Entwurfsbeschreibung: Datensuche für Linked Data

1 Allgemeines

Das Semantic Web und seine dazugehörigen Komponenten wachsen jeden Tag. Dazu gehört somit auch Linked Open Data. Um diese Daten effizient nutzen zu können benötigt man eine Suchmaschine. Diese wurde für Linked Data durch das Open-Source-Project "Tapioca" bereits implementiert. Allerdings gibt es einige Punkte, in denen Tapioca verbessert werden kann.
Im Rahmen des Softwaretechnik-Praktikums soll die Suchmaschine verbessert werden. Dabei wird eine benutzerfreundliche Webseite erstellt, auf der man eine Schlagwortsuche oder eine Suche durch das Hochladen einer RDF-Datei starten kann.

2 Produktübersicht

Das Produkt besteht aus einer Webseite, einem Indexgenerator, einem Benchmarking Tool und einem Metadata Extraction Tool.
Das Webinterface wird dabei auf einem Tomcat-Server aufgesetzt, da eine Kommunikation zwischen der Webseite und einem Java-Programm stattfinden soll. Die Suche wird, wie schon bemerkt, entweder durch Schlagwörter, oder durch eine RDF-Datei möglich sein.

3 Grundsätzliche Struktur- und Entwurfsprinzipien

3.1 Grobe Gliederung

Das Projekt lässt sich insgesamt in vier kleinere Teilprojekte gliedern. Aus diesem Grund haben wir uns für vier git-Projekte entschieden.
Für den Benutzer der Suchmaschine steht die Webseite im Vordergrund. Diese sollte somit einfach und intuitiv bedienbar sein. Es wird Wert auf eine Oberfläche gelegt, die einen nicht "überrumpelt", aber dennoch viel Funktionalität liefert.
Das Metadata Extraction Tool wird von den Benutzern der Webseite benötigt, um die Metadaten aus ihren Dateien zu filtern. Das macht das Vergleichen ihrer Datensätze einfacher.
Der Indexgenerator wird im Hintergrund laufen und ist somit für den Benutzer nicht sichtbar. Deshalb steht hier die Effizienz im Vordergrund.
Das Bechmarking Tool wird nur vom Administrator verwendet.

3.2 MVC-Architektur

Da dieses Projekt hauptsächlich mit Daten und Dateien arbeitet, hat sich das Team für eine MVC-Architektur entschieden. Somit teilt sich das Programm nach dem EVA-Prinzip in Eingabe, Verarbeitung und Ausgabe. Die Eingabe erfolgt sowohl über den Benutzer, in dem er ein Schlagwort oder eine Datei zur Suche eingibt, als auch durch das Auslesen der bereits vorbearbeiteten Daten aus der LOD-Cloud. Dann wird durch Tapioca die Daten verarbeitet und der Ähnlichkeitsgrad berechnet und ausgewertet. Diese werden dann wiederum auf der Webseite ausgegeben.

3.3 JSF - Java Server Faces

Da das Projekt eine Kommunikation zwischen einer Webseite und einem Java-Programm benötigt, wurde sich hier für das Framework JSF entschieden. Die Verbindung zwischen Webseite und Programm entsteht dann mittels Managed-Beans. Dabei ist es wichtig, dass alle Variablen, die durch den Benutzer eingegeben werden sollen, eine set-Methode und eine get-Methode besitzen. Diese werden dann automatisch bei der Eingabe, bzw. Ausgabe, aufgerufen.

3.4 Maven

Liest man sich Bücher über JSF durch, so wird oftmals empfohlen Maven mit zu benutzen. Maven liefert eine einfache Ordnerstruktur und teilt das Projekt in drei Teile auf: java, resources und webapp. Für das Webinterface werden dann alle .java-Dateien in den Ordner java, und alle .html-Dateien sowie zugehörige .css-Dateien in den Ordner webapp gespeichert.

