Tallinna Tehnikaülikool
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Text autocompletion and prediction REST service based on a graph database.

Bakalaurusetöö

Üliõpilane:	Ilja Gužovski
Üliõpilaskood:	112645
Juhendaja:	Raul Liivrand

Tallinn
2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---	--
(*kuupäev*)	(*allkiri*)

Annotatsioon

Antud töö seletab, kuidas luuakse teksti sisestamist soovitussüsteemi REST teenuste ja graafandmebaasi toel. Käesoleva bakalaureusetöö eesmärgiks on koondada teadusuuringuid teksti ennustamises, graafiteoorias, lingvistikas, andmebaasides, päringu optimeerimises ja dokumenteerida arendamise protsessi ja arhitektuurpilti. Esiteks kirjeldatakse, kuidas töötab algoritm. Teiseks kirjeldatakse tarkvara arhitektuuri. Lõpuks antakse järeldus: kas antud arhitektuur ja algoritm on efektiivsed.

Peamine probleem, millega ma olen kokku puutunud, oli teadmatus, kuidas disainida infosüsteeme, mis kasutavad graafandmebaase, kuidas graafandmebaas skaleerub, kuidas optimeerida päringuid ja kas üldse graafandmebaasid sobivad antud probleemi lahendamiseks.

Töötav prototüüp, mis kasutab optimeeritud päringuid ja on piisavalt kiire selleks, et töötada live-is, on loodud käesoleva bakalaureusetöö tulemusena. Lisaks eelnevale, kui ma lõpetan tarkavara arendamise ja optimisatsiooni, tekib mul võimalus teha järeldus: millised olid antud graafi mudeli plussid ja miinused, ja kas on üldse mõistlik kasutada graafandmebaasi antud probleemi lahendamiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 47 leheküljel, 4 peatükki, 13 joonist, 24 tabelit.

Abstract

The current thesis describes the development of the fast working text prediction and suggestion REST service based on the graph database. The aim of this thesis is to assemble research in the text prediction, graphs, linguistics, databases, and query optimizations and to document the implementation process of the new design. Firstly, the design principles of algorithm are described. Secondly, the software architecture is described. Lastly, I give a conclusion: could the designed solution and algorithm be efficient while working under the real stress.

The main problem with which I dealt was the lack of knowledge how to design such a system with the use of the graph database, how does the graph database scales and suits for this goal and how to optimize the database queries.

An implementation of the software, which uses optimized queries and has sufficient speed to work in live, is created as a result of this thesis. In addition, when the software implementation and optimization would be ended I could give a conclusion, what were the cons and pros of designed graph model and is it reasonable to use it.

The thesis is in English and contains 47 pages of text, 4 chapters, 13 figures, 24 tables., etc.

Abbreviations and glossary of terms

UI	User interface
JSON	JavaScript Object Notation
MVC	Model–view–controller
REST	Representational state transfer
API	Application programming interface
SOAP	Simple Object Access protocol
SOLID	Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion
TDD	Test-driven development
BDD	Behavior-driven development
DAO	data access object
RAM	Random-access memory
JDBC	Java database connectivity
ORM	Object-relational mapping

Table of figures

Table of tables

Table of Contents

1. Introduction

It is relatively easy today to see many kinds of auto completion services. For instance, Google Search autocomplete API [1], or next work prediction and suggestion in iOS and Android devices. At the same time, technologies of this kind, which are used in Google, Apple and Microsoft are mostly proprietary. Except Apache Lucene [2] search engine based solutions, there are not so many well-known open-source text auto-completion, prediction and suggestion services. Furthermore, I also have discovered that there is still no well-known search prediction and suggestion engines, which use graphs as a representation model for the sentences and text. That is why I decided to write a service of this kind and its engine, which will be based on graph database [3].

1.1 Background and problem

As mentioned above, there are not so many text suggestion engines. The main problem I solve is the exploration how this type of software should be designed and optimized in order to work in live and behave under the load.

The reason why designed software could be useful are the following: you could use it for learning a new language, it could assist you when you write or type some text, and it could also help handicap people to write more quickly. If I could make my work more commercial, it could be used as a JavaScript widget in form fields, where you have to write a lot, because usually there are many mistakes and typos done by the non-native speakers.

The reason why this work is useful are the following:

- Explore the usefulness of graphs in text search and prediction.
- Find the ways how to design this kind of service.
- Give an evaluation of the graph database Neo4j [4] and understand does this database suit for those kind of tasks.

1.2 Aims and goals of thesis

The main goal of the diploma is the working REST service [5] which could predict next word depending on the previous given input of words. In the terms of speed it should be fast enough to handle user typing and frequent queries. The service will be written in Java [6]. An additional goals of my work are research in graph databases and linguistics. From the point of view of a software engineer my application could be interesting in terms of architecture. After reading my thesis, the reader must understand the difficulties which I met while designing the solution based on graphs, it cons and pros.

1.3 Methodology

To reach the original goal I implemented 3 layered architecture [29], fully covered with unit and integration tests. The designed system is designed in object oriented way (this mean that I will use as much SOLID [7] rules, as I could while designing), with the use of modern software development practices like TDD (Test Driven Development) [8]. At the end of the work I expect to see well build and loosely coupled modules of my application, which would be easy to optimize, redesign and connect.

1.4 Thesis overview

The second part of my thesis explains basic idea of algorithm, its optimization and implementation considerations.

The third part of my thesis explains which technologies I have chosen for designing. It also describes why I believe some technologies suit better my service than others. It also contains use cases, deployment instructions and documentation for each package and class.

In fourth part, I would give an evaluation, measure the performance of my application and give a conclusion to designed architecture of my application.

