

INSTITUT FÜR INFORMATIK
Fachbereich 12 - Informatik und Mathematik



Bachelorarbeit

SPARQL-Abfrageschnittstelle ikonografischer
Daten im Resource Description Framework

Alicia Wirth

Abgabe am 26. September 2019

eingereicht bei
Prof. Dott.-Ing. Roberto V. Zicari
Datenbanken und Informationssysteme

Erklärung

gemäß Ordnung für den Bachelorstudiengang Informatik §25 ABS.11

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Frankfurt am Main, den 26. September 2019

Alicia Wirth

INHALTSVERZEICHNIS

1 Einleitung	1
1.1 Motivation	1
1.2 Forschungsstand	2
1.3 Aufgabenstellung	3
2 Grundlagen	4
2.1 Datengrundlage	4
2.1.1 Resource Description Framework	4
2.1.2 NLP-Pipeline	6
2.2 SPARQL	10
2.3 Vorteile von SPARQL und RDF	12
2.4 Entwicklung von Webapplikationen	12
2.4.1 Java Servlets	13
2.4.2 AJAX	13
3 Konzept	14
3.1 Analyse der Nutzergruppe	14
3.2 Anforderungen	14
3.3 Funktionalität	15
3.4 Design	17
4 Implementierung	19
4.1 SPARQL-Generierung	19
4.1.1 Implementierungsweise	20
4.1.2 Eine einfache Abfrage	20
4.1.2.1 Nutzung der Benutzerschnittstelle	20
4.1.2.2 Übersetzung in SPARQL	21
4.1.2.3 Implementierung	22
4.1.3 Platzierung auf Münzen	24
4.1.3.1 Nutzung der Benutzerschnittstelle	24
4.1.3.2 Eine neue Münze	25
4.1.3.3 Dieselbe Münzseite	27
4.1.3.4 Die andere Münzseite	33
4.1.3.5 Dieselbe Münze	34

INHALTSVERZEICHNIS

4.1.4 Schlüsselwortsuche	35
4.2 Entfernen von Abfrageteilen	36
4.3 Ergebnisanzeige	37
4.4 Design	37
5 Evaluation	40
5.1 Anforderungserfüllung	40
5.2 Evaluation durch Lautes Denken	41
6 Zusammenfassung und Fazit	43
Benutzerschnittstelle	46
Literaturverzeichnis	48

ABBILDUNGSVERZEICHNIS

2.1	Graph zu Listing 2.1	5
2.2	Graph zu Listing 2.1 und 2.3	6
2.3	In den vorliegenden RDF-Daten durch [KGTP18] enthaltene Münze	8
2.4	Graph zu Teilen der Münze aus Abbildung 2.3 der zugrundeliegenden RDF-Daten durch [KGTP18]	9
3.1	Grobes Konzept der Funktionalität	15
3.2	Grobes Konzept des Designs	18
4.1	Erste Abfrage „Artemis holding bow“ über die Schnittstelle	21
4.2	div-Blöcke einer Münze	24
4.3	Hinzufügen einer neuen Münze zu Abbildung 4.1 über die Schnittstelle	25
4.4	Farbpalette der Schnittstelle	38
4.5	Design der Schnittstelle nach der vollständigen Implementierung . .	39

ABKÜRZUNGSVERZEICHNIS

RDF Resource Description Framework

URI Uniform Resource Identifier

NLP Natural Language Processing

SPARQL SPARQL Protocol And RDF Query Language

LISTINGVERZEICHNIS

2.1	RDF-Beispiel des Subjekts für „Artemis“	5
2.2	RDF-Beispiel zu Literalen	5
2.3	Erweitertes RDF-Beispiel zu Listing 2.1 und leeren Knoten	6
2.4	SPARQL-Beispiel zu Abbildung 2.2	10
2.5	Ergebnis von Listing 2.4 und 2.6	11
2.6	Zweites SPARQL-Beispiel zu Abbildung 2.2	11
4.1	PREFIX und Variablen folgender Listings	19
4.2	SPARQL-Beispiel zu „Artemis holding bow“	22
4.3	Initialisierung von <code>startQuery</code> und <code>endQuery</code>	22
4.4	Inkrementierung von <code>addQuery</code>	23
4.5	Inkrementierung von <code>addQuery</code> für „Artemis“	23
4.6	SPARQL-Beispiel zum Hinzufügen einer neuen Münze über UNION . .	25
4.7	<code>addQuery</code> einer neuen Münze zu Listing 4.6	26
4.8	SPARQL-Beispiel zum Hinzufügen zur selben Münzseite	27
4.9	SPARQL-Beispiel zu hierarchischen Instanzen auf derselben Münzseite	28
4.10	Möglicher alternativer Zusatz zu Listing 4.9	29
4.11	Inkrementierung von <code>newQuery</code> zum Hinzufügen zur selben Münzseite	30
4.12	<code>ASK</code> -Beispiel zu Hierarchie-Überprüfung zwischen Subjekt-Kriterien .	31
4.13	Inkrementierung von <code>newQuery</code> zum Hinzufügen zur vorherigen Entität	32
4.14	SPARQL-Beispiel zum Hinzufügen zur anderen Münzseite	33
4.15	Inkrementierung von <code>newQuery</code> für erstmaliges Hinzufügen zur anderen Seite	34
4.16	SPARQL-Beispiel zu Schlüsselworten	35

KAPITEL EINS

Einleitung

Noch vor einigen Jahren war es für Archäologen unumgänglich zum Betreiben von Forschung entsprechende Institutionen persönlich aufzusuchen. Im Zeitalter des Internets können viele Zeiteinsparungen realisiert werden, doch auch hier bleibt es größtenteils nötig, verschiedene Online-Datenbanken zu besuchen. Die Digitalisierung steht noch am Anfang und sorgt für unvollständige Datenbanken, zum Beispiel aufgrund von Kreuzungen verschiedener Kollektionen mit noch nicht öffentlich zugänglichen Daten, so [Gru18]. Kollektionen von kleineren Institutionen gehen neben ihren größeren Konkurrenten gänzlich verloren. Die Schaffung eines sicheren aussagekräftigen Bildes von Geschichte, Kunst und Wirtschaft der damaligen Zeit gestaltet sich damit schwierig.

1.1 Motivation

Um dem entgegenzustehen verbinden Seiten wie *Corpus Nummorum*¹ die Daten verschiedener Datenbanken und machen eine umfassende Suche nach Aspekten wie *Mint*, aber vor allem eine ikonografische Suche und Ergebnispräsentationen möglich.

[KGTP18] beschreiben neben ihrem eigentlich Fokus der Entwicklung einer NLP-Pipeline, ikonografische Darstellungen auf Münzen als wichtige Informationsquelle der damaligen Zeit und schreiben ihnen demnach eine große Rolle zur Erforschung dieser Zeit zu. Das übliche Vorgehen einer ikonografischen Suche, auch *Schlüsselwortsuche* genannt, bedient sich der Beschreibung der Repräsentationen in natürlicher Sprache als Text und dem Identifizieren von Schlüsselworten in jenem Text. In der tatsächlichen Praxis, in der gerade im Feld der Numismatik viele Verlinkungen zwischen einzelnen Datenobjekten bestehen, ist dieser Ansatz problematisch. Er liefert zu viele Daten, deren relevante Vollständigkeit nicht sicher überprüft und nur schlecht anderen Bedingungen, wie zum Beispiel einer anderen Sprache, angepasst werden kann.

Ein besserer Ansatz bietet die *Semantische Suche*, die das Verständnis der Wörter eines Textes und ihres Kontextes fokussiert. Nach [DGV12] erlaubt sie es, Daten verschiedener Quellen zu verlinken und sich diese Verlinkungen zu Nutze zu machen.

¹Corpus Nummorum, <https://www.corpus-nummorum.eu/>, besucht am 18.09.2019

KAPITEL 1. EINLEITUNG

Als Antwort einer Suchanfrage können so zusätzlich qualitätsfördernde Synonyme oder verwandte Begriffe und Materialien erhalten werden. Umsetzungen erfolgen nach [Gru18] beispielsweise über das *Resource Description Framework* (RDF), eine technische Beschreibung von Verlinkungen zwischen Dokumenten oder Entitäten. Ihre Darstellung erfolgt vorwiegend über *Uniform Resource Identifiers* (URIs). Die Abfragesprache SPARQL ist ein Beispiel, diese Daten nach entsprechenden Bedingungen gefiltert abzufragen und anzuzeigen.

Für die Numismatik existieren bereits Webseiten zu Münzdatenbanken, die auf diesem Vorgehen basieren. Ein Beispiel ist *Nomisma*². Häufig besteht jedoch nicht die Möglichkeit, ikonografische Inhalte wie Relationen zwischen Entitäten abzufragen oder auf hierarchische Strukturen zurückzugreifen. Gleichzeitig mangelt es oft an Zuschneidung auf die eigentliche Nutzerbasis der Numismatiker und die Nutzung wird durch viele Eingabemöglichkeiten verkompliziert. Nichtsdestotrotz existieren Beispiele, die jene funktionellen Aspekte vorweisen.

1.2 Forschungsstand

Es existieren zahlreiche Webseiten mit Suchmasken für numismatische Daten. Einige dieser Webseiten bieten ikonografische Suchen an, die jedoch in den meisten Fällen auf einer *Schlüsselwortsuche* beruhen und demnach keinerlei Rücksicht auf eventuell erwünschte semantische Beziehungen zwischen Abfragekriterien nehmen, wie es zum Beispiel bei *Roman Provincial Coinage* (RPC)³ der Fall ist. RPCs *Advanced Search* ermöglicht es, Schlüsselworte spezifisch auf der Vorder- und Rückseite einer Münze abzufragen.

Neben diesen Webseiten steht das prägnante Beispiel *Digital Iconographic Atlas of Numismatics in Antiquity* (DIANA)⁴, welches eine ikonografische Suche abseits von Schlüsselworten bietet und bereits von [KGTP18] aufgeführt wurde.

DIANAs ikonografische Suche ist in vier Makro-Kategorien unterteilt: „Personages“, „Animals/Mythical Creatures“, „Flora“ und „Objects“. Nach ihrer Wahl können weitere entsprechende Kategorien gewählt und abgefragt werden. Dem Benutzer steht es frei, jene mit diversen anderen Filtern weiter zu spezifizieren. So können insbesondere Attribute wie „bow“ ergänzt werden, die über eine semantische Beziehung mit dem gewählten Subjekt verbunden sind und demnach zwangsläufig auf einer Münzseite in Verbindung stehen. Erst über die Wahl einer Kategorie kann ein direktes Subjekt wie „Artemis“ ausgewählt werden. Es besteht zusätzlich die Möglichkeit, die Abfrage um ein weiteres Subjekt zu ergänzen. Damit bietet DIANA bereits wichtige Funktionen einer ikonografischen Suche.

Der größte Nachteil von DIANA ist [KGTP18] folgend jedoch die Tatsache, dass jegliche Münzdaten manuell hinzugefügt werden müssen. Insbesondere mit der stets wachsenden Anzahl digitaler Münzdaten scheint diese Vorgehensweise nicht im Einklang zu stehen und macht die eigentlich erleichternde Verbindung mit Techniken

²Nomisma, <http://nomisma.org/>, besucht am 18.09.2019

³RPC, <https://rpc.ashmus.ox.ac.uk/search>, besucht am 21.09.2019

⁴DIANA, <http://ww2.unime.it/diana/?q=node/25>, besucht am 21.09.2019

wie der NLP-Pipeline von [KGTP18] schwer.

1.3 Aufgabenstellung

Ziel der Bachelorthesis ist es, eine grafisches Benutzerschnittstelle für die ikonografische Suche über durch die Professur bereitgestellten Münzdaten in RDF zu entwickeln. Der Fokus liegt auf der Umwandlung von Benutzereingaben in eine SPARQL-Abfrage, die anschließend über einen Endpunkt entsprechend ihrer Bedingungen ausgegeben werden soll. Die Art der Ausgabe ist dabei nicht relevant und die Beschränkung auf den Typus „Person“ als Subjekt auf einer Münze gestattet, zum Beispiel bei der Anfrage „Artemis holding bow“. „Tiere“, „Objekte“ und „Pflanzen“ als Subjekte sind noch nicht in dem Datensatz integriert und können damit größtenteils außer Acht gelassen werden. Zusätzlich sollen die hierarchischen Strukturen der Münzen beachtet und Anfragen wie „Alle Münzen, die Gottheiten abbilden“ ermöglicht und in die Benutzerschnittstelle eingebaut werden.

Die **Struktur** der Thesis umfasst im ersten Teil die Schaffung einer Grundlage für die kommenden Kapitel. Vor allem die Themen RDF, die Schaffung der zugrundeliegenden Münzdaten und SPARQL werden aufgegriffen. Wichtige Aspekte der Entwicklung von Webapplikationen werden zusätzlich hervorgehoben.

Der Hauptteil beinhaltet die Implementierung der Benutzerschnittstelle anhand selbstdefinierter Anforderungen an sie. Die Verbindung zu dem *Apache Jena Fuseki*⁵ SPARQL-Endpunkt und vor allem die Übersetzung von Benutzereingaben in eine nutzbare SPARQL-Abfrage spielen dabei eine wichtige Rolle. Unter anderem werden dazu SQL-Daten herangezogen, die den Benutzer bei der Eingabe unterstützen und Tippfehler oder alternative Bezeichnung dennoch korrekt interpretieren können. Insbesondere die Platzierung von Abfragen auf Münzen und entstandene Probleme mit dieser werden behandelt und umgesetzt. Die erarbeitete Benutzeroberfläche erhält schließlich eine moderne Darstellung als HTML. Anschließend wird sie über die zuvor definierten Anforderungen und durch die *Lautes Denken*-Methode von Testpersonen evaluiert.

Zum Schluss werden eine Zusammenfassung und ein Fazit aus den erarbeiteten Ergebnissen gezogen und mögliche zukünftige Verbesserungsmöglichkeiten angesprochen.

⁵ Apache Jena Fuseki, <https://jena.apache.org/documentation/fuseki2/index.html>, besucht am 23.09.2019

KAPITEL

ZWEI

Grundlagen

2.1 Datengrundlage

Der Bachelorthesis liegen ikonografische Münzdaten im *Resource Description Framework* zugrunde. Zur Entwicklung der Abfrageschnittstelle ist es essentiell zu verstehen, wie diese aufgebaut sind. Die folgenden Kapitel sollen der Erlangung jenen Verständnisses dienen.

2.1.1 Resource Description Framework

[RDF14] und [MMM⁺14] folgend ist das *Resource Description Framework* oder auch RDF eine Modellierungssprache zur strukturierten Darstellung von Informationen von Ressourcen im Web. Es dient hauptsächlich der Informationsverarbeitung und -interpretation durch Maschinen. Der größte Vorteil ergibt sich jedoch aus der einfachen Datenzusammenfassung und dem -austausch zwischen Applikationen.

Die Grundstruktur von RDF ist immer gleich. Laut [Ver04] besteht sie aus einem oder mehreren Tripeln der Form (*Subjekt*, *Prädikat*, *Objekt*). Über das Subjekt wird dabei eine Aussage gemacht. Das Prädikat ist die Eigenschaft, die beschrieben wird und das Objekt stellt den Wert dieser Eigenschaft dar. Jedes Subjekt kann auch die Rolle eines Objekts annehmen, sodass verschachtelte Ausdrücke möglich werden. Im effektivsten Fall sind alle drei Komponenten *Uniform Resource Identifiers* oder URIs. Sie identifizieren eine Ressource im Web eindeutig und verlinken sie mit anderen Informationsteilen. Dabei können mehrere URIs auf dieselbe Ressource zeigen. Synonyme Begriffe oder Schreibfehler spielen für sie keine Rolle mehr.

Anschaulich kann RDF als Graph dargestellt werden, wobei Subjekte oder Objekte als Knoten und Prädikate als gerichtete Kanten zwischen ihnen fungieren. Ein Subjekt verweist dabei immer auf ein Objekt. [Krc05]

Listing 2.1 ist ein Beispiel eines RDF-Ausdrucks aus der vorliegenden Münzdatei. Die aufgeführten PREFIX dienen der Übersicht und Vereinfachung des Codes und können entsprechend PREFIX:SUFFIX mit einem Suffix ergänzt werden.

`dump:subjects.php\?id\=15` hat sowohl eine Verbindung `skos:exactMatch` zu `bm:`

57039 („Artemis“), als auch eine Verbindung `rdf:type` zu `dump:Olympic`. Verständlich ausgedrückt ordnen diese Ausdrücke einem bestimmten Subjekt den URI für „Artemis“ und „Olympierin“ zu. Das bedeutet, dass beide URIs eine Instanz eben jenes Subjekts sind. Abbildung 2.1 stellt die Ausdrücke vereinfacht als Graph dar.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX dump: <file:///C:/Users/research/d2rq/dump_nlp_14_11_18.ttl#>
PREFIX bm: <https://public.researchspace.org/resource/?uri=http://
           collection.britishmuseum.org/id/person-institution/>

dump:subjects.php\?id\=15 skos:exactMatch bm:57039 .
dump:subjects.php\?id\=15 rdf:type dump:Olympic .
```

Listing 2.1: RDF-Beispiel des Subjekts für „Artemis“

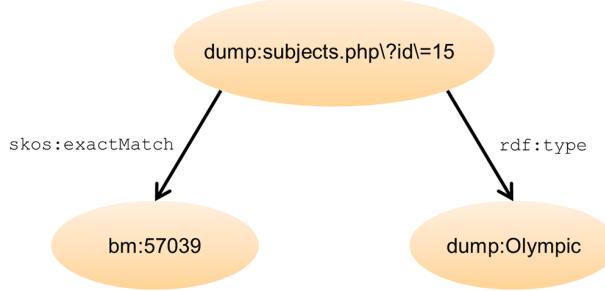


Abbildung 2.1: Graph zu Listing 2.1

[MMM⁺14] verdeutlichen, dass Informationen nicht immer in dieser Struktur aufgeführt sind. Tatsächlich ist die Realität komplizierter. Anstelle eines URI kann auch ein einfaches Literal das Objekt eines Subjekts darstellen. So könnte in Listing 2.1 beispielsweise statt `bm:57039` ‘‘Artemis’’ auftauchen. Probleme können durch die Interpretation von Literalen verursacht werden, denn nicht jedes Literal muss explizit ein String sein. Zwar besteht die Möglichkeit alle Literale, die durch ein bestimmtes Prädikat mit einem Subjekt verbunden sind, über ein Programm entsprechend zu verarbeiten, im schlechtesten Fall kann so jedoch der Datenaustausch zwischen Applikationen erschwert werden.

Abhilfe schaffen zum Objekt hinzugefügte URIs zur Identifikation eines bestimmten Datentyps.

