Entwicklung einer Anwendung zur Aufzeichnung von Provenancedaten in Webanwendungen

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

im Studiengang

Informatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Benjamin Bauer

29. August 2023

Bearbeitungszeitraum	12 Wochen
Matrikelnummer	8590497
Kurs	TINF20IT1
Ausbildungsfirma	DLR, Berlin-Adlershof
Betreuer	Carina Haupt, Jürgen Schultheis

Sperrvermerk

Die vorliegende Bachelorarbeit mit dem Titel *Entwicklung einer Anwendung zur Aufzeichnung von Provenancedaten in Webanwendungen* ist mit einem Sperrvermerk versehen und wird ausschließlich zu Prüfungszwecken am Studiengang Informatik der Dualen Hochschule Baden-Württemberg Mannheim vorgelegt. Jede Einsichtnahme und Veröffentlichung – auch von Teilen der Arbeit – bedarf der vorherigen Zustimmung durch das DLR.

Titeländerung

Im Rahmen der Bachelorarbeit wurde der Titel von *Integration von Provenance in die BACARDI-Software* zu *Entwicklung einer Anwendung zur Aufzeichnung von Provenancedaten in Webanwendungen* geändert. Durch ein Verschieben der Schwerpunkte während des Bearbeitens wurde dieser für sinnvoller erachtet. Das originale Anmeldeformular ist auf den letzten 4 Seiten des Dokuments zu finden.

Quellen und Quellcode

Die für die Arbeit verwendeten Quellen sind im Literaturverzeichnis zu finden und zusätzlich hinterlegt. Dem physischen Exemplar wurde ein USB-Stick mit Ausdrucken aller Quellen und dem verwendeten Quellcode angehängt. Das digitale Exemplar wird in einer Zip-Datei abgegeben, in der das gleiche enthalten ist, wie auf dem USB-Stick. Ausdrucke der verwendeten Bücher und Paper sind nicht beigelegt.

Zusammenfassung

Mit steigendem Bedarf von Anwendungen zur elektronischen Datenverarbeitung steigt auch die Komplexität der dahinter stehenden Software, wodurch die darin ablaufenden Prozesse immer schwerer nachvollziehbar werden. Um dennoch Aussagen über Qualität, Zuverlässigkeit und Vertrauenswürdigkeit der dabei verarbeiteten Daten treffen zu können, wird im Rahmen der Arbeit eine Anwendung zur Aufzeichnung von Provenance in Webanwendungen entwickelt. Diese orientiert sich am W3C PROV-Datenmodell und ermöglicht das Erfassen von Änderungen an Objekten und das Aufzeichnen verschiedener Views in Django-Anwendungen mit anschließendem Export der dabei erzeugten Provenance-Dokumente. Zur Veranschaulichung von Provenance wird zunächst eine Beispielanwendung implementiert, die als Grundlage für die Demonstration und Tests der Provenance-Anwendung dient. Basierend auf den daraus gewonnenen Erkenntnissen wird danach ein Gesamtkonzept erstellt, anhand dessen die Implementierung der Provenance-Anwendung und damit verbundene kritische Entscheidungen beschrieben werden.

Abstract

With rising software complexity, the associated background processes become difficult to understand. Capturing the origin of the process-related data is a possible fix for this issue. The present thesis describes the implementation of a provenance recording application. The tool relies on the W3C PROV-DM and works for web applications that base on Django. It allows users to capture changes to database objects and to record different views. This enables statements to be made about the quality, reliability and trustworthiness of the data collected. Another possibility is the export of the generated document in various ways. To illustrate provenance itself, the thesis presents a sample application. Based on the insights, it follows up with a concept for capturing provenance. Further on, this concept is used to develop the provenance application itself.

Erklärung

Ich versichere hiermit:

1. dass ich meine Bachelorarbeit mit dem Thema *Entwicklung einer Anwendung zur Aufzeichnung von Provenancedaten in Webanwendungen* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe;
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;
3. dass ich meine Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe;
4. dass die eingereichte elektronische Fassung exakt mit der eingereichten schriftlichen Fassung übereinstimmt;
5. dass ich mit meiner nicht genderneutralen Wortwahl niemanden diskriminieren möchte und sich alle Personenbezeichnungen auf alle Geschlechter beziehen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, 29. August 2023

Benjamin Bauer

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Mit einer steigenden Zahl von Webanwendungen, steigt auch die Komplexität der dahinter stehenden Software. Da diese zum Teil kritische Informationen verbreiten, ist es essenziell, die Korrektheit der damit verbundenen Daten durch die Aufzeichnung von Provenance sicherzustellen. Dabei sollen alle Schritte, in denen eine Webanwendung Daten verarbeitet und erzeugt, chronologisch dokumentiert werden. Durch die Darstellung dieser anhand des standardisierten W3C PROV-Datenmodells kann anschließend zurückverfolgt werden, wann welcher Prozess welche Daten wodurch verändert hat. Dadurch lassen sich anschließend nicht nur Aussagen über die Qualität, Zuverlässigkeit und Vertrauenswürdigkeit der Daten treffen, sondern auch Fehler in der aufgezeichneten Anwendung finden.

1.2 Aufbau der Arbeit

Nach einer sauberen Auflistung der Aufgabenstellungen in Kapitel 2, wird die für die Arbeit verwendete Hard- und Software in Kapitel 3 dargestellt. Um eine Anwendung zur Aufzeichnung von Provenance zu implementieren, wird zuerst ein Beispiel entworfen, an dem die für das Thema notwendigen Grundlagen beschrieben werden (Abschnitt 4). Für dieses Beispiel wird anschließend eine Webanwendung mit dem Framework Django implementiert, anhand derer später die Anwendung zur Aufzeichnung von Provenance demonstriert werden soll (Abschnitt 5.2). Darauf folgend wird ein Konzept erläutert, das eine einheitliche Verarbeitung verschiedener Provenance-Strukturen in Django-Anwendungen ermöglicht. Dazu zählt sowohl das Aufzeichnen von ausgeführten Funktionen, die Änderungen an Datenbankobjekten auslösen als auch der Export von dadurch entstandenen Provenance-Dokumenten in verschiedene

Formate (Abschnitt 5.3). Anhand des Konzeptes wird zuletzt die dafür vorgesehene Anwendung mit Django umgesetzt und implementiert. Dabei werden verschiedene Methoden abgewogen und kritische Entscheidungen im Entwicklungsprozess erläutert (Abschnitt 5.4).

1.3 Ziel der Arbeit

Ziel ist die Implementierung einer Django-Anwendung, die für andere Webanwendungen auf Basis der darin ablaufenden Prozesse und Datenbankänderungen kontinuierlich abgeschlossene Provenance-Dokumente generiert. Dabei sollen Anwender die Provenance-Anwendung für ihre Bedürfnisse über eine saubere Schnittstelle konfigurieren können. Im Rahmen der Bachelorarbeit soll bewiesen werden, dass die Umsetzung der genannten Anwendung möglich und das Konzept als Prototyp für verschiedene Anwendung nutzbar ist.

2 Strukturierte Aufgabenstellung

Im folgenden Kapitel werden die Aufgaben gelistet, die in der Bachelorarbeit bearbeitet werden sollen. Diese werden in Kapitel 5 in der hier angegebenen Reihenfolge erläutert.

1. **Entwurf eines einfachen Beispiels zur Demonstration von Provenance**
   - 1.1 Erklärung des W3C PROV-Datenmodells an diesem Beispiel
   - 1.2 Entwurf eines ER-Modells der verwendeten Klassen im Beispiel
   - 1.3 Auswahl von zwei spezifischen Use Cases für Provenance

2. **Implementieren der Django-Anwendung für das Beispiel**
   - 2.1 Erstellen und Migrieren der Modelle für die Datenbank
   - 2.2 Erstellen von klassenbasierten Views für einen übersichtlichen Ausleihprozess
   - 2.3 Erstellen von funktionsbasierten Views für die Rückgabe von Ausleihen

3. **Erarbeiten eines Konzeptes für die Django-Anwendung zur Aufzeichnung von Provenance**
   - 3.1 Definieren eines abgeschlossenen Provenance-Dokuments
   - 3.2 Erarbeiten von Konzepten für das Verarbeiten von Entities, Activities und Agents
   - 3.3 Erarbeiten eines Konzeptes zum standardisierten Export der Provenance-Daten

4. **Implementieren der Django-Provenance-Anwendung**

   4.1 Implementieren der Anwendung innerhalb des Bibliothekprojektes

   4.2 Implementieren von Verarbeitungsmöglichkeiten für Entities, Activities und Agents

   4.3 Bereitstellen einer Schnittstelle der Anwendung für den Export der Daten

   4.4 Darstellen eines mit der Anwendung aufgezeichneten Provenance-Dokuments

3 Methoden und Verfahren (Verwendete Soft- und Hardware)

Die Arbeit wurde vollständig auf einem Dell Latitude 7400 mit dem Betriebssystem Windows 10 21H2 geschrieben.

Die Implementation der Anwendung erfolgte in PyCharm Professional 2023.1.2 in einer virtuellen Umgebung, in der Python 3.9 verwendet wurde. Zur Installation von externen Pythonbibliotheken wurde Pip in der Version 21.1.2 verwendet [22]. Eigens installierte Pakete sind *prov* in Version 2.0.0 [25] und *Django* in Version 4.2.3 [9]. In Zusammenhang mit diesen Bibliotheken wurden deren Abhängigkeiten automatisch heruntergeladen, diese werden hier nicht aufgelistet. Der bei der Arbeit entstandene Code wurde auf dem internen GitLab des Deutschen Zentrums für Luft- und Raumfahrt versioniert. Die für die Bachelorarbeit erzeugten Grafiken wurden mit dem Plugin „PlantUML Integration“ für Pycharm in Version 6.1.0-IJ2022.2 erstellt. Die verwendete Hard- und Software ist zudem übersichtlich in den Tabellen 3.1 und 3.2 dargestellt.

Hersteller	Modell	Verwendung
Dell	Latitude 7400	Recherche und Implementierung

Tabelle 3.1: Verwendete Hardware

Name	Versionsnummer	Verwendung
Windows 10	21H2	Betriebssystem
PyCharm Professional	2023.1.2	Entwicklungsumgebung
Python	3.9.1	Interpreter
VirtualEnv	20.4.7	Unabhängige Python-Umgebungen
Pip	21.1.2	Installation von externen Bibliotheken
Django	4.2.3	Framework für implementierte Anwendungen
Prov	2.0.0	Schnittstelle zum W3C PROV-Modell
PlantUML Integration	6.1.0-IJ2022.2	Erstellen von Grafiken

Tabelle 3.2: Verwendete Software

4 Grundlagen der Arbeit

In diesem Kapitel werden die Grundlagen der Arbeit dargestellt und erläutert. Dabei wird auch das im Kontext von Aufgabe 1 entstandene Beispiel an verschiedenen Stellen eingebunden und zur Erklärung verschiedener Zusammenhänge verwendet. Welche Aufgabenteile an welchen Stellen beantwortet wurden, ist in Abschnitt 5.1 näher beschrieben.

4.1 Provenance

Der Begriff *Provenance* (dt. Provenienz) leitet sich vom lateinischen *provenire* ab und wird im Duden als „Bereich, aus dem jemand, etwas stammt, Herkunft/-sland“ definiert [5].

Provenance war ursprünglich vor allem im Kunsthandel für den Wert eines Objektes essenziell. Namhafte Vorbesitzer oder Lücken in der Historie konnten den Wert eines Kunstwerks maßgeblich in beide Richtungen beeinflussen [20](S.1). Dass Provenance heute auch in weiteren Feldern Anwendung findet, zeigte die Aufklärung des Pferdefleischskandals in Tiefkühlhamburgern im Januar 2013. Durch die Zurückverfolgung der gesamten Lieferkette konnte der Schlachthof ausfindig gemacht werden, von dem das Pferdefleisch stammte [17]. Da Provenance jedoch nicht nur für den physischen Teil der Welt von Bedeutung ist und in dieser Arbeit eine Anwendung zur Aufzeichnung von Provenance in Webanwendungen implementiert werden soll, wird folgend der Standard des World Wide Web Consortiums (W3C) angewendet:

„Provenance is defined as a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing.“ [21].

Demnach wird die Information über Personen, Institutionen, Entitäten und Aktivitäten, die an der Erzeugung, Beeinflussung oder Weiterleitung eines Objekts beteiligt sind, als *Provenance* bezeichnet. Durch die Aufzeichnung der Provenance, können sowohl Urheber anerkannt, als auch kritische Analysen über die Herkunft von Objekten durchgeführt werden. Mit Hilfe der Analysen sind dann Aussagen über Qualität, Zuverlässigkeit und Vertrauenswürdigkeit der verwendeten Informationen möglich. Im Buch „Provenance: An Introduction to PROV“ von Luc Moreau wird das am Beispiel des Datenjournalismus verdeutlicht. Hier sollen Nachrichten auf Basis von digitalen Informationen erzeugt werden. Durch die Verarbeitung und Vermischung alter Daten können dort bei neuen Veröffentlichungen auch schnell Fehler auftreten. Damit die Journalisten die Korrektheit der Daten verifizieren können, müssen Herkunft und Verarbeitungsschritte nachvollziehbar sein [20](S.2).

4.1.1 Das PROV-Datenmodell des W3C

Das *PROV-Datamodel (PROV-DM)* ist das vom W3C entworfene konzeptionelle, domänenunabhängige Datenmodell für die standardisierte Umsetzung von Provenance. Ein Ziel der Standardisierung ist, Provenance für Maschinen in unterschiedlichen Weisen prozessierbar zu machen. Vom W3C werden derzeit drei Serialisierungsformate bereitgestellt:

1. Die *PROV-Ontology (PROV-O)* beschreibt das PROV-DM in der Web Ontology Language in Version 2 (OWL), was wiederum die Abbildung in das Resource Description Framework (RDF) ermöglicht [19].

2. *PROV-Extensible Markup Language (PROV-XML)* bildet das PROV-DM auf die Extensible Markup Language ab. XML ist ein flexibles Textformat, das vor allem dem Transfer verschiedener digitaler Daten zwischen Systemen dient [26].

3. *PROV-Notation (PROV-N)* soll Provenance-Daten ebenfalls in einer leicht lesbaren Art darstellen und eignet sich durch die Formalität vor allem für die Veranschaulichung von Beispielen [18]. Um folgend die Kernstrukturen von PROV zu veranschaulichen, wird diese Notation verwendet.

Neben diesen gibt es ein weiteres Format, welches nicht vom W3C standardisiert wurde: *PROV-JavaScript Object Notation (PROV-JSON)* repräsentiert das PROV-DM im JSON-Format [13]. Die JavaScript Object Notation (JSON) ist ein leichtgewichtiges,

sprachenunabhängiges, textbasiertes Dateiformat, das hauptsächlich für den Austausch von strukturierten Daten zwischen Clients und Web Services verwendet wird [15].

Namespaces und Qualified Names

Namespaces (dt. Namensräume) bieten die Möglichkeit verschiedene Informationen so zu gruppieren, dass auf sie an anderen Stellen einer Anwendung einheitlich zugegriffen werden kann. In ihrer Empfehlung „Namespaces in XML“ legt das W3C unter anderem die folgenden Merkmale von Namespaces und ihrer Anwendung in XML fest [2]:

- Ein XML Namespace ist eine Sammlung von Namen, welche anhand eines Uniform Resource Identifier (URI) eindeutig identifizierbar sind.
- URIs sind identisch, wenn sie Buchstabe für Buchstabe übereinstimmen.
- Ein Qualified Name besteht aus einem Prefix und einem Localpart. Mit dem Prefix wird einer der im Dokument angegebenen Namespaces referenziert und der Lokalpart stellt einen Identifier in diesem Namespace dar.

Diese Merkmale sollen in dieser Arbeit unabhängig der Deklaration für XML auch für die anderen Serialisierungsformate von Provenance angewendet werden.

4.1.2 Aufbau des PROV-DM

Die Informationen dieses Abschnitts entstammen der Veröffentlichung des W3C über das PROV-DM [21]. Die Beispiele werden im PROV-N-Format dargestellt [18].

Das PROV-DM wird vom W3C in Kernkomponenten für grundlegende Provenance und Erweiterungen für spezifische Use Cases unterteilt. Der Kern von Provenance nach dem W3C ist in Abb. 4.1 dargestellt und beschreibt die Erzeugung von *Entities* durch *Activities*, welche durch *Agents* beeinflusst werden. Dabei kann Provenance in drei Ansichten dargestellt und aufgezeichnet werden [20](S.21,22):

1. Die *Data flow view* (dt. Datenansicht) stellt den Fluss und die Transformation von Informationen in digitaler und physischer Welt durch Entities, die voneinander abgeleitet werden können, dar.

2. In der *Process flow view* (dt. Prozessansicht) werden in einem System ablaufende Prozesse, die Daten verändern können, durch Activities dargestellt.

3. In der *Responsibility view* (dt. Verantwortungsansicht) ist es möglich, Personen oder Institutionen die Verantwortung für Dinge zu geben, die in einem System passiert sind.

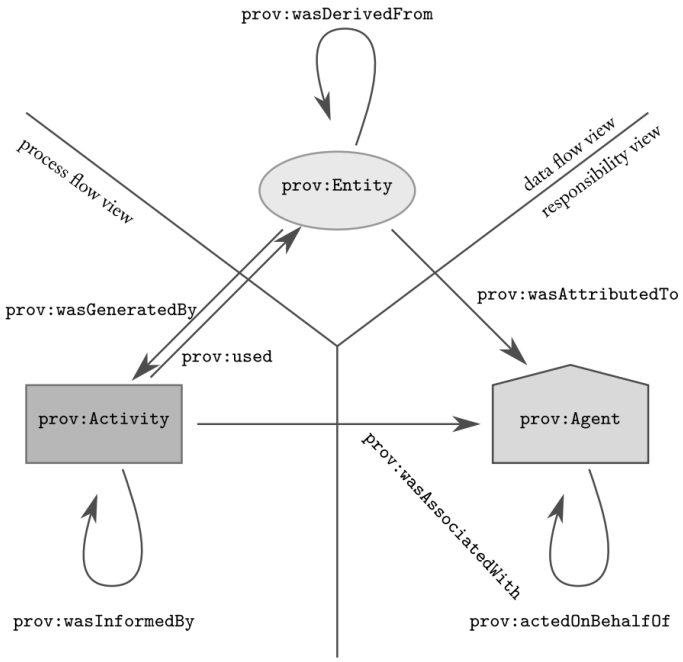


Abbildung 4.1: Kernstrukturen des PROV-DM

Um Objekte in verschiedenen Ansichten im Kern unter- und miteinander zu verbinden, können 7 verschiedene Relationen eingesetzt werden. Auch der Einsatz von Erweiterungen, welche die neuen Strukturen *Bundles* und *Collections*, zusätzliche Subtypen und weitere Relationen in Provenance einbinden ist möglich. Für die Umsetzung der Provenance-Anwendung sind lediglich die Collections relevant.

Kernstrukturen

Folgend werden die Kernstrukturen des PROV-DM erläutert und anhand von für die Arbeit gewählten Beispielen dargestellt. Dabei wird sich mit dem Ablauf einer Ausleihe in der Bibliothek der Dualen Hochschule Baden-Württemberg (DHBW) beschäftigt. Das Datenmodell für dieses Beispiel wird in Abschnitt 4.2.2 erläutert und ist

in diesem Abschnitt zum Verständnis des PROV-DM nicht notwendig. Außerdem wird darauf hingewiesen, dass nicht alle Beispiele und Kernstrukturen auch in der finalen Anwendung in Kapitel 5 vorkommen.