4 Struktur- und Entwurfsprinzipien einzelner Pakete

4.1 Webseite

Die Webseite ist die zentrale Anlaufstelle für den Nutzer. Diese sollte, wie bereits erwähnt, einfach und intuitiv bedienbar sein. Das Eingabefeld kann der Nutzer benutzen, um ein Schlagwort einzugeben, oder eine Datei per Drag-And-Drop hochzuladen um so danach zu suchen. Außerdem wird es ein Menü geben, welches weitere Informationen und Funktionen bereitstellt. Das Menü liegt in der oberen rechten Ecke und teilt sich in drei Punkte auf. Der Menüpunkt “Readme“ liefert weiterführende Informationen zur Benutzung der Suchmaschine. Im Punkt “Tools“ werden für Benutzer Werkzeuge, wie das Metadata Extraction Tool, zur Verfügung stehen. Als letzten gibt es den “Contact“, welches das Impressum der Seite darstellt.
Nach der Suche gelangt der Nutzer auf eine Ergebnisseite. Auf dieser findet man sowohl das Menü, als auch eine Suchleiste wieder, sodass der User nicht

auf die Homepage zurück muss, um eine weitere Suche zu starten.
Die Ergebnisse bestehen aus einem Kopf (Link zur Datei), der Ähnlichkeit (zwischen 0 und 1) und dem Körper mit einem kleinen Auszug aus der Datei.
Da dieses das erste Paket ist, ist die eigentliche Suchfunktion noch nicht eingebaut. Stattdessen liefert die Ergebnisseite vorprogrammierte Suchergebnisse, welche die Suchanfrage in den Köpfen stehen haben.

4.2 Metadata Extraction

Das Metadata Extraction Tool ist eine Anwendung, die der Benutzer vor der Suche auf seinen eigenen Daten benutzen sollte.
Die eingegebenen RDF-Daten können Dabei die Formate JSON-LD, N-TRIPLES, N3, RDF/JSON, RDF/XML, RDF/XML-ABBREV oder TURTLE haben. Dabei kann der Nutzer sogar entscheiden welches Ausgabeformat die resultierende Datei haben soll. Man kann die Dateien also ineinander umwandeln. Als erstes werden VoID-Information extrahiert. Dabei wird ein VoID-Model erstellt. Danach werden die Labels der RDF-Datei erfasst und zum Model hinzugefügt. Nach diesen Schritten wird das Model in eine Datei mit entsprechenden Ausgabeformat geschrieben.

5 Datenmodell

5.1 Metadata Extraction

Da die Datenmodellierung (Klassendiagramme) ein größeres und komplexes System ist, wurde diese dreigeteilt. Der erste Teil (Anhang A1) enthält einen groben Überblick, das zweite Diagramm (Anhang A2) zeigt das Extrahieren der VoID-Information und im Anhang A3 sieht man die Label-Extraction.