```
PREFIX nm: <http://nomisma.org/id/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

nm:alexander_iii rdf:age "27"^^xsd:integer .
```

Listing 2.2: RDF-Beispiel zu Literalen

Ferner ist es [MMM⁺14] folgend möglich einen Knoten oder dessen URI durch einen leeren Knoten (*blank node*) zu ersetzen, sollte dieser nie von außen referenziert wer-

KAPITEL 2. GRUNDLAGEN

den und sollten seine Objekte weiterhin zusammengefasst bleiben. Gekennzeichnet werden müssen leere Knoten durch *blank node identifiers*, die die Struktur `_:name` aufweisen. Ohne jegliche Kennzeichnung würden mehrere leere Knoten die RDF Tripel-Form verletzen.

Fügt man dem Beispiel aus Listing 2.1 den entsprechenden Code aus Listing 2.3 hinzu, erhält man einen leeren Knoten, der sowohl zuvor aufgeföhrte ikonografische Subjekt-Informationen, als auch Objekt-Informationen, also den URI für `bow_object`, zusammenfasst. Auf einer Münze wären damit ein Bogen und Artemis, die Olympierin ist, zu sehen.

```
PREFIX dbis: <http://www.dbis.cs.uni-frankfurt.de/cnt/id/>

_:node rdf:subject dump:subjects.php?id=15 .
_:node rdf:object dump:objects.php?id=23 .
dump:objects.php?id=23 skos:exactMatch dbis:bow_object .
```

Listing 2.3: Erweitertes RDF-Beispiel zu Listing 2.1 und leeren Knoten

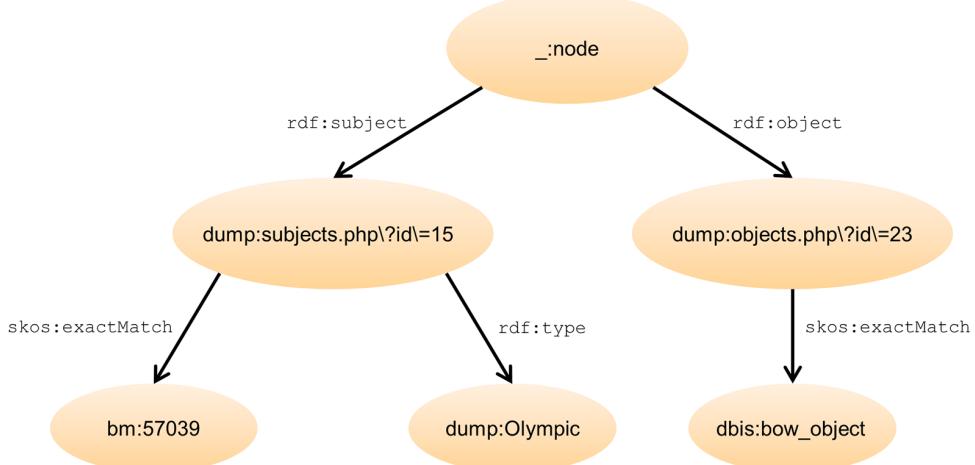


Abbildung 2.2: Graph zu Listing 2.1 und 2.3

2.1.2 NLP-Pipeline

[KGTP18] entwickelten einen Weg, *Natural Language Processing* auf Datensätzen anzuwenden und die Semantik aus ihren nicht standardisierten textuellen Beschreibungen ikonografischer Repräsentationen zu extrahieren. Nachdem das System mit einem Datensatz trainiert wurde, kann es ohne manuelles Eingreifen angewandt werden. Dazu wurde ein numismatischer Datensatz verwendet, doch das System bietet die Möglichkeit, auch auf Datensätze anderer Disziplin angewandt zu werden. Die der Ausarbeitung zugrundeliegenden RDF-Münzdaten wurden auf diese Weise mit den Daten des *Corpus Nummorum Thracorum* (CNT)¹ generiert. Das CNT bein-

¹CNT, <https://www.corpus-nummorum.eu/thrace/home>, besucht am 18.09.2019

haltet thrakische Kollektionen aus Museen und privaten Kollektionen.

Ikonografische Aspekte der einzelnen Münzen sind als Text in menschlicher Sprache beschrieben und werden in der Publikation als „Designs“ bezeichnet. Eine *Schlüsselwortsuche* ist zwar in der Lage zu überprüfen, ob ein Wortpaar in einem solchen Design enthalten ist, jedoch nicht, ob eine semantische Relation zwischen dem Paar besteht. Die entwickelte *Semantische Suche* kann hingegen nicht nur eine Relation erzwingen, sondern auch zwischen einer Menge von Relationen unterscheiden (z.B. „holding“ und „seated“).

Dazu wird eine Pipeline basierend auf *Natural Language Processing* (NLP) verwendet, welche Maschinen lehren soll, relevante Informationen in menschlicher Sprache zu finden. Es sind zwei Hauptaufgaben der Informationsextraktion identifiziert worden: Die *Named Entity Recognition* (NER) und die *Relation Extraction* (RE).

Die erste Aufgabe stellt NER dar. Dabei wird die NLP-Pipeline mit numismatischen Designs trainiert. Die Grundlagen dazu bilden manuell generierte Listen, die DIANA, genannt in Kapitel 1.2, folgend Personen, Objekte, Tiere und Pflanzen kategorisieren. Verschiedene Bezeichnungen und Schreibarten für denselben Ausdruck werden mitberücksichtigt.

Anschließend ist die Pipeline dazu in der Lage, definierte Nominalphrasen, auch *Named Entities* (NEs) genannt, zu erkennen. Sie referenzieren bestimmte Individuen, wie zum Beispiel „Artemis“ und werden entsprechend einer Entitätsklasse von Personen, Objekten, Tieren oder Pflanzen kategorisiert. Der gegenwärtige Fokus liegt auf Relationen zwischen Personen und Objekten. Des Weiteren werden Hierarchien für jeden NE-Typ in einer separaten Tabelle für Klassen und Subklassen definiert. So gehört „Artemis“ zum Beispiel zur Klasse „Deities“.

Vor dem RE-Schritt werden die zugeordneten NEs zu Paaren zusammengefasst (Kreuzprodukt). So könnte (‘‘Artemis’’, ‘‘bow’’) ein Endprodukt von NER darstellen.

In der zweiten Aufgabe wird sich auf die Relation zwischen NEs konzentriert. RE generiert Tripel der Form (NE1, Relation, NE2), wobei Relationen Verben oder Prädikate repräsentieren, die im Design ein Subjekt semantisch mit einem Objekt verbinden. Die Anzahl der Relationen ist dabei vordefiniert, was RE zu einem Multiklassen-Klassifikationsproblem macht. Neue Relationen können nicht gelernt werden, aber das Generieren von Trainings-Daten ist vergleichsweise leicht. Jede Relation repräsentiert eine Klasse, sodass RE automatisch erkennen kann, ob ein bestimmtes NE-Paar eine bestimmte Relation besitzt.

Die beste Klassifikationsmethode ist hierbei als *Logistic Regression* oder *Support Vector Machines* erkannt worden. Die Performanz der Aufgabe ist mit *F-Maß* auf 88% berechnet worden, ist aber offensichtlich durch die Performanz des NER-Schrittes beschränkt. Ausschließlich falsche positive und falsche negative Ergebnisse können fälschlicherweise in der Ergebnismenge vorhanden sein.

Um eine Abfrage mittels SPARQL zu ermöglichen, werden die Ergebnisse der NLP-Pipeline in RDF übersetzt. Dazu ist eine Mapping-Datei notwendig, die via *D2RQ*

KAPITEL 2. GRUNDLAGEN

*Mapping Language*² generiert worden ist. Die Hierarchie wird hingegen durch `rdfs:subClassOf`-Relationen zwischen Klassen und ihren Subklassen dargestellt, die Instanziierungen durch `rdf:type`. Mehrfachvererbung wird unterstützt.



Abbildung 2.3: In den vorliegenden RDF-Daten durch [KGTP18] enthaltene Münze³

Abbildung 2.4 zur Münze aus Abbildung 2.3 zeigt Teile des Aufbaus der entstandenen RDF-Datei am Beispiel der Münze der ID 1187 und gibt damit vor, wie im weiteren Verlauf die Abfragen-Generierung aufgebaut sein muss. Code-Beispiele können Kapitel 2.1.1 entnommen werden.

Die markierten Ausgangspunkte stellen nach erwünschter Kriterien-Selektion dar, welche Werte der Benutzerschnittstelle zur Verfügung stehen können. Die entsprechenden Werte der `SELECT`-Knoten werden in der Ergebnismenge aufgeführt.

Jedes Subjekt, Prädikat und Objekt enthält seinen entsprechenden URI und weitere Spezifizierungen in Form von Kategorien als Instanzen. Unterschiedliche Subjekt-, Prädikat- und Objekt-Kombinationen werden in Form eines Statements zu einem leeren Knoten zusammengefasst, welcher eine semantische Verbindung zwischen ihnen erzwingt. So ist es zum Beispiel möglich, explizit „Artemis holding bow“ abzufragen und dabei nicht gleichzeitig Münzen als Ergebnis präsentiert zu bekommen, die alle drei Komponenten ohne jegliche Verbindung abbilden. Darunter fällt zum Beispiel eine Münze mit dem ikonografischen Inhalt „Apollo holding bow, Artemis wearing chiton“.

Entsprechende Knoten werden schließlich zu Instanzen eines weiteren leeren Knotens, einer Bag. Zusätzlich werden alle identifizierten NE einer zweiten Bag zugeordnet. Dadurch werden NEs ohne Relationen nicht verloren und das Finden von derartigen Münzen ermöglicht. In der Abbildung wurde hingegen auf deren Darstellung verzichtet.

Bags werden wiederum zu Instanzen eines zuvor erwähnten Designs. Jenes Design stellt eine Seite einer Münze dar und bildet in jedem Fall seine verbundenen Subjekt-, Prädikat- und Objekt-Kombinationen ab. `design67` beinhaltet demnach unter anderem zwangsläufig „Artemis holding bow“.

Jede Münze kann zur Instanz zweier Designs werden, welche dann die Inhalte ihrer beiden möglichen Seiten abbilden. Im vorliegenden Beispiel also `design3638` und `design67`, welches über mehrere Instanzen unter anderem auf „Artemis holding bow“ verweist.

²D2QR Mapping Language, <http://d2rq.org/d2rq-language>, besucht am 23.09.2019

³Bizye CN_1187, in: Corpus Nummorum (http://www.corpus-nummorum.eu/CN_1187), Lizenz: CC BY-NC-SA 3.0 DE (<https://creativecommons.org/licenses/by-nc-sa/3.0/de/legalcode>), verändert, besucht am 22.09.2019

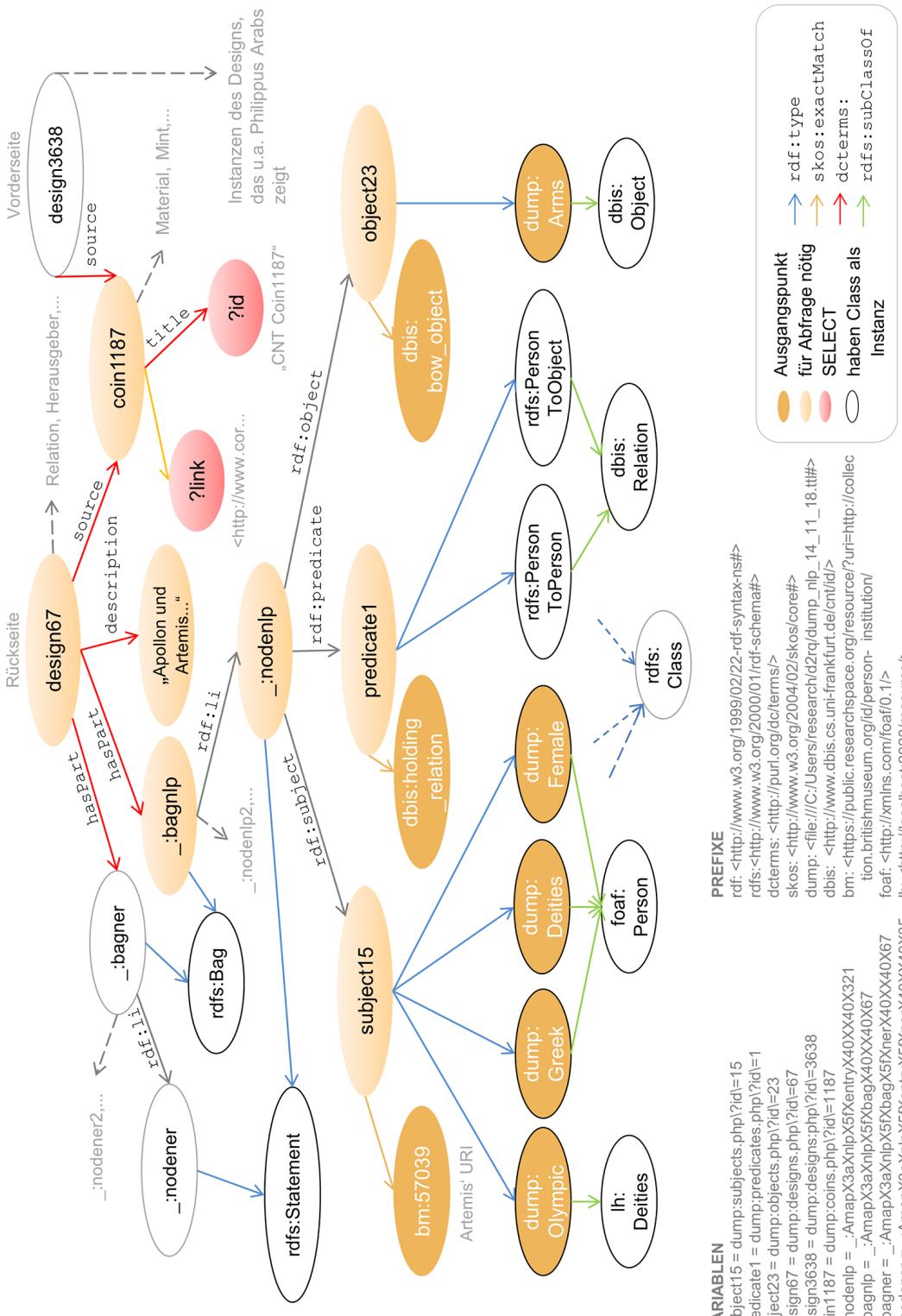


Abbildung 2.4: Graph zu Teilen der Münze aus Abbildung 2.3 der zugrundeliegenden RDF-Daten durch [KGTP18]

2.2 SPARQL

Das Akronym SPARQL steht für *SPARQL Protocol And RDF Query Language* [BV10]. [PAG06] beschreiben SPARQL als eine Abfragesprache der Graphzuordnung des Semantischen Webs. Eine Abfrage bestimmter Struktur wird mit einer Datenquelle abgeglichen und erhaltene Werte werden als Antwort auf die Abfrage verarbeitet. Die Datenquelle kann dabei aus verschiedenen Quellen zusammenge stellt sein.

Laut [SP13] wurde SPARQL spezifisch konstruiert, um die Anwendungsfälle und Anforderungen von RDF folgend [Cla05] zu erfüllen. Eine Abfrage besteht aus einer Menge an Tripeln, auch *basic graph patterns* genannt, die den RDF-Tripeln gleicht, jedoch die Möglichkeit bietet, auch Variablen als Subjekt, Prädikat oder Objekt einzuführen. Ein Subgraph der RDF-Daten wird der Abfrage zugeordnet, wenn RDF-Terme des Subgraphs durch entsprechende Variablen der Abfrage ersetzt werden können. Das Ergebnis wird als Ergebnismenge oder RDF-Graph ausgegeben.

Beeinflussbar ist die Ausgabe durch die Wahl einer der vier möglichen Abfrage formen. **CONSTRUCT** gibt einen einzigen RDF-Graphen als Ergebnis zurück, der durch die Aufnahme jeder Abfragelösung in eine Lösungssequenz, die Ersetzung durch Variablen und die Kombination der Tripel durch Vereinigung generiert wird. Ähnlich ist die Rückgabe von **DESCRIBE**. Auch hier wird ein einzelner RDF-Graph generiert, der allerdings die gefundenen Ressourcen festgelegt durch den SPARQL Abfrageprozessor beschreibt. **ASK** gibt den Datentyp **Boolean** zurück, welcher angibt, ob eine Abfragestruktur zuordenbar ist. Weitere Informationen über das Ergebnis werden nicht zurückgegeben.

Die in dieser Thesis fokussierte und daher im Verlauf näher ausgeführte Form stellt hingegen **SELECT** dar. Als direkte Rückgabe werden alle Variablen und ihre Bindungen generiert. Fast alle Abfragen umfassen außerdem eine **WHERE**-Klausel, die das *basic graph pattern* für die Zuordnung zum Datengraph und den anschließend zurückzugebenden Variablen liefert.

Mit dem Beispiel des Listings 2.2 aus Kapitel 2.1.1 als RDF-Grundlage, kann folgende Abfrage 2.4 und ihr Ergebnis 2.5 generiert werden. Die Abfrage filtert nach einem Objekt oder einer Instanz, die zu einem Subjekt `dump:subjects.php?id\=15` in Beziehung `skos:exactMatch` steht. Erneut gilt die Nutzung von PREFIX einzig der Vereinfachung und kann durch ein Suffix durch die Form `PREFIX:SUFFIX` ergänzt werden.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX dump: <file:///C:/Users/research/d2rq/dump_nlp_14_11_18.ttl#>
PREFIX bm: <https://public.researchspace.org/resource/?uri=http://
           collection.britishmuseum.org/id/person-institution/>
PREFIX dbis: <http://www.dbis.cs.uni-frankfurt.de/cnt/id/>

SELECT ?person WHERE {
  dump:subjects.php?id\=15 skos:exactMatch ?person .
  