Entities (dt. Entitäten) sind der Hauptbestandteil der Datenansicht, in welcher der Fokus auf dem Fluss der Informationen und den Änderungen der Entities innerhalb eines Systems liegt [20](S.24). Sie stellen physische, digitale oder konzeptuelle Objekte dar und verfügen immer über einen eindeutigen Identifier. Außerdem ist es möglich, ihnen in einer Liste weitere Attribute über Paare mit Schlüssel und dazugehörigem Wert zuzuweisen. Im Beispiel ist eine Entity eines Buchs von Luc Moreau dargestellt. Dabei bildet sich der Identifier aus der International Standard Book Number (ISBN) und dem spezifischen Exemplar. Zusätzliche Attribute sind hier lediglich die Klasse und der Titel des Exemplars.

```
entity(dhbw:book-9781627052214-0,
    [prov:type="dhbw:book",
    dhbw:title="Provenance: An Introduction to PROV"])
```

Activities (dt. Aktivitäten) sind Prozesse, die über einen Zeitraum laufen und Entities modifizieren, verschieben, nutzen oder verarbeiten können. Interagiert eine Activity mit einer Entity, entsteht daraus immer mindestens eine neue Entity. Das Ziel der Prozessansicht ist darzustellen, welche Activities in welchem Kontext an der Veränderung einer Entity beteiligt waren. Zusätzlich zum Identifier und optionalen Attributen in einer Liste können einer Activity eine Start- und Endzeit zugewiesen werden. Im Beispiel ist die Activity vom Bestellen eines Buches, um es auszuleihen dargestellt.

```
activity(dhbw:order-0000,
    2023-08-29T9:00:00, 2023-08-29T15:00:00,
    [prov:type="dhbw:order-a-borrowing"])
```

Agents (dt. Agenten) wird in der Verantwortungsansicht die Verantwortung für Ereignisse gegeben, die in einem System passiert sind. Dabei können Agents sowohl für Entities und Activities als auch für andere Agents verantwortlich sein und sie verfügen über einen eindeutigen Identifier und zusätzliche Attribute. Im Beispiel wird ein Student der DHBW als Agent dargestellt.

```
agent(dhbw:student-8590497,
    [prov:type="dhbw:student", dhbw:name="Benjamin Bauer"])
```

Zusätzlich zu den drei Haupttypen gibt es im Kern 7 Relationstypen, deren Relationen diese (wie in Abb. 4.1 dargestellt) miteinander verknüpfen. Relevant ist dabei die Unterscheidung zwischen unqualifizierten und qualifizierten Relationen. Die simple binäre Relation zwischen zwei Instanzen der Haupttypen ist eine unqualifizierte Relation, die aufgrund der Abgeschlossenheit von Provenanceaufzeichnungen in der Vergangenheit steht. Für komplexere Zusammenhänge wird mit dem qualifizierten Relationsmuster eine neue Instanz für die Relation eingefügt, der zusätzliche Informationen angehangen werden können [10]. Für diese Arbeit reichen unqualifizierte Relationen aus, die anders als die 3 Haupttypen keine zusätzlichen Identifier benötigen.

Die **Derivation** (dt. Ableitung) verknüpft zwei Entities miteinander und ist das Ergebnis einer Transformation der ersten Entity in die zweite. Während hier der Identifier nur optional ist, müssen die genutzte (an zweiter Stelle) und daraus entstandene (an erster Stelle) Entity angegeben werden. Außerdem ist es optional möglich, die Activity, durch die die Derivation entstanden ist, und die damit verbundene Generation oder Usage von weiteren Entities anzugeben. Spezielle Formen der Derivation sind die Revision, die Quotation und die Primary Source. Bei einer Revision ist die neue Entitiy lediglich eine neuere Version der alten. Die Quotation ist eine Zitation einer Entity, bei der die neue eine Kopie der originalen ist. In einer Primary Source wurde die neue Entity direkt von einem Agent mit den dafür benötigten Kenntnissen erzeugt. Im Beispiel wird aus einer angelegten Bestellung die dazugehörige Ausleihe abgeleitet.

```
wasDerivedFrom(dhbw:borrowing-0000, dhbw:order-0000,-,-,-,-)
```

Die **Generation** (dt. Erzeugung) ist die abgeschlossene Entstehung einer neuen Entity durch eine Activity. Das einzig verpflichtende Argument ist hier die entstandene Entity. Optional sind der Identifier, die erzeugende Activity, die Endzeit der Generation und die weiteren Attribute. Im Beispiel wurde die Entity `book-9781627052214-0` durch die Activity `buy-book` erzeugt.

```
wasGeneratedBy(dhbw:book-9781627052214-0, dhbw:buy-book
    ,2022-02-02T11:00:00,-,)
```

Die **Usage** bezeichnet die Nutzung einer Entity von einer Activity für andere Aktionen. Bei der Usage wird lediglich die Activity benötigt, die eine Entity verwendet hat. Optional sind Identifier, die genutzte Entity, die Startzeit der Nutzung und die optionalen Attribute. Im Beispiel wird die Entity `book-9781627052214-0` von

der Activity `borrow-book` genutzt. Das bedeutet, dass die Ausleihe gestartet wurde.

```
wasUsedBy(dhbw:borrow-book, dhbw:book-9781627052214-0,
    2023-08-29T15:01:00)
```

Die **Communication** ist der Austausch von Informationen zwischen zwei Activities. Wird eine Entity von einer Activity erzeugt, kann diese die andere, abhängige Activity darüber informieren. Dafür wird zuerst die informierte und danach die informierende Activity angegeben. Optional sind Identifier und die zusätzlichen Attribute. Im Beispiel wird die Activity `borrow-book` von der Activity `accept-order` darüber informiert, dass sie ihre Arbeit abgeschlossen hat.

```
wasInformedBy(dhbw:borrow-book, dhbw:accept-order,
    2023-08-29T15:00:00)
```

Die **Attribution** wird verwendet, um einem Agent die Zuständigkeit für eine Entity zuzuschreiben. Der Attribution werden die Identifier der zugeschriebenen Entity und des verantwortlichen Agents mitgegeben. Optional sind ein Identifier der Attribution selbst und zusätzliche Attribute. Im Beispiel wird dem Agent Student, die Zuständigkeit für die laufende Ausleihe zugeschrieben.

```
wasAttributedTo(dhbw:borrowing-0000, dhbw:student-8590497)
```

Die **Association** ist ähnlich zur Attribution, schreibt aber dem Agent die Zuständigkeit für eine Activity zu. Verpflichtend ist hier nur der Identifier der zugewiesenen Activity, optional sind der Identifier der Association, der Identifier des Agents und die zusätzlichen Attribute. Außerdem ist es möglich auf eine zusätzliche Entity zu verweisen, welche den Plan des Agents beim Umgang mit der Activity beschreibt. Im Beispiel ist der Agent Bibliothekar für das Akzeptieren der einzelnen Bestellungen zuständig. Hierbei wird standardmäßig im mitgegebenen Plan geprüft, ob das Buch für die Ausleihe bereitsteht.

```
wasAssociatedWith(dhbw:accept-order, dhbw:librarian-0, dhbw:
    checklist-for-orders)
```

Die **Delegation** ist die letzte der Relationen im Kern des PROV-DM. Hierbei überträgt ein Agent einem anderen Agent die Zuständigkeit für eine Activity, um diese für ihn auszuführen. Verpflichtend sind die Identifier des delegierenden und des annehmenden Agents, optional sind die Identifier der Delegation, der übertragenen

Activity und die zusätzlichen Attribute. Im Beispiel überträgt der Bibliothekar mit dem Identifier 0, das Akzeptieren einer Bestellung an einen Kollegen, da er kurzfristig ausfällt.

```
actedOnBehalfOf(dhbw:librarian-1, dhbw:librarian-0, dhbw:accept-
    order)
```

Erweiterungen

Zusätzlich zu den Kernkomponenten gibt es verschiedene Erweiterungen dieser im PROV-DM.

Ein **Bundle** ist eine Sammlung von Provenanceinformationen anderer Entities. Das Bundle ist dabei selbst eine Entity und dient somit der Beschreibung der Provenance von Provenance.

Eine **Specialization** einer Entity ist eine neue Entity, die alle Eigenschaften der originalen teilt und zusätzlich spezifischere Informationen enthält.

Ein **Alternate** einer Entity ist eine weitere Entity, die der Darstellung der gleichen Aspekte wie die der originalen Entity dient, ohne identisch sein zu müssen. Alternates können zusätzliche Informationen enthalten und sich über unterschiedliche Zeiträume strecken.

Eine **Collection** ist eine Entity, die anderen Entities eine umfassende Struktur bietet. Die in der Collection enthaltenen Entityen heißen *Member* und die Beziehung zwischen der Collection und ihren Mitgliedern heißt Membership.

4.1.3 Das PROV-DM als Python-Bibliothek

Die Bibliothek **prov** wurde 2014 von Trung Dong Huynh veröffentlicht und ist eine unter der MIT-Lizenz stehende Python-Implementation des PROV-DM. Prov ermöglicht die Erstellung von durch Pythonobjekte modellierten Provenance-Dokumenten, die anschließend in die Formate PROV-DM, PROV-DM oder PROV-DM serialisiert werden können. Zusätzlich ist es möglich, die Dokumente in verschiedenen grafischen Formaten oder als NetworkX Graph zu exportieren [24]. Die Klassen `ProvEntity`,

`ProvAgent` und `ProvActivity` bekommen ihre Eigenschaften in der Bibliothek immer von der Klasse `ProvRecord` vererbt. Zusätzlich setzt die Bibliothek auch die in Abschnitt 4.1.1 dargestellten Empfehlungen des W3C für Namespaces und Qualified Names um und wendet sie in den Provenance-Dokumenten an. Werden im späteren Teil der Arbeit Strukturen des PROV-DM mit dem Vorwort „Prov“ (z.B. ProvDocument, ProvEntity etc.) verwendet, handelt es sich dabei immer um Klassen oder Objekte dieser Bibliothek.

```
1 doc = ProvDocument()
2 ...
3 e1 = ProvDocument.entity('dhbw:book-9781627052214-0', (
4     (prov.PROV_TYPE, "dhbw:Book"),
5     ('dhbw:title', "Provenance: An Introduction to PROV")))
6 ...
7 doc.serialize("./dhbw.json")
```

Quellcode 4.1: Erzeugung und Serialisierung eines ProvDocuments in Python

In Quellcode 4.1 ist ein Teil des Quellcodes, mit dem Abb. 4.2 erzeugt wurde, dargestellt. Dabei wird in Zeile 1 die Grundstruktur des Provenance-Dokuments mit einer Instanz der Klasse `ProvDocument` deklariert. In Zeile 3 wird eine ProvEntity des Buchs aus der Bibliothek mit einem Identifier und zusätzlichen Attributen erzeugt. Zusätzlich kann das abgeschlossene Dokument in die verschiedenen Formate, hier JSON, serialisiert oder als Grafik exportiert werden.

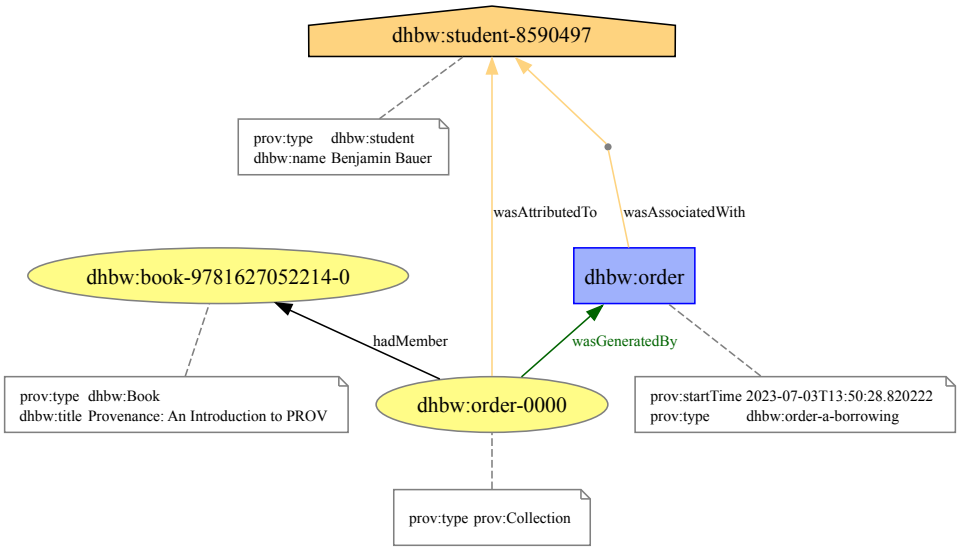


Abbildung 4.2: Beispiel der Provenanceaufzeichnung einer Bestellung in der Bibliothek

4.1.4 Provenance in der Bibliothek der DHBW

Sowohl zur Demonstration von Provenance selbst als auch zur Implementation der Anwendung zur Aufzeichnung von Provenance wurde ein eigenes Beispiel gewählt. Nachdem bereits die Kernstrukturen des PROV-DM bereits anhand des Beispiels erläutert wurden, soll nun dieses selbst dargestellt werden.

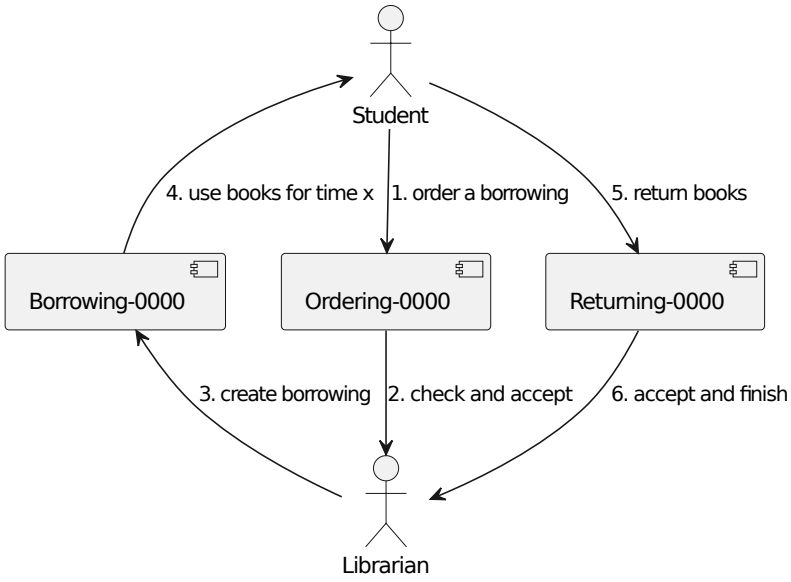


Abbildung 4.3: Vereinfachter Bestellprozess in der Bibliothek

Im Beispiel wird eine Anwendung zum Bestellen von Büchern auf einer Webseite der Bibliothek der DHBW implementiert, die danach zum Ausleihen an der DHBW abgeholt werden können. Dabei gibt es von einem Buch immer nur genau ein Exemplar und es ist möglich, in einer Bestellung mehrere Bücher zu bestellen und diese auch bereits vor Ende der Abgabefrist wieder zurückzugeben. Der Prozess einer Bestellung ist in Abb. 4.3 dargestellt.

Nachdem der Student sich für die auszuleihenden Bücher entschieden hat, legt er eine Bestellung an. Der Bibliothekar prüft danach, ob die Bücher für den gewählten Zeitraum zur Verfügung stehen und in welchem Zustand sie sich befinden. Im optimalen Fall kann er die Bestellung akzeptieren und die Ausleihe starten, die Bücher stehen dem Studenten folgend zur Abholung bereit. Nachdem er sie über den beantragten Zeitraum verwendet hat, gibt der Student die Bücher zurück. Zudem hat der Bibliothekar die Möglichkeit, die Bestellung aus verschiedenen Gründen zu beenden.

Anwendung von Provenance in der Bibliothek

Mithilfe der Aufzeichnung der Provenance der verschiedenen Bestellungen und ablaufenden Prozesse, könnten Mitarbeiter der DHBW z. B. folgende Fragen beantworten:

1. Ein Buch wurde nach eigentlicher Rückgabe nicht als wieder verfügbar dargestellt. Wo ist der Fehler aufgetreten?
2. Ein Buch wurde fälschlicherweise zwei Ausleihen gleichzeitig zugewiesen. Wo ist hier der Fehler aufgetreten und welche Ausleihe bekommt das Buch zuerst?

Da diese Fragen von kritischem Charakter sind und bei einem Eintreten den Betrieb des gesamten Systems beeinträchtigen können, ist ein schnelles Lösen der Probleme unvermeidlich. Da die Provenance-Anwendung alle Änderungen einer Bestellung aufzeichnet, können auch die entscheidenden Fehler dementsprechend schnell ausfindig gemacht und behoben werden.

4.2 Datenbanksysteme

Eine Datenbank (DB) ist eine organisierte Sammlung von strukturierten Informationen oder Daten und wird normalerweise von einem Datenbankmanagementsystem (DBMS) gesteuert. Das DBMS ist für die Organisation und die Strukturierung der Daten in der DB und die Kontrolle der lesenden und schreibenden Zugriffe zuständig. Eine Abfolge von Operationen auf der Datenbank wird Transaktion genannt und die Kombination aus DB und DB heißt Datenbanksystem (DBS). Die Informationen des gesamten Abschnittes stammen, sofern nicht anders gekennzeichnet, aus der Vorlesung und dem zugehörigen Skript „Datenbanken“ an der DHBW Mannheim [4].

4.2.1 Relationale Datenbanken

Eine relationale Datenbank organisiert die Informationen in mehreren miteinander verknüpften Tabellen. welche je eine Sammlung verwandter Daten enthalten. Jede Tabelle besitzt dabei in ihren Zeilen eine Sammlung von gleich aufgebauten Daten, deren Attribute in den Spalten der Tabelle gespeichert werden. In allen Tabellen der Datenbank wird eine Spalte verwendet, um den *Primärschlüssel* (eng. Primary Key (PK)) zu

speichern, über welchen jeder Datensatz eindeutig identifiziert und abgerufen werden kann. Um Beziehungen zwischen den Tabellen darzustellen, wird ein *Fremdschlüssel* (eng. Foreign Key (FK)) verwendet. Dabei handelt es sich um den Primärschlüssel der zu verknüpfenden Tabelle. Relationale Datenbanken folgen dem ACID-Prinzip, welches Regeln für Transaktionen aufstellt und damit für die Verlässlichkeit des Systems sorgt:

Atomizität (eng. Atomicity): Eine Transaktion wird ganz durchgeführt oder gar nicht.

Konsistenz (eng. Consistency): Transaktionen erzeugen einen gültigen Zustand oder fallen in den alten Zustand zurück.

Isolation: (eng. Isolation): Während einer Transaktion greift man auf die Daten zu, als ob es keine konkurrierenden Zugriffe gäbe.

Dauerhaftigkeit (eng. Durability): Nach einer erfolgreichen Transaktion bleiben die Daten dauerhaft gespeichert.