6 Glossar

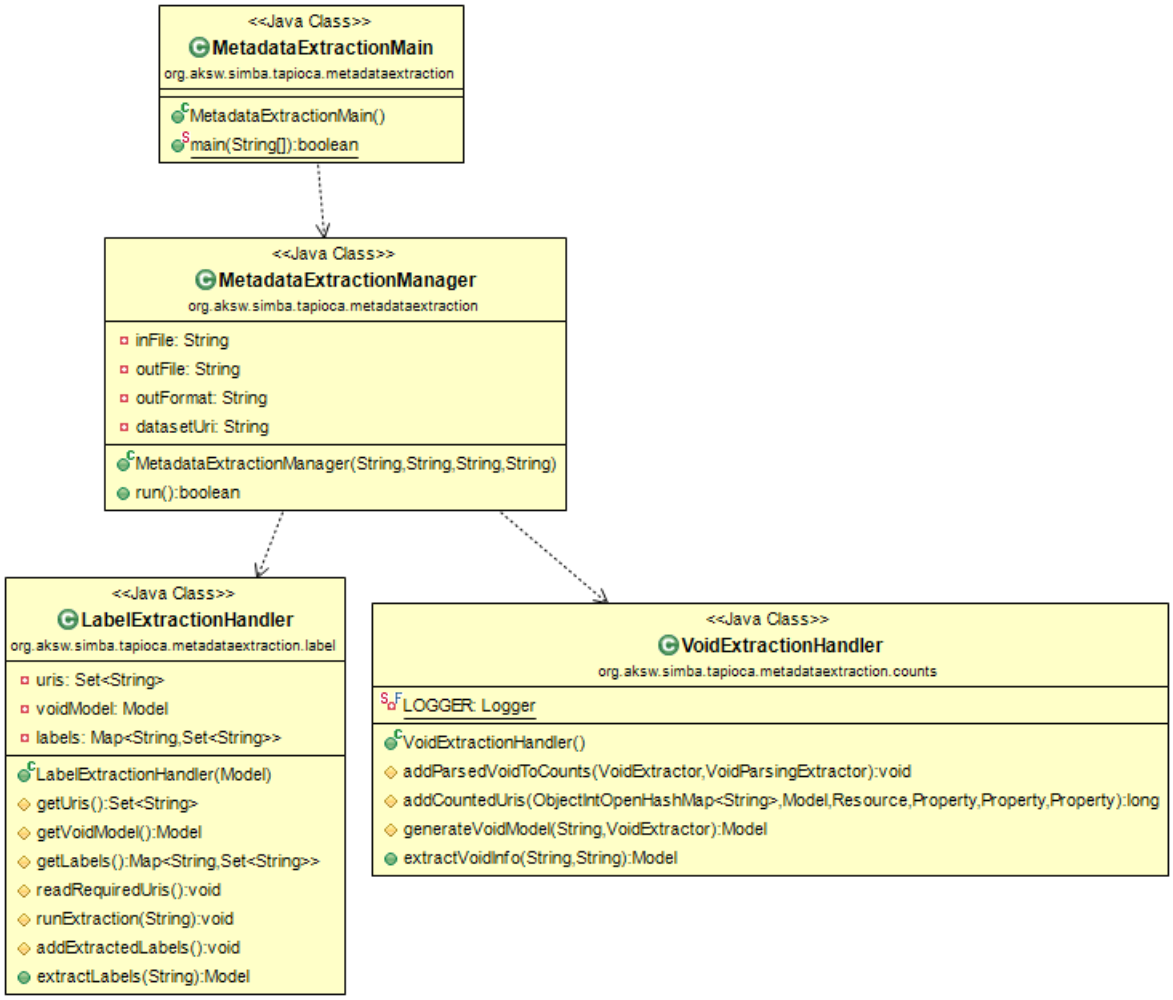
MVC steht für Model-View-Controller und ist ein Architekturmodell der objektorientierten Programmierung. Es gibt eine strikte Rollenverteilung und teilt ein Programm in ein Datenmodell (model), der Datenpräsentation (view) und der Programmsteuerung (controller) auf.

RDF steht für Resource Description Framework und erweitert das Web um die Möglichkeit Inhalten miteinander zu verbinden. In RDF-Dateien werden Aussagen über Ressourcen getroffen, wobei Ressourcen eindeutig bezeichnete Dinge der Welt sind. Dabei werden durch einen Graphen (bzw. Tripel) Ressourcen mit anderen Ressourcen verbunden.