```

}

Listing 2.4: SPARQL-Beispiel zu Abbildung 2.2

```
-----
| person |
=====
| bm:57039 |
-----
```

Listing 2.5: Ergebnis von Listing 2.4 und 2.6

Eine komplexere Abfrage einer Person, die in Verbindung mit dem Objekt `dbis:bow_object` steht, kann unter Verwendung derselben PREFIX wie folgt lauten und generiert dasselbe Ergebnis 2.5, nämlich den URI für „Artemis“.

```
SELECT ?person WHERE {
    ?object skos:exactMatch dbis:bow_object .
    ?node rdf:object ?object ;
        rdf:subject ?subject .
    ?subject skos:exactMatch ?person .
}
```

Listing 2.6: Zweites SPARQL-Beispiel zu Abbildung 2.2

Variablen wie `?subject` sind temporär und stets durch ein Fragezeichen gekennzeichnet. Sie werden lediglich für die `WHERE`-Klausel verwendet, wohingegen die Zuordnung zu `?person` über die `SELECT`-Abfrage abgebildet wird. Sowohl `?object`, als auch `?subject` sind über den leeren Knoten `:node` verbunden und müssen dementsprechend in der Abfrage zusammengeführt werden. Hierzu wird die Variable `?node` verwendet.

Dieses Verfahren findet im späteren Verlauf der Thesis zur einfachen und komplexeren Abfragegenerierung erneut Anwendung, sollten zum Beispiel zwei Personen auf derselben Münze abgefragt werden.

Ebenso ist die Kombination von *basic graph patterns* relevant, die einem oder mehreren alternativen RDF-Graphen zuordenbar sind. Zu einem einzelnen Ergebnis können diese mittels des `UNION` Schlüsselworts zusammengefasst werden. Nutzen findet dies bei der Suche mehrerer Kriterien, die nicht explizit auf derselben Münze sein müssen.

Durch Modifikatoren nach der `WHERE`-Klausel, wie zum Beispiel `LIMIT` zur Limitierung der Ergebnismenge oder `ORDER BY` zur absteigenden oder aufsteigenden Sortierung nach einer bestimmten Variable, kann das Ergebnis weiter angepasst werden.

Logische Operationen können durch `FILTER` in der `WHERE`-Klausel realisiert werden. Des Weiteren existieren die üblichen Aggregatfunktionen für die `SELECT`-Klausel, wie zum Beispiel `COUNT` oder `MAX`.

Weitere Informationen über mögliche Optionen zur Generierung von Abfragen sind in SPARQLs Dokumentation [HS13] nachlesbar.

2.3 Vorteile von SPARQL und RDF

Man möge sich fragen, aus welchem Grund eine eher abstrakte Abfragesprache wie SPARQL über einer verbreiteteren und intuitiveren wie SQL gewählt wird. Kapitel 2.2 beschreibt, dass die Abfragesprache SPARQL spezifisch für den Umgang mit RDF entwickelt wurde. Demnach sollte man sich tatsächlich zuerst mit der Frage beschäftigen, weshalb auf RDF anstelle von relationalen Datenbanken zurückgegriffen wird. Im Folgenden wird jener Grund anhand seiner Vorteile erörtert.

[Ber09] beschreibt viele Vorteile von RDF. So ist die Serialisierung von RDF als XML leicht durchzuführen und auch seine Graphenstruktur birgt viele Vorteile. Der größte Vorteil ergibt sich jedoch aus dem Datenaustausch und der Interoperabilität.

Durch die Schemaunabhängigkeit von RDF gestaltet sich die Integration von Daten äußerst einfach. Dabei spielt es keine Rolle, ob komplexe Datenmengen oder einzelne Attribute eingeführt werden sollen. Daten, die einem Schema nur partiell oder gar nicht folgen, können ohne weitere Anpassung in RDF überführt werden. Dazu zählen auch Daten ohne oder abweichender Typisierung oder Referenz. Die Flexibilität ist eine der wichtigsten Eigenschaften von RDF.

Damit ist RDF leicht weiterzuentwickeln und kann als datengesteuert beschrieben werden, wohingegen relationale Datenbanken im Vergleich gar starr erscheinen. Ihr vordefiniertes Schema kann zu erheblicher Anpassungsnot von Daten oder dem Schema selbst führen. Sind eigentlich zu gruppierende Attribute zweier Datenmengen nicht vereinbar, kann das Schema sogar um ein weiteres Attribut erweitert werden müssen. Dies ist nur ein Beispiel, das einen größeren Speicherbedarf mit sich ziehen würde.

Gerade im Gebiet der Numismatik, an welches sich die Bachelorthesis lehnt, erscheint dieser Vorteil von großer Bedeutung. Wie bereits in Kapitel 1 erwähnt steht es zur Aufgabe, verschiedene Datensätze zusammenzuführen, die sich im schlechtesten Fall in ihrer Struktur deutlich unterscheiden.

Zwar besteht laut [Ber09] die Möglichkeit, RDF mit verschiedenen Tools ohne großen Aufwand in eine relationale Datenbank zu überführen, was im Nachhinein die Verwendung von SQL ermöglichen würde, jedoch erscheint dies im Falle der Numismatik nicht sinnvoll.

An der Vollständigkeit vieler numismatischer Datenbanken wird noch gearbeitet. Neue Daten werden stets ergänzt. Das Feld und die Datenmenge verändern sich damit stetig, was immerzu eine Überführung von RDF mit sich ziehen würde.

RDF und SPARQL erweisen sich also insbesondere für die Ausarbeitung als sinnvolle Herangehensweise zum Lösen der Aufgabenstellung 1.3.

2.4 Entwicklung von Webapplikationen

Bei der Entwicklung von Webapplikationen finden Client- und Server-Seite zusammen. Entsprechend muss die Übertragung von Daten zwischen beiden Seiten korrekt vernommen werden.

Die Client-Seite beinhaltet, was dem Client beziehungsweise dem Browser angezeigt wird oder zur Anzeige beiträgt. Damit umfasst sie alles, was ohne Kommunikation mit einem Server verarbeitet werden kann. Typische Sprachen sind vor allem JavaScript, *Hypertext Markup Language* (HTML) und *Cascading Style Sheets* (CSS).

Während die Client-Seite Skripte nachdem die Webseite geladen wurde ausführt, führt die Server-Seite ihren Code schon aus, bevor sie geladen wird. Eine dynamische userabhängige Inhaltsgenerierung wird mit dem Zugriff über Server, wie zum Beispiel auf eine Datenbank, ermöglicht. Die Sprache PHP (*Hypertext Preprocessor*) ist ein prägnantes Beispiel für Sprachen der Server-Seite. [Cod13]

Eine andere Möglichkeit ist die Verwendung von Java Servlets, die bereits in vorherigen Anwendungen der Professur Nutzung fanden und entsprechend in der Ausarbeitung weiter verwendet werden.

2.4.1 Java Servlets

Servlets sind Klassen der Programmiersprache Java und dienen der Interaktion zwischen Client- und Server-Seite. Wie Java selbst sind sie plattformunabhängig. Die Implementierung erfolgt mittels der `javax.servlet.http.HttpServlet` Klasse aus dem `javax.servlet.http` oder der `javax.servlet.GenericServlet` Klasse aus dem `javax.servlet` Package. Für HTTP-Protokolle wird ersteres verwendet.

[And01] folgend sendet der Client eine Anfrage (*Request*) an den Server. Dieser konvertiert die Anfrage in ein `HttpServletRequest`-Objekt, welches an das Servlet weitergereicht wird.

Üblicherweise geschieht dies mittels eines `form`-Tags mit dem Klicken eines `input`-Buttons des Typs `submit`. Anschließend kann die Anfrage verarbeitet und anderen Komponenten weitergeleitet werden, wie zum Beispiel einer Datenbank. Dabei werden häufig die Methoden `doGet()` zum Erfassen der Daten als Antwort auf eine GET-Methode oder `doPost()` zum Posten der Daten als Antwort auf eine POST-Methode überschrieben.

Ein `HttpServletResponse`-Objekt wird von einem Servlet generiert, das der Web-Server dem Client als Antwort (*Response*) zurückgibt. In Form des oben angesprochenen `submit`-Buttons wird dazu eine neue HTML- oder JSP-Seite konstruiert, die angefragte Informationen enthält.

2.4.2 AJAX

[G⁺05] beschreiben die jQuery-Methode **AJAX**. Sie ermöglicht hingegen auch asynchrone Interaktionen mit einem Servlet, ohne dass dabei eine Seite neu geladen werden muss. Eine weitere Schicht zwischen Server und Client, die Ajax-Engine, wird hinzugefügt. Sie verwandelt Anfragen in ein `XMLHttpRequest`-Objekt und sendet sie an den Server. Seine Antwort erhält die Ajax-Engine in Form von XML-Daten zurück, die dann von ihr in HTML und CSS umgewandelt und zurück an JavaScript geleitet werden. Damit ist eine Dynamiksteigerung unter Verwendung von Servlets ermöglicht.

KAPITEL

DREI

Konzept

In diesem Kapitel werden erste Konzepte bezüglich Funktionalität und Design der Benutzeroberfläche entwickelt und aufgezeigt. Sie sollen dem besseren Verständnis und als grober Überblick dienen, insbesondere was den Datenfluss zwischen den einzelnen Komponenten betrifft. Dazu werden ersten Skizzen digital überführt und im Folgenden erläutert.

3.1 Analyse der Nutzergruppe

Zur der Entwicklung der Benutzerschnittstelle muss sich stets vor Augen gehalten werden, bei welchen Nutzern sie Anwendung finden wird. Im Falle der Bachelorthesis liegt der Fokus auf der Nutzerbasis der Numismatiker aus aller Welt. Zur Weiterentwicklung oder als Grundlage kann sie außerdem für Informatiker interessant sein. Dennoch sollte sie vorrangig Menschen dienen, die weder Programmier noch SPARQL-Kenntnisse besitzen.

Zwar besteht die Möglichkeit das Konzept ähnlich der Datengrundlage entwickelt durch [KGTP18] und bereitgestellt durch die Professur auf andere Aspekte der Archäologie wie beispielsweise Tonwaren anzuwenden, jedoch muss es dafür angepasst werden. Münzspezifische Funktionen würden ihren Sinn verlieren und im schlimmsten Fall zu falschen Ergebnissen führen. Dementsprechend bleibt der Fokus auf eben jener Spezifikation.

3.2 Anforderungen

Aus der Nutzergruppen-Analyse ergeben sich folgende Anforderungen an die Schnittstelle:

A1 Semantische Benutzeranfragen ikonografischer Repräsentationen auf Münzen sind ermöglicht und können in die Abfragesprache SPARQL umgewandelt werden

A1.1 Auch ikonografische Kategorien können abgefragt werden

A1.2 Mehrere Kombinationen einer Abfrage sind ermöglicht

A1.2.1 Die Position der Abfragekriterien auf den Münzen ist wählbar

A1.3 Jeder URI eines jeden Abfragekriteriums ist einsehbar

A2 Eingaben alternativer Bezeichnungen (in Zukunft auch anderer Sprachen) und von Tippfehlern werden unterstützt und verarbeitet

A2.1 Ikonografische Kategorien sind als solche gekennzeichnet

A3 Die Suche nach ikonografischen Repräsentationen kann zusätzlich durch eine *Schlüsselwortsuche* ergänzt werden

A4 Der SPARQL-Code ist einsehbar und manuell anpassbar

A5 Nutzern werden ihre erwählten Abfragekriterien angezeigt und sie können diese auch wieder entfernen

A5.1 Die Unterscheidung der Platzierung auf den Münzen nach **A1.2.1** wird verdeutlicht

Und auch, wenn die genaue Art und Weise der Ausgabe nicht relevant in dieser Thesis ist, soll sie dennoch erfolgen. Dementsprechend wird beigefügt...

A6 Die Ergebnisse werden übersichtlich ausgegeben

3.3 Funktionalität

Alle im vorherigen Kapitel genannten Anforderungen sollen in die Benutzerschnittstelle integriert werden können.

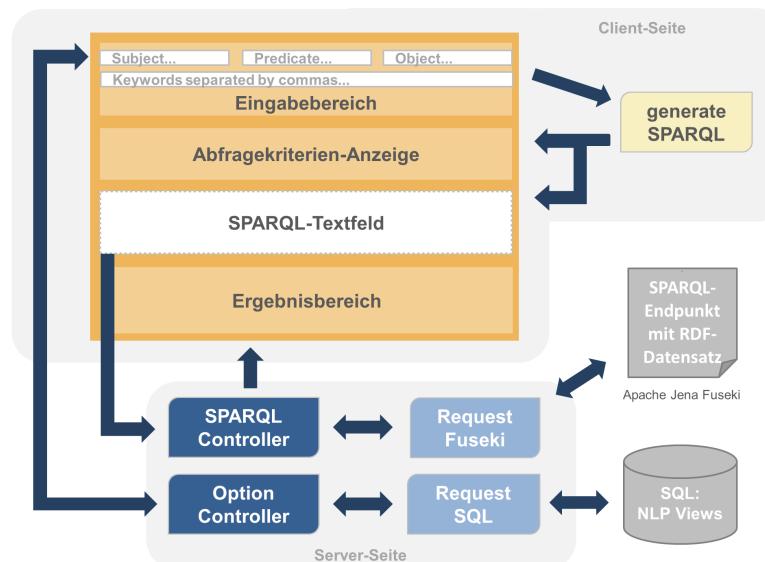


Abbildung 3.1: Grobes Konzept der Funktionalität

KAPITEL 3. KONZEPT

Zur Abfrage ikonografischer Repräsentationen und der Realisierung von **A1** wird sich für drei Eingabefelder der Form *Subjekt*, *Prädikat* und *Objekt* entschieden. Die zugrundeliegende RDF-Datenmenge und die Entscheidung für die Abfragesprache SPARQL unterstützen diese Entscheidung durch ihren Aufbau in eben jener Form. Einzelne Komponenten können damit in ihr vorgesehenes Feld eingetragen werden.

Die Eingabefelder umfassen sowohl direkte Subjekte, wie zum Beispiel „Artemis“, als auch **A1.1** folgend Kategorien, wie zum Beispiel „Deities“. Dies ermöglicht eine einfache und schnelle Nutzung im Vergleich zu anderen Webseiten wie DIANA aus Kapitel 1.2.

Um wie von **A1.2** gestellt eine Suche nach mehreren Kombinationen zu ermöglichen, ist ein Button vorgesehen, welcher die gemachten Eingaben an ein Programm übergibt. Dieses konvertiert sie in einen SPARQL-Code **String** und gibt die Eingabefelder für weitere Eingaben frei. Werden diese getätigt, wiederholt sich der Vorgang und der zuvor generierte SPARQL-Code wird entsprechend angepasst.

A1.2.1 sieht nun außerdem eine Entscheidung über die Platzierung der Abfragekriterien auf Münzen vor. Dazu werden die Auswahlmöglichkeiten „neue Münze“, „selbe Seite“ und „andere Seite“ in Form einer HTML-Selektion beigefügt. Sie beeinflussen das Programm zur Generierung des Codes, treten aber erst auf, wenn bereits eine Abfragekriterien-Kombination generiert wurde.

Anforderung **A2** bedeutet die Notwendigkeit einer Autovervollständigung der Eingabefelder, die alternativen Bezeichnungen oder Tippfehlern die tatsächlich korrekte Bezeichnung zuweist. Es bieten sich die von der Professur erstellten SQL-Tabellen der Daten an, die oben genannten Aspekte in Form von Spalten beinhalten. Da die Daten nur gelesen werden, Kategorien enthalten und **A2.1** folgend gekennzeichnet werden sollen, eignet sich die Erstellung von Sichten über jene Tabellen am besten. Für jedes Eingabefeld wird demnach eine Sicht erstellt, die den Namen der Entität, alternative Bezeichnungen, Tippfehler, ihren URI und ein zugeschriebenes Label enthält. Das Label dient dabei der Unterscheidung von Kategorien und direkten Subjekten.

Eine Übertragung der Labels und eindeutigen URIs der erwählten Entitäten zur Anzeige nach **A1.3** erscheint hier, wie auch zur Weiterverwendung sinnvoll. Die Benutzerschnittstelle ist demnach in der Lage, auf jene Sichten zuzugreifen und sie zur Eingabeunterstützung zu nutzen.

Zur Ergänzung der Suche nach ikonografischen Repräsentationen um die *Schlüsselwortsuche*, wie von **A3** gefordert, wird ein weiteres Eingabefeld hinzugefügt und in SPARQL übersetzt. Mit der Trennung durch Kommata können mehrere Schlüsselworte auf einmal eingegeben und ähnlich der vorherigen Optionen auf den Münzen platziert werden.