Andere Datenbanktypen

Neben den relationalen Datenbanken gibt es auch andere logische Datenmodelle, die für die Arbeit nicht relevant sind, hier aber trotzdem erwähnt werden sollen:

In einem *hierarchischen* Datenmodell werden die Daten entlang eines Baumes mit Eltern- und Kindknoten gespeichert. Dieses Datenmodell wird in verschiedenen Verzeichnissystemen angewendet.

In einem *Netzwerk*-Datenmodell werden Daten ähnlich zum hierarchischen Datenmodell gespeichert, es ist aber zusätzlich möglich auch nichthierarchische oder bidirektionale Beziehungen abzubilden. Netzwerke können z. B. in Neo4j umgesetzt werden und eignen sich gut für die Speicherung von Provenance-Daten. Dabei können Entities, Agents und Activities durch Knoten und Relationen durch Kanten dargestellt werden.

In einem *objektorientierten* Datenmodell werden Informationen als Objekte gespeichert und können so direkt als Objekte abgerufen werden. Eine ojektorientierte Datenbank lässt sich z. B. mit db4o umsetzen.

In einem *NoSQL*-Datenmodell werden Informationen nicht in Tabellen sondern in verschiedenen anderen Dokumenten wie z. B. JSON-Dateien oder Diagrammen gespeichert. Ein Beispiel dafür ist MongoDB.

4.2.2 Datenbankmodell der Bibliothek

Das **Entity-Relationship (ER)**-Modell ist ein vom Datenbanktyp unabhängiges, konzeptionelles Datenmodell zur Visualisierung von Datenbankstrukturen, welches 1976 von Peter Pin-Shan Chen entwickelt wurde [3]. Das Modell basiert auf den folgenden grundlegenden Elementen:

Entities sind Objekte, die eindeutig identifiziert werden können, z.B. Personen oder Events.

Relationen sind Assoziationen zwischen Entities, z.B. Vater-Sohn ist eine Relation zwischen zwei Entities.

Attribute sind Informationen über Entities, die in Paaren aus Attributen und Werten dargestellt sind, z.B. das Attribut „Name“ mit dem Wert „Benjamin“.

Die Relationen unterscheiden sich anhand der Kardinalität. Diese beschreibt, wie viele Objekte von verschiedenen Entities miteinander in einer Beziehung stehen können:

- Eins zu Eins (1..1): Eine Entity der einen Tabelle ist mit genau einer anderen Entity einer anderen Tabelle verbunden.
- Eins zu Viele (1..m): Eine Entity der einen Tabelle ist mit mehreren Entities einer anderen Tabelle verbunden. Die jeweils anderen Entities sind jedoch nur mit genau dieser originalen Entity verbunden.
- Viele zu Viele (m..n): Eine Entity ist mit mehreren Entities einer anderen Tabelle verbunden. Die Entities der anderen Tabelle können auch mit mehreren Entities verbunden sein.

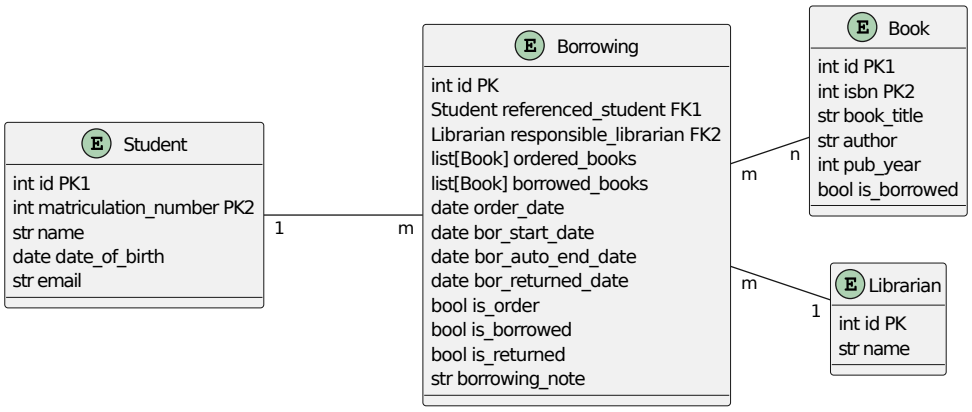


Abbildung 4.4: ER-Modell der Bibliothek

In Abb. 4.4 ist das ER-Modell für das gewählte Beispiel der Bibliothek dargestellt. Als zentrale Entity steht die **Borrowing** im Mittelpunkt der Anwendung. Diese hat einen eindeutige Identifier, über den sie referenziert werden kann und zwei einfache Fremdschlüssel: Den **Student**, der die Bestellung abgegeben hat, und der **Librarian**, der die *Borrowing* bearbeitet. Jeder *Student* ist hierbei theoretisch neben seinem automatisch zugewiesenen Identifier auch über seine siebenstellige Matrikelnummer eindeutig identifizierbar. Da die Identifizierung über den Identifier für das Beispiel ausreicht, bleibt dieser der für die spätere Provenanceaufzeichnung relevante. Außerdem besteht eine `Viele zu viele`-Relation zwischen **Book** und *Borrowing*. Ein *Book* kann in mehreren, sich zeitlich nicht überschneidenden *Borrowings* bestellt werden und ebenso können in einer *Borrowing* mehrere *Books* angefragt werden.

Anhand des Bestellprozesses wurde zwischen zwei Datenmodellen abgewogen. Das letztendlich gewählte ist in Abb. 4.4 dargestellt und die Alternative dafür in Abb. 4.5. Der Unterschied besteht im Verarbeiten einer *Borrowing*. In der Alternative gibt es statt einer großen *Borrowing*-Klasse drei verschiedene, die gemeinsam eine ähnliche Logik umsetzen. Dementsprechend erzeugt ein *Student* bei der Bestellung ein *Order*-Objekt, dass als Referenz in einem neuen *Borrowing*-Objekt verwendet wird. Zusätlich gibt es ein *Review*-Objekt, welches bei der Rückgabe der *Borrowing* erzeugt wird und nach dem Prüfen vom *Librarian* den Abschluss der *Borrowing* darstellt. Das alternative Datenmodell wurde verworfen, da es keinen Gewinn von zusätzlichen Informationen darstellt und für die gleichen Aktionen mehr Operationen durchgeführt werden müssten.

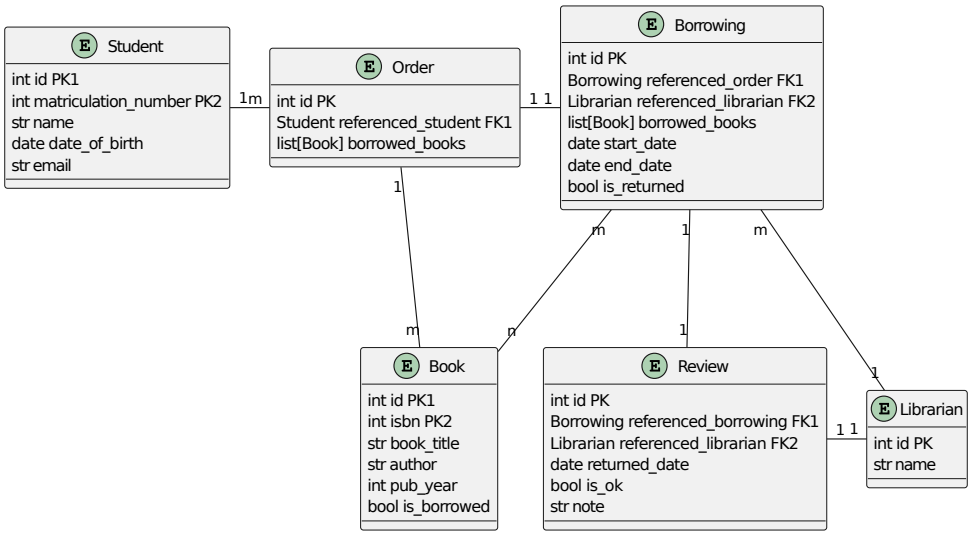


Abbildung 4.5: Alternatives (verworfenes) ER-Modell der Bibliothek

Obwohl das ausgewählte ER-Modell bereits in hoher Simplizität vorliegt, ist es möglich dieses zusätzlich zu normalisieren. Das Ziel einer Normalisierung ist die Reduzierung redundanter Daten und damit die Qualitätssicherung eines Datenbankentwurfs. Dabei gibt es verschiedene Normalformen, die durch eine Normalisierung erreicht werden können:

1. Normalform: Alle Attribute liegen atomar vor.
2. Normalform: Die Bedingungen der 1. Normalform sind erfüllt. Zusätzlich muss jedes Nichtschlüsselattribut vollständig von den Schlüsselkandidaten abhängig sein.
3. Normalform: Die Bedingungen der 2. Normalform sind erfüllt. Zusätzlich darf kein Nichtschlüsselattribut von einem Schlüsselkandidaten transitiv abhängig sein.

Bei einem Schlüsselkandidaten handelt es sich um einen Teil einer Kombination von Attributen, durch die die Einträge einer Tabelle der relationalen Datenbank eindeutig identifizierbar sind. Zusätzlich zu den ersten drei Normalformen gibt es auch die 4. und 5. sowie die Boyce-Codd-Normalform, die jedoch nur selten angewendet werden und hier nicht relevant sind. Im ausgewählten ER-Modell in Abb. 4.4 liegt das Datenbankmodell bereits in der 1. Normalform vor, da die Informationen jeder Tabelle bereits atomar verschiedenen Tabellenfeldern zugeordnet sind. Das Modell steht aber nicht in

der 2. Normalform, da es in den Tabellen *Student* und *Book* mehrere Schlüsselkandidaten gibt, die theoretisch aus den Tabellen entfernt werden könnten. Dabei könnte man die normalen Identifier aus beiden Tabellen entfernen, da die Primärschlüssel bereits durch die Matrikelnummer bzw. die ISBN dargestellt werden. Da eine weitere Normalisierung des Modells für die spätere Anwendung in Django durch automatisches Erzeugen von Identifiern keinen Vorteil bringt, wird auf diese verzichtet.

4.3 Das Django Webframework

Ein Webframework ist ein Programmgerüst, das als Grundlage für die Entwicklung von Web-Anwendungen dient und bereits eine umfassende Auswahl an Basisfunktionen und -klassen bietet. Durch die klare Strukturierung der grundlegenden Komponenten ist es möglich, in kurzer Zeit lauffähige Software zu entwickeln. Dabei wird vor allem auf die Designprinzipien „Don't repeat yourself" (DRY, Vermeidung von Wiederholungen), „Keep it short and simple" (KISS, Einsatz der einfachsten Lösung) und „Convention over Configuration" (je mehr Standardisierung, desto einfacher) gesetzt. Das Grundgerüst jedes Webframeworks sind die miteinander kommunizierenden Klassen, welche statisch und unveränderlich oder flexibel an die Aufgabe anpassbar sein können. Zusätzlich gilt die „Inversion of Control"(IvC), durch die das Framework die Aufgabe eines Hauptprogramms übernimmt und die Komponenten aufruft, wenn sie benötigt werden. Da Webframeworks auf unterschiedlichen Technologien und Architekturen aufbauen, kann man sie unterschiedlich klassifizieren. Während alle Benutzereingaben auf Single-Page-Webanwendungen auf einer einzigen Seite erfolgen, navigiert der Anwender bei Multi-Page-Webanwendungen zwischen verschiedenen Seiten. Auch anhand des Ortes der Anwendungslogik wird zwischen serverzentrischen und clientzentrischen Webanwendungen unterschieden. Während bei serverzentrischen Anwendungen die gesamte Logik auf dem Server bleibt und die spezifischen Daten nach einer Anfrage zur Verfügung gestellt werden, werden die Logik und die Benutzeroberfläche beim Start clientzentrischer Anwendungen auf das Gerät des Clients geladen und dort ausgeführt. Auch die hybride Anwendung auf dem Server und auf dem Client ist möglich, dabei können die jeweiligen Verarbeitungsanteile variieren. Alle Informationen über Webframeworks sind dem Digital Guide von Ionos entnommen [16].

Django ist ein in Python geschriebenes Webframework, welches 2005 von der Django Software Foundation veröffentlicht wurde und unter der BSD-3-Lizenz steht. Die Open-Source Software ermöglicht das Implementieren von pythonbasierten Webanwendungen und bietet zusätzliche Tools, mit denen verschiedene Standardprozesse bei der Webentwicklung schnell umsetzbar sind [6]. Die wichtigsten der Tools werden folgend erklärt und sind ebenfalls in der Dokumentation von Django zu finden.

Architektur - MVC Prinzip

Die Architektur von Django beruht auf dem **Model-View-Controller (MVC)**-Prinzip, kann aber auch anders interpretiert werden [7]. In der Regel wird die Logik einer Webanwendung basierend auf dem MVC-Prinzip dabei auf drei Bereiche aufgeteilt [12]:

1. Im *Model* ist das Datenmodell und die dazugehörigen Funktionen hinterlegt.
2. In der *View* werden die Daten dargestellt. Benutzereingaben, die in der View getätigt werden, werden an den Controller weitergeleitet. Vor allem ist hier relevant, wie die Daten dargestellt werden.
3. Der *Controller* regelt entsprechend der Benutzereingaben, welche Aktionen durchgeführt oder welche Daten und Views angezeigt werden sollen.

In Djangos Views ist es wichtiger, welche Daten man in den Views sieht und nicht auf welche Art dies geschieht. Dafür gibt es verschiedene Template-Klassen, durch die der Fokus des Entwicklers mehr auf das Verarbeiten als auf die Darstellung der Daten gelegt werden kann. Die Art auf die die Daten dargestellt werden, wird durch HTML- und CSS-Templates festgelegt. Auch den Controller als zentrale Schnittstelle gibt es nicht. Das Entwicklerteam verweist darauf, dass Django selbst als Controller arbeitet [7].

Djangos Object Relational Mapper

Auch wenn Django auf dem MVC-Prinzip beruht, muss kein Datenmodell bereitgestellt werden, um eine Anwendung zu implementieren. Soll hinter der Webanwendung dennoch eine relationale Datenbank stehen, bietet Django mit einem **Object Relational Mapper (ORM)** die Möglichkeit, alle Daten durch Pythonobjekte zu modellieren.

```
1 class Book(models.Model):
2     isbn = models.IntegerField(validators=[
3     MaxValueValidator(9799999999999), MinValueValidator
          (9780000000000)])
4     title = models.TextField()
5     author = models.CharField(max_length=60)
```

Quellcode 4.2: Ausschnitt aus dem Datenbankmodell der Bibliothek in Python

Dafür wird in der `models.py` für jede Klasse des Datenbankmodells eine Pythonklasse angelegt, die ihre Eigenschaften von der von Django bereitgestellten Klasse `Model` erbt. Zusätzlich ist es möglich, die Attribute einer Klasse mit verschiedenen Feldtypen zu modellieren. Dabei kann auch festgelegt werden, welche Attribute als Primärschlüssel verwendet und welche anderen Modelle durch Fremdschlüssel verknüpft werden sollen. Durch die Fremdschlüssel in Django werden dabei die in Abschnitt 4.2.2 erwähnten (1..1)- und (1..m)-Beziehungen umgesetzt. Auch die Verwendung von (m..n)-Beziehungen ist in Django möglich und wird durch `ManyToMany`-Felder bereitgestellt. In Quellcode 4.2 ist ein einfaches Modell eines *Books* mit drei zusätzlichen Feldern dargestellt. Dabei ist es möglich dem *Book* eine 13-stellige ISBN nach den Vorgaben der International ISBN Agency [14], einen Titel und einen Autor zuzuweisen. Nachdem die für die Datenbank benötigten Modelle implementiert sind, können diese durch verschiedene Kommandos in Tabellen der Datenbank übertragen werden, was in Abschnitt 5.2.1 genauer erläutert wird.

Administrations-Interface

Zur Verwaltung der Einträge der Datenbank bietet Django ein automatisch erzeugtes Administrations-Interface, mit dem autorisierte Nutzer Objekte hinzufügen, ändern oder löschen können. Dafür müssen gewünschte Modelle in der `admin.py` registriert werden (siehe Quellcode 4.3). Zusätzlich ist es auch möglich, verschiedene Attribute eines Objekts zu ändern.

```
1 admin.site.register(models.Book)
```

Quellcode 4.3: Registrieren eines vorhandenen Modells für das Administrations-Interface

Authentifizierungssystem zur Nutzerverwaltung

Neben dem Admin-Interface bietet Django ein System, das die Authentifizierung und Autorisierung von Nutzern verwaltet und durch das Backend `django.contrib.auth` in Anwendungen eingebundenwerden kann. Neben der Klasse `User` für die Nutzer selbst, bietet dieses Optionen zum Verwalten verschiedener Berechtigungen, Nutzergruppen, Passwörter und zusätzliche Login und Registrierungs-Templates. Es ist zusätzlich möglich, User mit Administrationsrechten in der Commandline mit dem Kommando `python manage.py createsuperuser` zu erzeugen.

Darstellung der Daten mit Views

Bei der Beschreibung von Djangos MVC-Interpretation wurden bereits verschiedene Template-Klassen zur Darstellung der Daten erwähnt. Dabei handelt es sich um `Views`, die nach einer HTTP-Request entweder den angeforderten Inhalt oder eine Fehlerseite zeigen. Anhand der in der Anfrage mitgegebenen Parameter wird ein HTML-Template mit den angeforderten Daten gerendert. Neben der funktionsbasierten Darstellung (siehe Quellcode 4.4) gibt es auch die klassenbasierte Darstellung, welche Standarddarstellungen vereinfacht und dem Code durch die Wiederverwendbarkeit der Klassen mehr Struktur verleiht (siehe Quellcode 4.5). In beiden Beispielen wird eine View zur Darstellung einer Liste von allen Objekten des Book-Modells implementiert.

```
1 def book_list(request):
2     list = Book.objects.all()
3     context = {"list": list}
4     return render(request, "book-list.html", context)
```

Quellcode 4.4: Funktionsbasierte View zum Darstellen aller Book-Objekte

```
1 class BookListView(views.ListView):
2     model = Book
3     template_name = 'book-list.html'
4 
5     def get_queryset(self):
6         return Book.objects.all()
```

Quellcode 4.5: Klassenbasierte View zum Darstellen aller Book-Objekte

Django Signals

Ein für die Umsetzung der Provenance-Anwendung wichtiges Feature von Django sind **Signals**. Da mit Django mehrere kleine Anwendungen in einem großen Projekt betrieben werden können, ist es möglich diese bei verschiedenen Ereignissen in anderen Bereichen des Systems zu informieren [8]. Dabei sendet ein *Sender* dem *Receiver* die Informationen zu, die dieser für sich verarbeiten kann. In Django gibt es vorgefertigte Model-Signale, Management-Signale, Request- und Response-Signale, Signale für Tests und Signale für die Verbindung mit den Datenbanken, zusätzlich können auch eigene Signale definiert werden. Für die Provenance-Anwendung sind vor allem die Model-Signale wichtig, die auslösen, wenn Datenbankeinträge initialisiert (`init`), geändert und gespeichert (`save`) oder gelöscht (`delete`) werden. Für diese Signale gibt es je zwei unterschiedliche Versionen, `pre` und `post`. Diese werden entweder vor oder nach der eigentlichen Aktion gesendet und enthalten dadurch unterschiedliche Informationen. Zusätzlich gibt es das Signal `m2m_changed`, das über Änderungen an ManyToMany-Feldern der Modelle informiert.