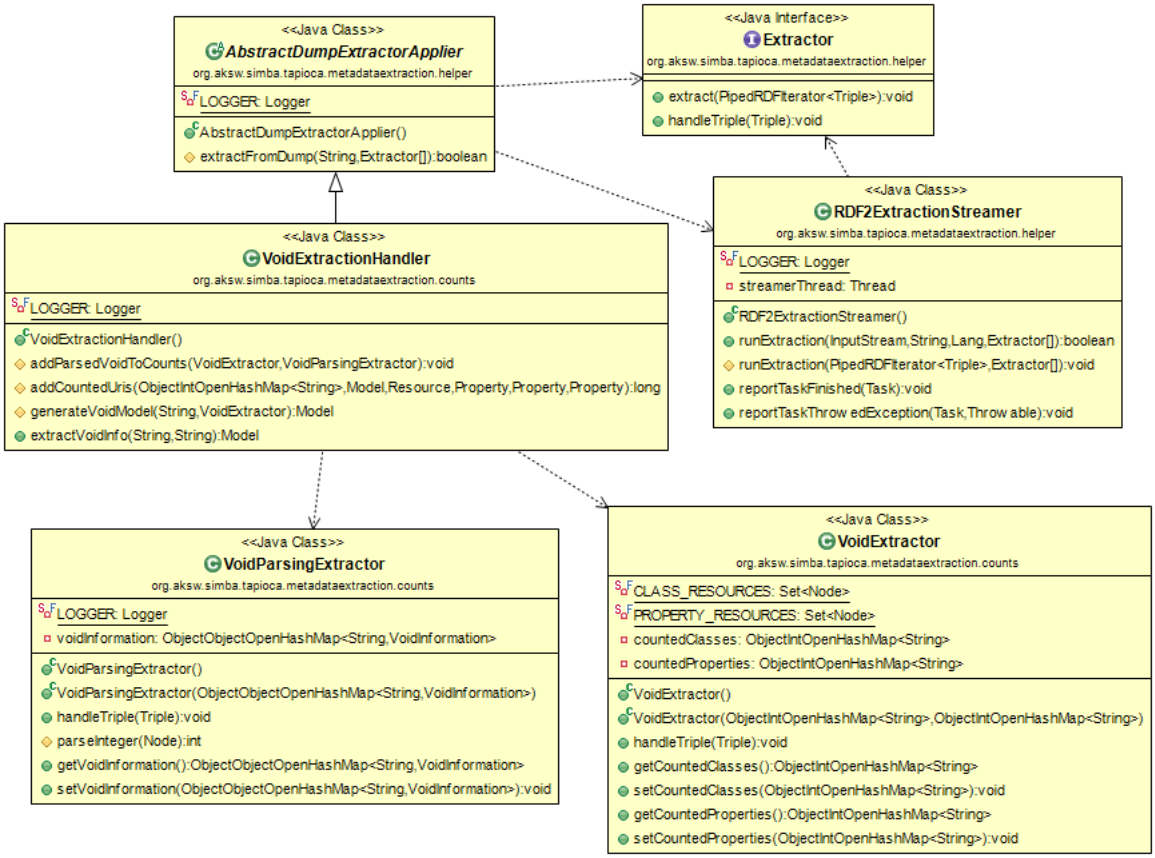
Metadata beinhalten Merkmale über andere Daten. So werden oft größere Datensammlungen (Dokumente, Bücher, etc.) beschrieben.

VoID ist ein RDF-Vokabular um Metadata ausdrücken zu können. So wird eine Schnittstelle zwischen Ersteller und Nutzer von RDF-Daten geschaffen.