2. Algorithm

In this section I will try to explain the initial idea of algorithm, which I want to design. I also included performance requirements and implementation considerations.

2.1 Design principles

At first, we have to clarify how we could transform the words and sentences into graphs.

Consider the following sentence: To be, or not to be, that is the question.

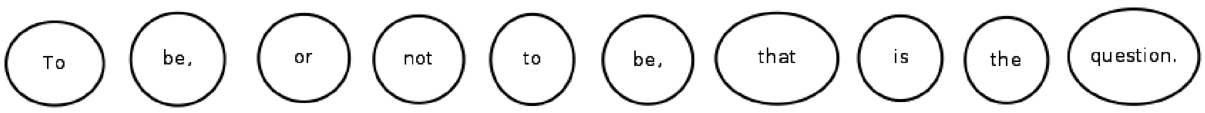

Figure 1: Ordinary sentence

We could represent each word as a point, so each next point will be in a relationship with previous point.

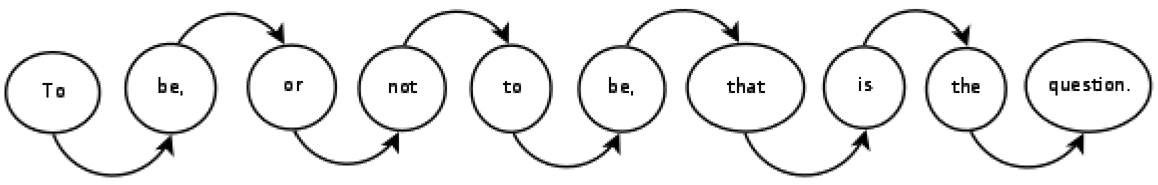

Figure 2: Sentence words relationships

We could realize that this is an ordinary graph, where the word is vertex [9] and so called relationship is edge [10]:

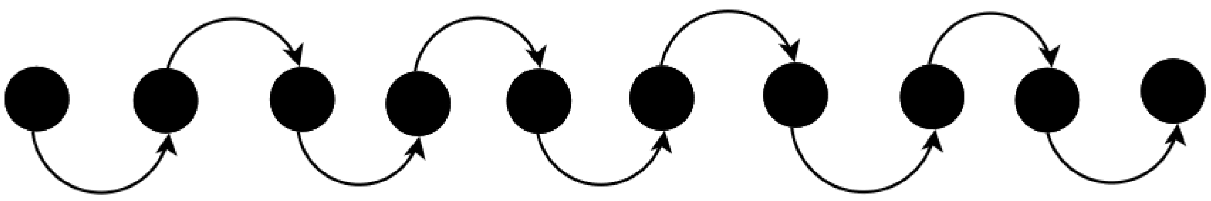

Figure 3: Graph look on sentence words relationships

However, each relationship should have power (or popularity), so, for instance, relationship of words “to be,” should have power (popularity) of two. And if, in addition to sentence “To be, or not to be, that is the question”, we had a sentence: “To say something one time”, then if you suggest a next word for a word “to”, then the first result will be a word “be,” and the second “say”.

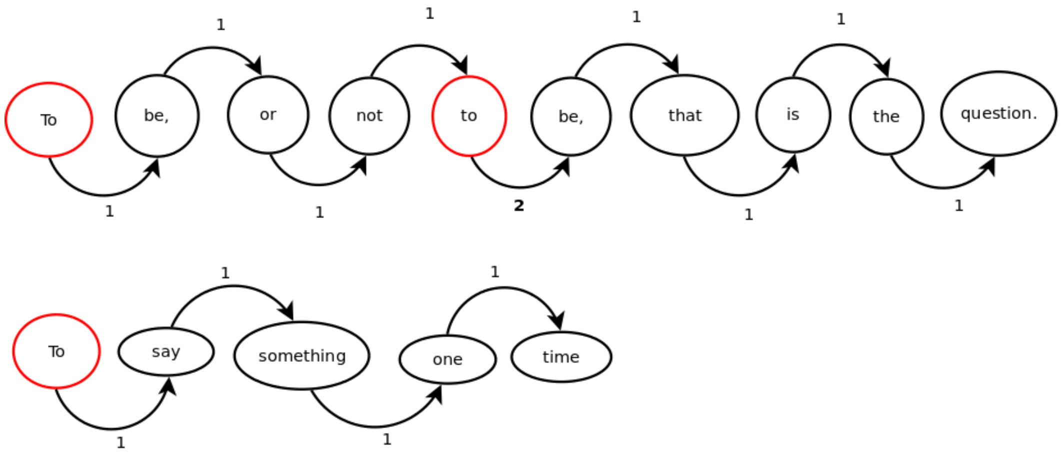

Figure 4: Powers of edges

2.2 Optimization

However, by iteratively designing my algorithm I had found that the algorithm does not understand context well. Due to this reason, I implemented the following idea: Each 4 points of the graph are considered as one point. So, if you give an input of three words, you will get fourth word, which will be more suitable for the place, because it knows, which 3 words preceded before. For instance, the sentence “To be, or not to be, that is the question.” consists of following “subgraphs”:

1. To be, or not
2. be, or not to
3. or not to be
4. not to be, that
5. to be, that is
6. be, that is the
7. that is the question.
8. is the question. <nullword>
9. the question. <nullword> <nullword>

Where the <nullword> means that there is no word followed.

In addition, to improve the prediction result, right after sorting the results by the power of relationship, I sort those results by the popularity of each word (I also hold popularity of each word in database). Algorithm for this task is easy: on each save of the relationship, we also increment popularities of words presented in given relationship.