Soll ein spezifisches Signal verarbeitet werden, wird zuerst eine Receiver-Funktion geschrieben und diese mit dem Signal verbunden (siehe Quellcode 4.6). Wird ein Eintrag in der Datenbank geändert und gespeichert, wird danach das `post_save`-Signal ausgelöst und die Methode `post_save_handling()` aufgerufen.

```
1 from django.db.models.signals import post_save
2
3 def post_save_handling(sender, instance, **kwargs)
4     print(f"This is the post save handling of {instance}"}
5
6 post_save.connect(post_save_handling)
```

Quellcode 4.6: Verbinden eines Signals mit einem Receiver

Ein Model-Signal in Django verfügt immer über folgende Argumente:

1. *Sender:* Die Klasse, von der ein Objekt geändert wurde.
2. *Instance:* Das Objekt, das geändert wurde.

Zusätzlich gibt es verschiedene weitere Keyword-Arguments, die sich zwischen den Signalen unterscheiden und für die Umsetzung der Provenance-Anwendung irrelevant sind.

5 Durchführung

In diesem Kapitel wird die Bearbeitung der Aufgaben aus der strukturierten Aufgabenstellung in Kapitel 2 erklärt. Dabei sollen die während der Bachelorarbeit erarbeiteten Anwendungen vorgestellt und vor allem auf spezielle Überlegungen bei verschiedenen kritischen Entscheidungen eingegangen werden.

5.1 Entwurf eines einfachen Beispiels zur Demonstration von Provenance

Aufgabenstellung 1 wurde bereits in Kapitel 4 bei der Darstellung der Grundlagen von Provenance eingebunden und soll hier nicht wiederholt werden. Dennoch wird in diesem Abschnitt gezeigt, an welchen Stellen im Text die einzelnen Aufgabenteile bearbeitet wurden:

1.1 Das W3C PROV-Datenmodell wurde an diesem Beispiel erläutert. Die Beispiele sind in Abschnitt 4.1.2 eingebunden.

1.2 Das ER-Modell für die Bibliothek wurde in Abschnitt 4.2.2 erläutert und ist in Abb. 4.4 dargestellt.

1.3 Zwei Use Cases / Fragen, die mit Hilfe der Provenanceaufzeichnung beantwortet werden sollen, sind in Abschnitt 4.1.4 beschrieben.

5.2 Implementieren der Django-Anwendung für die Bibliothek

In diesem Abschnitt wird die Implementierung der Anwendung für das Verarbeiten der Ausleihen von Büchern an der DHBW erläutert. Das Projekt, in dem die Anwendung laufen soll, trägt den Namen `dhbwlib` und wurde mit dem Kommando `django-admin startproject dhbwlib` erzeugt. Dieses initialisiert das Django-Projekt mit verschiedenen Einstellungen und Python-Template-Skripten. Darunter zählt auch die `manage.py`, welche die Interaktion mit dem Django-Projekt über die Commandline ermöglicht. Mit dem Kommando `python manage.py startapp libbackend` wurde die Anwendung `libbackend` erzeugt. Auch dieses Kommando erzeugt verschiedene Python-Skripte, die der Umsetzung der Anwendungslogik und dem Einbinden in das Projekt dienen. In Abb. 5.1 sind alle Dateien des Django-Projektes mit Bibliotheks- und Provenance-Anwendung dargestellt, da sich im folgenden Abschnitt vermehrt darauf bezogen wird.

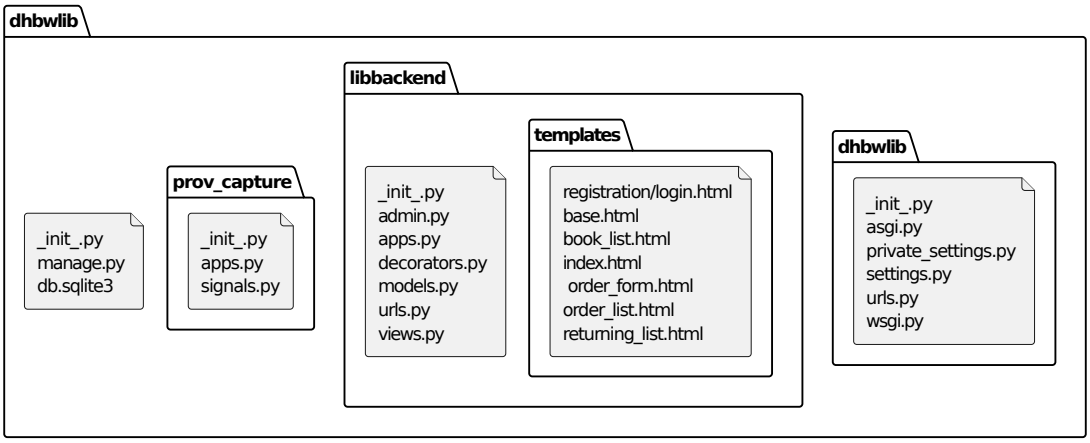


Abbildung 5.1: Verzeichnisübersicht über das gesamte Django-Projekt

5.2.1 Erstellen und Migrieren der Modelle für die Datenbank

Eine der im `libbackend` erzeugten Dateien ist `models.py`. In dieser sollen die Datenbankmodelle aus Abb. 4.4 entsprechend dem ORM aus Abschnitt 4.3 als Pythonobjekte modelliert werden. Die fertigen Modelle sind in Abb. 5.2 dargestellt.

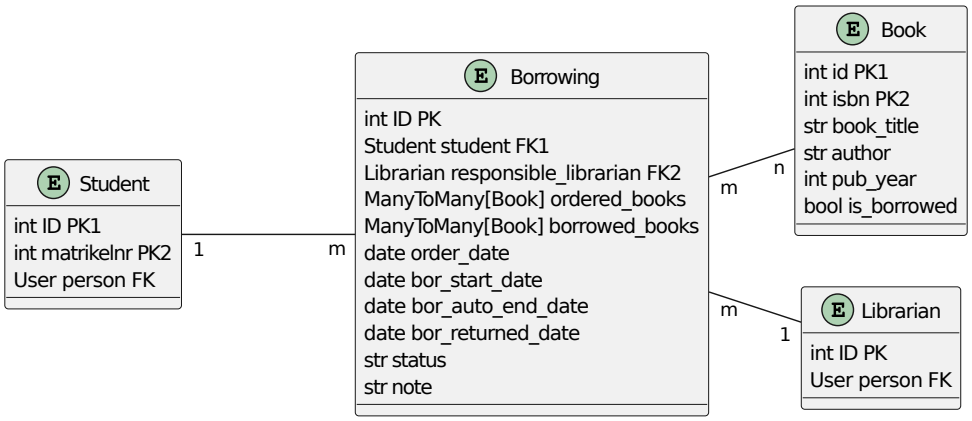


Abbildung 5.2: ER-Modell der models.py in der Anwendung libbackend

Auffallend sind drei Unterschiede zum ER-Modell in Abb. 4.4:

1. Die Klassen *Student* und *Librarian* nutzen einen Fremdschlüssel auf die Instanz „Person“ der Klasse User. Da Django mit dem Authentifizierungssystem eine User-Klasse mitliefert (siehe Abschnitt 4.3), muss die Logik für diesen nicht mehr implementiert werden. Stattdessen wird im *Student* und im *Librarian* lediglich der Fremdschlüssel auf einen vorher angelegten User genutzt. Eine andere Möglichkeit, die Personen darzustellen, wäre auch das Ausbauen der existierenden User-Klasse indem *Students* und *Librarians* nicht mehr von der Klasse `Model`, sondern von der Klasse `AbstractUser` abgeleitet werden. Darauf wird jedoch verzichtet, da der einfache Fremdschlüssel in diesem Fall ausreicht.

2. Die bestellten und ausgeliehenen *Books* werden nicht in einer Liste gespeichert, sondern in einem ManyToMany-Feld. Da es sich hier um eine (m..n)-Beziehung handelt, ist das von Django mitgelieferte Konstrukt eine gute Wahl, um diese zu modellieren.

3. Statt der drei Boolean Variablen für den Status der Bestellung wird ein String verwendet. Dieser String hat drei Auswahlmöglichkeiten: „O“ für Order, „B“ für Borrowing und „R“ für Returned.

Nach dem Erstellen der Modelle in Python müssen diese auf die Datenbank abgebildet werden, dieser Prozess wird in Django durch *Migrationen* ermöglicht. Mit `python manage.py makemigrations libbackend` wird Django über Änderungen an den Datenbankmodellen informiert und aufgefordert, diese als Datei zu speichern. In den

Dateien werden die Migrationen als Python-Code in der Klasse `Migration` für Menschen les- und modifizierbar dargestellt, wodurch die Einsicht auf bevorstehende Aktualisierungen der Datenbank ermöglicht wird. Mit `python manage.py migrate` werden alle noch nicht angewendeten Migrations auf die einzelnen Tabellen der Datenbank angewendet. Für vorher nicht existente Modelle werden dadurch die zugehörigen Tabellen erzeugt und für bereits existierende Modelle diese synchronisiert und gegebenenfalls einzelne Felder angepasst. Damit die Nutzung der Datenbank auch ohne eine direkte Kommunikation über die Webanwendung möglich ist, stellt Django eine automatisch erzeugte API bereit, mit der Datenbankobjekte in Python abgefragt, erzeugt, modifiziert oder gelöscht werden können.

5.2.2 Erstellen von klassenbasierten Views für einen übersichtlichen Ausleihprozess

Durch das von Django mitgelieferte Administrations-Interface (siehe Abschnitt 4.3) ist es möglich, die Tabellen der Datenbank im Browser darzustellen. Dort können einzelne Objekte und ihre zugehörigen Felder bereits vor der Implementierung von eigenständigen Views erstellt und modifiziert werden. Da jedoch die Darstellung der Daten im Administrations-Interface für eine realistische Umsetzung von Bestellungen in der Bibliothek nicht ausreicht, werden folgend Views für unterschiedliche Anwendungsfälle implementiert. Für jede View wird zusätzlich ein HTML-Template erstellt, welches in der Anwendung `libbackend` im Verzeichnis `templates` liegt und der Darstellung der View im Browser dient. Damit die Views untereinander verknüpft werden können, muss für jede selbst implementierte View ein Pfad in der Datei `urls.py` der Anwendung `libbackend` deklariert werden. Zusätzlich gibt es auf Projektlevel ebenfalls eine `urls.py`, in der die URLs der verschiedenen Anwendungen eingebunden werden, das `libbackend` wird mit „lib/“ eingebunden. Folgend werden die für das `libbackend` implementierten Views erläutert.

LoginView

Zuerst sollen sich die User (sowohl *Students* als auch *Librarians*) der Bibliothek einheitlich einloggen können. Da das Authentifizierungssystem von Django bereits existiert und eingebunden ist, sind auch die für den Login benötigten Views bereits vorhanden. Deshalb muss hier nur das HTML-Template `login.html` erstellt werden, welches im

Verzeichnis `registration` gespeichert wird. Durch diesen Speicherort muss Django nicht zusätzlich mitgeteilt werden, wo das Template zu finden ist. Auch eine View für Registrierungen ist möglich, wird hier aber nicht verwendet, da die User für das Beispiel in der Commandline erzeugt werden. In der `urls.py` auf dem Projektlevel wurde das komplette Authentizierungssystem unter dem Pfad „accounts/“ eingebunden. Dadurch liegen theoretisch alle zu diesem Backend gehörige Views unter diesem Pfad, die Login View ist dabei unter „accounts/login/“ erreichbar.

IndexView

Loggt sich ein User ein, wird er zum Index weitergeleitet. Hier wird nach einer Request lediglich das HTML-Template `index.html` mit Links für verschiedene Aktionen in der Anwendung gerendert. Die Funktion `index()` und die Links im Browser sind in Quellcode 5.1 und Abb. 5.3 dargestellt. Der Index ist über den Pfad „lib/“ erreichbar.

```
1 from django.shortcuts import
      render
2 def index(request):
3     return render(request,
4                   'index.html')
```

Quellcode 5.1: Index der Bibliothek

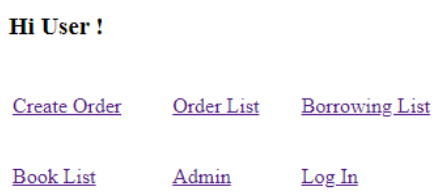


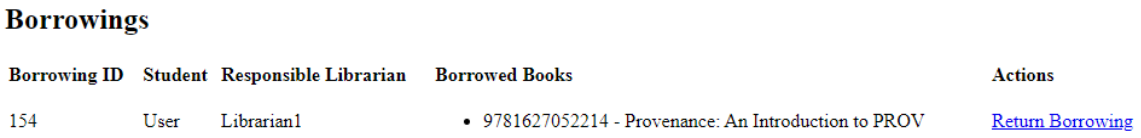
Abbildung 5.3: Index der Bibliothek

Verschiedene ListViews

Klickt man im Browser auf den Link „Book List“, wird man zu einer View weitergeleitet, die unter dem Pfad „lib/books/“ erreichbar ist und alle vorhandenen *Books* in der Bibliothek auflistet. Die dahinterliegende Struktur ist die von Django bereitgestellte `ListView`-Klasse, die mit verschiedenen Argumenten initialisiert wird und anhand einer Query gefilterte Objekte in einem HTML-Template zurückgibt und rendert. Für die Bibliothek wird die Bücherliste mit nur zwei Argumenten initialisiert, dem Modell und dem HTML-Template. Da die Liste aus Büchern bestehen soll, werden dafür das `Book` aus der `models.py` und die Datei `book_list.html` verwendet. Mit der Funktion `get_queryset()` wird die Liste der darzustellenden *Books* erzeugt und zurückgegeben. Da alle existierenden *Books* angezeigt werden sollen, ist ein zusätzlicher Filter nicht notwendig. Die BookListView ist in Quellcode 5.2 dargestellt.

```
1 class BookListView(ListView):
2     model = Book
3     template_name = 'book_list.html'
4
5     def get_queryset(self):
6         return Book.objects.all()
```

Quellcode 5.2: ListView für alle Books der Bibliothek



Borrowings

Borrowing ID	Student	Responsible Librarian	Borrowed Books	Actions
154	User	Librarian1	• 9781627052214 - Provenance: An Introduction to PROV	Return Borrowing

Abbildung 5.4: BorrowingListView von laufenden Borrowings

Auch eine Liste der Bestellungen, Ausleihen und Rückgaben soll abrufbar sein. Dafür wird erneut die `ListView` verwendet, das Modell ist hier aber die *Borrowing*. Die HTML-Templates ähneln dem `book_list.html`, stellen aber unterschiedliche Attribute der *Borrowing* dar. In der Funktion `get_queryset()` werden für jeden Status der *Borrowing* unterschiedliche Filter implementiert. Da sich dieser für Bestellungen nach „O“, für Ausleihen nach „B“ und für Rückgaben nach „R“ unterscheidet, werden diese zum Filtern verwendet. Dafür wird die Funktion `Borrowing.objects.filter(status)` eingesetzt, die die gefilterten Objekte in einer Liste zurückgibt. Anhand dieses Konzepts wurden die drei Klassen *OrderListView*, *BorrowingListView* und *ReturningListView* implementiert. Die Listen sind über „lib/orders/“, „lib/borrowings/“ und „lib/returnings/“ abrufbar. In Abb. 5.4 sind die laufenden *Borrowings* im Browser dargestellt.

OrderCreateView

In Abb. 5.3 ist auch der Link „Create Order“ gegeben. Mit diesem kann ein *Student* die verfügbaren *Books* zum Ausleihen bestellen. Die Klasse, die die Logik für diesen Prozess bereitstellt, ist die *OrderCreateView*, welche von der `CreateView` von Django erbt. Diese gibt auf eine HTTP GET-Request ein Formular zurück, in dem die Attribute eines neuen Objektes gesetzt werden können. Wird das Formular in einer POST-Request zurückgeschickt, kann für das Objekt anhand der mitgegebenen Attribute ein Eintrag

in der Datenbank erzeugt werden. In der *OrderCreateView* werden erneut das Modell *Borrowing* und das dafür vorgesehene HTML-Template gesetzt. Zusätzlich gibt es die Option `fields`, mit der die auf dem Formular auszufüllenden Felder gesetzt werden können. Für die Bestellung reicht in diesem Fall das Feld `ordered_books`. In diesem ManyToMany-Feld können theoretisch beliebig viele von allen vorhandenen Büchern ausgewählt werden. Damit ein *Student* aber nicht die *Books* bestellt, die derzeit in einer *Borrowing* sind, müssen diese gefiltert werden. In der Methode `get_form()` wird zunächst die gleiche Methode in der Elternklasse aufgerufen, um das Formular als Instanz zu erhalten. In diesem wird danach das Queryset vom Feld `ordered_books` so gefiltert, dass nur die *Books* abgebildet werden, die gerade nicht ausgeliehen sind (`is_borrowed=False`). Die Methode ist in Quellcode 5.3 dargestellt.

```
1 def get_form(self, *args, **kwargs):
2     form = super().get_form(*args, **kwargs)
3     form.fields['ordered_books'].queryset = form.fields['
          ordered_books'].queryset.filter(is_borrowed=False)
4     return form
```

Quellcode 5.3: get_form-Methode der OrderCreateView

Da `ordered_books` nicht das einzige Attribut ist, was in einem neuen *Borrowing*-Objekt gesetzt werden soll, soll nach Abschicken des Formulars auch der zugehörige *Student* gesetzt werden. In der Methode `form_valid()` wird dem bereits vorhandenen Formular im neuen Feld „Student“ dieser zugewiesen. Auch dafür muss zuerst der passende *Student* aus der Datenbank gefunden werden. Mit der Query `Student.objects.get(person_id=self.request.user.id)` wird dieser anhand des Users, der die HTTP-Request gesendet hat, gefunden. Durch die vorherige Deklaration im Modell wird auch das Bestelldatum automatisch gesetzt, wenn eine *Borrowing* erzeugt wird. In diesem Fall liegt das nicht am Formular, sondern an einem Default-Wert in der `models.py`. Beim erfolgreichen Verarbeiten des Formulars wird der *Student* mit Hilfe der Methode `get_success_url()` zur Liste aller vorhandenen Bestellungen weitergeleitet, wo auch seine neue Bestellung aufgelistet ist.

Da die Bestellungen ausschließlich von den *Students* angelegt werden sollen, ist es möglich, den *Librarians* den Zugriff auf die *OrderCreateView* zu verweigern. Mit Hilfe von einer Decorator-Funktion können Funktionen der Klasse so erweitert werden, dass diese ihr Verhalten ändern, ohne selbst geändert zu werden. Um den Zugriff auf die *OrderCreateView* für den *Librarian* zu sperren, wurde in der Datei `decorators.py`

die Funktion `user_is_student()` implementiert. Ist die anfragende Person kein *Student*, bekommt sie, die Antwort „Access denied“, was dem HTTP-Response Code 403 entspricht, da die Identität der Person dem Server trotzdem bekannt ist [11]. Die fertige Decorator-Funktion kann folgend mit dem Method-Decorator

```
@method_decorator(user_is_student, name='dispatch')
```

über die *OrderCreateView* geschrieben werden. Dieser ermöglicht die Nutzung des Decorators nicht nur für Funktionen, sondern auch Methoden von Klassen. Damit bereits vor der Darstellung des Formulars geprüft wird, ob ein User das Recht hat dieses anzuzeigen, wird die Methode `dispatch` dekoriert. Diese wird beim Start der *OrderCreateView* nach dem Setup aufgerufen und gibt eine HTTP-Response zurück. Mit dem Decorator wird dort eingegriffen und die Möglichkeit geschaffen, die „Access Denied“-Response zurückzugeben. Die *OrderCreateView* ist unter dem Pfad „lib/order/create/“ erreichbar.

