF-8	UCS Transformation Format 8 Bit: Eine weitverbreitete Codierung von Unicodezeichen. UCS steht dabei für Universal Character Set.
XML	Extensible Markup Language: XML ist eine Auszeichnungssprache um Daten strukturiert, je nach Formatierung auch hierarchisch darzustellen. Sie bildet heute für viele Anwendungen die Basis zum Austausch von Daten über das Internet. Ein Beispieldokument finden Sie in Anhang C.2.
.py	Dateiendung für Pythonquellcodedateien

C. Quellcode

C.1. User-Daten als JSON-Dokument

Listing C.1: JSON-Dokument der Repräsentation eines Users

```
{
    "name": "Max Mustermann",
    "E-Mail": "max.mustermann@bsp.de"
}
```

C.2. User-Daten als XML-Dokument

Listing C.2: XML-Dokument der Repräsentation eines Users

```
<?xml version="1.0" encoding="utf-8"?>
<root>
    <name>Max Mustermann</name>
    <E-Mail>max.mustermann@bsp.de</E-Mail>
</root>
```

C.3. Die Datei settings.py

Listing C.3: Quellcode der globalen Einstellungen im Projekt euSDB

```
import os
SITE_ROOT = os.path.realpath(os.path.dirname(__file__))

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
    ('marius', 'mail@pausmarius.de'),
)

DEFAULT_FILE_STORAGE = 'db_file_storage.storage.DatabaseFileStorage'

MANAGERS = ADMINS
```

```
15 DATABASES = {
16     'default': {
17         'ENGINE': 'django.db.backends.mysql',
18         'NAME': 'euSDB',
19         'USER': 'eusdb',
20         'PASSWORD': 'xxx',
21         'HOST': '',
22         'PORT': '',
23     }
24 }

26 HAYSTACK_CONNECTIONS = {
27     'default': {
28         'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
29         'URL': 'http://127.0.0.1:9200/',
30         'INDEX_NAME': 'sdb',
31     },
32 }

34 HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'

36 ALLOWED_HOSTS = []

38 TIME_ZONE = 'Europe/Berlin'

40 LANGUAGE_CODE = 'de'

42 SITE_ID = 1

44 USE_I18N = True

46 USE_L10N = True

48 USE_TZ = True

50 MEDIA_ROOT = ''

52 MEDIA_URL = ''

54 STATIC_ROOT = ''

56 STATIC_URL = '/static/'

58 STATICFILES_DIRS = (
59     os.path.join(SITE_ROOT, '..', 'static'),
60 )

62 STATICFILES_FINDERS = (
63     'django.contrib.staticfiles.finders.FileSystemFinder',
64     'django.contrib.staticfiles.finders.AppDirectoriesFinder',
65 )
```

```
67  SECRET_KEY = ')x^+9zw#b@gp0*ig1eq9790!*0g2jt#rwy0zq4y5uixef1j$eu'

69  TEMPLATE_LOADERS = (
70      'django.template.loaders.filesystem.Loader',
71      'django.template.loaders.app_directories.Loader',
72  )

74  MIDDLEWARE_CLASSES = (
75      'django.middleware.common.CommonMiddleware',
76      'django.contrib.sessions.middleware.SessionMiddleware',
77      'django.middleware.csrf.CsrfViewMiddleware',
78      'django.contrib.auth.middleware.AuthenticationMiddleware',
79      'django.contrib.messages.middleware.MessageMiddleware',
80  )

82  ROOT_URLCONF = 'euSDB.urls'

84  WSGI_APPLICATION = 'euSDB.wsgi.application'

86  TEMPLATE_DIRS = (
87      os.path.join(SITE_ROOT, '..', 'templates'),
88      os.path.join(SITE_ROOT, 'templates'),
89  )

91  INSTALLED_APPS = (
92      'django.contrib.auth',
93      'django.contrib.contenttypes',
94      'django.contrib.sessions',
95      'django.contrib.sites',
96      'django.contrib.messages',
97      'django.contrib.staticfiles',
98      'django.contrib.admin',
99      'django.contrib.admindocs',
100     'chemicals',
101     'userauth',
102     'rest_framework',
103 )


106 LOGIN_URL = '/benutzer/anmelden/'
107 LOGOUT_URL = '/benutzer/abmelden/'
108 LOGIN_REDIRECT_URL = '/'

110 LOGGING = {
111     'version': 1,
112     'disable_existing_loggers': False,
113     'filters': {
114         'require_debug_false': {
115             '()': 'django.utils.log.RequireDebugFalse'
116         }
117     },
118     'handlers': {
```

```
        'mail_admins': {
            'level': 'ERROR',
            'filters': ['require_debug_false'],
            'class': 'django.utils.log.AdminEmailHandler'
        }
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': True,
        },
    }
}

REST_FRAMEWORK = {
    'DEFAULT_MODEL_SERIALIZER_CLASS':
        'rest_framework.serializers.HyperlinkedModelSerializer',
    'DEFAULT_RENDERER_CLASSES': (
        'rest_framework.renderers.TemplateHTMLRenderer',
        'rest_framework.renderers.UnicodeJSONRenderer',
        'rest_framework.renderers.XMLRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ),

    'PAGINATE_BY': 2,
    'PAGINATE_BY_PARAM': 'page_size',
    'MAX_PAGINATE_BY': 100
}
```