2.2 Implementation considerations

At first, the text suggestion is not efficient if it works slowly. So the first requirement of algorithm is to work instantly. The instantly means that suggestion must work on every key press. The word record in typing is 256 wpm (words per minute) [11]. Each word is by definition 5 characters [12]. After performing simple calculations I got (256*5) / 60 = 21.3. This means that 22 character per second is absolute maximum. Then we could count how much time each request and response should take: 1000/22 = 45,454545455 ~ 45 milliseconds/request. However, the average user will hardly notice 100 millisecond delay [13]. So the desired response time lies between 40ms and 100ms.

Secondly, from the point of memory the service should not be memory hog. In fact, English vocabulary contains at most 250000 words [14]. However, we should consider that capitalized word, or word with coma is also considered a word in our application, as a result, we could multiply our value with a factor of four. Therefore our database will contain around 1 million words. Which is not a big value for modern database. Because of this reason, I suppose, that service will take no more than 256 megabytes of RAM.

Because there are some performance limitations, I looked towards to compiled languages, like Java, because compared to Ruby, for instance, it runs more than 100 times faster [15]. But we have to keep in mind that usually database is the bottleneck of the system performance.

The graph database was chosen because it suits my model – it also consists of graphs. The reason why I did not chose relational database, was the fact, that they are not efficient on big amount of join operations. My algorithm implies a lot of join operations if we use relational model.

As I mentioned before, the number of records in the database would be around 1-2 million. I am not sure, but I expect that the size of the database will be less than 10 gigabytes.

3. Implementation

3.1 Platform & language selection

There are plenty of technologies available today for designing the web system. The most mature and general are: Java, Python [16], Ruby [17], PHP [18]. For designing the REST service, from my point of view, there are only 3 languages suitable for this goal: Java, Python or Ruby.

However, despite the fact that you could work with neo4j via Python, Ruby and PHP, originally it was intended to work more with Java (Because Neo4j is also written in Java [19]).

As a result, I have chosen libraries like Spring MVC [20] and Spring Data Neo4j [22], because I suppose they will prevent me from writing “boilerplate” code.

3.2 Web application architecture

From the graph databases I have chosen Neo4j, because of its maturity and big community. Currently, Neo4j is the most used graph database.

As a primary Java Framework I have chosen Spring MVC 3, also because of its maturity. At the same time there were many available artefacts for Spring MVC. I have added one of them: Spring Data Neo4j – which is actually Neo4j database ORM. The Spring MVC, in my case, is configured as a REST service, which produces JSON responses with Jackson. Everything mentioned over are back-end technologies. From the front-end technologies I use AngularJS [23], but for demonstrating the features I had written small jQuery [24] widget, which detects user typing in textarea and brings handy popup with predicted next word. My REST service also could use SOAP for presentation protocol, but I consider this as a bad idea, because JSON is already standard de-facto for the newer web applications [25].

The server side architecture follows ordinary 3 layer architecture: we have model layer, Spring repository layer (which is actually DAO layer), DAO layer (which actually delegates to Spring repository layer), service layer and controller layer.

From the front end side, we use JQuery html textarea widget.

3.3 Technology selection

For my system there were a variety of technologies, but I had chosen the following:

- Neo4j - open-source graph database, implemented in Java.
- Spring MVC - open source application framework and inversion of control container for the Java platform.
- Spring Data Neo4J – which offers advanced features to map annotated entity classes to the Neo4j Graph Database.
- Jackson - suite of data-processing tools for Java
- AngularJS - open-source web application framework
- jQuery - cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

For the testing purposes I used:

- JUnit [26] is a unit testing framework for the Java programming language.