BorrowStartView (UpdateView)

Nach der Bestellung eines *Students* soll der *Librarian* prüfen, ob die bestellten *Books* vorhanden sind und die *Borrowing* starten. Da nach einer Bestellung bereits das passende *Borrowing*-Objekt existiert, wird dafür nicht erneut auf eine `CreateView` zurückgegriffen. Die *BorrowStartView* basiert auf der von Django bereitgestellten `UpdateView` und kann ein bereits existierendes Objekt modifizieren. In der Klasse werden zunächst wieder das Modell, das HTML-Template und die modifizierbaren Felder gesetzt. Zusätzlich zu den bestellten Büchern, können nun auch die ausgeliehenen *Books* geändert werden. Dafür wird in der Methode `get_form()` zuerst nach allen vom *Student* ausgewählten, bestellten Büchern gesucht und danach anhand dieser die für eine *Borrowing* verfügbaren *Books* im Feld `borrowed_books` bereitgestellt. Da nach einer Bestellung eine andere Bestellung möglicherweise zuerst bearbeitet wurde und bestellte *Books* nicht mehr verfügbar sind, ist es möglich, nicht alle bestellten *Books* ausleihen zu können. In der Methode `form_valid()` werden erneut zusätzliche Werte zugewiesen und die Korrektheit des Formulars geprüft. Die zugewiesenen Werte sind:

1. Der zuständige *Librarian*. Dieser wird anhand des Request Users bestimmt.
2. Das Startdatum der *Borrowing*.
3. Das automatische Enddatum der *Borrowing*.

4. Der Status wird von „O“ auf „B“ geändert.

Nach erfolgreichem Start der *Borrowing* wird der *Librarian* auf die Liste aller bestehenden *Borrowings* weitergeleitet. Die *BorrowStartView* ist unter dem Pfad „borrowing/-start/<int:pk>“ erreichbar. Der Platzhalter <int:pk> dient der eindeutigen Identifikation der bereits bestehenden Datenbankeinträge von *Borrowings* anhand ihres Primärschlüssels. Wird dort z. B. der Primary Key 1 eingetragen, wird die *BorrowStartView* für das *Borrowing*-Objekt mit dem Identifier 1 geladen. Durch die erneut gesetzte `get_success_url()` wird der *Librarian* nach dem Starten einer *Borrowing* zur Liste aller laufenden *Borrowings* weitergeleitet.

5.2.3 Erstellen von funktionsbasierten Views für die Rückgabe von Ausleihen

Auch wenn es möglich wäre, alle für die Bibliothek benötigten Views klassenbasiert aufzubauen, sollen für eine umfangreiche Funktionalität der Provenance-Anwendung auch funktionsbasierte Views implementiert werden. Aus diesem Grund sind die Views für Rückgaben von *Borrowings* funktionsbasiert implementiert. Dabei gibt es drei Views für die Rückgabe von Bestellungen:

1. `return_all_borrowings_auto()`
2. `return_all_borrowings()`
3. `return_a_borrowing()`

Die erste Funktion dient der Rückgabe aller Bestellungen, die zu einem bestimmten Zeitpunkt auslaufen. In einem reellen Beispiel würde dies jedoch keinen Sinn ergeben, da die *Books* als physische Exemplare verliehen sind und nicht auf einmal zurückgegeben werden würden. Für die Provenance-Anwendung sollte diese Funktion dennoch einen guten Testfall darstellen, da hier mehrere Datenbankobjekte durch den Aufruf einer Funktion verarbeitet werden. Dafür werden nach einer Anfrage zuerst alle *Borrowing*-Objekte anhand ihres Status „B“ gefiltert, da nur die laufenden *Borrowings* betrachtet werden sollen. Anschließend wird für jedes Objekt geprüft, ob das automatische Enddatum weiter zurück liegt, als das aktuelle Datum. Ist das der Fall, wird eine Helferfunktion aufgerufen, in der die *Borrowing* zurückgegeben (der Status auf

„R“ gesetzt) und für jedes darin enthaltene *Book* die `is_borrowed`-Variable auf False gesetzt wird.

In der zweiten Funktion wird die Rückgabe aller Bestellungen unabhängig des End-Datums umgesetzt. Sie ist nur von *Librarians* ausführbar und ruft für jede der laufenden *Borrowings* ebenfalls die Helferfunktion zur Rückgabe der *Books* auf.

Damit auch *Students* selbst ihre *Borrowings* zurückgeben können, gibt es mit der dritten Funktion die Möglichkeit, eine spezifische *Borrowing* zurückzugeben. Dafür wird beim Aufruf der Funktion der zugehörige Primärschlüssel mitgegeben und anhand dessen die gewünschte *Borrowing* bestimmt. Für diese wird ebenfalls die Helferfunktion aufgerufen und enthaltene *Books* zurückgegeben. Am Ende jeder Rückgabemöglichkeit werden die angemeldeten User zur Liste der Rückgaben weitergeleitet.

5.3 Erarbeiten eines Konzeptes für die Django-Anwendung zur Aufzeichnung von Provenance

Da die Provenance-Anwendung für andere Django-Projekte wiederverwendbar sein soll, reicht eine einfache Anpassung auf die Bibliothek nicht aus. Deshalb soll in diesem Abschnitt ein Konzept entwickelt werden, das es ermöglicht die Provenance-Anwendung auch für andere Projekte gleichermaßen anzuwenden.

5.3.1 Definieren eines abgeschlossenen Provenance-Dokuments

Damit die Provenance-Anwendung ihre Informationen nicht kontinuierlich in das selbe Provenance-Dokument schreibt, muss zunächst ein abgeschlossenes Dokument definiert werden. Dabei sollte es einen sauberen Start- und Endpunkt geben, anhand dessen das fertige Dokument in ein gewünschtes Format serialisiert wird. Da die Provenance-Anwendung beim Start des Django-Servers mit allen anderen Anwendungen im Projekt gestartet wird, soll an diesem Punkt das erste Dokument erzeugt werden und Provenance aufzeichnen. Jegliche Änderungen an den Daten in der Commandline oder im Admin-Interface werden daduch ab dem Start mitgezeichnet und ins Dokument geschrieben. Mit dem Aufruf einer View vom User beginnt ein neues isoliertes

Dokument, für das zuerst alle vorherigen Aufzeichnungen serialisiert, ausgegeben und gelöscht werden. Sollten in der View weitere Views oder andere Funktionen aufgerufen werden, von denen auch die Provenance aufgezeichnet werden soll, gehören diese zum aktuellen Dokument. Erst nach Abschluss der zuerst aufgerufenen View wird auch dieses Dokument serialisiert, ausgegeben und gelöscht. Erfolgen zwischen der letzten aufgerufenen und einer neuen View wieder Änderungen an den Daten, gehören diese erneut zu einem eigenen Dokument.

In einem validen Provenance-Dokument müssen die verschiedenen Namespaces deklariert werden, bevor Records hinzugefügt werden können. Dabei gibt es einen Default-Namespace für den der Name des Django-Projektes eingesetzt wird und verschiedene zusätzliche Extra-Namespaces. Diese sind unbegrenzt möglich und sollen anhand der im Projekt genutzten Anwendungen erzeugt werden. Da es neben den selbst implementierten Anwendungen auch von Django bereitgestellte gibt, ist es nötig zwischen diesen zu unterscheiden. Dabei soll der Anwender selbst entscheiden, welche der Apps aufgezeichnet werden sollen.

5.3.2 Erarbeiten von Konzepten für das Verarbeiten von Entities, Activities und Agents

In einer Django-Anwendung bieten sich verschiedene Komponenten als Repräsentation der Kernstrukturen des PROV-DM an. Da es sich bei Entities und Agents um Objekte, Personen oder Daten handelt, ist es sinnvoll diese durch verschiedene Modelle der aufgezeichneten Anwendung darzustellen. Die verschiedenen Datenbankeinträge von Modellen und zugehörige Änderungen an diesen können direkt als ProvEntity repräsentiert werden. ProvAgents dagegen werden durch Personen oder Institutionen dargestellt, die für verschiedene Aktionen verantwortlich sind. Dadurch ist es möglich, dass Modelle der Anwendung auch als Agent gelten können. In der Bibliothek bieten sich dafür theoretisch die Klassen *Student* und *Librarian* an. Da diese in unterschiedlichen Anwendungsfällen sowohl als Entity als auch als Agent dargestellt werden können, muss auch für beide Fälle eine Verarbeitung ermöglicht werden. Damit die Aufzeichnung der Provenance der Modelle in beiden Repräsentationen möglich ist, soll der Anwender selbst entscheiden, in welcher Repräsentation er die Modelle anwenden will. Soll auch die User-Klasse von Django aufgezeichnet werden, wird empfohlen alle eigenen Modelle als Entities aufzuzeichnen, da zuständige Agents immer über User identifizierbar sein werden. ProvActivities sollen durch Views und Funktionen in den

jeweiligen Anwendungen dargestellt werden. Dabei soll auch hier möglich sein, selbst zu entscheiden, welche Funktionen aufgezeichnet werden sollen und welche irrelevant sind.

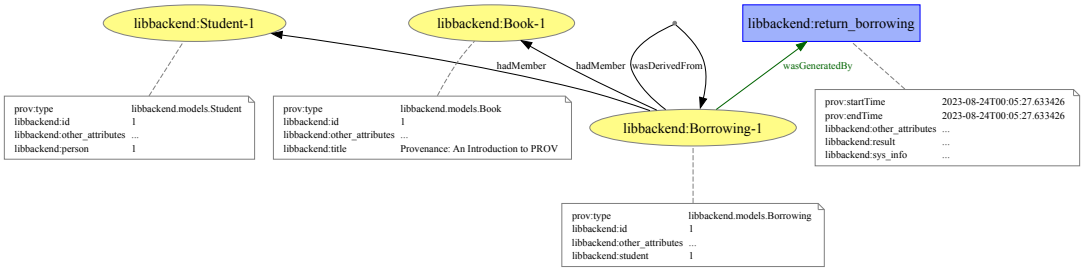


Abbildung 5.5: Unzureichendes Konzept der Provenanceaufzeichnung einer ausgeführten Funktion in der Bibliotheks-Anwendung

In Abb. 5.5 wurde die Funktion `return_borrowing()` vom Agent *Student*-1 ausgelöst. In dieser wurde die ProvEntity *Borrowing*-1 erzeugt, welche wiederum auf die ProvEntity *Book*-1 und *Student*-1 verweist. Problematisch ist hierbei, dass die zurückgegebene *Borrowing*-1 als neue ProvEntity mit dem gleichen Identifier wie die zuvor ausgeliehene *Borrowing*-1 gespeichert wird und dadurch im ProvDocument möglicherweise durch einen ProvRecord dargestellt wird. Um die Darstellung zweier verschiedener ProvRecords als einen einzigen zu umgehen, gibt es folgende Möglichkeiten:

1. Jedes Objekt bekommt einen zweiten Counter im Identifier, der für jedes abgeschlossene Dokument neu hochgezählt wird.
2. Jedem Objekt wird im Identifier der genaue Zeitstempel der Erzeugung zugewiesen.

Der erste Weg ist zwar für jedes abgeschlossene Dokument sinnvoll, möchte man aber Objekte aus mehreren Dokumenten in eine sinnvolle Reihenfolge bringen und miteinander vergleichen, ist das nicht möglich, da keine Zeitinformationen vorliegen. Es wäre auch möglich, eine Kombination aus beiden Ansätzen zu implementieren und den Zeitstempel als zusätzliches Attribut aufzunehmen. Da dies keinen weiteren Informationsgewinn bringen würde und zum Ordnen von Objekten zusätzliche Operationen angewendet werden müssten, wird auf den hybriden Ansatz verzichtet. Demnach wird die zweite Option, das Verwenden eines Zeitstempels in jedem Identifier, gewählt. Eine Provenanceaufzeichnung nach diesem Muster ist in Abb. 5.6 dargestellt.

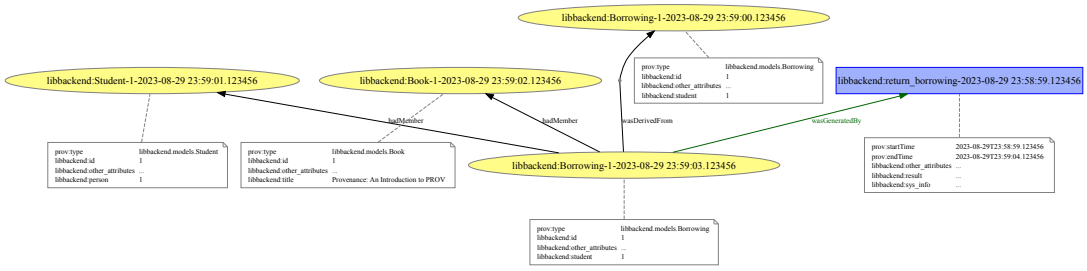


Abbildung 5.6: Konzept der Provenanceaufzeichnung einer ausgeführten Funktion in der Bibliotheks-Anwendung mit Zeitstempeln

5.3.3 Erarbeiten eines Konzeptes zum standardisierten Export der Provenance-Daten

Bereits in Abschnitt 4.1.1 wurden 4 mögliche Serialisierungsformate des PROV-DM vorgestellt, diese werden auch in der prov-Bibliothek in Python unterstützt. Zusätzlich ist es möglich, Dokumente in verschiedenen Bildformaten zu exportieren. Der Anwender soll die Möglichkeit bekommen selbst zu bestimmen, in welchen Formaten und an welchem Speicherort er die Dokumente ausgegeben bekommen möchte. Neben der automatischen Generierung der Dokumente beim Aufruf einer neuen View (siehe Abschnitt 5.3.1) soll es z. B. auch die Möglichkeit geben, jederzeit einen Button in anderen Anwendungen drücken zu können, um das aktuelle Dokument auszugeben.

Die Serialisierung ist vor allem in Hinblick auf die Weiterverarbeitung der Daten wichtig. Durch die einheitliche Serialisierung in standardisierten Formaten ist es möglich, die Daten auch für andere Anwendungen bereitzustellen und beispielsweise in einem großen Store gemeinsam zu lagern.

5.4 Implementieren der Django-Provenance-Anwendung

Nach dem Entwurf der Konzepte für den Umgang mit verschiedenen Provenance-Instanzen behandelt dieser Abschnitt die Implementierung dieser in einer eigenen Django-Anwendung, welche im gleichen Projekt wie die Beispielanwendung laufen soll. Vor allem bei der Implementierung dieser Anwendung gab es verschiedene kritische Entscheidungen, die folgend erläutert werden sollen.

5.4.1 Implementieren der Anwendung innerhalb des Bibliothekprojektes

Der erste Schritt der neuen Anwendung ist der Start dieser im Projekt `dhbwlib`. Mit dem Kommando `python manage.py startapp prov_capture` wurden die Anwendung `prov_capture` und verschiedene Python-Skripte erzeugt. Während diese noch in der Anwendung `libbackend` genutzt wurden, werden sie hier größtenteils nicht eingebunden. Für die Anwendung `prov_capture` werden weder Models für eine Datenbank, noch Views oder das Admin-Interface genutzt. Es werden nur die Dateien `apps.py` zur Konfiguration der Anwendung und `signals.py` zur Verarbeitung der in Django eingebauten Signale verwendet (siehe Abschnitt 4.3).

Die Klasse ProvenanceGenerator als Grundstruktur

Für das Verarbeiten der Entities, Agents und Activities anhand ausgeführter Signale und Views ist eine umfassende Handler-Klasse in der `signals.py` vorgesehen. Von der selbst implementierten Klasse **ProvenanceGenerator** soll dafür beim Start der Anwendung genau ein Objekt instanziiert werden, das bis zum Stopp der Anwendung die Provenance kontinuierlich aufzeichnen und ausgeben soll.

Bei der ersten Implementierung dieses Ansatzes wird ein Objekt des *ProvenanceGenerators* einer lokalen Variable in der `ProvCaptureConfig` in der `apps.py` zugewiesen. Nach dem Verbinden der Django-Signale mit einzelnen Receiver-Funktionen wird jedoch bemerkbar, dass die verbundenen Signale nicht senden, obwohl Objekte in der Datenbank geändert wurden. Der Grund dafür ist der Garbage Collector in Python, der den Speicher der lokalen Variablen nach dem Start der App freigibt, da dieser nicht mehr benötigt wird. Garbage Collection ist eine Methode zum Verwalten von Speicher, welche automatisch Speicher zurückgewinnt, wenn dieser von der Anwendung nicht mehr benötigt wird oder erreichbar ist. Dadurch können Datenlecks verhindert und eine effiziente Speichernutzung gewährleistet werden [1]. Damit dieses Problem nicht mehr auftritt, wurde statt der lokalen Variable die Klassenvariable `generator_instance` in der ProvCaptureConfig deklariert. Beim Start der Anwendung wird diese jetzt mit einer Instanz des *ProvenanceGenerators* initialisiert und empfängt durchgehend die gesendeten Signale. Da es immer nur genau eine Instanz des *ProvenanceGenerators* geben und diese auch an anderen Stellen der Anwendung erreichbar sein soll, wird für die Klasse ein Singleton-Pattern implementiert. Dieses ist ein Designmuster, das

dafür sorgt, dass von einer Klasse nur eine einzige Instanz existiert, welches dennoch in der gesamten Anwendung zugänglich ist [12](S. 127). Im *ProvenanceGenerator* wird dafür die private Klassenvariable `_instance` deklariert, in der diese Instanz später gespeichert wird. Damit geprüft werden kann, ob bereits eine Instanz existiert und wenn nicht, diese erzeugt wird, wird die Klassenmethode `get()` implementiert. Gibt es noch keine, erzeugt diese eine neue, speichert sie in _instance und gibt sie zurück. Existiert _instance bereits, wird diese direkt zurückgegeben. Durch die get()-Methode ist das Objekt an jeder Stelle der Anwendung erreichbar.

Initialisieren des ProvenanceGenerators und Erzeugen eines neuen Provenance-Dokuments

```
1  PROVENANCE = {
2      "ENTITIES": [ 'libbackend.Book', ...],
3      "AGENTS": [...],
4      "NAMESPACES": {
5          "DEFAULT": f"{ROOT_URLCONF.split('.')[0]}.org/",
6          "EXTRA": [ "auth", "django", "libbackend", "prov_capture"]
7      },
8      "OUTPUT": {"PATH": f"captures/",
9          "SERIALIZE": ["xml", ...],
10         "GRAPHIC": ["svg", ...],
11     },}
```

Quellcode 5.4: Dictionary in der settings.py zum Konfigurieren des ProvenanceGenerators

Wird der *ProvenanceGenerator* initialisiert, wird diesem ein Default-Namespace mitgegeben, der in der Instanz-Variable `default_namespace` gespeichert wird. Der Default-Namespace kann vom Anwender in der `settings.py` auf Projekt-Ebene im Dictionary „`PROVENANCE`“ in dem darunterliegenden Dictionary „`NAMESPACES`“ unter dem Key „`DEFAULT`“ gesetzt werden. Da in der `settings.py` auch der Name des Projektes mit dem Kommando `ROOT_URL_CONF.split('.')[0]` abrufbar ist, ist dieser zwar unter dem „`DEFAULT`“-Wert gesetzt, bleibt aber änderbar. Das Dictionary „`PROVENANCE`“ dient grundsätzlich der Konfiguration der Provenance-Anwendung an einer Stelle und wird in Quellcode 5.4 dargestellt. Im Verlauf von Abschnitt 5.4 werden auch die hier noch nicht erwähnten Einträge erläutert.