C.4. Die models-Klassen der chemicals-APP

Listing C.4: Inhalt der gesamten models-Klasse der chemicals-APP

```
# encoding: utf-8
from django.db import models
from django.utils.timezone import now

class CompanyData(models.Model):
    id = models.AutoField(u'id', primary_key=True, editable=False)
    name = models.CharField(u'Name', max_length=255, default='-')
    slug_company = models.SlugField(unique=True)
    previous_data = models.OneToOneField('self', blank=True,
        null=True, related_name='Vorgaenger_von')
    replacement_data = models.OneToOneField('self', blank=True,
        null=True, related_name='Nachfolger_von', editable=False)
    last_update = models.DateTimeField(editable=False)
```

```
15      class Meta:
16          verbose_name = u'Hersteller'
17          verbose_name_plural = u'Hersteller'
18
19      def __unicode__(self):
20          return self.slug_company
21
22      def save(self, *args, **kwargs):
23          if not self.id:
24              self.created = now()
25          self.last_update = now()
26          super(CompanyData, self).save(*args, **kwargs)
27          if self.previous_data:
28              CompanyData.objects.filter(pk=self.previous_data.id).
29                  update(replacement_data=self)
30
31  class Hazard(models.Model):
32      id = models.AutoField(u'id', primary_key=True, editable=False)
33      hazard_symbol_text = models.CharField(u'Hazard_symbol_text',
34          max_length=255, default='-')
35      slug_hazard = models.SlugField(unique=True)
36      previous_data = models.OneToOneField('self', blank=True,
37          null=True, related_name='Vorgaenger_von')
38      replacement_data = models.OneToOneField('self', blank=True,
39          null=True, related_name='Nachfolger_von', editable=False)
40      last_update = models.DateTimeField(editable=False)
41
42      class Meta:
43          verbose_name = u'Hazard Symbol'
44          verbose_name_plural = u'Hazard Symbole'
45
46      def __unicode__(self):
47          return self.hazard_symbol_text
48
49      def save(self, *args, **kwargs):
50          if not self.id:
51              self.created = now()
52          self.last_update = now()
53          super(Hazard, self).save(*args, **kwargs)
54          if self.previous_data:
55              Hazard.objects.filter(pk=self.previous_data.id).
56                  update(replacement_data=self)
57
58  class Ghs(models.Model):
59      id = models.AutoField(u'id', primary_key=True, editable=False)
60      ghs_indicator_danger_text = models.CharField(u'Ghs_indicator_danger_text',
61          max_length=255, default='-')
62      slug_ghs = models.SlugField(unique=True)
63      previous_data = models.OneToOneField('self', blank=True,
64          null=True, related_name='Vorgaenger_von')
65      replacement_data = models.OneToOneField('self', blank=True,
```

```
66          null=True, related_name='Nachfolger_von', editable=False)
67      last_update = models.DateTimeField(editable=False)

69      class Meta:
70          verbose_name = u'GHS Gefahren Kennung'
71          verbose_name_plural = u'GHS Gefahren Kennungen'

73      def __unicode__(self):
74          return self.ghs_indicator_danger_text

76      def save(self, *args, **kwargs):
77          if not self.id:
78              self.created = now()
79          self.last_update = now()
80          super(Ghs, self).save(*args, **kwargs)
81          if self.previous_data:
82              Ghs.objects.filter(pk=self.previous_data.id).
83                  update(replacement_data=self)

85  class Eu(models.Model):
86      id = models.AutoField(u'id', primary_key=True, editable=False)
87      eu_indicator_danger = models.CharField(u'Eu_indicator_danger',
88          max_length=255)