3.4 Deployment Diagram

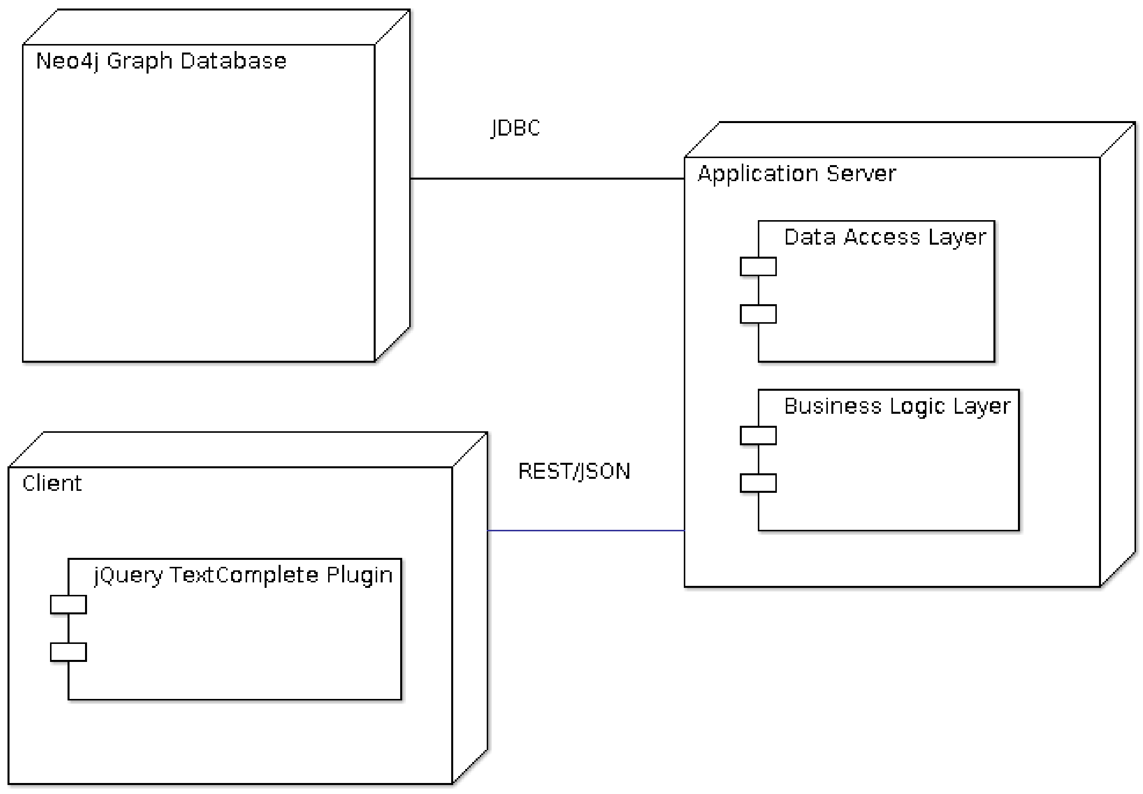

Figure 5: Deployment Diagram

There are 2 main possibilities to organize Neo4j graph database: you can use the standalone explicit one or use embedded database, which comes bundled with Spring Data Neo4j.
As I mentioned before, I use Spring Data Neo4j ORM for Neo4j, so you only have to hardcode the JDBC [27] source path. If properly configured, your DAO (Data Access Object) layer should be connected to database. Of course, you should manually create all models and map them. In its turn, those models interact with business logic layer and produce JSON. Those JSON responses should be intercepted by client (client is jQuery widget).

To build my application from scratch you should have Java 7 and Maven 3.2.x installed.
To define a jdbc url please explore ApplicationConfig.java class. Then navigate to target folder and run from command line “mvn clean install –DskipTests”. After that copy WordService.war from target folder and deploy an app with appropriate Java EE 7 container [28] like Jetty 9.1 or Tomcat 8. If you compile from sources and agree to use jetty embedded

container, then you can use command “ mvn clean install –DskipTests jetty:run” to quickly run application.

Then open a browser and navigate to http://localhost:8080.

3.5 Use cases

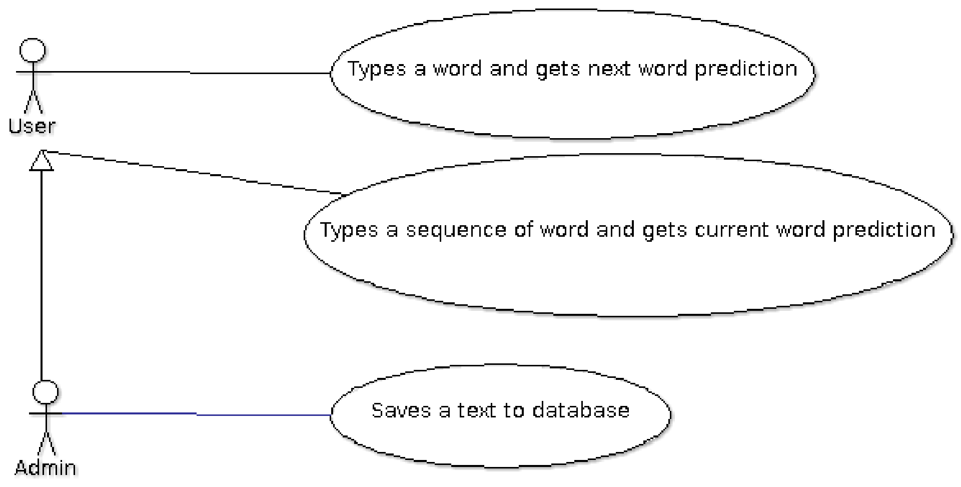
Types a word and gets next word prediction
User
Types a sequence of word and gets current word prediction
Saves a text to database
Admin

Figure 6: Use case diagram

Use case: User types a word and gets next word prediction

Participants: User, Admin

Description: When user types a whole word and presses whitespace, then a selectbox with predicted words should appear. Words in this selectbox should be ordered ascending to their suitability and popularity. User can navigate with arrow keys and select a word with Enter key, after that the whole selected word should be typed into textarea.

Example: User types a phrase “A long time”, presses whitespace, then selectbox with words “ago” and “until” should appear. By default, first word is always selected, so the user presses Enter key and sees the phrase “A long time ago”.

Use case: User types a sequence of word and gets current word prediction

Participants: User, Admin

Description: When user types a part of a word, then a selectbox with predicted words should automatically appear. This should happen after each keypress. This means that words in this selectbox should be ordered ascending to their suitability and popularity. User can navigate with arrow keys and select a word with Enter key, after that the selected word should be typed in textarea.

Example: User types a phrase “accom”, presses whitespace, then selectbox with words “accommodate” and “accommodation” should appear. By default, first word is always selected, so the user presses Enter key and sees the word “accommodate”.

Use case: Administrator saves a text to database

Participants: Admin

Description: Administrator writes, or pastes the text to textbox and presses save button.
Then saving process should start in the background. Then the words should be persisted to the database.

3.6 Architecture model

Application Architecture is designed as a relaxed three-layer architecture [29]. The layers are the following:

1. Presentation layer.
2. Service layer.
3. DAO layer.

Each layer will provide maximum testability and maximize the loose coupling by using the dependency injection frameworks.

However in real life application has 5 layers:

1. Model layer – contains domain objects, which are mapped with Spring data Neo4j ORM.
2. Repository layer – which is user for writing queries, it contains only interfaces, which are derived from Spring data neo4j GraphRepository interface. They allow to generate queries and to write them on you own, by using Cypher Query language.
3. DAO layer – originally, we needed only repository for our three layer architecture, but because of speed requirements and "buginess" of repository layer, I have implemented additional layer, which caches some results and prepares data for repository.
4. Service layer – contains business logic, parses the text and finds the text. In general, it is used for data processing.
5. Controller layer – because of the fact, that I am designing REST service, controller produces JSON only.