Nachdem der Default-Namspace gesetzt wurde, werden aus dem „`PROVENANCE`"-Dictionary unter den Keys „`ENTITIES`" und „`AGENTS`" alle zugehörigen Modelle abgerufen, für die die Provenance aufgezeichnet werden soll. Da es möglich sein soll nicht alle vorhandenen Modelle aufzuzeichnen, wird so das spezifische Verbinden der Signale mit ausgewählten Modellen ermöglicht. Die daraus entstehenden Listen werden in den Instanzvariablen `entities` und `agents` des *ProvenanceGenerators* gespeichert.

Damit beim Start des *ProvenanceGenerators* ein neues Provenance-Dokument in einem ProvRecord angelegt wird, wird bei der Initialisierung einer Instanz die Methode `create_new_document()` aufgerufen. In dieser wird die Instanzvariable `executing_activities`, die eine Liste aller noch ausführenden Activities speichern soll, deklariert und mit einer leeren Liste initialisiert. Warum die ausführenden Activities gespeichert werden sollen, wird später in Abschnitt 5.4.2 erläutert. Danach wird der Instanzvariable `document` ein leeres `ProvDocument` zugewiesen. Diesem wird zuerst der in `default_namespace` gespeicherte Default-Namespace und danach die in den Einstellungen gesetzten Extra-Namespaces hinzugefügt. Für diese gibt es in den Einstellungen im Dictionary „`PROVENANCE`" unter dem Dictionary NAMESPACES den Key EXTRA, in dem eine Liste von zusätzlichen Namespaces angegeben wird. Vorgesehen sind dafür die eigens implementierten Anwendungen, von denen Provenance aufgezeichnet werden soll, möglich sind aber alle installierten Apps des Projektes. Um die Extra-Namespaces hinzuzufügen wurde die Methode `add_extra_namespace` implementiert. Damit es keine ungewollten Dopplungen und damit neue Namespaces im ProvDocument gibt, wird hier überprüft, ob ein potenzielles Prefix bereits existiert und dieses hinzugefügt, wenn das nicht der Fall ist. Neben den aus dem Django-Projekt gewonnen Namespaces ist es möglich, zwei weitere hinzuzufügen: Mit dem Namespace `auth` wird die Aufzeichnung der User-Klasse von Django als ProvAgent und mit dem Namespace `sys` die Aufzeichnung von zusätzlichen Systeminformationen ermöglicht.

5.4.2 Implementieren von Verarbeitungsmöglichkeiten für Entities, Activities und Agents

Im folgenden Abschnitt wird genauer auf die Klasse *ProvenanceGenerator* eingegangen und erläutert, wie die in Abschnitt 5.3 dargestellten Konzepte umgesetzt wurden. Dafür

ist in Abb. 5.7 der gesamte Aufbau des *ProvenanceGenerators* dargestellt, auf den sich über den gesamten Abschnitt hinweg immer wieder bezogen werden wird.

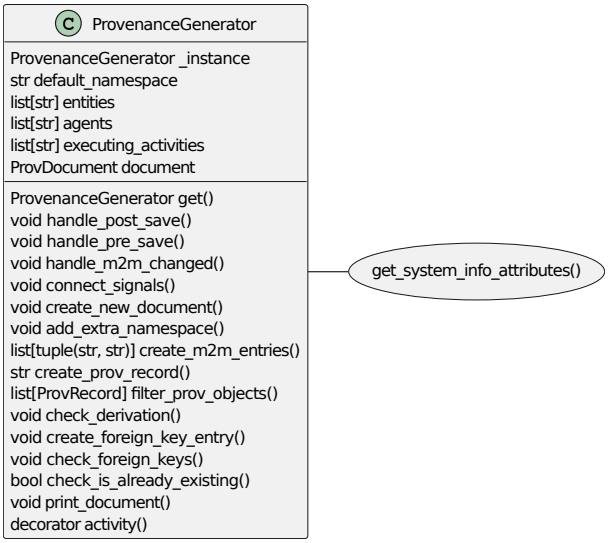


Abbildung 5.7: Klassendiagramm des ProvenanceGenerators

Verständnis und dynamisches Verbinden der Signale

Der letzte Schritt der Initialisierung eines *ProvenanceGenerators* ist das Verbinden der Signale mit den einzelnen Modellen. Dafür wird die Methode `connect_signals()` aufgerufen und für jeden Eintrag darin die dafür vorgesehenen Signale entsprechend Abschnitt 4.3 mit den Receiver-Funktionen verbunden. Von den in Abschnitt 4.3 erwähnten Signalen sind hierbei nur die folgenden für die Provenanceaufzeichnung relevant:

1. **pre_save**
2. **post_save**
3. **m2m_changed**

Das `pre_save`-Signal wird ausgelöst, bevor ein Objekt in der Datenbank gespeichert wird, um den Zustand von diesem vor einer Änderung zu erhalten. Im `post_save`-Signal ist es möglich, den Zustand des Objektes nach der Speicherung abzurufen. Prinzipiell ist dieser auch im `pre_save` als zusätzliches Argument gespeichert und es könnte sinnvoll sein, die Informationen des `post_save`-Signals nicht separat zu

verarbeiten. Da jedoch die Möglichkeit besteht, dass Datenbankinteraktionen während der Speicherung einen Fehler erzeugen und unterbrochen werden, hat man nur durch das `pre_save`-Signal keine Information über den Erfolg einer Transaktion. Dadurch könnte die Provenance-Aufzeichnung inkonsistent werden, weshalb auch das `post_save`-Signal verwendet werden soll. Da aber der Anwender in diesem Signal wiederum zu wenig Informationen über geänderte ManyToMany-Felder in einem Objekt hat, muss auch das Signal `m2m_changed` verwendet werden. Dabei gibt es verschiedene Aktionstypen, die in dem Signal mitgesendet werden können: `add`, `remove` und `clear` und davon je die Versionen `pre` und `post`. Für den *ProvenanceGenerator* sollen nur die Aktionen `post_add`, welche beim Hinzufügen von Elementen zur Relation ausgelöst wird und `post_remove`, welches beim Löschen von Elementen aus der Relation ausgelöst wird, betrachtet werden. Bei der späteren Verarbeitung von ManyToMany-Feldern ist außerdem besonders zu beachten, dass das Signal `m2m_changed` erst nach dem `post_save` ausgelöst wird. Das liegt an der Speicherung der Inhalte von ManyToMany-Feldern in eigenen Tabellen und nicht in der Tabelle der Klasse, in der sie verwendet werden. Durch die Änderung eines ManyToMany-Felds in einer Instanz und der damit verbundenen Speicherung in der dafür vorgesehenen Tabelle wird zuerst das `post_save`-Signal für diese Instanz gesendet. Dort gibt es allerdings noch keine Informationen über die Änderungen der ManyToMany-Felder und es ist dadurch nicht möglich herauszufinden, ob der post_save-Zustand das finale Objekt nach einer Aktion repräsentiert. Die Tabelle des ManyToMany-Felds kann erst danach bearbeitet werden, weil der Fremdschlüssel zur originalen Instanz erst nach ihrer Speicherung zum Referenzieren bereitsteht.

In der Methode `connect_signals()` wird anschließend durch jeden Eintrag der Entities und Agents iteriert und dabei die Signale `pre_save`, `post_save` und wenn vorhanden `m2m_changed` mit den Handler-Funktionen `handle_pre_save`, `handle_post_save` und `handle_m2m_changed` verbunden. Diese werden immer mit den in den Signalen enthaltenen Sendern, Instanzen und zusätzlichen Keyword-Arguments aufgerufen (siehe Abschnitt 4.3).

Verarbeitung von Entities und Agents

Die in Abschnitt 5.3.2 vorgestellten Konzepte für Entities und Agents werden nun im *ProvenanceGenerator* umgesetzt. Bei der Verarbeitung der verschiedenen Signale sollen dabei für die jeweiligen Objekte die zugehörigen ProvRecords erzeugt werden, dieser

Prozess ist für Agents und Entities bis auf die Erstellung des finalen ProvRecords gleich. Dafür wird in allen Handler-Funktionen die Methode `create_prov_record()` mit unterschiedlichen Informationen zum Objekt, für das ein ProvRecord erzeugt werden soll, aufgerufen:

1. **handle_pre_save():** Von der erhaltenen Instanz wird das zugehörige originale Datenbankobjekt vor der Speicherung abgerufen. Für dieses wird anschließend die Methode `create_prov_record()` mit dem Sender, der originalen Instanz und den Keyword-Arguments aufgerufen. Gibt es kein originales Objekt, bedeutet das, das Objekt wurde neu erzeugt und es soll nichts getan werden.
2. **handle_post_save():** Es wird direkt `create_prov_record()` mit dem Sender, der erhaltenen Instanz und den Keyword-Arguments aufgerufen.
3. **handle_m2m_changed():** Nachdem die Einträge eines ManyToMany-Feldes geändert wurden, wird `create_prov_record()` aufgerufen, wenn es noch keine zugehörigen ProvRecords gibt. Anschließend werden diese Records mit dem ProvRecord des Objektes verbunden, in dem die ManyToMany-Felder geändert wurden.

Bei einem Aufruf von `create_prov_record()` werden für das mitgegebene Objekt zuerst die für die ProvRecords benötigten Attribute erstellt. Damit dies für jedes Objekt unabhängig ist, wird durch alle in dessen Modell deklarierten Felder iteriert. Dadurch wird ein neues Attribut immer nach folgendem Muster erzeugt:

```
1 for f in obj._meta.fields:
2     new_attribute = (f'{obj._meta.app_label}:{f.attname}',
3             str(getattr(obj, f.attname)))
4 identifier =
```

Quellcode 5.5: Erzeugen neuer Attribute für ProvRecords

Gibt es in einem Modell zusätzliche ManyToMany-Felder, werden mit der Methode `create_m2m_entries()` auch durch diese iteriert und für jedes Feld ein neues Attribut mit den darin enthaltenen Objekten erzeugt. Nachdem alle Attribute erzeugt wurden, wird der Identifier mit einem ähnlichen Muster anhand des Konzeptes in Abschnitt 5.3.2 erstellt. Danach soll mit der Methode `check_is_already_existing()` geprüft werden, ob es bereits einen ProvRecord des neuen Identifiers mit identischen Attributen gibt, da die Ableitung eines ProvRecords vom selben ProvRecord nicht sinnvoll ist. Weil es aber trotzdem möglich ist, dass in der Historie ein Objekt mit

den gleichen Attributen existiert, darf diese Prüfung nicht für alle existierenden ProvRecords eines Objektes durchgeführt werden. Deshalb wird zuerst mit der Methode `filter_prov_objects()` nach allen Objekten mit gleichem Identifier, ausgeschlossen ist dabei der Zeitstempel, gesucht. Da diese in einer Liste nach dem Zeitstempel geordnet zurückgegeben werden, reicht es nur für das letzte Objekt der Liste die Methode `check_is_already_existing()` aufzurufen. In dieser wird die Gleichheit der bereits existierenden Attribute des ProvRecords und der vorher neu erstellten Attribute geprüft. Sind diese identisch, gibt die Methode True zurück, sind sie es nicht, wird False an `create_prov_record()` zurückgegeben. Da `create_prov_record()` in jedem Fall einen Identifier an den aufrufenden Handler zurückgeben soll, gibt die Methode im Fall von identischen Attributen immer den Identifier des bereits existierenden ProvRecords zurück. Dadurch kann dieser im weiteren Prozess durch den Einsatz von Relationen mit anderen Records verknüpft werden. Gibt es jedoch keinen identischen ProvRecord, wird geprüft, ob das Modell des Objektes in den Einstellungen den Entities oder den Agents zugeordnet ist und je nach Zuordnung eine ProvEntity oder ein ProvAgent mit dem Identifier und den Attributen erzeugt. Wenn das Modell keinem der beiden ProvRecords zugeordnet werden kann, wird eine Warnung ausgegeben und das zugehörige Objekt übersprungen. Abhängig des Typs des neuen ProvRecords folgen nun unterschiedliche Schritte. Ist der ProvRecord den Agents zugeordnet, wird lediglich der ProvAgent erzeugt und dessen Identifier zurückgegeben. Ist der ProvRecord jedoch eine ProvEntity, folgt die Überprüfung nach Derivations, bei denen neue Entities von bestehenden Entities abgeleitet werden (siehe Abschnitt 4.1.2). Dafür wird die Methode `check_derivation()` aufgerufen, die wiederum anhand eines mitgegebenen Identifiers nach allen ähnlichen ProvEntities sucht. Zu diesen zählen dabei alle, die den mit Ausnahme des Zeitstempels gleichen Identifier besitzen. Für eine saubere Überprüfung werden außerdem vorbereitend die bereits existieren ProvDerivations in dem Dokument gesucht, da es später keine Dopplungen geben soll. Danach wird folgender Algorithmus angewendet:

1. Wenn die Liste aller ähnlichen Entities länger als 1 ist:
2. Für i in der Länge der Liste - 1:
3. Variable für eine identische Derivation = None
4. Für jede Derivation in allen Derivationen:

5. Wenn die Derivation von Entity[i+1] und Entity[i] bereits existiert, prüfe die nächste Derivation (Schritt 4).

6. Wenn sie nicht existiert, dann erstelle sie neu.

Nachdem dies für die neue Entity durchgeführt wurde, wird der letzte Schritt in `create_prov_record()` durchgeführt, die Überprüfung von Fremdschlüsseln im Objekt. Wenn das Objekt Fremdschlüssel oder ManyToMany-Felder mit Einträgen besitzt, sollen auch für diese die zugehörigen ProvRecords und Relationen im Dokument erstellt werden. Dafür wird die Methode `check_foreign_keys()` mit der zu überprüfenden Klasse und dem Objekt aufgerufen und geprüft ob im Objekt enthaltene Felder Instanzen eines Fremdschlüssels oder eines ManyToMany-Feldes ist. Ist eine der genannten Instanzen vorhanden, dann wird für die darin referenzierten Objekte die Methode `create_foreign_key_entry()` aufgerufen und ProvRecords erzeugt, wenn diese noch nicht existieren. Dabei wird zuerst das zugehörige Objekt, das über den Fremdschlüssel oder das ManyToMany-Feld referenziert wurde, aus der Datenbank abgerufen und danach für dieses die Methode `create_prov_record()` aufgerufen. Anhand des neuen ProvRecords wird je nach dem Typ dessen folgende Relation angelegt:

1. Ist der neue ProvRecord vom Typ ProvEntity, wird eine `Membership` erzeugt. Dabei ist die bereits existierende Entity die Collection und der neue ProvRecord der Member.

2. Ist der neue ProvRecord vom Typ ProvAgent, wird eine `Attribution` zwischen der existierenden Entity und dem neuen Agent erzeugt.

Verarbeitung von Activities

Für die Darstellung von Activities im Provenance-Dokument wird es zwei verschiedene Fälle geben, für die eine Verarbeitung bereitgestellt werden muss:

1. Eine klassenbasierte View wird aufgerufen und dabei ein Objekt erzeugt oder modifiziert.

2. Eine funktionsbasierte View oder eine Funktion wird aufgerufen und dabei ein Objekt erzeugt oder modifiziert.

In beiden Fällen könnte dafür entweder ein selbst implementierter Dekorator oder ein eigenes Signal verwendet werden, da Django keine Signale für Views oder Funktionen bereitstellt. Für das Verbinden der einzelnen Views und Funktionen mit dem *ProvenanceGenerator* gibt es dadurch drei Optionen:

1. Dekorieren der Funktion direkt im Code mit dem „@“-Operator
2. Deklarieren der zu dekorierenden Funktionen in der `settings.py` im „`PROVENANCE`“ Eintrag.
3. Implementieren eines eigenen Signals, dieses wird beim Start der Anwendung mit den aufzuzeichnendenden Funktionen verbunden.

Alle drei Varianten sind theoretisch möglich, im Rahmen der Arbeit wurde jedoch nur die erste umgesetzt. Dafür wurde in der Klasse *ProvenanceGenerator* unter der Methode `activity()` die Funktion `_decorator()` und darunter die Funktion `wrapped_func()` implementiert. Die drei Funktionen haben folgende Zwecke:

1. Die **activity** ist die Funktion, mit der andere Funktionen dekoriert werden können. Dabei kann der zusätzliche Parameter `name` mitgegeben werden, durch den der Name der Funktion im ProvDocument verändert werden kann.
2. Durch die Funktion **_decorator** wird die Information der dekorierten Funktion beibehalten, da diese sonst nur noch als `wrapped_func()`-Objekt identifizierbar wäre. Wird beim Dekorieren der Funktion ein spezieller Name mitgegeben, wird dieser hier als neuer Name der Funktion gesetzt. Die Funktion gibt zum Abschluss das `wrapped_func()`-Objekt zurück.
3. In der Funktion **wrapped_func** wird die eigentliche Funktion dekoriert. Dabei wird vor und nach dem Aufruf und Speichern des Rückgabewertes der Funktion neuer Code eingefügt. Die Funktion gibt zum Schluss das Ergebnis zurück, das ihr die dekorierte Funktion zurückgegeben hat.