A4 sieht die Einsicht und Änderung des SPARQL-Codes vor. Dementsprechend wird er in ein HTML-Textfeld eingefügt. Ist die Abfrage-Erstellung komplett und ein entsprechender Button zum Abfragen gedrückt, wird der Code aus eben jenem Textfeld übertragen. Die Abfrage wird über eine SPARQL-Engine, welche die vorliegen Münzdaten als RDF enthält, ausgeführt.

Um Nutzern auch anderweitig ihre gewählten Abfragekriterien sichtbar und entfernt zu machen, werden sie vom Programm zur Generierung des SPARQL-Codes auf der Webseite pro zu suchender Münze angezeigt. Jede Münze enthält einen Button, mit dem man sie wieder entfernen und den SPARQL-Code entsprechend automatisch anpassen kann. Damit ist Anforderung **A5** erfüllt. Zur Erfüllung von **A5.1** wird die Anzeige der Suchkriterien für den Benutzer angepasst. Näheres dazu ist in Kapitel 3.4 Design nachzulesen.

Zuletzt wird **A6** über die Rückgabe der Ergebnisse der SPARQL-Engine realisiert. Mit Hilfe von Funktionen der von der Professur bereitgestellten *QueryAFE*-Schnittstelle, beziehungsweise insbesondere des Java-Programms *SPARQLXMLEHandler* werden die Ergebnisse in eine HTML-Tabelle überführt und dem Client über die Webseite direkt ausgegeben.

3.4 Design

Um das Verständnis der Benutzerschnittstelle der hauptsächlich ohne Programmier- oder SPARQL-Kenntnisse ausgestatteten Nutzerbasis zu erleichtern, ist ein gutes Design unabdingbar. Insbesondere Ikonen und Farben können dem und der Kennzeichnung wichtiger Aspekte dienen. Die Funktionalität soll weiterhin den Kern des Designs bilden, nicht aber durch etwaige Überladungen abgelenkt werden. Entsprechend wird ein eher minimalistisches Konzept angestrebt, welches auf Basis von [BG14] entwickelt wird.

Struktur

Begonnen wird mit der Wahl der Struktur der Benutzerschnittstelle. Der Auflöckerung und Balance zugrunde liegend wird ein negativer weißer Raum ober- und unterhalb der Webseite eingefügt. Zur weiteren optischen Abgrenzung, wird in dessen Hintergrund eine Farbe oder ein Bild hinterlegt. Der Inhalt selbst wird darüber hinaus aufmerksamkeitsregend in einem unsichtbaren Block in diesem Bereich zentriert und von weiteren inhaltslosen Flächen umgeben. Die dadurch erschaffene Nähe gruppiert den Inhalt sinngemäß und zeigt damit die Zugehörigkeit zur Abfragefunktion der Benutzerschnittstelle auf. [BG14]

Beim erstmaligen Aufrufen sind lediglich der Titel der Webseite und Radio-buttons in Form von Ikonen sichtbar. Diese kategorisieren das spätere ikonografische Subjekt der Professur folgend in die Superkategorien „Person“, „Animal“, „Plant“ und „Object“ und sollen damit das Finden von erwünschten Subjekten durch verkürzte Mengen erleichtern. Nachdem sich erstmalig für eine Superkategorie entschieden wurde, wird der Rest der Abfragemaske sichtbar.

Ihre Funktionen in Form von Eingabefeldern für **A1** und **A3**, Buttons und Selektion sind in der Reihenfolge ihrer möglichen Anwendung platziert. Die Darstellung der Ergebnismenge in Form einer Tabelle nach **A6** erfolgt unter der Abfragemaske. Zur Erfüllung der Anforderung **A5** werden außerdem gewählte Abfrage-Kriterien in einem Feld zwischen Add- und Query-Button in Form von Münzen eingefügt. Das Textfeld des generierten SPARQL-Codes wird versteckt, kann aber **A4** folgend mit-

KAPITEL 3. KONZEPT

hilfe eines Buttons angezeigt und editiert werden.

Farbwahl

Um nicht mit der womöglich typischen grauen oder braunen Farbwahl für das Themengebiet der Numismatik einherzugehen, wird eine auffällige energische Farbe wie Orange als Hauptton gewählt [BG14]. Diese soll sowohl wichtige Aspekte wie Überschriften, als auch Ikonen, Buttons und Selektionen kennzeichnen und hervorheben.

Unterschiedlich auf einer Münze platzierte Abfragekriterien werden hingegen durch Graustufen gekennzeichnet. Insgesamt wird sich auf wenige Farbtöne beschränkt, um nicht vom eigentlichen Inhalt abzulenken und ein harmonisches Gesamtbild zu schaffen.

Ikone

Laut dem üblichen Sprichwort und [BG14] sagen Bilder mehr als tausend Worte. Entsprechend werden Ikonen zur einfachen Darstellung und Verständnisunterstützung einiger Funktionen gewählt. So werden insbesondere die Superkategorien als Ikone dargestellt.

Schrift

[BG14] empfiehlt, zwei unterschiedliche Schriftarten zu verwenden. Unter Berücksichtigung dessen und dem Gedanken weiterer Hervorhebung ist die Idee, eine Serifenschrift zur Kennzeichnung von Überschriften und eine seriflose Schrift für restliche Aspekte zu verwenden.

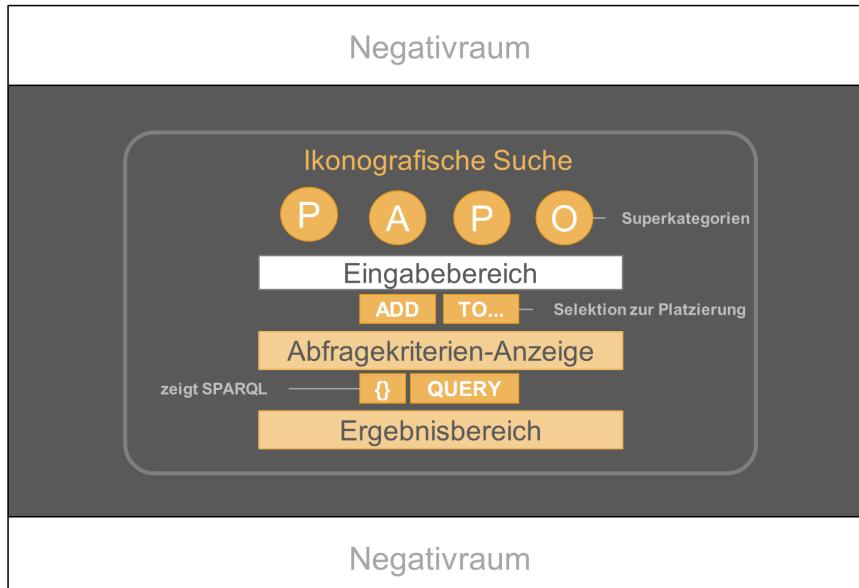


Abbildung 3.2: Grobes Konzept des Designs

Implementierung

In diesem Kapitel wird die Art und Weise, wie auch der Grund für die Implementierung der zuvor ausgearbeiteten Konzepte aufgeführt und genauer erläutert. Mitunter werden verschiedene Möglichkeiten zur Umsetzung aufgezeigt und abgewogen. Als Entwicklungsumgebung und Server wird sich der vorherigen Arbeit der Professur folgend für *NetBeans IDE 8.0.2* und den *Glassfish Server 4.1* entschieden.

4.1 SPARQL-Generierung

Der mitunter wichtigste Punkt der Ausarbeitung stellt die Übersetzung der Benutzereingaben in funktionstüchtigen SPARQL-Code zur Abfrage gewünschter ikonografischer Kriterien auf Münzen dar.

Als Grundlage dienen die in Kapitel 2.1.2 von [KGTP18] mittels NLP-Pipeline und *D2RQ* generierten Daten in RDF. Die Ergebnismenge soll die den Daten enthaltenen IDs der Münzen und ihren URI zu *Corpus Nummorum* der Klasse *coins* beinhalten. Es bietet sich an, zum besseren Verständnis des SPARQL-Codes Abbildung 2.4 heranzuziehen, welche die Verbindungen der einzelnen RDF-Tripel der zugrundeliegenden Münzdaten an einem kleinen Beispiel anschaulich darstellt.

Ziel des Kapitels ist nicht nur die Übersetzung in SPARQL, sondern auch die verschiedenen Möglichkeiten und auftretenden Problematiken zu erläutern. Dazu wird sich grundsätzlich für alle Codeausschnitte an folgenden PREFIX und Variablen orientiert.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

@Artemis = <https://public.researchspace.org/resource/?uri=http://collection.
britishmuseum.org/id/person-institution/57039>
@Isis = <https://public.researchspace.org/resource/?uri=http://collection.
britishmuseum.org/id/person-institution/54025>
@Apollo = <https://public.researchspace.org/resource/?uri=http://collection.
britishmuseum.org/id/person-institution/56988>
@Deities = <file:///C:/Users/research/d2rq/dump_nlp_14_11_18.ttl#Deities>
@Male = <file:///C:/Users/research/d2rq/dump_nlp_14_11_18.ttl#Male>
```

KAPITEL 4. IMPLEMENTIERUNG

```
@holding = <https://www.dbis.cs.uni-frankfurt.decnt/id/holding_relation>
@bow = <https://www.dbis.cs.uni-frankfurt.decnt/id/bow_object>
@chiton = <https://www.dbis.cs.uni-frankfurt.decnt/id/chiton_object>
@spear = <https://www.dbis.cs.uni-frankfurt.decnt/id/spear_object>
```

Listing 4.1: PREFIX und Variablen folgender Listings

4.1.1 Implementierungsweise

Für die Konstruktion des SPARQL-Codes anhand von Benutzereingaben wird sich für die Skriptsprache JavaScript entschieden. Sie ermöglicht den einfachen Zugriff und die Verarbeitung der Benutzereingaben in HTML auf Client-Seite.

Zu Beginn und in ersten Testläufen der Implementierung wurde die Übersetzung noch mit einem Java Servlet vollzogen, bevor sie an den SPARQL-Endpunkt weitergeleitet wurde. Unter Berücksichtigung von Anforderung **A4** ist diese Vorgehensweise zur bloßen Übersetzung jedoch als wenig sinnvoll erachtet worden. Durch stetige Übertragungen der Benutzereingaben an das Servlet wäre die Dynamik und Performanz der Seite deutlich eingeschränkt worden.

Stattdessen vernimmt ein JavaScript-Programm die Übersetzung und überträgt den Code erst an das entsprechende Servlet zur Verbindung mit dem SPARQL-Endpunkt, wenn er vom Nutzer als vollendet betrachtet und der `Submit`-Button zum tatsächlichen Abfragen gedrückt wird.

4.1.2 Eine einfache Abfrage

Zur Generierung einer grundlegenden Abfrage wird sich am Beispiel „Artemis holding bow“ auf einer Seite einer Münze orientiert.

4.1.2.1 Nutzung der Benutzerschnittstelle

Dazu wird zuerst der Radiobutton für „Person“ zur Festlegung der Subjekt-Superkategorie gewählt und anschließend die Abfrage in die neu erschienenen Subjekt-, Prädikat- und Objekt-Eingabefelder entsprechend eingetragen. Der Benutzer wird dabei durch die jQuery-Methode `autocomplete()` unterstützt. Wie in Kapitel 3.3 beschrieben greift diese entsprechend des zuvor gewählten Radiobuttons für das Subjekt-Eingabefeld und der Fokussierung eines der Eingabefelder auf erstellte Sichten der Form Name, alternative Bezeichnung, Tippfehler, URI und Label zu. Mittels einer SQL-Abfrage des `LIKE`-Operators für Name, alternative Bezeichnung und Tippfehler können Name, URI und Label zurückgegeben werden. Dem Benutzer wird der Name der passenden Instanzen in Form einer Dropdown-Liste angezeigt, wobei Kategorien mithilfe des Labels als solche gekennzeichnet werden. Zugehörige URIs und Labels werden nach der Selektion in versteckten Feldern für die spätere Nutzung gespeichert. Generell gilt, dass zumindest eines der vier Eingabefelder ausgefüllt sein muss, bevor der Add-Button zum Hinzufügen der Kriterien aktiviert wird.

Wird der Add-Button nun gedrückt, geschieht sowohl die Übersetzung der Eingaben in SPARQL, als auch die Anzeige der mit URI versehenen Kriterien in nutzerfreundlicher Form auf einer Münze. Die Eingabefelder werden geleert und eine Selektion

zur Platzierung auf Münzen wird angezeigt. Des Weiteren ist der Query-Button nun aktiviert und der erstellte Code über den {}-Button sowohl einsehbar, als auch editierbar. Der Add-Button selbst wird hingegen deaktiviert, denn alle Eingabefelder sind leer. Werden sie wieder befüllt, können weitere Kriterien dem bereits erstellten SPARQL-Code hinzugefügt werden. Abbildung 4.1 zeigt den Verlauf unter Nutzung der Benutzerschnittstelle.

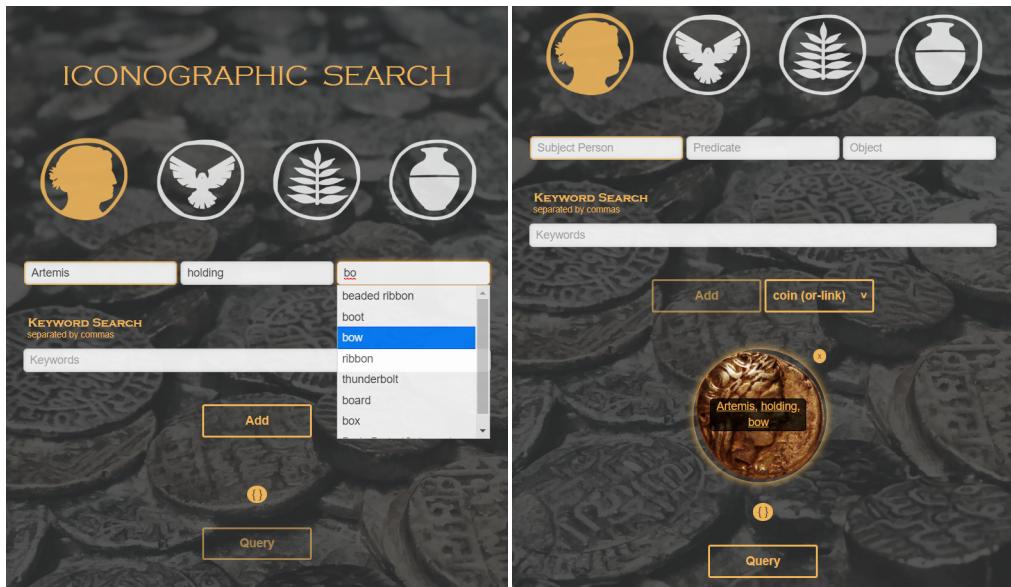


Abbildung 4.1: Erste Abfrage „Artemis holding bow“ über die Schnittstelle

4.1.2.2 Übersetzung in SPARQL

Zur Übersetzung der Nutzereingaben wird sich zuerst überlegt, wie ein entsprechender SPARQL-Code zur Abfrage aufgebaut sein muss.

Wäre die Benutzerschnittstelle primär als *Schlüsselwortsuche* zu implementieren, würde es genügen, unter einer mit der Klasse `design` über `dcterms:description` verbundenen Instanz nach einem übereinstimmenden `Substring` zu suchen. Dass dieses Vorgehen der *Semantischen Suche* unterliegt, wurde bereits in den Kapiteln 1.1 und 2.1.1 aufgeführt.

Stattdessen sollen die einzelnen URIs der ikonografischen Repräsentationen als Abfragegrundlage dienen. Sie sind unabhängig von beschreibenden `Strings` und ermöglichen die Funktionalität des Codes auch für unterschiedliche Bezeichnungen. Durch die von Anforderung **A2** angestrebte Lösung mittels Autovervollständigung der Nutzereingaben und damit gleichermaßen erfüllbaren Anforderung **A1.3**, stehen die URIs dem Programm bereits in versteckten Eingabefeldern zur Verfügung.

Wird nun nach dem Abfragebeispiel „Artemis holding bow“ gesucht, dessen Design schon in Abbildung 2.4 enthalten ist, muss der Graph startend bei den URIs der

KAPITEL 4. IMPLEMENTIERUNG

Repräsentationen der Klassen `subject`, `predicate` und `object` bis zu `coins` durchlaufen werden. Macht man sich das in Kapitel 2.1.1 und 2.2 erlangte Wissen über die Graphendarstellung von RDF zu Nutze, ergibt sich folgender SPARQL-Code. Die durch Nutzereingaben eingefügten URIs sind in diesen Fall rot gekennzeichnet.