Application architecture is shown in the following figure:

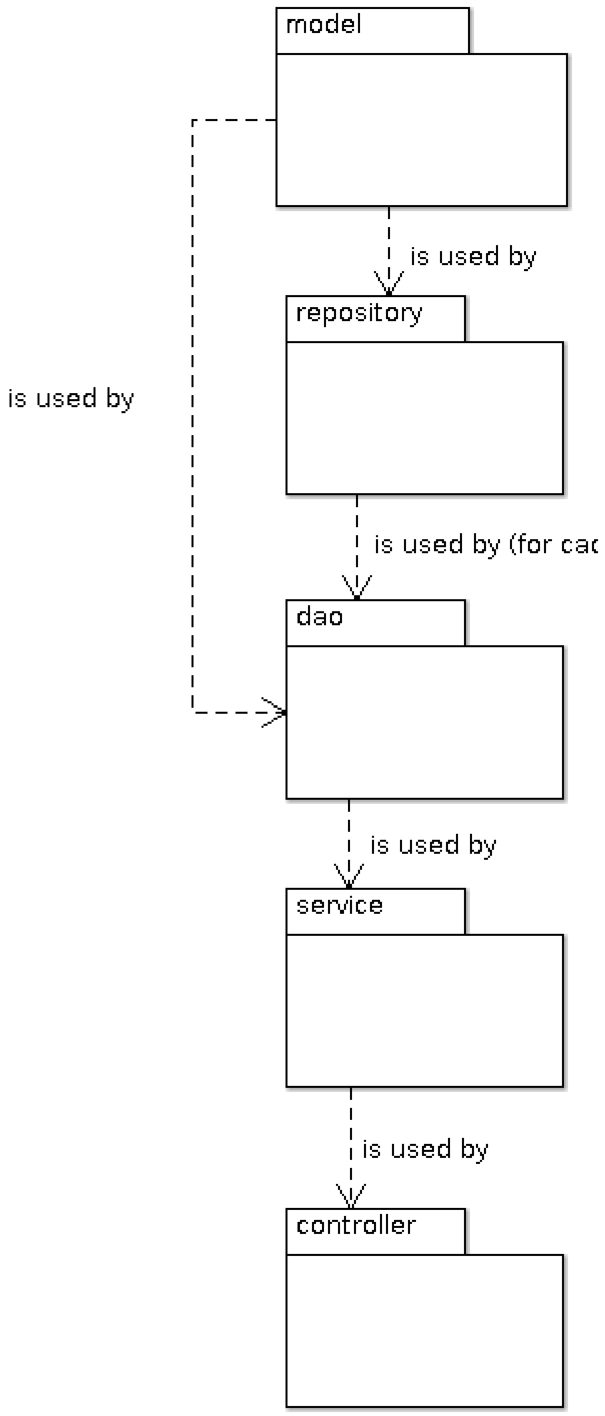

Figure 7: Package diagram

3.6.1 Model layer

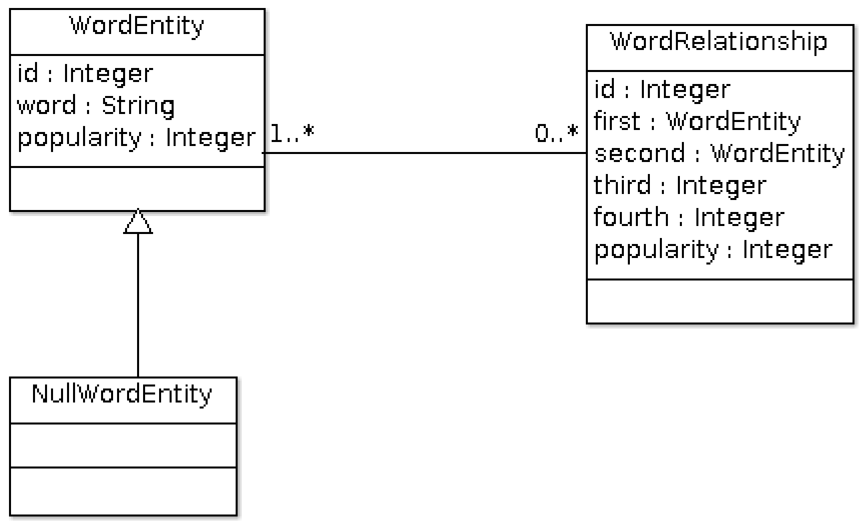

Figure 8: Model layer package

Package contains classes of domain objects which display the response to the user and forward user's requirements to business layer. Those classes are mapped with Spring Data to Neo4j database.

WordEntity

Table 1: WordEntity attributes

Attribute name	**Description**	**Example**
id	Id represents the numeric identifier value in the database, it is primary key.	345
word	The word itself, may contain comma, point, exclamation mark and question mark at the end of the word. Could contain uppercase or lowercase letters.	Really?

popularity	The power or popularity of word field. This means how much this word was found during the text parse or how much the word represented in all persisted texts.	12

NullWordEntity

This class is a WordEntity nullvalue. It is a representation of Null Object pattern.