Der Großteil der Logik für das Erstellen von ProvActivities liegt in der dritten Funktion, dementsprechend wird diese folgend genauer erklärt. Da in einer Activity sowohl die Start- als auch die Endzeit aufgezeichnet werden kann (siehe Abschnitt 4.1.2), ist das Speichern der Startzeit der erste Schritt, wenn die `wrapped_func()` aufgerufen wird. Danach muss der im ProvDocument referenzierte Namespace für die dekorierte Funktion abgerufen werden. Da diese innerhalb einer Anwendung des

Django-Projektes liegt, und eine Anwendung ein Python-Modul ist, lässt sich der zugehörige Namespace durch `label = func.__module__.split('.')[0]` identifizieren. Liegt die Funktion nicht innerhalb des Django Projektes, muss dafür eine andere Verarbeitung bereitgestellt werden, worauf für die Bachelorarbeit vorerst verzichtet wurde. Ist der Namespace bestimmt, sollen für die ProvActivity alle zusätzlichen Attribute gesammelt werden, auf die bereits vor dem Rückgabewert der dekorierten Funktion zugegriffen werden kann. Dabei ist der erste Eintrag immer die Funktion und ihre Speicheradresse selbst, gefolgt von den `args` und den `kwargs`, wenn diese vorhanden sind. Zusätzlich ist es möglich, auch Systeminformationen in den Attributen durch den Aufruf der außerhalb des *ProvenanceGenerators* liegenden Funktion `get_system_info_attributes()` zu speichern. Dabei werden neue Attribute mit Informationen über das Betriebssystem, die Interpreterversion und den auf dem System eingeloggten Nutzer erzeugt, die danach an die vorhandenen Attribute angehangen werden können. Da diese Systeminformationen vom Anwender selbst konfigurierbar sein sollen, liegt die Funktion außerhalb des *ProvenanceGenerators*. Kurz bevor die dekorierte Funktion tatsächlich aufgerufen wird, wird sie der in Abschnitt 5.4.1 erwähnten Liste von aktuell ausführenden Funktionen hinzugefügt. Nachdem der Rückgabewert der Funktion in der Variable `result` gespeichert wurde, wird die Funktion wieder aus der Liste entfernt. Das kurzzeitige Speichern aller ausführenden Funktionen in dieser geschieht aus folgendem Grund: Da eine Funktion auch eine andere aufzuzeichnende Funktion aufrufen kann, muss die Reihenfolge aller Funktionen stets nachvollziehbar sein. Durch die Liste, die einen Stack implementiert, gibt es an jedem Punkt des Prozesses die Information, welche Funktionen noch ausgeführt werden. Erst mit Abschluss der letzten Funktion und somit einer wieder leeren Liste, kann auch das ProvDocument abgeschlossen werden. Nachdem die ausgeführte Funktion aus der Liste entfernt wurde, wird die Endzeit gespeichert und ihr Rückgabewert den Attributen angehangen. Danach wird in dem ProvDocument eine ProvActivity mit dem dazugehörigen Identifier, der Start- und Endzeit und den Attributen erzeugt. Der Identifier setzt sich dabei aus dem Namespace, dem Namen der Funktion und der Startzeit zusammen.

Nachdem die ProvActivity erzeugt wurde, müssen die verschiedenen Relationstypen für ProvRecords, die mit ihr in Verbindung stehen, geprüft werden. Die Prüfung der zugehörigen ProvEntities ist dabei der erste Schritt. Da in den `args` verschiedene Objekte der Datenbank referenziert sein können, wird für diese zuerst der zugehörige ProvRecord bestimmt und anschließend die Relation `Generation` (siehe Abschnitt 4.1.2) mit der referenzierten ProvActivity im ProvDocument gespeichert. Gibt es danach in der

Liste der ausführenden Funktionen noch mindestens einen Eintrag, kann daraus auch die Existenz von mindestens zwei verschiedenen Activities gefolgert werden. Dabei wurde die ProvActivity, von der zum aktuellen Zeitpunkt der Identifier vorliegt von der aufgerufen, die in der Liste an letzter Stelle steht. Dadurch kann für diese Activities die Relation `Communication` im ProvDocument erzeugt werden. Außerdem ist es möglich, wenn die erste aufgerufene Funktion eine Request und damit eine Instanz der Klasse WSGIRequest war, den aufrufenden User zu bestimmen. Ist in den `settings.py` deklariert, auch Objekte des Authentication-Backends aufzuzeichnen, wird für den User ein ProvAgent erstellt und dieser über eine `Association` mit der Activity verbunden. Zum Schluss wird geprüft, ob es noch Einträge in `executing_activities()` gibt. Ist das nicht der Fall, kann das ProvDocument abgeschlossen, ausgegeben und zurückgesetzt werden, damit in einem neuen Aufruf in ein neues ProvDocument geschrieben werden kann.

Um die Logik der Methode `activity()` auf verschiedene klassen- oder funktionsbasierte Views und Funktionen anzuwenden, muss in der `views.py` zuerst die Instanz des *ProvenanceGenerators* abgerufen werden. Dafür wird nach allen Import-Statements die Instanz mit `generator = ProvenanceGenerator.get(")` in der lokalen Variablen `generator` gespeichert. Die Dekorierung von klassenbasierten Views unterscheidet sich danach von funktionsbasierten Views und Funktionen: Klassenbasierte Views werden wieder mit dem `method_decorator()`, der bereits in Abschnitt 5.2.2 erläutert wurde, wie folgend dargestellt dekoriert:

```
@method_decorator(generator.activity(name="foo123"),name='dispatch')
```

Der Method-Decorator wird beim Aufruf der Funktion `dispatch` aufgerufen und setzt den Namen der Activitiy im ProvDocument auf „foo123“. Da eine klassenbasierte View in Django immer je einmal mit einer GET- und mit einer POST-Request aufgerufen wird, kann in der Methode `activity()` zusätzlich gefiltert werden, um welche Art von Request es sich handelt. Weil beim Senden einer GET-Request in der Regel nur Daten angefragt, aber nicht geändert werden, wäre diese theoretisch für die Aufzeichnung von Provenance nicht relevant. Da dennoch die Möglichkeit besteht, dass in der GET-Request eine andere Funktion aufgerufen wird, die Daten ändert, wird der Filter vorerst nicht eingebaut. Zuletzt ist die Dekorierung von Funktionen und funktionsbasierten Views einfacher, als die der klassenbasierten Views, dabei kann der Method-Decorator weggelassen werden:

```
@generator.activity(name="foo456")
```

5.4.3 Bereitstellen einer Schnittstelle der Anwendung für den Export der Daten

In Abschnitt 5.3.3 wurde bereits beschrieben, welche Serialisierungs- und Grafikformate vom *ProvenanceGenerator* unterstützt werden sollen. Dabei soll der Anwender selbst entscheiden, in welchen Formaten die Proveance-Dokumente auf welchem Pfad exportiert werden. Dafür wird im Dictionary „`PROVENANCE`“ in den Einstellungen das neue Dictionary „`OUTPUT`“ bereitgestellt, das über folgende Keys verfügt:

1. Der „`PATH`“ gibt an, wo die serialisierten Dokumente gespeichert werden sollen. Wird kein „`PATH`“ angegeben, werden sie in der Commandline ausgegeben.
2. In „`SERIALIZE`“ ist es möglich, die Formate „n“ oder „txt“ für eine Serialisierung nach PROV-N, „json“ für eine Serialisierung nach PROV-JSON, „xml“ für eine Serialisierung nach PROV-XML und „rdf“ für eine Serialisierung nach PROV-O anzugeben.
3. In „`GRAPHIC`“ ist der Export der Dokumente in „pdf“, „jpg“, „png“ und „svg“ möglich. Ist dabei kein „`PATH`“ angegeben, wird für jedes ProvDocument ein neuer Matplotlib-Plot geöffnet.

Werden ein Pfad und verschiedene Exportformate deklariert, werden alle Dokumente mit dem aktuellen Zeitstempel als Dateinamen in diesem Pfad gespeichert. Zusätzlich zum Export der aufgezeichneten Provenance nach dem Abschluss einer ausgeführten View oder Funktion soll es auch möglich sein, das ProvDocument an beliebigen Stellen auszugeben. Dafür soll ein Button bereitgestellt werden, der in unterschiedlichen Views eingebunden werden kann. So wird zunächst eine Funktion geschrieben, die durch den Klick des Buttons und damit einer Request das aktuelle ProvDocument des *ProvenanceGenerators* ausgibt. Danach soll die Funktion den Anwender aber nicht zu einer anderen Stelle der Anwendung sondern zur View, in der der Button ausgelöst wurde, weiterleiten. Die Funktion `print_document_on_click()` setzt genau das um und kann in der `views.py` jeder Anwendung eingesetzt werden.

```
1 def print_document_on_click(request):
2     generator.print_document()
3     orig_page = request.META.get('HTTP_REFERER', '/')
4     return redirect(orig_page)
```

Quellcode 5.6: Funktion zum Ausgeben des aktuellen Provenance-Dokuments

Da die Instanz des *ProvenanceGenerators* in der `views.py` durch das Dekorieren der aufzuzeichnenden Views bereits vorhanden ist, kann mit diesem direkt die Methode `print_document()` ausgelöst werden. Damit der Anwender danach zur ursprünglichen View weitergeleitet wird, wird aus den Metadaten der Request der `HTTP-Referer` abgerufen, da in diesem der Pfad der aufrufenden View gespeichert ist. Mit `redirect()` wird zum Schluss eine `HTTPResponseRedirect` ausgelöst, die den Anwender zu der passenden Seite weiterleitet. Damit die Funktion `print_document_on_click()` tatsächlich in der laufenden Anwendung eingebunden wird, muss für diese ein Pfad in der `urls.py` deklariert werden. Beim Aufruf des Pfades „print-doc/“ wird die Funktion ausgelöst und das ProvDocument in den in den Einstellungen gesetzten Serialisierungsformaten ausgegeben. Damit der Pfad auch durch das Auslösen eines Buttons an verschiedenen Stellen der Anwendung aufrufbar ist, kann in den HTML-Templates der Views eine Form hinzugefügt werden, die den Button modelliert. Wird die Form durch den Klick des Buttons abgeschickt, wird eine HTTP-Post-Request an den im Feld `action` angegebenen Pfad gesendet und dadurch `print_document_on_click()` aufgerufen.

5.4.4 Darstellen eines mit der Anwendung aufgezeichneten Provenance-Dokuments

Mit dem vorherigen Abschnitt ist die Implementierung der Provenance-Anwendung vorerst abgeschlossen. Da beim Umsetzen der Anwendung verschiedene Verbesserungsmöglichkeiten und Erweiterungen der Software aufgefallen sind, werden diese später in Kapitel 6 erläutert. In diesem Abschnitt soll folgend ein in einer Instanz der Klasse ProvDocument aufgezeichnetes Provenance-Dokument der Bibliothek dargestellt und erläutert werden.

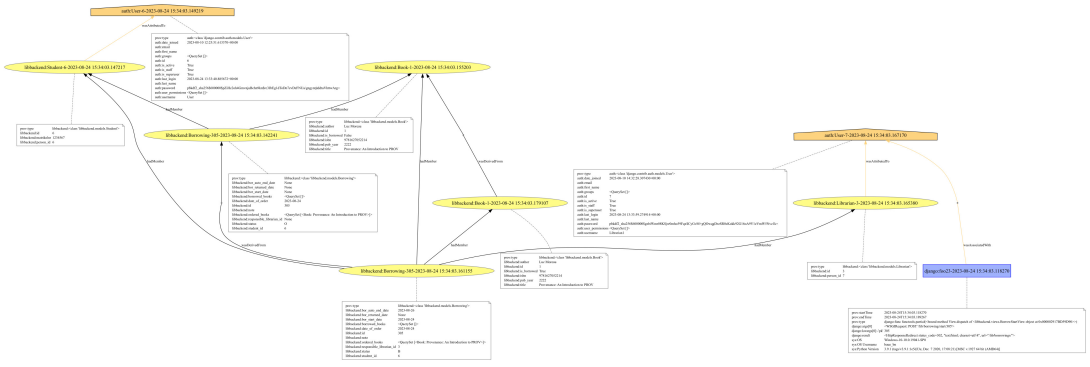


Abbildung 5.8: Aufgezeichnete BorrowStartView

In Abb. 5.8 ist eine Provenanceaufzeichnung nach dem Ausführen einer View in der Bibliothek dargestellt, diese ist auch in Abb. A1, Abb. A2 und Quellcode 1 nochmals in einem größeren Format mit dem dazugehörigen Quellcode angehangen. Im Beispiel wurde die Funktion mit dem Namen `foo23` von User 7 ausgeführt, anhand der aufgezeichneten Attribute der Funktion ist unter anderem erkennbar, dass es sich dabei um die Methode `dispatch` von der *BorrowStartView* handelt. Anhand des ausführenden Users konnte die Verbindung zum dafür zuständigen *Librarian*, der die *Borrowing* gestartet hat, hergestellt werden. Durch die ProvEntity *Borrowing-305-2023-08-24 15:34:03.161155* und ihrer Membership-Relation zum *Librarian* kann auch die Verbindung zu allen anderen ProvRecords hergestellt werden: Die ProvEntity *Borrowing-305-2023-08-24 15:34:03.161155* wurde von der ProvEntity *Borrowing-305-2023-08-24 15:34:03.142241* abgeleitet und anhand der jeweiligen Attribute wird deutlich, dass ein neuer *Librarian* referenziert und das bestellte *Book* ausgeliehen wurde. Dabei ist jedoch auch bemerkbar, dass in den Attributen von *Borrowing-305-2023-08-24 15:34:03.161155* das QuerySet `borrowed_books` leer ist, obwohl das ausgeliehene *Book* bereits verknüpft sein sollte. Das liegt am Senden des Signals `m2m_changed` nach dem Signal `post_save` (siehe Abschnitt 5.4.2). Dadurch wird zwar die Membership richtig erstellt, in den Attributen kann die Änderung aber danach nicht nachgetragen werden. Stattdessen ist bei der Derivation von *Book-1-2023-08-24 15:34:03.179107* aus *Book-1-2023-08-24 15:34:03.155203* die Änderung des Attributs `is_borrowed` von „False“ zu „True“ erkennbar. Beiden *Borrowing*-Entities kann auch der ausleihende *Student* und der damit verbundene User zugeordnet werden: Beim *Student* mit dem Identifier 6 handelt es sich um den User mit dem `username` „User“.

6 Ergebnisse, kritische Reflexion und Zukunftsaussichten

In Tabelle 6.1 ist erkennbar, dass alle der in Kapitel 2 gesetzten Aufgabenstellungen gelöst werden konnten. Daraus könnte man schließen, dass die Provenance-Anwendung ideal gelöst ist und ohne Probleme funktioniert, was aber nicht der Fall ist. Je weiter die Anwendung vorangeschritten ist, desto mehr Probleme und Verbesserungsmöglichkeiten sind bemerkbar geworden. Folgend sollen die verschiedenen Probleme erläutert und Lösungen dafür bereitgestellt werden.

Usability

Die Gebrauchstauglichkeit (eng. Usability) ist bei der Provenance-Anwendung zum jetzigen Zeitpunkt zu gering, um sie in einem produktiven System zu verwenden oder sie zu veröffentlichen. Sie produziert zwar kontinuierlich Provenance-Aufzeichnungen, diese sind aber nicht immer einwandfrei und korrekt. In den nächsten Schritten müssten die Fehlerquellen ausfindig gemacht und behoben werden. Folgend werden Fehler und Probleme, die kurz vor der Fertigstellung der Arbeit aufgefallen sind, gelistet:

1. Die Anwendung wurde nur an der Bibliotheksanwendung und nicht an anderen Projekten getestet. Dadurch kann ungewollter Code im Einsatz sein, der zwar für die Bibliothek, aber nicht bei anderen Anwendungen funktioniert. Dies muss überprüft und die Provenance-Anwendung in anderen Projekten getestet werden.
2. In Abschnitt 5.4.4 wurde erläutert, dass Attribute von ProvRecords, die anhand von ManyToMany-Feldern erzeugt wurden, nach einem `m2m_changed`-Signal nicht mehr geändert werden können und somit falsche Werte enthalten. Theoretisch könnten diese Attribute weggelassen werden, da die Verknüpfung der

Aufgabenstellung	Erfüllt?
1. Entwurf eines einfachen Beispiels zur Demonstration von Provenance	
1.1 Erklärung des W3C PROV-Datenmodells an diesem Beispiel	ja
1.2 Entwurf eines ER-Modells der verwendeten Klassen im Beispiel	ja
1.3 Auswahl von zwei spezifischen Use Cases für Provenance	ja
2. Implementieren der Django-Anwendung für das Beispiel	
2.1 Erstellen und Migrieren der Modelle für die Datenbank	ja
2.2 Erstellen von klassenbasierten Views für einen übersichtlichen Ausleihprozess	ja
2.3 Erstellen von funktionsbasierten Views für die Rückgabe von Ausleihen	ja
3. Erarbeiten eines Konzeptes für die Django-Anwendung zur Aufzeichnung von Provenance	
3.1 Definieren eines abgeschlossenen Provenance-Dokuments	ja
3.2 Erarbeiten von Konzepten für das Verarbeiten von Entities, Activities und Agents	ja
3.3 Erarbeiten eines Konzeptes zum standardisierten Export der Provenance-Daten	ja
4. Implementieren der Django-Provenance-Anwendung	
4.1 Implementieren der Anwendung innerhalb des Bibliotheksprojektes	ja
4.2 Implementieren von Verarbeitungsmöglichkeiten für Entities, Activities und Agents	ja
4.3 Bereitstellen einer Schnittstelle der Anwendung für den Export der Daten	ja
4.4 Darstellen eines mit der Anwendung aufgezeichneten Provenance-Dokuments	ja

Tabelle 6.1: Ergebnisse

verschiedenen Entitäten durch die ProvMembership geschieht. Falls die Attribute aber beibehalten werden sollen, müsste bei einem `m2m_changed`-Signal die alte Entität, in der die ManyToMany-Felder existieren, gelöscht und dann eine neue im `m2m_changed`-Signal erzeugt werden, da eine Änderung von bereits bestehenden ProvRecords nicht möglich ist.

3. Ein weiteres Problem bei der Verarbeitung von `m2m_changed`-Signalen ist die Darstellung der möglichen Aktionen `post_add` und `post_remove`. In der Bibliothek werden aus den ManyToMany-Feldern zwar keine Einträge entfernt, passiert das aber in anderen Anwendungen, wird dies nicht anders dargestellt als das Hinzufügen von Einträgen. Da die Entitäten der ManyToMany-Felder immer durch ProvMemberships mit den anderen Entitäten verknüpft werden, wird schnell unübersichtlich, welche Aktion durchgeführt wurde. Das Lösen von Problem 2 und damit das Erstellen einer neuen Entität bei einem `m2m_changed`-Signal würde aber auch dieses Problem beheben.

4. Beim Verarbeiten von ManyToMany-Feldern tritt auch das Problem auf, dass verschiedene ProvEntities mehrfach mit der gleichen ProvMembership verknüpft

werden. Um das zu lösen, muss eine Überprüfung ähnlich zur Methode `check_derivations()` eingebaut werden. In dieser wird dann nur eine neue ProvMembership erzeugt, wenn sie zwischen zwei ProvEntities noch nicht existiert.

5. Wird in den Einstellungen ein Modell als Entity und als Agent eingetragen, wird es immer als Entity aufgezeichnet und nie als Agent. Auch wenn es an verschiedenen Stellen Sinn ergäbe, Objekte gleichermaßen aufzuzeichnen, würde es einerseits das Provenance-Dokument unleserlicher und andererseits die Entscheidung, welcher Typ wann sinnvoller wäre, schwerer machen. Damit die zweifache Eintragung von Modellen unterbunden wird, muss beim Start der Anwendung ein Fehler ausgegeben werden.

6. Es gibt noch keine saubere Schnittstelle, um neue Attribute zu den Systeminformationen hinzuzufügen. Dafür kann in den Einstellungen ein weiterer Key im Dictionary „`PROVENANCE`“ hinzugefügt werden, in dem eine eigens implementierte Funktion referenziert wird.

7. Der Code ist nicht einwandfrei modular und sauber geschrieben. An verschiedenen Stellen werden Argumente mitgegeben, die dort gar nicht sinnvoll sind, oder Codeteile wiederholt. Die Lösung dafür ist lediglich das Refaktorisieren des Codes.