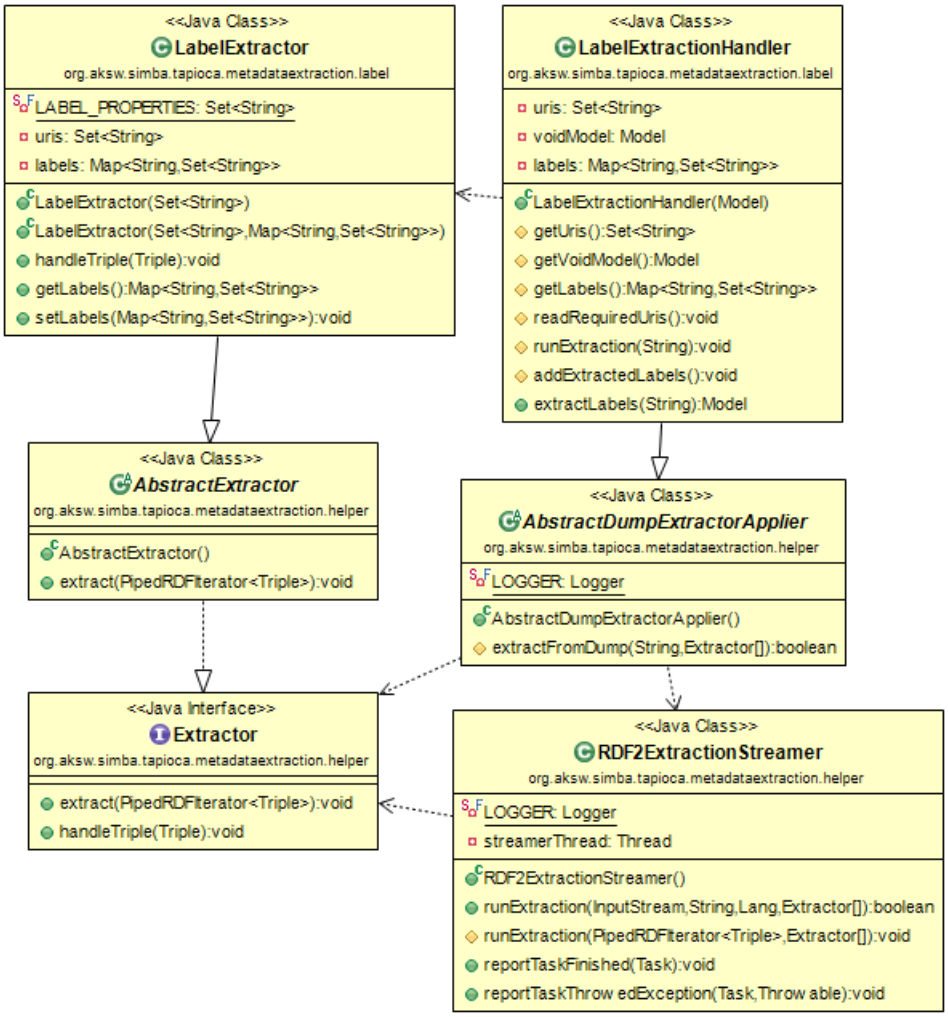
Anhang A1


<<Java Class>>
MetadataExtractionMain
org.aksw.simba.tapioca.metadataextraction
MetadataExtractionMain()
main(String[]):boolean
<<Java Class>>
MetadataExtractionManager
org.aksw.simba.tapioca.metadataextraction
inFile: String
outFile: String
outFormat: String
datasetUri: String
MetadataExtractionManager(String,String,String,String)
run():boolean
<<Java Class>>
LabelExtractionHandler
org.aksw.simba.tapioca.metadataextraction.label
uris: Set<String>
voidModel: Model
labels: Map<String,Set<String>>
LabelExtractionHandler(Model)
getUris():Set<String>
getVoidModel():Model
getLabels():Map<String,Set<String>>
readRequiredUris():void
runExtraction(String):void
addExtractedLabels():void
extractLabels(String):Model
<<Java Class>>
VoidExtractionHandler
org.aksw.simba.tapioca.metadataextraction.counts
LOGGER: Logger
VoidExtractionHandler()
addParsedVoidToCounts(VoidExtractor,VoidParsingExtractor):void
addCountedUris(ObjectIntOpenHashMap<String>,Model,Resource,Property,Property,Property):long
generateVoidModel(String,VoidExtractor):Model
extractVoidInfo(String,String):Model