Table 2: NullWordEntity attributes

Attribute name	Description	Value
id	Same as WordEntity	0
word	Same as WordEntity	null
popularity	Same as WordEntity	0

WordRelationship

Table 3: WordRelationship attributes

Attribute name	Description	Example
id	Id represents the numeric identifier value in the database, it is primary key.	555
first	Represents the starting WordEntity object	
second	Represents the WordEntity object which is followed right after the first WordEntity object.	
third	Represents the WordEntity object id which is followed after the second WordEntity object.	123

fourth	Represents the WordEntity object id which is followed right after the third WordEntity object.	765
popularity	Popularity of this relationship. (How much this combination of words represented in all parsed texts)	2

3.6.2 Repository layer

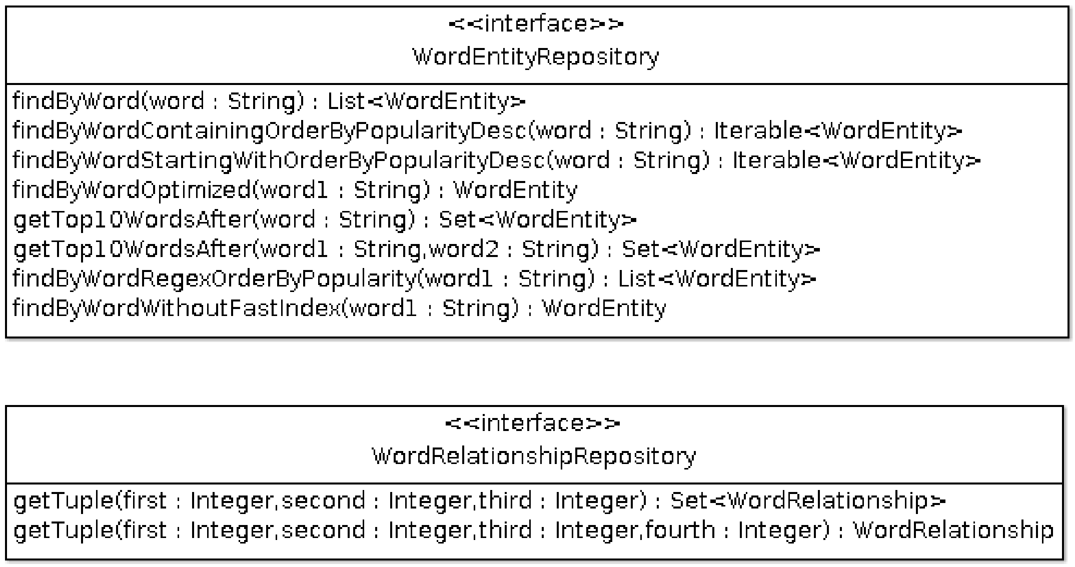
<<interface>>
WordEntityRepository
findByWord(word : String) : List<WordEntity>
findByWordContainingOrderByPopularityDesc(word : String) : Iterable<WordEntity>
findByWordStartingWithOrderByPopularityDesc(word : String) : Iterable<WordEntity>
findByWordOptimized(word1 : String) : WordEntity
getTop10WordsAfter(word : String) : Set<WordEntity>
getTop10WordsAfter(word1 : String,word2 : String) : Set<WordEntity>
findByWordRegexOrderByPopularity(word1 : String) : List<WordEntity>
findByWordWithoutFastIndex(word1 : String) : WordEntity
<<interface>>
WordRelationshipRepository
getTuple(first : Integer,second : Integer,third : Integer) : Set<WordRelationship>
getTuple(first : Integer,second : Integer,third : Integer,fourth : Integer) : WordRelationship

Figure 9: Repository layer package

This layer represents a Spring Data ORM repositories. The idea of Spring Data is to name a functions of interface in an appropriate pattern, so the implementations will be generated automatically.

WordEntityRepository

Table 4: WordEntityRepository methods

Method name	**Notes**	**Parameters**
findByWord() WordRelationship Public	Method returns a list of WordEntities. The list is returned because Spring data neo4j does not respect case of the word. So if you query word “came”, you get also WordEntities with word “Came” and “CAMEL”, for instance.	word

findByWordContaining OrderByPopularityDesc Iterable<WordEntity> Public	Method returns Iterable of WordEntities, which contain a sequence from the word parameter. For instance, if you query a word "large", it will also produce WordEntity with word "enlarge". All returned WordEntitites will be ordered by popularity attribute.	word
findByWordStartingWith OrderByPopularityDesc Iterable<WordEntity> Public	Method returns Iterable of WordEntities, which start from a sequence taken from word parameter. For instance, if you query a word "large", will also produce WordEntity with word "largest". All returned WordEntitites will be ordered by popularity attribute in descending order.	word
findByWordOptimized WordEntity Public	Same as findByWord method, but written in native cypher query.	word1
getTop10WordsAfter Set<WordEntity> Public	Finds a words, which are followed right after the word. Returned set is ordered by popularity. Set consists from 10 elements.	word
getTop10WordsAfter	Finds words, which are	word1,word2

Set<WordEntity> Public	followed right after the word1 and word2. Returned set is ordered by popularity. Set consists from 10 elements.	
findByWordRegexOrderByPopularity List<WordEntity> Public	Method returns a list of WordEntities, where the attribute matches regular expression.	word1
findByWordWithoutFastIndex WordEntity Public	Same as findByWord() method, by works slowly and respects the case of the words.	word1

WordRelationshipRepostitory

Table 5: WordRelationshipRepository method

Method name	**Notes**	**Parameters**
getTuple Set<WordRelationship> Public	Method returns a set of WordRelationships where the word attribute matches first, second and third argument.	first second third
getTuple WordRelationship Public	Method returns aWordRelationship where the word attribute matches first, second, third and fourth argument.	first second third fourth

3.6.3 DAO layer

The idea of DAO layer is to provide persisting functions to service layer. It is needed because of caching, and the fact that repository layer functions contain bugs or unexpected behaviour. For example if you query a word which contains semicolons, then functions, which use indexes will throw an exception.