89      eu_indicator_danger_text_de = models.CharField(
90          u'Eu_indicator_danger_text_de', max_length=255, default='-')
91      slug_eu = models.SlugField(unique=True)
92      previous_data = models.OneToOneField('self', blank=True,
93          null=True, related_name='Vorgaenger_von')
94      replacement_data = models.OneToOneField('self', blank=True,
95          null=True, related_name='Nachfolger_von', editable=False)
96      last_update = models.DateTimeField(editable=False)

98      class Meta:
99          verbose_name = u'EU Gefahren Kennung'
100         verbose_name_plural = u'EU Gefahren Kennungen'

102     def __unicode__(self):
103         return self.eu_indicator_danger_text_de

105     def save(self, *args, **kwargs):
106         if not self.id:
107             self.created = now()
108         self.last_update = now()
109         super(Eu, self).save(*args, **kwargs)
110         if self.previous_data:
111             Eu.objects.filter(pk=self.previous_data.id).
112                 update(replacement_data=self)

114 class P_phrase(models.Model):
115     id = models.AutoField(u'id', primary_key=True, editable=False)
116     p_phrase = models.CharField(u'P_phrase', max_length=255)
117     p_phrase_text_de = models.CharField(u'P_phrase_text_de',
```

```
118         max_length=255, default='-')
119     slug_p_phrase = models.SlugField(unique=True)
120     previous_data = models.OneToOneField('self', blank=True,
121         null=True, related_name='Vorgaenger_von')
122     replacement_data = models.OneToOneField('self', blank=True,
123         null=True, related_name='Nachfolger_von', editable=False)
124     last_update = models.DateTimeField(editable=False)

126     class Meta:
127         verbose_name = u'P-Satz'
128         verbose_name_plural = u'P-Saetze'

130     def __unicode__(self):
131         return self.p_phrase_text_de

133     def save(self, *args, **kwargs):
134         if not self.id:
135             self.created = now()
136         self.last_update = now()
137         super(P_phrase, self).save(*args, **kwargs)
138         if self.previous_data:
139             P_phrase.objects.filter(pk=self.previous_data.id).
140                 update(replacement_data=self)

142 class S_phrase(models.Model):
143     id = models.AutoField(u'id', primary_key=True, editable=False)
144     s_phrase = models.CharField(u'S_phrase', max_length=255)
145     s_phrase_text_de = models.CharField(u'S_phrase_text_de',
146         max_length=255, default='-')
147     slug_s_phrase = models.SlugField(unique=True)
148     previous_data = models.OneToOneField('self', blank=True,
149         null=True, related_name='Vorgaenger_von')
150     replacement_data = models.OneToOneField('self', blank=True,
151         null=True, related_name='Nachfolger_von', editable=False)
152     last_update = models.DateTimeField(editable=False)

154     class Meta:
155         verbose_name = u'S-Satz'
156         verbose_name_plural = u'S-Saetze'

158     def __unicode__(self):
159         return self.s_phrase_text_de

161     def save(self, *args, **kwargs):
162         if not self.id:
163             self.created = now()
164         self.last_update = now()
165         super(S_phrase, self).save(*args, **kwargs)
166         if self.previous_data:
167             S_phrase.objects.filter(pk=self.previous_data.id).
168                 update(replacement_data=self)
```

```
170 class R_phrase(models.Model):
171     id = models.AutoField(u'id', primary_key=True, editable=False)
172     r_phrase = models.CharField(u'R_phrase', max_length=255)
173     r_phrase_text_de = models.CharField(u'R_phrase_text_de',
174         max_length=255, default='-')
175     slug_r_phrase = models.SlugField(unique=True)