```
SELECT ?id ?link WHERE {
    ?subject skos:exactMatch @Artemis .
    ?node rdf:subject ?subject .
    ?predicate skos:exactMatch @holding .
    ?node rdf:predicate ?predicate .
    ?object skos:exactMatch @bow .
    ?node rdf:object ?object .
    ?bag rdf:li ?node .
    ?design dcterms:hasPart ?bag ;
        dcterms:source ?coin .
    ?coin dcterms:title ?id ;
        skos:exactMatch ?link .
}
```

Listing 4.2: SPARQL-Beispiel zu „Artemis holding bow“

Die URIs der Nutzereingaben werden in diesem Fall alle Instanzen der Variable `?node`. Diese kann alle leeren Knoten des NLP-Datensets annehmen, welche diese Instanzen vorweisen und damit eine semantische Verbindung zwischen ihnen voraussetzen.

Über die Variable `?bag` werden alle Bags angenommen, die entsprechende leere Knoten `?node` zur Instanz haben. Gleiches gilt für Designs der Variable `?design`, die entsprechende `?bag` als Instanzen besitzen können. Sie beschreiben unter anderen die ikonografischen Repräsentationen auf einer Seite einer Münze. Münzen der Variable `?coin` gelten ebenfalls als Instanzen jenes Designs.

Der Variable `?coin` liegen ihre ID und ihr URI vor, die auf die im `SELECT`-Statement angegebenen Variablen abgebildet und damit in der Ergebnismenge aufgeführt werden.

4.1.2.3 Implementierung

Zur Generierung des zuvor aufgeführten SPARQL-Codes nach dem Drücken des Add-Buttons auf der Benutzerschnittstelle wird die Funktion `generateSPARQL()` des JavaScript-Programms `generateSPARQL` aufgerufen. Sie verwendet drei globale Variablen: `startQuery`, `newQuery` und `endQuery`. Dabei umfassen `startQuery` und `endQuery` Teile der typischen Abfrage, die immer gleich bleiben. Ihre Initialisierung begibt sich demnach folgendermaßen.

```
var startQuery = prefixes + "SELECT DISTINCT ?id ?link WHERE {";

var endQuery = "?bag rdf:li ?node ." +
    "?design dcterms:hasPart ?bag ;" +
    "dcterms:source ?coin ." +
    "?coin dcterms:title ?id ;" +
```

```
"skos:exactMatch ?link .";
```

Listing 4.3: Initialisierung von `startQuery` und `endQuery`

Hingegen stellt `newQuery` die Verbindung der Nutzereingaben zu ihrem Subjekt `?subject`, Prädikat `?predicate` und Objekt `?object` und deren Verbindung zum leeren Knoten `?node` dar. Entsprechend der Eingabe kann sich die Darstellung des SPARQL-Codes ändern. Insbesondere besteht die Möglichkeit, alle bis auf ein Eingabefeld unausgefüllt zu lassen, wodurch entsprechende Zeilen des Codes nicht mehr auftauchen. Dies wäre zum Beispiel der Fall, wenn lediglich nach „Artemis“ als Subjekt und „bow“ als Objekt gesucht werden würde.

Demnach wird die Variable `newQuery` iterativ befüllt. Das Programm greift auf die Eingabefelder Subjekt, Prädikat und Objekt zu und speichert sie in der Variable `coin_input`. Jedes Eingabefeld besitzt seine Bezeichnung als ID (zum Beispiel `subject` für Subjekt). Ebenso wird die selektierte Option zur Platzierung der Kriterien in `placement_input` gespeichert. Beim ersten Hinzufügen der Kriterien auf eine erste Münze, wie es beim Beispiel „Artemis holding bow“ der Fall ist, besteht für den Benutzer zuerst keine Möglichkeit zur Wahl der Platzierung. Stattdessen wird die Default-Option der ID `all_coins` oder „`coin`“ selektiert, welche Abfragekriterien zu einer neuen Münze hinzufügt.

Mittels einer je um 1 inkrementierenden `for`-Schleife über `coin_input` und einer Überprüfung, ob das Eingabefeld `coin_input[j]` nicht leer ist, greift die Funktion auf die entsprechenden URIs und Labels in vor dem Nutzer versteckten Eingabefeldern zu. Diese können korrekt über ihre ID mit Hilfe von `coin_input[j].id` bestimmt werden.

Unter Berücksichtigung des Labels kann außerdem die Relation zwischen `?subject` und des URIs festgelegt werden. Ist `label == 1`, so liegt eine Kategorie vor und `relation` würde den String ‘‘ `rdf:type` ’’ speichern. Andernfalls, wie auch im Beispiel, speichert `relation` den String ‘‘ `skos:exactMatch` ’’.

Da die Option `all_coins` selektiert ist, wird eine lokale Variable `addQuery` wie folgt inkrementiert.

```
addQuery += "? " + coin_input[j].id + relation + link + " ." +
"?node rdf:" + coin_input[j].id + " ?" + coin_input[j].id + " .";
```

Listing 4.4: Inkrementierung von `addQuery`

Für das Subjekt „Artemis“ ergibt dies den ersten Teil der Abfrage aus Listing 4.2.

```
addQuery += "?subject skos:exactMatch @Artemis ." +
"?node rdf:subject ?subject .";
```

Listing 4.5: Inkrementierung von `addQuery` für „Artemis“

Mit der Beendigung der `for`-Schleife ist `addQuery` um weitere mögliche Eingaben erweitert worden. Für das Beispiel werden „holding“, als auch „bow“ ähnlich des Listings 4.5 hinzugefügt.

KAPITEL 4. IMPLEMENTIERUNG

Nun wird nach `all_coins newQuery` sowohl um `addQuery`, als auch um `endQuery` inkrementiert. Der Zähler `count_all`, der die auftretenden Münzen zählt, wird ebenfalls um 1 inkrementiert.

Final werden die einzelnen Code-Abschnitte der Variablen `startQuery` und `newQuery` mit dem Zusatz des Strings ‘‘}’’ zusammengefügt und in einem vom Benutzer einsehbaren und veränderbaren Textfeld der ID `query` ausgegeben. Damit wurde der SPARQL-Code aus Listing 4.2 erstellt.

Zur schlussendlichen Darstellung der abzufragenden Kriterien für den Benutzer existiert die Funktion `generate_display()` derselben Datei. Mittels der jQuery-Methode `append()` werden Informationen in Form mehrerer `div`-Blöcke der `index`-Datei hinzugefügt und sichtbar.

Dem Design-Konzept aus Kapitel 3.4 folgend umfassen diese einen `div`-Block der Klasse `coins`. In späteren Ausführungen sollen alle Blöcke jener Klasse eine Münze darstellen. Innerhalb eines Blocks befindet sich ein weiterer `div`-Block der Klasse `criteria_wrapper`, der aufgrund der abgezielten abgerundeten Ecken von `.coins` eine feste Abmessung besitzt. `div`-Blöcke der Klasse `criteria` enthalten die benutzerdefinierten Abfragekriterien gruppiert zu Statements. Zusätzlich werden den Kriterien mittels eines `a`-Tags ihre URIs zugewiesen. Die Blöcke werden `.criteria_wrapper` hinzugefügt. Insbesondere `count_all` ist zur Bestimmung aller IDs nützlich.

Abbildung 4.2 stellt die Blöcke anschaulicher dar.

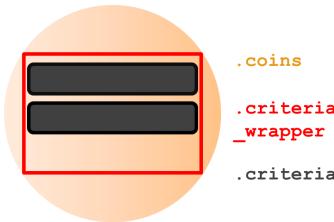


Abbildung 4.2: `div`-Blöcke einer Münze

Mit dem erstmaligen Hinzufügen von Abfragekriterien und damit der Generierung ihres SPARQL-Codes, kann die nächste Abfrage vom Benutzer platziert werden, indem das vorgesehene HTML-`select` eingeblendet wird.

4.1.3 Platzierung auf Münzen

Unter Verwendung des nach einer ersten Generierung von Code eingeblendeten HTML-`select` der ID `placement_input`, steht dem Benutzer die Wahl der Platzierung folgender Abfragekriterien auf einer Münze frei. Es wird zwischen „coin“, „to same side“, „to other side“ und „to coin“ unterschieden.

4.1.3.1 Nutzung der Benutzerschnittstelle

Für die möglichen Optionen wird die Benutzerschnittstelle gleich verwendet. Als Ausgangspunkt fungiert das Beispiel aus Kapitel 4.1.2.1. Soll nun das Subjekt „Male“

mit dem Objekt „chiton“ hinzugefügt werden, werden diese gleichermaßen in die entsprechenden Eingabefelder eingetragen. Die neu erschienene Selektion neben dem Add-Button ist zu beachten und die gewünschte Option zu wählen, sollte sie noch nicht selektiert sein.

Mit dem Drücken des Add-Buttons wird der von der vorherigen Abfrage generierte Code um die neuen Werte erweitert und dem Benutzer werden eben jene Werte in Form einer neuen Münze oder als Block auf der vorherigen Münze angezeigt. Alle Operationen werden auf der neusten Münze beziehungsweise des neusten div-Blocks der Klasse `coins` ausgeführt.

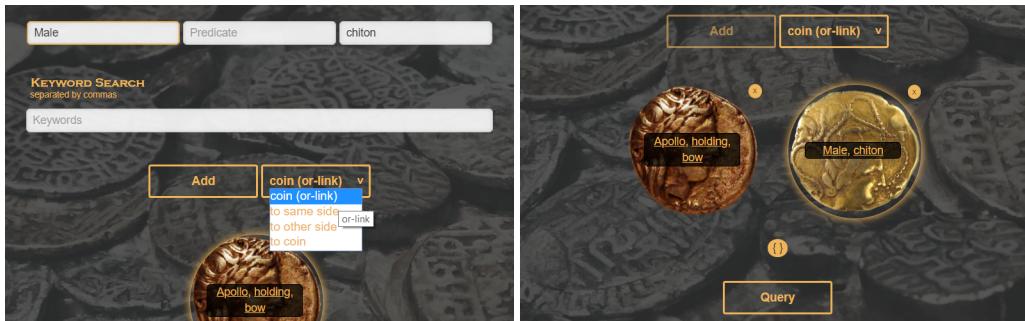


Abbildung 4.3: Hinzufügen einer neuen Münze zu Abbildung 4.1 über die Schnittstelle

4.1.3.2 Eine neue Münze

Um den Benutzern die Möglichkeit zu geben, nach mehreren Kriterien auf möglicherweise verschiedenen Münzen zu suchen, wird die Funktionalität der bereits verwendeten Option „coin“ erweitert. Als Beispiel wird jenes aus dem unmittelbar vorherigen Kapitel wieder aufgegriffen. Es verdeutlicht zudem die Möglichkeit der Abfrage von Kategorien.

Übersetzung in SPARQL

Für die Übersetzung der erwünschten Abfrage kann der `UNION`-Operator von SPARQL verwendet werden, welcher die Vereinigung mehrerer Ergebnismengen realisiert.

Mittels des Wissens aus Kapitel 4.1.2.2 und der Syntax und Funktionalität des `UNION`-Operators aus Kapitel 2.2, sowie der Dokumentation [HS13], ist folgender SPARQL-Code für das genannte Beispiel vorgesehen.

```
SELECT ?id ?link WHERE { 
    ?subject skos:exactMatch @Artemis .
    ?node rdf:subject ?subject .
    ?predicate skos:exactMatch @holding .
    ?node rdf:predicat ?predicate .
    ?object skos:exactMatch @bow .
    ?node rdf:object ?object .
    ?bag rdf:li ?node .
```

KAPITEL 4. IMPLEMENTIERUNG

```
?design dcterms:hasPart ?bag ;
  dcterms:source ?coin .
?coin dcterms:title ?id ;
  skos:exactMatch ?link .
} UNION {
?subject skos:exactMatch @Male .
?node rdf:subject ?subject .
?object skos:exactMatch @chiton .
?node rdf:object ?object .
?bag rdf:li ?node .
?design dcterms:hasPart ?bag ;
  dcterms:source ?coin .
?coin dcterms:title ?id ;
  skos:exactMatch ?link .
} }
```

Listing 4.6: SPARQL-Beispiel zum Hinzufügen einer neuen Münze über UNION

Es sei zu erwähnen, dass dabei in der Ergebnismenge sowohl einzig „Artemis holding bow“, als auch „Male“ und „chiton“ ikonografisch auf einer Münze enthalten sein, aber auch beide Kriterien auf einer Münze kombiniert sein können. Genauer gesprochen implementiert diese Funktionalität also eine *Disjunktion* angegebener Kriterien.

Implementierung

Die Option „coin“ beziehungsweise deren ID `all_coins` stellt die Default-Option und damit die Option des ersten Hinzufügens von Kriterien auf einer ersten Münze dar. Entsprechend wurde sie schon in Kapitel 4.1.2 aufgeführt und muss um den neuen Anforderungen zu entsprechen nur noch erweitert werden. Nach der genannten `for`-Schleife über `coin_input` muss demnach für `all_coins` lediglich überprüft werden, die wievielte Münze eingefügt wird. Dabei kommt der Zähler `count_all` ins Spiel. Ist dieser nicht 0, so wird `addQuery` einmalig für die neue Münze vorne um den String „`} UNION {` “ erweitert. Für das Beispiel einer neuen Münze mit „Male“ und „chiton“ sähe `addQuery` demnach wie folgt aus.

```
addQuery = "} UNION { ?subject rdf:type @Male ." +
  "?node rdf:subject ?subject ." +
  "?object skos:exactMatch @chiton ." +
  "?node rdf:object ?object .";
```

Listing 4.7: `addQuery` einer neuen Münze zu Listing 4.6

Erst im folgenden wird `newQuery` in beiden Fällen um `addQuery` und `endQuery` inkrementiert und eine neue sichtbare Münze in Form von `div`-Blöcken via `append()` hinzugefügt.

Schlussendlich ist noch ein Unterschied beim finalen Zusammenführen der einzelnen Code-Teile zu beachten. Aufgrund der Syntax für `UNION` werden nun `startQuery`, der `String '{', newQuery` und der `String '} }'` im Textfeld der ID `query` zusammengefügt. Damit ergibt sich der SPARQL-Code aus Listing 4.6.

4.1.3.3 Dieselbe Münzseite

Zur Implementierung der Möglichkeit, nach verschiedenen Kriterien-Kombinationen von Subjekt, Prädikat und Objekt auf derselben Münzseite zu suchen, wird die Funktionalität der Option „to same side“ benötigt. Diese soll zum Beispiel ermöglichen, eine Person mit mehreren Prädikaten oder Objekten, aber auch mehrere Personen auf einer Münzseite abzufragen. Zur Orientierung dient das Beispiel „Artemis holding bow“ in Kombination mit „Apollo holding spear“ auf einer Münzseite.

Übersetzung in SPARQL

Für die Übersetzung sollte nun zuerst die Struktur und Erklärung des SPARQL-Codes zu Kapitel 4.1.2.2 herangezogen werden. Es ist bekannt, dass das Design die Darstellung ikonografischer Repräsentationen auf einer Seite einer Münze abbildet und ein Design je zwei Bags besitzt, die leere Knoten mit Subjekten, Prädikaten und Objekten oder Subjekten und Objekten als Instanzen besitzen. Demnach müssen mehrere Abfragekriterien-Kombinationen auf einer Seite Instanzen desselben Designs der Variable `?design` werden. Tatsächlich genügt es, als Instanzen derselben Bag der Variable `?bag` zu gelten. Beide Bags eines Designs besitzen mit dem Unterschied des Prädikats dieselben Instanzen, weswegen prinzipiell eine weitere Unterscheidung überflüssig sein kann. Alle anderen Instanzen und Unterinstanzen, mit der Ausnahme der Münze der Variable `?coin`, können sich voneinander unterscheiden und sollten demzufolge nicht derselben Variable zugeordnet werden. Es ergibt sich folgender Beispiel-Code.

```
SELECT ?id ?link WHERE {
    ?subject skos:exactMatch @Artemis .
    ?node rdf:subject ?subject .
    ?predicate skos:exactMatch @holding .
    ?node rdf:predicate ?predicate .
    ?object skos:exactMatch @bow .
    ?node rdf:object ?object .
    ?bag rdf:li ?node .
    ?design dcterms:hasPart ?bag ;
        dcterms:source ?coin .
    ?coin dcterms:title ?id ;
        skos:exactMatch ?link .

    ?subject0 skos:exactMatch @Apollo .
    ?node0 rdf:subject ?subject0 .
    ?predicate0 skos:exactMatch @holding .
    ?node0 rdf:predicate ?predicate0 .
    ?object0 skos:exactMatch @spear .
    ?node0 rdf:object ?object0 .
    ?bag rdf:li ?node0 .
}
```

Listing 4.8: SPARQL-Beispiel zum Hinzufügen zur selben Münzseite

Problematik

Sollen hingegen ein direktes Subjekt wie die Person „Isis“ und die Kategorie „Deities“ auf einer Münzseite erscheinen, gestaltet sich die Abfrage problematisch. Da eine hierarchische Beziehung zwischen beiden Instanzen besteht, denn sie gehören demselben Subjekt `dump:subjects.php\?id\=71` an, würde unter Verwendung des Codes aus Listing 4.8 lediglich „Isis“ auf Münzen erscheinen.

Man möge argumentieren können, dass dies sehr wohl ein korrektes Ergebnis darstellt. Sinn ergibt eine derartige Abfrage jedoch nicht, da „Deities“ in diesem Fall höchstens als überflüssiger Filter dient. Besteht zwischen einem direkten Subjekt (oder Objekt) und einer Kategorie keine hierarchische Verbindung, tritt auch kein Problem auf.

Ähnliches gilt für die Abfrage zweier Kategorien auf einer Münzseite. Hierbei kann ein zusätzlicher Filter jedoch Sinn ergeben. Bei einer vorliegenden hierarchischen Beziehung zwischen erwählten Instanzen können diese entweder als selbe Entität oder als unterschiedliche Entitäten zählen. Ein gutes Beispiel bildet der Unterschied einer Abfrage nach „Female Deities“ und einer Abfrage nach „Female“ und „Deities“.

Die Benutzerschnittstelle soll in der Lage sein, derartige Problematiken mit hierarchischen Verbindungen abzufangen und entsprechend zu verarbeiten. Im Falle zweier Kategorien soll dem Nutzer die Wahl gelassen werden, wie die Eingabe zu interpretieren ist. Ist eine Kombination gewünscht, so muss dem entsprechenden `?subject` lediglich eine weitere Instanz als Kategorie hinzugefügt werden. Hingegen sollten eine Kategorie und ein direktes Subjekt als zwei verschiedene Entitäten interpretiert werden. Entsprechend muss der SPARQL-Code für diese Abfragen angepasst werden. Für „Isis“ und „Deities“ sähe diese Anpassung wie folgt aus.