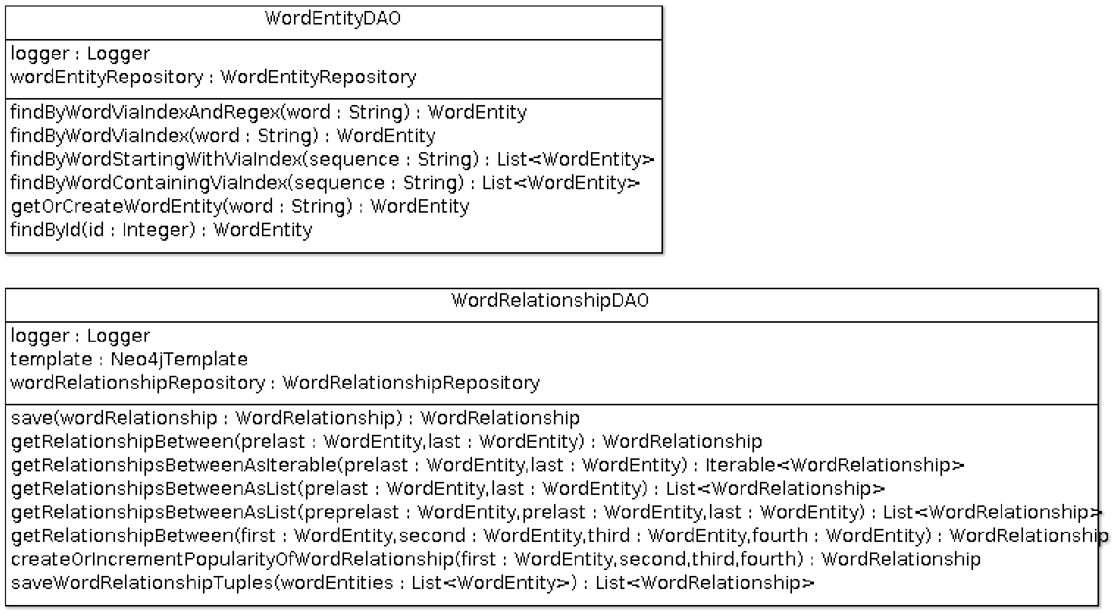
WordEntityDAO
logger : Logger
wordEntityRepository : WordEntityRepository
findByWordViaIndexAndRegex(word : String) : WordEntity
findByWordViaIndex(word : String) : WordEntity
findByWordStartingWithViaIndex(sequence : String) : List<WordEntity>
findByWordContainingViaIndex(sequence : String) : List<WordEntity>
getOrCreateWordEntity(word : String) : WordEntity
findById(id : Integer) : WordEntity
WordRelationshipDAO
logger : Logger
template : Neo4jTemplate
wordRelationshipRepository : WordRelationshipRepository
save(wordRelationship : WordRelationship) : WordRelationship
getRelationshipBetween(prelast : WordEntity,last : WordEntity) : WordRelationship
getRelationshipsBetweenAsIterable(prelast : WordEntity,last : WordEntity) : Iterable<WordRelationship>
getRelationshipsBetweenAsList(prelast : WordEntity,last : WordEntity) : List<WordRelationship>
getRelationshipsBetweenAsList(preprelast : WordEntity,prelast : WordEntity,last : WordEntity) : List<WordRelationship>
getRelationshipBetween(first : WordEntity,second : WordEntity,third : WordEntity,fourth : WordEntity) : WordRelationship
createOrIncrementPopularityOfWordRelationship(first : WordEntity,second,third,fourth) : WordRelationship
saveWordRelationshipTuples(wordEntities : List<WordEntity>) : List<WordRelationship>

Figure 10: DAO layer package

WordEntityDAO

Table 6: WordEntityDAO attributes

Attribute name	**Description**	**Example**
logger	Injected instance of Logger	
wordEntityRepository	Injected instance of WordEntityRepository	

Table 7: WordEntityDAO methods

Method name	**Notes**	**Parameters**

findByWordViaIndexAndRegex WordEntity Public	Method returns WordEntity where the word attribute matches word parameter.	word
findByWordViaIndex WordEntity Public	Method returns WordEntity where the word attribute matches word parameter. Uses indexes, fast, but not stable.	word
findByWordStartingWithViaIndex List<WordEntity> Public	Method returns list of WordEntities, which start from a sequence taken from a sequence parameter. For instance, if you query a word “large”, will also produce WordEntity with word “largest”. It uses indexes, and because of this, the search works almost instantly.	sequence
findByWordContainingViaIndex List<WordEntity> Public	Method returns list of WordEntities, which contain a sequence from sequence parameter. For instance, if you query a word “large”, will also produce WordEntity with word “enlarge”. It uses indexes, and because of this reason, the search works almost instantly.	sequence
getOrCreateWordEntity WordEntity Public	Gets (finds) or creates WordEntity, which has the given word parameter.	word
findById WordEntity Public	Method returns WordEntity, which has the given id parameter.	id

WordRelationshipDAO

Table 8: WordRelationshipDAO attributes

Attribute name	Description	Example
logger	Injected instance of Logger	
template	Injected instance of Neo4jTemplate	
wordEntityRepository	Injected instance of WordEntityRepository	

Table 9: WordRelationshipDAO methods

Method name	Notes	Parameters
save WordRelationship Public	Method persists the given WordRelationship	wordRelationship
getRelationshipsBetweenAsIterable Iterable<WordRelationship> Public	Method returns Iterable of WordRelationships, where the first WordEntity contains prelast as a word attribute, and the second contains word last as a word attribute.	prelast last
getRelationshipsBetweenAsList List<WordRelationship> Public	Same as getRelationshipsBetween AsIterable, but returns list with initialized values.	prelast last
getRelationshipsBetweenAsListWord Entity List<WordRelationship> Public	Method returns list of WordRelationships, where the first WordEntity contains preprelast as a word attribute, and the second contains word prelast as a word attribute.	preprelast prelast last