Anhang A2


<<Java Class>>
AbstractDumpExtractorApplier
org.aksw.simba.tapioca.metadataextraction.helper
LOGGER: Logger
AbstractDumpExtractorApplier()
extractFromDump(String,Extractor[]):boolean
<<Java Interface>>
Extractor
org.aksw.simba.tapioca.metadataextraction.helper
extract(PipedRDFIterator<Triple>):void
handleTriple(Triple):void
<<Java Class>>
RDF2ExtractionStreamer
org.aksw.simba.tapioca.metadataextraction.helper
LOGGER: Logger
streamerThread: Thread
RDF2ExtractionStreamer()
runExtraction(InputStream,String,Lang,Extractor[]):boolean
runExtraction(PipedRDFIterator<Triple>,Extractor[]):void
reportTaskFinished(Task):void
reportTaskThrowedException(Task,Throwable):void
<<Java Class>>
VoidExtractionHandler
org.aksw.simba.tapioca.metadataextraction.counts
LOGGER: Logger
VoidExtractionHandler()
addParsedVoidToCounts(VoidExtractor,VoidParsingExtractor):void
addCountedUris(ObjectIntOpenHashMap<String>,Model,Resource,Property,Property,Property):long
generateVoidModel(String,VoidExtractor):Model
extractVoidInfo(String,String):Model
<<Java Class>>
VoidParsingExtractor
org.aksw.simba.tapioca.metadataextraction.counts
LOGGER: Logger
voidInformation: ObjectObjectOpenHashMap<String,VoidInformation>
VoidParsingExtractor()
VoidParsingExtractor(ObjectObjectOpenHashMap<String,VoidInformation>)
handleTriple(Triple):void
parseInteger(Node):int
getVoidInformation():ObjectObjectOpenHashMap<String,VoidInformation>
setVoidInformation(ObjectObjectOpenHashMap<String,VoidInformation>):void
<<Java Class>>
VoidExtractor
org.aksw.simba.tapioca.metadataextraction.counts
CLASS_RESOURCES: Set<Node>
PROPERTY_RESOURCES: Set<Node>
countedClasses: ObjectIntOpenHashMap<String>
countedProperties: ObjectIntOpenHashMap<String>
VoidExtractor()
VoidExtractor(ObjectIntOpenHashMap<String>,ObjectIntOpenHashMap<String>)
handleTriple(Triple):void
getCountedClasses():ObjectIntOpenHashMap<String>
setCountedClasses(ObjectIntOpenHashMap<String>):void
getCountedProperties():ObjectIntOpenHashMap<String>
setCountedProperties(ObjectIntOpenHashMap<String>):void

Anhang A3


<<Java Class>>
LabelExtractor
org.aksw.simba.tapioca.metadataextraction.label
LABEL_PROPERTIES: Set<String>
uris: Set<String>
labels: Map<String,Set<String>>
LabelExtractor(Set<String>)
LabelExtractor(Set<String>,Map<String,Set<String>>)
handleTriple(Triple):void
getLabels():Map<String,Set<String>>
setLabels(Map<String,Set<String>>):void
<<Java Class>>
LabelExtractionHandler
org.aksw.simba.tapioca.metadataextraction.label
uris: Set<String>
voidModel: Model
labels: Map<String,Set<String>>
LabelExtractionHandler(Model)
getUris():Set<String>
getVoidModel():Model
getLabels():Map<String,Set<String>>
readRequiredUris():void
runExtraction(String):void
addExtractedLabels():void
extractLabels(String):Model
<<Java Class>>
AbstractExtractor
org.aksw.simba.tapioca.metadataextraction.helper
AbstractExtractor()
extract(PipedRDFIterator<Triple>):void
<<Java Class>>
AbstractDumpExtractorApplier
org.aksw.simba.tapioca.metadataextraction.helper
LOGGER: Logger
AbstractDumpExtractorApplier()
extractFromDump(String,Extractor[]):boolean
<<Java Interface>>
Extractor
org.aksw.simba.tapioca.metadataextraction.helper
extract(PipedRDFIterator<Triple>):void
handleTriple(Triple):void
<<Java Class>>
RDF2ExtractionStreamer
org.aksw.simba.tapioca.metadataextraction.helper
LOGGER: Logger
streamerThread: Thread
RDF2ExtractionStreamer()
runExtraction(InputStream,String,Lang,Extractor[]):boolean
runExtraction(PipedRDFIterator<Triple>,Extractor[]):void
reportTaskFinished(Task):void
reportTaskThrowedException(Task,Throwable):void