```

SELECT DISTINCT ?id ?link WHERE {
  ?subject skos:exactMatch @Isis .
  ?node rdf:subject ?subject .
  ?bag rdf:li ?node .
  ?design dcterms:hasPart ?bag ;
    dcterms:source ?coin .
  ?coin dcterms:title ?id ;
    skos:exactMatch ?link .

  ?subject0 rdf:type @Deities .
  ?node0 rdf:subject ?subject0 .
  ?bag rdf:li ?node0 .

  FILTER(?subject != ?subject0)
}

```

Listing 4.9: SPARQL-Beispiel zu hierarchischen Instanzen auf derselben Münzseite

Mit dem Zusatz `FILTER(?subject != ?subject0)` wird garantiert, dass sich beide Subjekte unterscheiden und damit verschiedene Entitäten bilden. Jedoch kann es

auch hierbei zu Problemen kommen. So besitzen einige URIs im RDF-Datensatz zwei Subjekte. Dies ist auf die Granularität der Sichtweise zurückzuführen. Insbesondere Gottheiten wie „Artemis“ sind betroffen und besitzen sowohl ein Subjekt als griechische, als auch ein Subjekt als römische Variante. Da es sich prinzipiell aber um dieselbe abgebildete Person handelt, haben sie denselben URI zur Instanz. Ein leerer Knoten der Variable `?node` der „Artemis“ enthält, besitzt dementsprechend immer beide Subjekte und macht den `FILTER` wirkungslos.

Eine Möglichkeit zur Umgehung des Problems wäre der Vergleich der URIs über einen Zusatz zu Listing 4.8 in Form folgendes SPARQL-Codes.

```
?subject0 skos:exactMatch ?compare0 .
FILTER(?compare0 != @Isis)
```

Listing 4.10: Möglicher alternativer Zusatz zu Listing 4.9

Dabei können sich `skos:exactMatch` und `@Isis` entsprechend der Eingabe ändern. Die Struktur des Subjekts in der Datenmenge soll simuliert und eindeutig unterschieden werden. Auch hierbei können Probleme festgestellt werden, zum Beispiel bei der Unterscheidung zweier Kategorien. Die Vermutung besteht, dass dies dem mehrmaligen Auftreten von `rdf:type`-Relationen pro Subjekt im Datensatz zu Grunde liegt. Hat ein Subjekt die nach SPARQL-Code nicht enthalten zu sein sollende Kategorie als Instanz, kann in den meisten Fällen dennoch ein passendes Muster des Subjekts mittels einer seiner anderen Instanzen über `rdf:type` gefunden werden. Damit würde jene Münze zur Ergebnismenge hinzugefügt werden. Hingegen tritt die `skos:exactMatch`-Relation pro Subjekt nur einmal auf und bietet damit keine weiteren alternativen Muster.

Für die Ausarbeitung wurde sich der Kürze und insgesamt weniger und besser nachvollziehbaren Fehlern wegen für das Vorgehen nach Listing 4.9 entschieden.

Implementierung

Zur Umsetzung aller im vorherigen Kapitel erwähnten Aspekte, wird die `index`-Datei der Webseite um eine Checkbox mit der ID `same_entity` erweitert. Ist sie abgehakt, sollen vom Nutzer getätigte Eingaben zur vorherigen Entität auf der entsprechenden Seite der Münze hinzugefügt werden. Andernfalls wird die Münze um eine neue Entität erweitert. Erkannte hierarchische Beziehung werden über ein Ikon dargestellt. Beide Erweiterungen sollen nur bei Bedarf angezeigt werden.

Insbesondere wird sich auf Subjekte und zwei Entitäten mit unterschiedlichen Subjekten pro Platzierungswahl auf einer Münze beschränkt. Da Subjekte die meisten Kategorien besitzen und die meisten Münzen nur selten mehr als zwei Subjekte abbilden, kann diese Entscheidung gerechtfertigt werden. Besteht hingegen keine hierarchische Beziehung zwischen den Subjekten oder anderen Eingabefeldern, können von Benutzern beliebig viele Entitäten hinzugefügt werden. Andernfalls wird eine Transitivität nicht umgesetzt und fehlerhafte Ergebnismengen können generiert werden.

KAPITEL 4. IMPLEMENTIERUNG

Zur näheren Beschreibung des Codes wird sich an folgenden Anwendungsfällen orientiert. Dabei wurde eine Abfragekriterien-Kombination Kapitel 4.1.2 oder 4.1.3.2 folgend schon zu einer neuen Münze hinzugefügt. Die Liste `array_same` der Form Label, Link, Relation und `count_same` enthält die Werte jenes bereits hinzugefügten Subjekt-Kriteriums.

1. „Apollo“ mit „spear“ zur selben Münzseite wie „Artemis holding bow“

Bei jeder Selektion eines neuen Subjekts aus der Dropdown-Liste der Autovervollständigung und damit einer Änderung des Felds `subject_label`, bestimmt das JavaScript-Programm `specifyPlacement` die Hierarchie zum vorherigen Subjekt derselben Seite, sollte eines existieren. Im Fall des Beispiels bleibt die globale Variable `hierarchy` auf ‘‘false’’ gesetzt, da sowohl der Wert von `array_same[0]`, dem Label von „Artemis“, als auch der Wert vom aktuellen Label der Subjekt-Eingabe „Apollo“ 0 ist.

Anstatt nun im JavaScript-Programm `generateSPARQL` wie in Kapitel 4.1.3.2 `addQuery` zu befüllen, wird hierbei im ersten Schritt `newQuery` auf ähnliche Art inkrementiert. Dies geschieht nur, da die Checkbox der ID `same_entity` mit `hierarchy = “false”` nicht abgehakt ist. Tatsächlich wird dem Benutzer nicht die Möglichkeit geben, dies zu ändern.

Zum Inkrementieren wird insbesondere der Zähler `count_same` benutzt, der die Vorkommnisse von hinzugefügten Abfragekriterien-Kombinationen auf einer Münze zählt und bei jedem Hinzufügen einer neuen Münze wieder den Wert 0 erhält. Ähnlich des Zählers `count_all` wird er im späteren Verlauf für die ID-Bestimmung der hinzugefügten Kriterien benutzt.

```
newQuery += "?subject" + count_same + " skos:exactMatch @Apollo ." +
    "?node" + count_same + " rdf:subject ?subject" + count_same + " .";
```

Listing 4.11: Inkrementierung von `newQuery` zum Hinzufügen zur selben Münzseite

Anschließend wird `array_same` über die Funktion `generate_entity()` mit den neuen Werten des Subjekts „Apollo“ befüllt. Außerhalb der `for`-Schleife wird `newQuery` zusätzlich um ‘‘?bag rdf:li ?node’’ + `count_same` + ‘‘.’’ + `sameQuery` inkrementiert. Die Variable `sameQuery` wird in der Regel bei jedem Funktionsaufruf von `generate_SPARQL()` mit einem leeren String initialisiert und wird im vorliegenden Anwendungsbeispiel auch nicht verändert. Kapitel 4.1.2 folgend werden die Code-Teile im Textfeld der ID `query` zusammengeführt. Listing 4.8 konnte generiert werden.

Schließlich wird mittels der Funktion `generate_display()` `count_same` um 1 inkrementiert. Ein `div`-Block der Klasse `criteria` wird dem entsprechenden Block der Klasse `coin_wrapper` über seine ID mit `count_all` hinzugefügt. `.criteria` selbst wird anschließend gleich Kapitel 4.1.2 mit dem Namen der Kriterien und ihren URIs erweitert.

2. „Deities“ zur selben Münzseite wie „Artemis holding bow“

Wie bereits in Kapitel 4.1.3.3 aufgeführt, werden Nutzereingaben von direkten Sub-

jekten wie Personen und Kategorien als unterschiedliche Subjekte interpretiert, insbesondere wenn eine hierarchische Verbindung zwischen ihnen besteht. Der Unterschied zum vorherigen Beispiel liegt dementsprechend einzig beim Zusatz eines SPARQL-FILTERs ähnlich des Listings 4.9.

Das zuvor für „Artemis“ gespeicherte und das neue Label der Benutzereingabe unterscheiden sich, entsprechend ändert sich der Ablauf des Programms `specifyPlacement`. Mittels eines synchronen AJAX-Aufrufs wird bestimmt, ob eine Hierarchie zwischen den beiden Subjekten der Nutzereingaben besteht. Dazu werden beide URIs und Relationen an das Servlet `HierarchyController` übertragen. Dieses generiert eine ASK-Abfrage in SPARQL als String und übergibt sie der Funktion `executeAsk` des Programms `RequestFuseki` als Parameter.

```
ASK { ?compare skos:exactMatch @Artemis .
      ?compare rdf:type @Deities . }
```

Listing 4.12: ASK-Beispiel zu Hierarchie-Überprüfung zwischen Subjekt-Kriterien

`RequestFuseki` dient der Kommunikation mit dem SPARQL-Endpunkt von *Apache Jena Fuseki*. Die Variable `hierarchy` speichert das zurückgegebene Ergebnis des AJAX-Aufrufs. Im Beispiel erhält sie als Wert den String ‘‘true’’. Dies hat zu Folge, dass das Ikon zur Kennzeichnung einer Hierarchie eingeblendet wird. Eine Auswahlmöglichkeit in Form der `same_entity` Checkbox besteht für den Benutzer hingegen nicht.

Der SPARQL-Code wird entsprechend des vorherigen Beispiels erstellt. `sameQuery` wird jedoch mittels der Funktion `generate_entity()` auf den Wert ‘‘`FILTER(?subject != array_same[3])`’’ gesetzt. Das Element `array_same[3]` gibt in diesem Fall den `count_same` Wert der zuletzt eingefügten Entität auf jener Seite an. Da „Artemis holding bow“ die einzige Kriterien-Kombination auf der Seite der Münze ist, ist ihr Wert ein leerer String. Es entsteht `sameQuery = 'FILTER(?subject != ?subject0)'` und `array_same[3]` übernimmt den Wert von `count_same`.

Alle nachfolgenden Nutzereingaben des Subjekts für dieselbe Seite beachten damit nur noch die neu hinzugefügte Entität „Deities“ und ermöglichen das Hinzufügen neuer Kategorien zu dieser Entität. Folgende Funktionalitäten des Programmes für die Anwendung gleichen dem vorherigen Anwendungsbeispiel.

3. „Female“ zur selben Münzseite wie „Deities“

Durch die auf eine Kategorie folgende Eingabe einer weiteren Kategorie auf der selben Münzseite soll dem Benutzer die Wahl geboten werden, die neue Kategorie zur Entität der vorherigen hinzuzufügen oder eine neue Entität zu generieren.

`specifyPlacement` erkennt mittels beiden Labels die Kategorien und ermittelt über den zuvor aufgefassten AJAX-Aufruf aus Listing 4.12, ob eine hierarchische Beziehung zwischen beiden Subjekten in der RDF-Grundlage besteht. Ist dem nicht so, werden sie als verschiedene Entitäten aufgefasst. Sollte jedoch eine Beziehung beste-

KAPITEL 4. IMPLEMENTIERUNG

hen, so wird im Gegensatz zu dem vorgegangenen Beispiel mit `hierarchy = “true”` nicht nur das entsprechende Ikon, sondern auch die Checkbox der ID `same_entity` angezeigt. Dem Nutzer sind damit zwei Möglichkeiten gegeben.

3.1 „Deities“ und „Female“

Der SPARQL-Code wird im Fall erwünschter verschiedener Entitäten gleich des vorherigen Beispiels der Form des Listings 4.9 generiert.

3.2 „Female Deities“

Sollen „Female“ und „Deities“ als gemeinsame Entität fungieren, so verändert sich `array_same` nicht. Hingegen wird `array_same[3]` zum passenden Inkrementieren von `newQuery` verwendet. Anstatt `newQuery` ähnlich des Listings 4.9 zu inkrementieren, genügt eine einzige zusätzliche Zeile SPARQL-Code. Diese verbindet das Subjekt des SPARQL-Musters mit der Instanz „Deities“ auch mit der Instanz „Female“.

```
newQuery += "?subject" + array_same[3] + " rdf:type @Female .";
```

Listing 4.13: Inkrementierung von `newQuery` zum Hinzufügen zur vorherigen Entität

`array_same[3]` wird zusätzlich verwendet, um die Nutzereingaben und ihre URIs dem vorherigen der Klasse `criteria` derselben Seite hinzuzufügen. Im Falle des Beispiels wird „Female“ durch `generate_display()` zum Block von „Deities“ hinzugefügt.

Würden an dieser Stelle weitere Kategorien zur selben Seite hinzugefügt werden, so würde lediglich das erste Subjekt einer Entität als Vergleich dienen. Folgende Kategorien würden also dennoch weiterhin mit „Deities“ verglichen werden, nicht aber mit „Female Deities“. Entsprechend werden Eingaben, die keine hierarchische Beziehung zu „Deities“, jedoch zu „Female“ besitzen, direkt als unterschiedliche Entität interpretiert.

Existiert wiederum keine hierarchische Beziehung zu „Female“, aber zu „Deities“ so besteht die Möglichkeit, das neue Subjekt dennoch dieser Entität zuzufügen. Prinzipiell ist dies inkorrekt und führt zu keinem sinnvollen Ergebnis. So würde die Addition von „Male“ zu „Female Deities“ wie im Beispiel eine leere Ergebnismenge produzieren, aber möglich sein, da „Deities“ das erste Subjekt der Entität ist und eine Hierarchie zwischen ihm und „Male“ besteht. Jedoch wird davon ausgegangen, dass der Nutzerbasis generelle Zusammenhänge hierarchischer Strukturen der Kategorien bekannt sind und die Benutzerschnittstelle dennoch entsprechend korrekt genutzt werden kann.

Besteht hingegen eine oder keine hierarchische Verbindung des neuen Subjekts zu beiden vorherigen Subjekten, genügt zur korrekten Ausführung auch die Überprüfung mit dem ersten Subjekt „Deities“.

4.1.3.4 Die andere Münzseite

Sollen Kriterien-Kombinationen nun nicht nur auf derselben Seite einer Münze, sondern auch auf unterschiedlichen Seiten platziert werden können, wird die Benutzerschnittstelle um die Funktion der Option „to other side“ der ID `other_side` ergänzt. Dabei spielt es keine Rolle, welche der beiden Seiten die Vorder- und welche die Rückseite ist. Sollte dies für Numismatiker von Relevanz sein, muss die Benutzerschnittstelle im späteren Verlauf um diesen Zusatz ergänzt werden.

Als unterstützendes Beispiel wird das Hinzufügen der Kriterien-Kombination „Male“ als Subjekt und „spear“ als Objekt auf die andere Seite der Münze mit „Artemis“ genutzt.

Übersetzung in SPARQL

Die Übersetzung in SPARQL greift die Erläuterung aus Kapitel 4.1.2 und 4.1.3.3 auf, führt sie aber anderweitig fort. Nun sollen viel mehr zwei verschiedene Designs der Variable `?design` genutzt werden, die die Seiten einer Münze darstellen. Um beide Seiten auf derselben Münze abzubilden, müssen sie dieselbe Instanz einer Münze der Variable `?coin` besitzen. Entsprechend können sich alle Variablen bis auf `?coin` voneinander unterscheiden.

Da Münzen in der Regel unterschiedliche Designs auf ihren Seiten besitzen, wird der SPARQL-Code um einen `FILTER` ergänzt, der dafür sorgt, dass beide Designs nicht gleich sind. Es entsteht folgender Code.

```
SELECT DISTINCT ?id ?link WHERE {
    ?subject skos:exactMatch @Artemis .
    ?node rdf:subject ?subject .
    ?bag rdf:li ?node .
    ?design dcterms:hasPart ?bag ;
        dcterms:source ?coin .
    ?coin dcterms:title ?id ;
        skos:exactMatch ?link .

    ?subject0 skos:exactMatch @Male .
    ?node0 rdf:subject ?subject0 .
    ?object0 skos:exactMatch @spear .
    ?node0 rdf:object ?object0 .
    ?bag0 rdf:li ?node0 .
    ?design0 dcterms:hasPart ?bag0 ;
        dcterms:source ?coin .

    FILTER(?design != ?design0)
}
```

Listing 4.14: SPARQL-Beispiel zum Hinzufügen zur anderen Münzseite

Implementierung

Prinzipiell verläuft die Implementierung von `other_side` gleich der von `same_side`

KAPITEL 4. IMPLEMENTIERUNG

aus Kapitel 4.1.3.3 mit Hilfe des Zählers `count_same`. Der Unterschied bildet das Inkrementieren von `newQuery` außerhalb der `for`-Schleife über `coin_input`, jedoch einzig beim ersten Hinzufügen von Kriterien auf die andere Münzseite. Dabei wird `newQuery` einmalig für eine Münze wie folgt inkrementiert.