176     previous_data = models.OneToOneField('self', blank=True,
177         null=True, related_name='Vorgaenger_von')
178     replacement_data = models.OneToOneField('self', blank=True,
179         null=True, related_name='Nachfolger_von', editable=False)
180     last_update = models.DateTimeField(editable=False)

182     class Meta:
183         verbose_name = u'R-Satz'
184         verbose_name_plural = u'R-Saetze'

186     def __unicode__(self):
187         return self.r_phrase_text_de

189     def save(self, *args, **kwargs):
190         if not self.id:
191             self.created = now()
192         self.last_update = now()
193         super(R_phrase, self).save(*args, **kwargs)
194         if self.previous_data:
195             R_phrase.objects.filter(pk=self.previous_data.id).
196                 update(replacement_data=self)

198 class H_phrase(models.Model):
199     id = models.AutoField(u'id', primary_key=True, editable=False)
200     h_phrase = models.CharField(u'H_phrase', max_length=255)
201     h_phrase_text_de = models.CharField(u'H_phrase_text_de',
202         max_length=255, default='-')
203     slug_h_phrase = models.SlugField(unique=True)
204     previous_data = models.OneToOneField('self', blank=True,
205         null=True, related_name='Vorgaenger_von')
206     replacement_data = models.OneToOneField('self', blank=True,
207         null=True, related_name='Nachfolger_von', editable=False)
208     last_update = models.DateTimeField(editable=False)

210     class Meta:
211         verbose_name = u'H-Satz'
212         verbose_name_plural = u'H-Saetze'

214     def __unicode__(self):
215         return self.h_phrase_text_de

217     def save(self, *args, **kwargs):
218         if not self.id:
219             self.created = now()
220         self.last_update = now()
221         super(H_phrase, self).save(*args, **kwargs)
```

```
222         if self.previous_data:
223             H_phrase.objects.filter(pk=self.previous_data.id).
224                 update(replacement_data=self)

226 class Chemical(models.Model):
227     id = models.AutoField(u'id', primary_key=True, editable=False)
228     name = models.CharField(u'Name', max_length=255, default='')
229     company = models.ForeignKey(CompanyData,
230         verbose_name=u'Hersteller-Firma', related_name='chemicals')
231     hazard = models.ManyToManyField(Hazard, verbose_name=u'Hazard Symbol')
232     ghs = models.ManyToManyField(Ghs, verbose_name=u'GHS Gefahren Kennung')
233     eu = models.ManyToManyField(Eu, verbose_name=u'EU Gefahren Kennung')
234     p_phrase = models.ManyToManyField(P_phrase, verbose_name=u'P-Satz')
235     s_phrase = models.ManyToManyField(S_phrase, verbose_name=u'S-Satz')
236     r_phrase = models.ManyToManyField(R_phrase, verbose_name=u'R-Satz')
237     h_phrase = models.ManyToManyField(H_phrase, verbose_name=u'H-Satz')
238     slug_chemical = models.SlugField(unique=True)
239     previous_data = models.OneToOneField('self', blank=True,
240         null=True, related_name='Vorgaenger_von')
241     replacement_data = models.OneToOneField('self', blank=True,
242         null=True, related_name='Nachfolger_von', editable=False)
243     last_update = models.DateTimeField(editable=False)

245     class Meta:
246         verbose_name = u'Chemikalie'
247         verbose_name_plural = u'Chemikalien'

249     def __unicode__(self):
250         return self.slug_chemical

252     def save(self, *args, **kwargs):
253         if not self.id:
254             self.created = now()
255         self.last_update = now()
256         super(Chemical, self).save(*args, **kwargs)
257         if self.previous_data:
258             Chemical.objects.filter(pk=self.previous_data.id).
259                 update(replacement_data=self)
```