User Experience

Die Nutzererfahrung (eng. User Experience) ist ein weiteres Problem, was eine Veröffentlichung oder anderweitige Verwendung der Provenance-Anwendung erschwert. Zum jetzigen Zeitpunkt gibt es fast keine Dokumentation und auf fehlerhafte Eingaben werden die Fehler nicht ordnungsgemäß ausgegeben. Deshalb sind weitere mögliche Schritte die Ausbesserung der Fehlerrückgaben und das Bereitstellen von Dokumentation, um Anwender bei der Nutzung zu unterstützen. Außerdem könnten zusätzliche Konfigurationsmöglichkeiten hinzugefügt werden:

- Spezifizieren von Dateinamen für exportierte Provenance-Dokumente.
- Spezifizieren der aufzuzeichnenden Activities in den Einstellungen und nicht in der `views.py`.

- Spezifizieren, welche Attribute eines Modells aufgezeichnet werden sollen. Dabei könnten z.B. vertrauliche Felder, die nicht aufgezeichnet werden sollen, ausgeschlossen werden.

Da durch das Einbinden des in Abschnitt 5.4.3 erläuterten Buttons derzeit nur das Ausgeben des aktuellen Dokuments möglich ist, könnte auch das Löschen eines aktuellen Dokuments durch einen anderen Button ermöglicht werden.

Langzeitperspektive

Im Blick auf größere Anwendungsfelder der Provenance-Anwendung könnte es auch möglich sein, Dokumente aus verschiedenen Projekten in einem großen Store zu sammeln (siehe Abschnitt 5.3.3). Bei dem Store könnte es sich um eine Graphdatenbank handeln, in der die verschiedenen Entitäten auch projektübergreifend gesammelt werden können. So könnte es dann möglich sein, auch Provenance von Projekten, die nicht Django nutzen, aufzuzeichnen und in dem Store zu lagern. Für eine solche Anwendung wird am Deutschen Zentrum für Luft- und Raumfahrt derzeit ein Konzept entwickelt, für das bereits ein Whitepaper existiert [23].

Eine weitere Möglichkeit ist auch das Bereitstellen eines Stores im kleineren Sinn. Die Provenance-Dokumente werden dann nicht mehr kontinuierlich in Dateien exportiert, sondern in eine nur dafür vorgesehene Datenbank im Django-Projekt. Dort kann es dann auch möglich sein, verschiedene Dokumente im Browser darzustellen und zu durchsuchen.

Fazit

Zusammengefasst gibt es viele verschiedene Verbesserungsmöglichkeiten und Erweiterungen der Provenance-Anwendung, dennoch beweist der Prototyp, dass die Umsetzung einer Provenance-Anwendung möglich ist. Somit stellt dieser ein Grundgerüst für andere Anwendungsfelder und dadurch auch ein erfolgreiches Ergebnis der Bachelorarbeit dar.

Abkürzungsverzeichnis

PROV-O	PROV-Ontology
PROV-N	PROV-Notation
PROV-DM	PROV-Datamodel
W3C	World Wide Web Consortium
OWL	Web Ontology Language in Version 2
RDF	Resource Description Framework
PROV-XML	PROV-Extensible Markup Language
PROV-JSON	PROV-JavaScript Object Notation
URI	Uniform Resource Identifier
DHBW	Duale Hochschule Baden-Württemberg
ISBN	International Standard Book Number
1..1	Eins zu Eins
1..m	Eins zu Viele
m..n	Viele zu Viele
DB	Datenbank
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem
PK	Primary Key
FK	Foreign Key
DRY	Don't repeat yourself
KISS	Keep it short and simple
IvC	Inversion of Control
MVC	Model-View-Controller
ORM	Object Relational Mapper
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
CSS	Cascading Style Sheets
ER	Entity-Relationship

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] ANSARI, A. Garbage Collection in Python, 2023. Verfügbar unter: https://www.geeksforgeeks.org/garbage-collection-python/ (zuletzt besucht: 29.08.2023).

[2] BRAY, T., HOLLANDER, D., AND LAYMAN, A. Namespaces in XML, 1998. Verfügbar unter: https://www.w3.org/TR/1998/PR-xml-names-19981117 (zuletzt besucht: 29.08.2023).

[3] CHEN, P. P.-S. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Trans. Database Syst.* (1976).

[4] COLGEN, R. Vorlesung Datenbanken (T3INF2004) - Skript, 2022.

[5] CORNELSEN VERLAG GMBH. Provenienz, 2023. Verfügbar unter: https://www.duden.de/node/115835/revision/1405316 (zuletzt besucht: 29.08.2023).

[6] DJANGO SOFTWARE FOUNDATION. Django Overview, 2005. Verfügbar unter: https://www.djangoproject.com/start/overview/ (zuletzt besucht: 29.08.2023).

[7] DJANGO SOFTWARE FOUNDATION. FAQ: General, 2022. Verfügbar unter: https://docs.djangoproject.com/en/dev/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names (zuletzt besucht: 29.08.2023).

[8] DJANGO SOFTWARE FOUNDATION. Signals Documentation, 2022. Verfügbar unter: https://docs.djangoproject.com/en/4.2/topics/signals/ (zuletzt besucht: 29.08.2023).

[9] DJANGO SOFTWARE FOUNDATION. Django Software, 2023. v4.2.3, Lizenz: BSD-3-Clause, Verfügbar unter: https://github.com/django/django (zuletzt besucht: 29.08.2023).

[10] DODDS, L., AND DAVIS, I. *Linked Data Patterns: A pattern catalogue for modelling, publishing, and consuming Linked Data*. 2022.

[11] FIELDING, R. T., NOTTINGHAM, M., AND RESCHKE, J. HTTP Semantics. RFC 9110, 2022. Verfügbar unter: https://www.rfc-editor.org/info/rfc9110 (zuletzt besucht: 29.08.2023).

[12] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] HUYNH, T. D., JEWELL, M. O., KESHAVARZ, A. S., MICHAELIDES, D. T., YANG, H., AND MOREAU, L. The PROV-JSON Serialization, 2013. Verfügbar unter: https://www.w3.org/Submission/prov-json/ (zuletzt besucht: 29.08.2023).

[14] INTERNATIONAL ISBN AGENCY. *ISBN Users Manual*, 7th ed., 2017. Verfügbar unter: https://www.isbn-international.org/content/isbn-users-manual/29 (zuletzt besucht: 29.08.2023).

[15] INTERNET ENGINEERING TASK FORCE. The JavaScript Object Notation (JSON) Data Interchange Format, 2017. Verfügbar unter: https://datatracker.ietf.org/doc/html/rfc8259 (zuletzt besucht: 29.08.2023).

[16] IONOS. Webframeworks - Überblick und Klassifizierung, 2022. Verfügbar unter: https://www.ionos.de/digitalguide/websites/web-entwicklung/webframeworks-ein-ueberblick/ (zuletzt besucht: 29.08.2023).

[17] KRAUSE, A.-M. Wie kam das Pferd ins Rindfleisch? (Tagesschau), 2013. Verfügbar unter: https://www.tagesschau.de/inland/faqpferdefleisch-ts-100.html (zuletzt besucht: 29.08.2023).

[18] LEBO, T., SAHOO, S., AND MCGUINNESS, D. PROV-N: The Provenance Notation, 2013. Verfügbar unter: https://www.w3.org/TR/2013/REC-prov-n-20130430/ (zuletzt besucht: 29.08.2023).

[19] LEBO, T., SAHOO, S., AND MCGUINNESS, D. PROV-O: The PROV Ontology, 2013. Verfügbar unter: https://www.w3.org/TR/2013/REC-prov-o-20130430/ (zuletzt besucht: 29.08.2023).

[20] MOREAU, L., AND GROTH, P. *Provenance: An Introduction to PROV*. Morgan & Claypool Publishers, 2013.

[21] MOREAU, L., AND MISSIER, P. PROV-DM: The PROV Data Model, 2013. Verfügbar unter: https://www.w3.org/TR/2013/REC-prov-dm-20130430/ (zuletzt besucht: 29.08.2023).

[22] PYTHON PACKAGING AUTHORITY. Pip Software, 2021. v21.1.2, Lizenz: MIT, Verfügbar unter: https://github.com/pypa/pip (zuletzt besucht: 29.08.2023).

[23] SCHREIBER, A., WEINERT, A., HECKING, T., STOFFERS, M., FÖRST, N., AND VON KURNATOVSKI, L. Provenance Messaging Architecture. Arbeitsentwurf.

[24] TRUNG DONG HUYNH. Prov Python package's documentation, 2014. Verfügbar unter: https://prov.readthedocs.io/en/latest/readme.html (zuletzt besucht: 29.08.2023).

[25] TRUNG DONG HUYNH. Prov Software, 2020. v2.0.0, Lizenz: MIT, Verfügbar unter: https://github.com/trungdong/prov (zuletzt besucht: 29.08.2023).

[26] XML CORE WORKING GROUP. Extensible Markup Language (XML), 2013. Verfügbar unter: https://www.w3.org/XML/ (zuletzt besucht: 29.08.2023).

Anhang

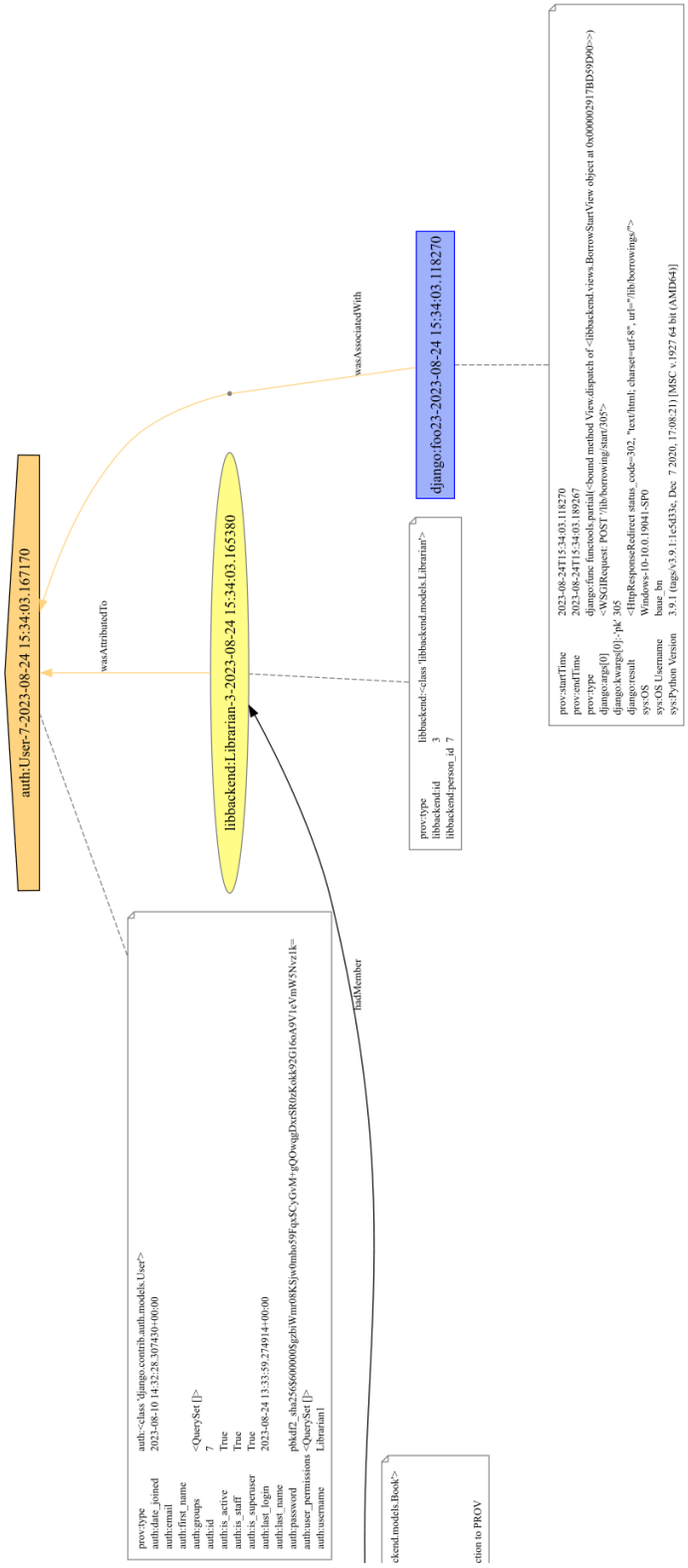


Abbildung A1: Aufgezeichnete BorrowStartView (Große Ansicht 1)

Abbildung A2: Aufgezeichnete BorrowStartView (Große Ansicht 2)

```
document
  default <dhbwlib.org/>
  prefix auth <dhbwlib.org/auth/>
  prefix django <dhbwlib.org/django/>
  prefix libbackend <dhbwlib.org/libbackend/>
  prefix prov_capture <dhbwlib.org/prov_capture/>
  prefix sys <dhbwlib.org/sys/>

  entity(libbackend:Borrowing-305-2023-08-24 15:34:03.142241, [prov:
      type="libbackend:<class 'libbackend.models.Borrowing'>",
      libbackend:id="305", libbackend:student_id="6", libbackend:
      responsible_librarian_id="None", libbackend:date_of_order
      ="2023-08-24", libbackend:bor_start_date="None", libbackend:
      bor_auto_end_date="None", libbackend:bor_returned_date="None",
       libbackend:status="O", libbackend:note="", libbackend:
      ordered_books="<QuerySet [<Book: Provenance: An Introduction
      to PROV>]>", libbackend:borrowed_books="<QuerySet []>"])
  entity(libbackend:Student-6-2023-08-24 15:34:03.147217, [prov:type
      ="libbackend:<class 'libbackend.models.Student'>", libbackend:
      id="6", libbackend:matrikelnr="1234567", libbackend:person_id
      ="6"])
  agent(auth:User-6-2023-08-24 15:34:03.149219, [prov:type="auth:<
      class 'django.contrib.auth.models.User'>", auth:id="6", auth:
      password="
      pbkdf2_sha256$600000$pZiHc2oh4GmvnjuBchn9kn$rc3JbEg1dToDn7evDzFNUa
      /gngyntjtddtuVhttwArg=", auth:last_login="2023-08-24
      13:33:40.885672+00:00", auth:is_superuser="True", auth:
      username="User", auth:first_name="", auth:last_name="", auth:
      email="", auth:is_staff="True", auth:is_active="True", auth:
      date_joined="2023-08-10 12:25:51.613370+00:00", auth:groups="<
      QuerySet []>", auth:user_permissions="<QuerySet []>"])
  wasAttributedTo(libbackend:Student-6-2023-08-24 15:34:03.147217,
      auth:User-6-2023-08-24 15:34:03.149219)
  hadMember(libbackend:Borrowing-305-2023-08-24 15:34:03.142241,
      libbackend:Student-6-2023-08-24 15:34:03.147217)
  entity(libbackend:Book-1-2023-08-24 15:34:03.155203, [prov:type="
      libbackend:<class 'libbackend.models.Book'>", libbackend:id
      ="1", libbackend:isbn="9781627052214", libbackend:title="
```

```
    Provenance: An Introduction to PROV", libbackend:author="Luc
    Moreau", libbackend:pub_year="2222", libbackend:is_borrowed="
    False"])
15 hadMember(libbackend:Borrowing-305-2023-08-24 15:34:03.142241,
    libbackend:Book-1-2023-08-24 15:34:03.155203)
16 entity(libbackend:Borrowing-305-2023-08-24 15:34:03.161155, [prov:
    type="libbackend:<class 'libbackend.models.Borrowing'>",
    libbackend:id="305", libbackend:student_id="6", libbackend:
    responsible_librarian_id="3", libbackend:date_of_order
    ="2023-08-24", libbackend:bor_start_date="2023-08-24",
    libbackend:bor_auto_end_date="2023-08-26", libbackend:
    bor_returned_date="None", libbackend:status="B", libbackend:
    note="", libbackend:ordered_books="<QuerySet [<Book:
    Provenance: An Introduction to PROV>]>", libbackend:
    borrowed_books="<QuerySet []>"])
17 wasDerivedFrom(libbackend:Borrowing-305-2023-08-24
    15:34:03.161155, libbackend:Borrowing-305-2023-08-24
    15:34:03.142241, -, -, -)
18 hadMember(libbackend:Borrowing-305-2023-08-24 15:34:03.161155,
    libbackend:Student-6-2023-08-24 15:34:03.147217)
19 entity(libbackend:Librarian-3-2023-08-24 15:34:03.165380, [prov:
    type="libbackend:<class 'libbackend.models.Librarian'>",
    libbackend:id="3", libbackend:person_id="7"])
20 agent(auth:User-7-2023-08-24 15:34:03.167170, [prov:type="auth:<
    class 'django.contrib.auth.models.User'>", auth:id="7", auth:
    password="pbkdf2_sha256$600000$gzbiWmr08KSjw0mho59Fqx$CyGvM+
    gQOwqgDxrSR0zKokk92G16oA9V1eVmW5Nvz1k=", auth:last_login
    ="2023-08-24 13:33:59.274914+00:00", auth:is_superuser="True",
     auth:username="Librarian1", auth:first_name="", auth:
    last_name="", auth:email="", auth:is_staff="True", auth:
    is_active="True", auth:date_joined="2023-08-10
    14:32:28.307430+00:00", auth:groups="<QuerySet []>", auth:
    user_permissions="<QuerySet []>"])
21 wasAttributedTo(libbackend:Librarian-3-2023-08-24 15:34:03.165380,
     auth:User-7-2023-08-24 15:34:03.167170)
22 hadMember(libbackend:Borrowing-305-2023-08-24 15:34:03.161155,
    libbackend:Librarian-3-2023-08-24 15:34:03.165380)
```

```
23  hadMember(libbackend:Borrowing-305-2023-08-24 15:34:03.161155,
        libbackend:Book-1-2023-08-24 15:34:03.155203)
24  entity(libbackend:Book-1-2023-08-24 15:34:03.179107, [prov:type="
        libbackend:<class 'libbackend.models.Book'>", libbackend:id
        ="1", libbackend:isbn="9781627052214", libbackend:title="
        Provenance: An Introduction to PROV", libbackend:author="Luc
        Moreau", libbackend:pub_year="2222", libbackend:is_borrowed="
        True"])
25  wasDerivedFrom(libbackend:Book-1-2023-08-24 15:34:03.179107,
        libbackend:Book-1-2023-08-24 15:34:03.155203, -, -, -)
26  hadMember(libbackend:Borrowing-305-2023-08-24 15:34:03.161155,
        libbackend:Book-1-2023-08-24 15:34:03.179107)
27  activity(django:foo23-2023-08-24 15:34:03.118270, 2023-08-24T15
        :34:03.118270, 2023-08-24T15:34:03.189267, [prov:type="django:
        func functools.partial(<bound method View.dispatch of <
        libbackend.views.BorrowStartView object at 0x000002917BD59D90
        >>)", sys:Python Version="3.9.1 (tags/v3.9.1:1e5d33e, Dec  7
        2020, 17:08:21) [MSC v.1927 64 bit (AMD64)]", sys:OS="Windows
        -10-10.0.19041-SP0", sys:OS Username="baue_bn", django:args
        [0]="<WSGIRequest: POST '/lib/borrowing/start/305'>", django:
        kwargs[0]:-'pk'="305", django:result="<HttpResponseRedirect
        status_code=302, \"text/html; charset=utf-8\", url=\"/lib/
        borrowings/\">"])
28  wasAssociatedWith(django:foo23-2023-08-24 15:34:03.118270, auth:
        User-7-2023-08-24 15:34:03.167170, -)
29 endDocument
```

Quellcode 1: PROV-N Darstellung des Beispiels