```
newQuery += "?design" + count_same + " dcterms:hasPart ?bag" + count_same +
"; dcterms:source ?coin . " +
"FILTER(?design != ?design" + count_same + ")";
```

Listing 4.15: Inkrementierung von `newQuery` für erstmaliges Hinzufügen zur anderen Seite

Wird nun mehr als eine Kriterien-Kombination zur anderen Seite hinzugefügt, wird dies genauso gehandhabt, wie die Generierung durch `same_side`. Die zweite Kriterien-Kombination durch `other_side` befindet sich schließlich auf derselben Seite wie die erste. Entsprechende Werte sind in der Liste `array_side` und der Variable `count_same_first` zur Zuordnung des richtigen Designs gespeichert.

4.1.3.5 Dieselbe Münze

Ist es irrelevant, auf welcher Seite einer Münze sich folgende Abfragekriterien befinden, ist die Option „to coin“ der ID `same_coin` zu wählen. Nutzereingaben können damit also sowohl auf derselben, als auch auf der anderen Seite einer bereits vorhandenen Münze stehen. Als Beispiel wird sich dem Hinzufügen der Kriterien-Kombination aus dem vorgegangenen Unterkapitel 4.1.3.4 bedient.

Übersetzung in SPARQL

Entfernt man die Zeile `FILTER(?design != ?design0)` aus Listing 4.14, so erlangt man die gewünschte Funktionalität des SPARQL-Codes. Designs müssen sich nicht mehr unterscheiden, was sowohl Kriterien auf derselben, als auch auf der anderen Seite zulässt.

Implementierung

Unter Berücksichtigung jener Änderung, gleicht auch die Implementierung der aus Kapitel 4.1.3.4 unter Einführung einer neuen Liste `array_coin`.

Hinzuzufügen sei jedoch, dass sich `same_coin` demnach bezüglich eventuell vorliegender Hierarchien und dem Hinzufügen von Kategorien nur auf sich selbst bezieht. Subjekte von Entitäten aus `same_side` und `other_side` werden nicht beachtet. Die in Kapitel 4.1.3.3 aufgeführte Problematik besteht also für `same_coin` in Verbindung mit den anderen Platzierungen weiterhin komplett. Sie macht es unmöglich ohne manuelle Anpassung des SPARQL-Codes eine Abfrage für unterschiedliche Entitäten verschiedener in hierarchischer Verbindung stehender Subjekte zu anderen Platzierungen zu generieren.

Durch das manuelle Editieren des SPARQL-Codes kann dieser Effekt jedoch mittels des Zusatzes von `FILTERn` über entsprechende Subjekte erreicht werden. Dazu

kann sich an Listing 4.9 orientiert werden.

4.1.4 Schlüsselwortsuche

Trotz der klaren zuvor aufgeführten Überlegenheit der *Semantischen Suche* kann es sinnvoll sein, ebenso die Möglichkeit einer *Schlüsselwortsuche* zu implementieren. Entsprechend wird ein vierter Eingabefeld eingeführt, in welches Benutzer ihre genau auf einer Münze vorhanden sein sollenden Schlüsselworte eingeben können.

Übersetzung in SPARQL

Ikonografische Beschreibungen als `Strings` liegen jedem Design als Instanz über die Beziehung `dcterms:description` vor. Abbildung 2.4 zeigt einen Ausschnitt des `Strings` als Instanz von `design67`. Die SPARQL-Abfrage kann also über das Design und seine direkte Verbindung zu einer Münze generiert werden. Im Beispiel mit dem Schlüsselwort „Border of Dots“ lautet sie wie folgt.

```
SELECT DISTINCT ?id ?link WHERE {
    ?design dcterms:description ?string ;
        dcterms:source ?coin .
    ?coin dcterms:title ?id ;
        skos:exactMatch ?link .

    FILTER regex(?string, "Border of Dots")
}
```

Listing 4.16: SPARQL-Beispiel zu Schlüsselworten

Der Operator `regex` vergleicht dabei einen dem Design vorliegenden `String` mit einem regulären Ausdruck. In Kombination werden nur jene Münzen zurückgegeben, die mindestens ein Design besitzen, das eine den `String` enthaltende Instanz der Beziehung `dcterms:description` besitzt.

Zur Kombination unter möglicherweise folgenden Eingaben ikonografischer Repräsentationen bietet es sich jedoch an, in jedem Fall `endQuery` aus Listing 4.3 einmalig für jede neue Münze zu generieren und dem die Verbindung des Designs mit der Variable `?string` und dem `FILTER` hinzuzufügen. Damit ist es außerdem auch nach einer ersten einzigen Schlüsselworteingabe möglich, alle in Kapitel 4.1.3 beschriebenen Funktionen zu nutzen.

Implementierung

Eine Funktion `generate_keywords()` wird eingeführt. Ihr erster Schritt ist die Bearbeitung der Eingabe. Da das Ziel ist, mehrere Schlüsselworte gleichzeitig eingeben zu können, werden sie mit der JavaScript-Methode `split()` über Kommata getrennt und als einzelne Elemente in eine Liste eingefügt.

`newQuery` enthält prinzipiell immer Standard-Codeausschnitte für die Platzierungen auf einer Münze, weshalb lediglich die Übersetzung der genannten zwei Zeilen implementiert werden muss.

KAPITEL 4. IMPLEMENTIERUNG

Ein globaler Zähler `count_same_keyword` wird initialisiert. Er hält die Anzahl aller Vorkommnisse von einzelnen Schlüsselworten auf einer Münzseite als Wert fest und kann damit die Ausgangsvariable `?string` mit einer `for`-Schleife über der generierten Liste aus Schlüsselworten immer unterschiedlich gestalten. Dies geschieht ähnlich des Effekts des Zählers `count_same` der vorigen Kapitel.

Ist dabei eine neue Münze oder dieselbe Seite als Option zur Platzierung gewählt, können beide Zeilen unter diesem Zusatz mittels `?design` einer Variable `keywordQuery` als `String` hinzugefügt werden. Sind die anderen beiden Optionen gewählt, muss `?design` entsprechend der des ersten Designs auf jener Platzierung angepasst werden. Dazu können existierende Variablen genau diesen Wertes verwendet werden.

Schließlich wird `newQuery` um `keywordQuery` inkrementiert und die Liste an Schlüsselworten entsprechenden `div`-Blöcken der Klasse `criteria_wrapper` den eventuell existierenden ikonografischen Repräsentationen der Eingabe angehängt.

4.2 Entfernen von Abfrageteilen

Den Benutzern sollte die Möglichkeit gegeben sein, Münzen auch wieder aus ihrer Suchanfrage zu entfernen, ohne dabei manuell in den Code eingreifen zu müssen.

Dazu wurde jeder `div`-Block zur Darstellung der Münzen mit einem Button versehen. Dieser ruft mit seiner ID als Parameter die Funktion `delete_SPARQL(id)` auf. Über jene `id` wird sowohl die entsprechende Münzdarstellung der Klasse `coins` über ihre ID aus HTML entfernt, als auch der entsprechende `Substring` aus dem SPARQL-Code. Bei jedem Funktionsaufruf wird außerdem `count_all` um eine Stelle verringert, die Selektions-Möglichkeit des Benutzers entfernt und ihr Default-Wert selektiert, sodass darauffolgende Nutzereingaben auf einer neuen Münze platziert werden.

Da `newQuery` den kompletten Code zwischen `startQuery` und dem letzten ‘‘}’’, oder ‘‘} }’’ umfasst und die Veränderung auch in den nächsten Schritten bestehen bleiben soll, genügt es, wenn es angepasst wird. Mit der Länge des `div`-Blocks der Klasse `coins` wird dabei unterschieden, nach welchem `Substring` gesucht werden muss. Orientiert wird sich dabei grundlegend an dem `String ‘‘} UNION {}’`, welcher immer den Code-Abschnitt einer neuen Münze einleitet. Zumeist wird sich dazu nach seinem `id`-fachen Vorkommnis gerichtet.

Für das Entfernen der letzten Münze geschieht dies über die Ersetzung jenen Anfangs bis hin zum Ende von `newQuery` durch einen leeren `String`. Wird hingegen eine mittlere Münze der Darstellung entfernt, so findet sich der zu ersetzenende `Substring` über das `id`-fache Vorkommen von ‘‘`UNION`’’ bis hin zum `id+1`-fachen Vorkommen von ‘‘}’’.

Der Code-Abschnitt der ersten Münze lässt sich in jedem Fall über den `Substring` von der ersten Position von `newQuery` bis zum ersten Vorkommnis von ‘‘{’’ entfernen. In beiden Fällen werden die IDs aller nachfolgenden Elemente im `div`-Block `search_container` angepasst, um seine korrekte Struktur beizubehalten.

`newQuery` wird gleich der Funktion `generate_SPARQL()` in das Textfeld der ID `query` gesetzt. Sind alle Münzen gelöscht, so wird das Textfeld der ID `query` durch einen leeren `String` ersetzt und Variablen wie `newQuery` und `count_all` zurück auf ihren Initialwert gesetzt.

4.3 Ergebnisanzeige

Zur tatsächlichen Abfrage muss der im Textfeld der ID `query` generierte Code an einen SPARQL-Endpunkt geleitet und von ihm verarbeitet werden. Wie bereits zuvor erwähnt, wurde sich diesbezüglich für den von der Professur empfohlenen Endpunkt über *Apache Jena Fuseki* entschieden.

Mit einem `form`-Tag der Methode `post` an ein Java Servlet kann dies erreicht werden. Es wird sich außerdem dazu entschieden, die Ergebnisse in einer neuen Registerkarte des Browsers darzustellen. Dies ermöglicht das bessere vergleichen mehrerer Ergebnismengen und das Erweitern zuvor abgefragter Kriterien. Entsprechend erhält der `form`-Tag das Ziel `_blank`.

Um dem Nutzer sichtbar zu machen, was er abgefragt hat, wird außerdem der HTML-Code des `div`-Blocks der ID `search_container` bei jedem Drücken des Add-Buttons in ein unsichtbares und unveränderliches Eingabefeld gespeichert und schließlich mit ans Servlet übertragen.

Das Servlet `SPARQLController` erhält die Werte über das Drücken des `QuerySubmits` und generiert die Ergebnismenge über die Methode `queryAsHTML` mit dem URI des Endpunktes und dem SPARQL-Code als Parameter. Dazu ruft es die Methode `executeSPARQLQuery` auf, welche mittels des `org.apache.jena.query` Packages von *Apache Jena* das Ergebnis der Abfrage in Form eines `XMLString`s serialisiert und zurückgibt. Jener `XMLString` wird anschließend mittels der Methode `asHTML` und dem Java-Programm `SPARQLXMLHandler` in die Form einer HTML-Tabelle überführt. Beide Funktionen wurden bereits von der Professur für die Benutzerschnittstelle *QueryAFE* entwickelt und sind ohne Veränderung übernommen worden.

Schließlich gibt der `SPARQLController` die Ergebnisse der JSP-Datei `result` zurück. Sowohl der SPARQL-Code, als auch der HTML-Code des `search_container` und die verarbeitete HTML-Tabelle werden dabei lediglich an die passende Stelle der neuen Datei übertragen (*Forward*).

4.4 Design

Das endgültige Design der Benutzerschnittstelle wird gemäß des Konzepts aus Kapitel 3.4 entwickelt und von Abbildung 4.5 gezeigt. Fast alle geplanten Aspekte werden umgesetzt und im Folgenden tiefgehender erläutert. Insbesondere eine Funktion mit hilfreichen Kommentaren für den Benutzer wird ergänzt.

Struktur

Die Struktur findet wie erwähnt Umsetzung. Zur genaueren Abgrenzung von den ikonografischen Eingabefeldern von **A1** und **A3** besitzen erstere [BG14] folgend eine höhere Proximität und sind außerdem in der Reihenfolge *Subjekt*, *Prädikat*, *Objekt* eines einfachen Satzes angeordnet. Die Anwendungsreihenfolge zur Darstellung der Funktionen wird eingehalten. Insbesondere die Proximität des Add-Buttons und der Selektion zur Platzierung der Abfragekriterien kann als gemeinsamer Befehl verstanden und damit als Satz gelesen werden, zum Beispiel „Add to same side“.

KAPITEL 4. IMPLEMENTIERUNG

Hingegen wird der Ergebnisbereich auf eine neue Seite ausgelagert, die beim Drücken des Query-Buttons über das Servlet aufgerufen wird. Ebenfalls werden Kopien der Abfragekriterien-Anzeige und das Textfeld des SPARQL-Codes auf der neuen Seite angezeigt, sind jedoch nicht mehr veränderbar und dienen nur der Verbindungsdarstellung der Ergebnismenge zur gemachten Abfrage.

Farbwahl

Abbildung 4.4 bildet die endgültige Farbpalette der Benutzerschnittstelle ab. Es wird darauf abgezielt, ein ansehnliches Ergebnis mit möglichst wenigen verschiedenen und einfachen Tönen zu erlangen.

Ein ausreichender Kontrast des Orangetons zum dunklen Hintergrund ist durch den Unterschied der Helligkeit (Hell-Dunkel-Kontrast) der beiden Farbtöne, wie auch der empfundenen Temperatur (Kalt-Warm-Kontrast) geschaffen und verleiht der Farbe damit eine größere Wichtigkeit. Restliche Farben sind ähnlich kontrastreich gewählt.

Farbe	Verwendung
#eeb55b	Titel, URIs, Button- & Selektion-Ränder & -Text, Superkategorie-Ikonen-Hover, Checkbox-Aktivierung, Hierarchie-Ikon, Fokus, Münzenkennzeichnung, Textselektionshintergrund
#a57d3f	URI-Hover & -Fokus
#fff	Eingabe-Felder, Ergebnis-Tabelle, Header & Footer, Textselektion, Button- & Selektion-Hover, Superkategorie-Ikonen, Hilfshintergrund
#cacaca	Hilfs- & Eingabefeld-Ränder, Checkbox-Hintergrund
#000	Eingabefelder-Schrift, .criteria- & .side_criteria-Hintergrund, Hilfsschrift
#ff0000	query_warn-Ikon

Abbildung 4.4: Farbpalette der Schnittstelle

Schriftart

Für die angekündigte Serifenschrift wird die freie Schriftart *Copperplate Gothic* gewählt. Ihr Ziel ist es, eine sowohl elegante, als auch moderne Wirkung hervorzurufen. *Arial* findet Nutzung als seriflose Schrift.

Zusätzlich wird der einsehbare SPARQL-Code mittels der für Code typischen Schriftart *Courier* dargestellt.

Abfragekriterien-Anzeige

Zur Gestaltung der Abfragekriterien-Anzeige wird die Darstellung in Form von Münzen umgesetzt. Dazu werden mehrere div-Blöcke verwendet. Der äußerste Block einer „Add coin“-Selektion stellt dabei die Münze selbst dar und enthält entsprechend ein Bild einer Münze als Hintergrund. Um nicht über seine runden Ecken

KAPITEL 4. IMPLEMENTIERUNG

hinauszugehen, wird ein weiterer `div`-Block mit vorgegebener Höhe und Breite eingeführt. Erst in ihm werden die vom Benutzer gewählten Abfragekriterien-Kombinationen, ebenfalls in Form eines `div`-Blocks, eingefügt.

Ihre Platzierung kann anhand verschiedener Graustufen und Deckkraft unterschieden werden. So ist die Deckkraft eines Abfragekriterien-Blocks für Kriterien auf derselben Münzseite weitaus höher als die von Abfragekriterien-Blöcken für die andere Münzseite. Blöcke zur Darstellung von Abfragekriterien-Kombinationen irgendwo auf einer Münze werden grau gefärbt.

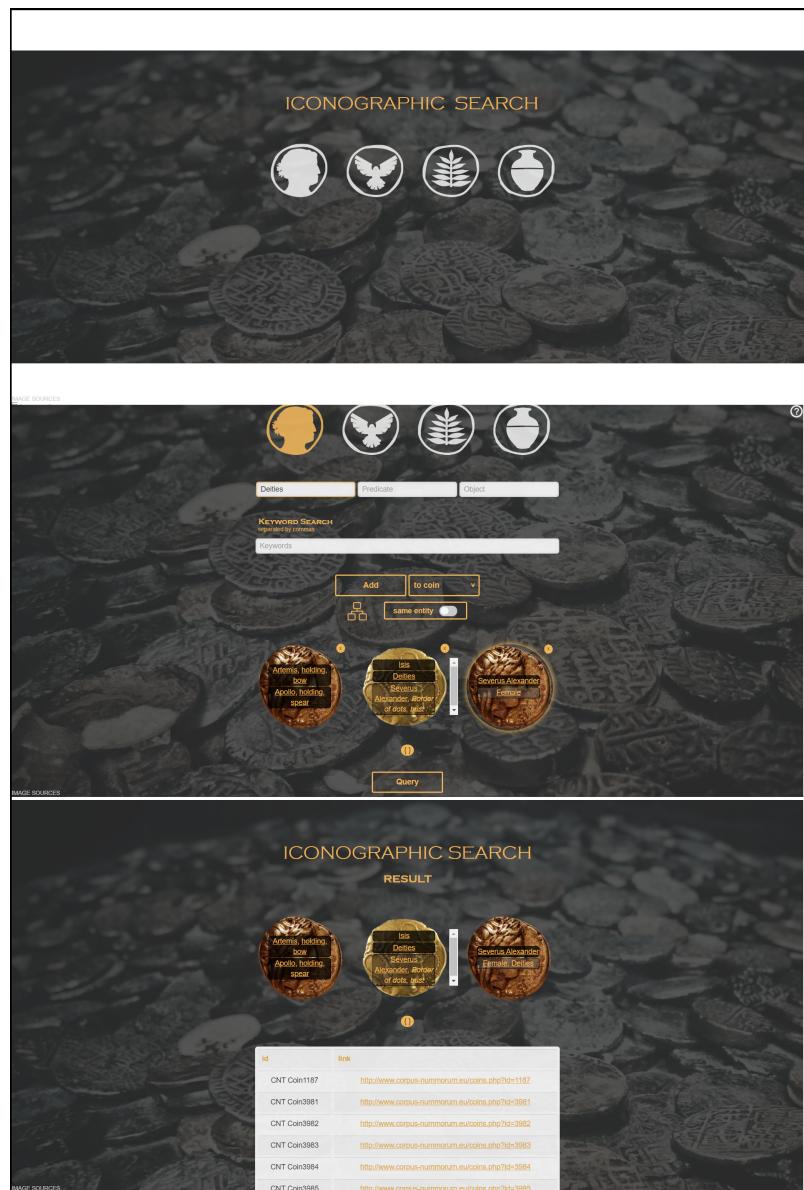


Abbildung 4.5: Design der Schnittstelle nach der vollständigen Implementierung

KAPITEL **FÜNF**

Evaluation

Die entwickelte Benutzerschnittstelle wird schließlich einer Evaluation unterzogen, durch welche mögliche Mängel aufgedeckt und verbessert werden sollen.

Im ersten Schritt wird dabei eigenständig auf die Anforderungen zurückgegriffen und überprüft, ob sie implementiert wurden. Im zweiten Schritt wird eine Evaluation anhand der *Lautes Denken*-Methode mit Testpersonen durchgeführt.

5.1 Anforderungserfüllung

Folgende Tabelle soll aufzeigen, ob und in welchen Kapiteln die vordefinierten Anforderungen aus Kapitel 3.2 implementiert wurden.

Anforderung	Implementierung	Kapitel
A1, A1.1		
A2, A2.1	✓	4.1.2
A4		
A1.2, A1.2.1	✓	4.1.3
A3	✓	4.1.4
A5, A5.1	✓	4.1.2, 4.4 und 4.2
A6	✓	4.3

Tabelle 5.1: Überprüfung der Anforderungserfüllung

Es ist zu schließen, dass alle Anforderungen implementiert worden sind. Hingegen besteht insbesondere bei Anforderungen **A5** und **A5.1** in Zügen der Nutzerfreundlichkeit weitere Verbesserungsmöglichkeiten, wie unter anderem im folgenden Kapitel aufgedeckt.

Auch bleibt Anforderung **A1.2.1** in einigen Fällen unerfüllt. So ist es nur bei keinerlei hierarchischen Beziehungen zwischen den Subjekten einer Platzierung möglich, auf ihr mehr als zwei Subjekte korrekt abzufragen. Mit der mehrerer Kategorien in hierarchischer Beziehung können fehlerhafte Abfragen wie „Female Deities Male“ umgesetzt werden. Des Weiteren sind etwaige Ansätze zur Lösung der Probleme

durch hierarchische Beziehungen zu diesem Zeitpunkt nur für das Subjekt umgesetzt worden. Genaueres wurde bereits in Kapitel 4.1.3.3 festgehalten.

5.2 Evaluation durch Lautes Denken

Zur Aufdeckung von Schwachstellen der Benutzerschnittstelle ist eine Evaluation durch Testpersonen unverzichtbar. Es bietet sich der Analyse der Nutzerbasis aus Kapitel 3.1 folgend an, die Evaluation mit Numismatikern durchzuführen. Da eine größere Evaluation und das Herstellen der Kontakte zu jenen Testpersonen allerdings den zeitlichen Rahmen der Thesis überschreiten würde, wurde sich stattdessen auf insgesamt drei Testpersonen aus Mitarbeitern der Professur und Kommilitonen beschränkt. Offensichtlich ist die Aussagekraft der Evaluation damit eingeschränkt, dennoch konnten erste Probleme mit der Nutzung der Benutzerschnittstelle festgestellt werden.

Den Testpersonen wurden folgende fünf Aufgaben zur Generierung von Abfragen auf der durch Kapitel 4 weitestgehend vollendeten Benutzerschnittstelle vorgegeben. Sie sollen alle möglichen Funktionalitäten im Groben darstellen können. Bei der Lösung jener Aufgaben sollte anhand der zeitsparenden, aber durchaus aussagekräftigen *Lautes Denken*-Methode vorgegangen werden. Sie setzt voraus, dass alle Gedankengänge bei der Bearbeitung der Aufgaben laut ausgesprochen und damit mitgeteilt werden. [DR99]

1. Eine Münze, auf der „**Artemis holding bow**“ zu sehen ist
2. Eine Münze, auf der „**Artemis holding bow**“ und „**Apollo**“ mit dem **Objekt „spear“** auf derselben Seite zu sehen sind
3. Eine Münze, auf der „**Isis**“ und „**Deities**“ auf derselben Seite zu sehen sind
4. Eine Münze, auf der „**Isis**“ und „**Deities**“ auf derselben und „**Severus Alexander**“ mit den Schlagwörtern „**Border of dots**“ und „**bust**“ auf der anderen Seite zu sehen sind. (Schlagwörter bitte genau wie angegeben übernehmen!)
5. Eine Münze, auf der „**Severus Alexander**“ auf einer Seite und „**Female Deities**“ irgendwo auf der Münze zu sehen sind

Bei allen drei Testpersonen konnten ähnliche Ergebnisse beobachtet werden. So sagte die Gestaltung der Benutzeroberfläche zu, die Funktion der Radiobuttons der Superkategorien wurden verstanden und man fand schnell heraus, wo und wie ikonografische Abfragekriterien und Schlüsselworte eingegeben werden müssen.

Hingegen ereignete sich auch dasselbe Problem. Dass der Add-Button vor dem zu Beginn deaktivierten Query-Button gedrückt werden muss, sorgte bei den Testpersonen für Verwirrung. Gleichermaßen taten sie sich im ersten Moment schwer, die Selektion für die Platzierung der Kriterien auf den Münzen zu finden und generierten damit falsche Abfragen in Form von neuen Münzen. Das mit sich ziehende

KAPITEL 5. EVALUATION

Entfernen jener fälschlicher Münzen aus der Abfrage konnte leicht durchgeführt werden. Jedoch wurde die Einschränkung, die vorherige Münze nicht mehr verändern und um Kriterien erweitern zu können als umständlich wahrgenommen.

Nach den ersten zwei Aufgaben fanden sich die Testpersonen allerdings schnell zurecht und konnten die Benutzerschnittstelle ohne weitere Probleme nutzen. Die Unterscheidung der Darstellung der unterschiedlichen Platzierungen auf einer Münze durch Graustufen wurde schnell erkannt und verstanden. Lediglich die Bezeichnung der „same entity“ Checkbox wurde zurecht bemängelt.

Erste Verbesserungen können durch nähere Kennzeichnungen der Buttons und einer besseren Wahl des Begriffs für „same entity“ leicht umgesetzt werden. Der Vorschlag neben der Funktionalität der Hilfsfunktion ein Tutorial anhand eines Beispiels hinzuzufügen würde erwähnte Anfangsschwierigkeiten reduzieren und fließt damit ebenfalls in die letzten Änderungen der Entwicklung ein.

Zusammenfassung und Fazit

Durch die stetig wachsende Anzahl digital überführter numismatischer Daten nach [Gru18] wird Flexibilität und eine eindeutige Identifikation von Datenobjekten als sinnvoll erachtet, insbesondere wenn jeweilige Daten in einer großen Datenbank kombiniert werden sollen. Das *Resource Description Framework* (RDF) stellt diesbezüglich eine gute Lösungsbasis dar.

Die der Bachelorthesis zugrundeliegenden von [KGTP18] mittels NLP-Pipeline generierten ikonografischen Münzdaten bedienen sich jener Basis. Anhand der Abfragesprache SPARQL und eines SPARQL-Endpunkts können sie über die Benutzerschnittstelle abgefragt werden.

Mit der schnellen Eingabe einer Abfrage in Form von *Subjekt*, *Prädikat* und *Objekt*, wobei auch Kategorien enthalten sein können, kann ein entsprechender SPARQL-Code anhand der Struktur der NLP-Daten und mithilfe von SQL-Sichten generiert werden. Manuell oder über die Benutzerschnittstelle kann jener Code unter Berücksichtigung gewählter Platzierungen auf einer Münze erweitert werden. Die Optionen umfassen: „new coin“, „to same side“, „to other side“ und „to coin“.

Hierarchische Beziehungen hinzuzufügender und bestehender Abfragekriterien über die letzteren drei Optionen können dabei Probleme verursachen. So können Kategorien in Verbindung mit direkten Subjekten oder Kategorien als überflüssige Filter wirken, sollte eine hierarchische Beziehung zwischen ihnen bestehen. Die Unterscheidung von Subjekten durch den SPARQL-Code schafft dem zwar Abhilfe, löst das Problem aber vorerst nur für höchstens zwei Subjekte einer Münzseite. Insbesondere die Granularität von Kriterien sorgt weiterhin für fälschliche Ergebnisse.

Hingegen ist die Kombination zweier Kategorien auf einer Münzseite über Verbindung mit demselben Subjekt bei einer hierarchischen Beziehung ermöglicht. Auch die Kombination von ikonografischen Repräsentationen mit einer *Schlüsselwortsuche* ist umgesetzt, durch welche Kriterien gleichermaßen auf Münzen platziert werden können.

Der Nutzerfreundlichkeit wegen können angezeigte Münzen wieder aus der Abfrage entfernt werden und das Ergebnis wird in einem neuen Tab in Form einer Tabelle präsentiert. Zwar besteht durch das umgesetzte Design eine klare Aufwertung, doch anhand der Evaluationsergebnisse können weitere Schwachstellen erkannt werden, deren grobe Bearbeitung jedoch noch erfolgt.

KAPITEL 6. ZUSAMMENFASSUNG UND FAZIT

Schlussfolgernd ist zu schließen, dass die erarbeitete Benutzerschnittstelle zwar noch nicht alle zuvor geschilderten Probleme lösen kann, jedoch in jedem Fall durch ihre vereinfachte Nutzung eine Basis bildet, auf der aufgebaut werden kann.

Dazu können vorliegende teilweise Lösungen des Problems der hierarchischen Beziehungen zwischen Eingaben gleichermaßen auf Prädikat- und Objekt-Eingaben angewendet werden. In jedem Fall sollte jedoch zuerst vorgesehen werden, den Teil des SPARQL-Codes zur Unterscheidung von Subjekt-Eingaben einer Platzierung in hierarchischer Beziehung transitiv zu gestalten und damit für mehr als zwei Subjekt-Eingaben umzusetzen.

Die Kombination von Kategorien zu einem Subjekt sollte zudem in nächsten Schritten alle seine Kategorien zur Überprüfung von Hierarchie und daraus folgenden Funktionalitäten in Betracht ziehen, auch wenn mögliche falsche Abfragen durch das Wissen der Nutzerbasis umgangen werden können.

Über eine Lösung des Problems der Granularität der Daten, die die Unterscheidung von Subjekten erschwert, sollte außerdem nachgedacht werden. So könnten etwaige doppelte Subjekte auch zu einem einzigen kombiniert werden, insbesondere da jeder entsprechende leere Knoten sowieso beide Subjekte zur Instanz hat.

Weitergehend kann die Nutzerfreundlichkeit gesteigert werden, indem zum Beispiel über den Mausfokus einer Münze weitere Kriterien zu eben jener Münze hinzugefügt werden können. Dies könnte beispielsweise über JavaScript-Listen einer jeden Münze erreicht werden, die über ihre IDs zugänglich gemacht werden würden. Ähnliches gilt für die Kombination von Kategorien, die bis jetzt lediglich mit unmittelbar vorkommenden Kategorien einer Münzseite in hierarchischer Beziehung kombiniert werden können.

Für beide Verbesserungen muss die entsprechende Stelle des SPARQL-Codes gefunden werden, was sich als kompliziert herausstellen kann. Jedoch besteht die Möglichkeit eines ähnlichen Vorgehens wie dem aus Kapitel 4.2 zum Entfernen von Münzen. Dabei wurde sich an der Anzahl der vorkommenden ‘‘} UNION {}’’ des SPARQL-Codes orientiert.

Sind derartige Aspekte in der Benutzerschnittstelle umgesetzt, so ist ihre ikonografische Funktionalität weitestgehend ausgereift. Ergänzende andere Aspekte wie *Mint* oder *Publisher* können über die zugrundeliegenden Daten leicht integriert werden, die entstandene Suchmaske erweitern und damit umfangreichere Abfragen ermöglichen.

Mit angedachter Erweiterungen der zugrundeliegenden NLP-Daten durch die Professur um die Superkategorien Tiere, Pflanzen und weitere über dasselbe Prinzip wie vorliegende Personen mit Objekten, kann die Benutzerschnittstelle auch jene abfragen. Die Generierung des SPARQL-Codes für diese Superkategorien ist zu diesem Punkt anhand der SQL-Sichten teilweise ermöglicht, doch aufgrund der fehlenden Grundlage noch nicht abfragbar. Wird diese Grundlage geschaffen, existiert hingegen die Möglichkeit das Vorgehen für Personen zu übernehmen oder leicht anzupassen.

Ähnlich kann eine Orientierung für ikonografische Darstellungen anderer archäologischer Kategorien geschaffen werden. Diesbezüglich müssten jedoch münzspezifische

KAPITEL 6. ZUSAMMENFASSUNG UND FAZIT

Funktionen und ihr SPARQL-Code offensichtlich entsprechend angepasst werden.

Damit ist es offensichtlich, dass im Gebiet der numismatischen Datenverarbeitung noch viel Forschung betrieben werden muss. Insbesondere bietet es sich an, einen gemeinsamen Konsens über das Händeln der bereits großen und stets größer werdenden Datenmengen zu treffen, um nicht im Nachhinein zu viel Zeit in vielzählige nötige Anpassungen investieren zu müssen, die vielleicht an anderer Stelle besser investiert wäre.

Benutzerschnittstelle

Die entwickelte Benutzerschnittstelle, samt ihrer zugrundeliegenden RDF-Daten, SQL-Tabellen und -Sichten, sowie verwendeter Quellen, kann durch folgenden Link heruntergeladen werden.

```
https://github.com/aliciawrt/IGSApplication.git
```

Etwaige einfache Verbesserungsmöglichkeiten erkannt durch die Evaluation des Kapitels 5.2, wie eine bessere Kennzeichnung der Buttons und das Hinzufügen eines Tutorials, wurden umgesetzt. Entsprechend kann sich die Gestaltung der Benutzerschnittstelle leicht von der des Kapitels 4.4 unterscheiden.

Zur Benutzung müssen lediglich beigelegte SQL-Sichten erstellt und ein SPARQL-Endpunkt, wie zum Beispiel *Apache Jena Fuseki*, hergestellt und mit vorliegenden RDF-Daten beladen werden. Werte des Java-Programms `StoredData` des Ordners `query` aus `src` der Applikation müssen entsprechend der eigenen angepasst werden.

Die Applikation wurde unter der Verwendung des *Google Chrome* Browsers getestet, weshalb dessen Nutzung empfohlen wird. Mit der Nutzung anderer Browser kann es zu leicht veränderten Anzeigen kommen, der Funktionalität sollte es aber nicht schaden.

LITERATURVERZEICHNIS

- [And01] Mark Andrews. Story of a Servlet: An Instant Tutorial, 2001. <https://www.oracle.com/technetwork/java/tutorial-138750.html>, besucht am 13.08.2019.
- [Ber09] Michael Bergman. Advantages and Myths of RDF. *AJ3, April*, 2009.
- [BG14] Jason Beaird und James George. *The principles of beautiful web design*. SitePoint, 2014.
- [BV10] David Bernstein und Deepak Vij. Using Semantic Web Ontology for Intercloud Directories and Exchanges. In *International Conference on Internet Computing*, S. 18–24, 2010.
- [Cla05] Kendall Clark. RDF Data Access Use Cases and Requirements, 2005. <https://www.w3.org/TR/rdf-dawg-uc/>, besucht am 09.07.2019.
- [Cod13] Code Conquest. Client Side vs. Server Side, 2013. <https://www.codeconquest.com/website/client-side-vs-server-side/>, besucht am 13.08.2019.
- [DR99] Joseph S Dumas und Janice Redish. *A practical guide to usability testing*. Intellect books, 1999.
- [DVG12] Roberto De Virgilio, Francesco Guerra und Yannis Velegrakis. *Semantic Search over the Web*. Springer Science & Business Media, 2012.
- [G⁺05] Jesse James Garrett et al. Ajax: A new approach to web applications. 2005.
- [Gru18] Ethan Gruber. Linked Open Data and Hellenistic Numismatics. In *Alexander the Great: A Linked Open World*, S. 17–33. 2018.
- [HS13] Steven Harris und Andy Seaborne. SPARQL 1.1 Query Language, 2013. <https://www.w3.org/TR/sparql11-query/>, besucht am 09.07.2019.
- [KGTP18] Patricia Klinger, Sebastian Gampe, Karsten Tolle und Ulrike Peter. SEMANTIC SEARCH BASED ON NATURAL LANGUAGE PROCESSING—A NUMISMATIC EXAMPLE. *JOURNAL OF ANCIENT HISTORY AND ARCHAEOLOGY*, 5(3), 2018.

- [Krc05] Helmut Krcmar. Informationsmanagement, 4., überarb. und erw. Aufl., Berlin ua, 2005.
- [MMM⁺14] Frank Manola, Eric Miller, Brian McBride et al. RDF Primer, 2014. <https://www.w3.org/TR/rdf-primer/>, besucht am 09.07.2019.
- [PAG06] Jorge Pérez, Marcelo Arenas und Claudio Gutierrez. Semantics and Complexity of SPARQL. In *International semantic web conference*, S. 30–43. Springer, 2006.
- [RDF14] RDF Working Group. RDF, 2014. <https://www.w3.org/RDF/>, besucht am 09.07.2019.
- [SP13] Andy Seaborne und Eric Prud'hommeaux. SPARQL Query Language for RDF, 2013. <https://www.w3.org/TR/rdf-sparql-query/>, besucht am 09.07.2019.
- [Ver04] Cristina Vertan. Resource Description Framework (RDF), 2004. <https://nats-www.informatik.uni-hamburg.de/pub/SemanticWeb04/EinfuehrungsFolien/rdf.pdf>, besucht am 09.07.2019.