	The last WordEntity contains last parameter as a word attribute.	
getRelationshipBetween WordRelationship Public	Method returns WordRelationship, where the first WordEntity contains first as a word attribute, the second contains word second as a word attribute, the third contains word third as a word attribute, and the last WordEntity contains fourth as a word attribute	first second third fourth
createOrIncrementPopularityOfWord Relationship WordRelationship Public	Creates a relationship between the words, or increments the popularity of relationship and persists it to database.	first second third fourth
saveWordRelationshipTuples List<WordRelationship> Public	Accepts a list of wordentities (which are just generated from sentences) and persists or increments popularity of each created WordRelationship to database. Relationship will be saved according to order of WordEntities in list. Normally this order is similar to sentence.	wordEntities

3.6.4 Service layer

This layer is responsible for all business logic like saving, parsing the text, splitting the whole text to sentences to words, generating WordEntities from those words etc.

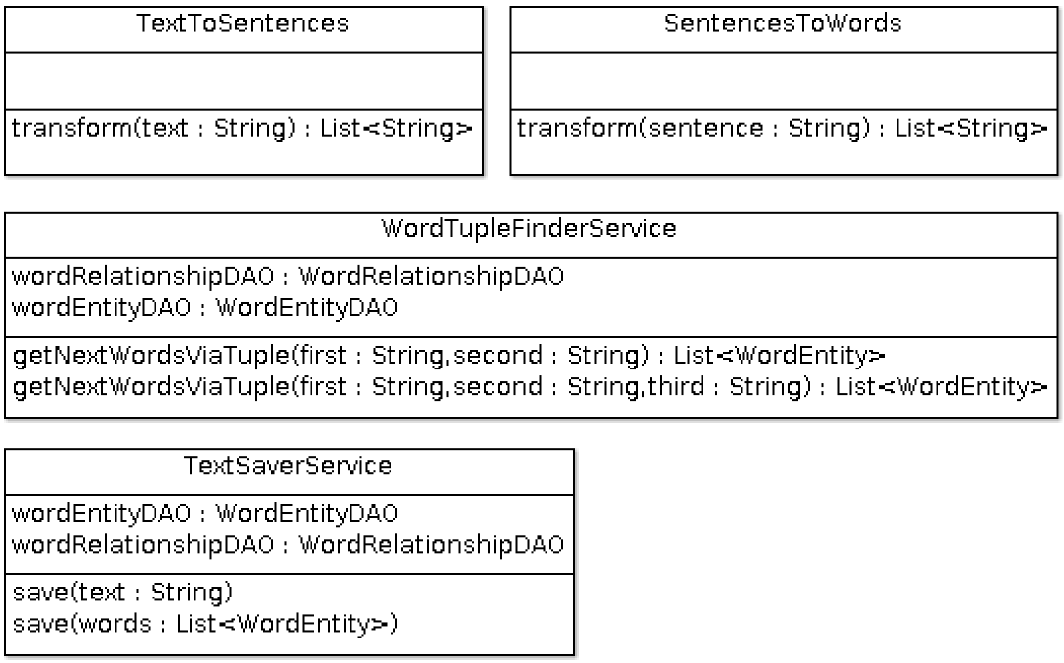
TextToSentences
transform(text : String) : List<String>
SentencesToWords
transform(sentence : String) : List<String>
WordTupleFinderService
wordRelationshipDAO : WordRelationshipDAO
wordEntityDAO : WordEntityDAO
getNextWordsViaTuple(first : String,second : String) : List<WordEntity>
getNextWordsViaTuple(first : String,second : String,third : String) : List<WordEntity>
TextSaverService
wordEntityDAO : WordEntityDAO
wordRelationshipDAO : WordRelationshipDAO
save(text : String)
save(words : List<WordEntity>)

Figure 11: Service layer package

TextToSentences

Table 10: TextToSentences methods

Method name	**Notes**	**Parameters**
transform List<String> Public	Method splits text into list of sentences.	text

SentencesToWords

Table 11: SentencesToWords methods

Method name	**Notes**	**Parameters**
transform List<String> Public	Method splits sentence into list of sentences.	sentence

WordTupleFinderService

Table 12: WordTupleFinderService attributes

Attribute name	**Description**	**Example**
wordRelationshipDAO	Injected instance of WordRelationshipDAO	
wordEntityDAO	Injected instance of WordEntityDAO	

Table 13: WordTupleFinderService methods

Method name	**Notes**	**Parameters**
getNextWordsViaTupleList List<WordEntity> Public	Method finds a list of WordEntities, where the first and second parameter represent a first and second WordEntity word attribute. If some word is missing, then the method returns empty list.	first second
getNextWordsViaTupleList List<WordEntity> Public	Method finds a list of WordEntities, where the first, second and third parameter represents first, second and third WordEntity word attribute. If some word is missing, then the method returns empty list.	first second third

TextSaverService

Table 14: TextSaverService attributes

Attribute name	**Description**	**Example**
wordRelationshipDAO	Injected instance of WordRelationshipDAO	
wordEntityDAO	Injected instance of WordEntityDAO	

Table 15: TextSaverService methods

Method name	**Notes**	**Parameters**
save void Public	Method transforms text to WordEntities and WordRelationships and saves them to database.	text
save void Public	Method saves WordEntities and their WordRelationships to database.	words

3.6.5 Controller layer

This layer is responsible for forwarding user requests to service layer and producing responses. All controllers produce JSON only.

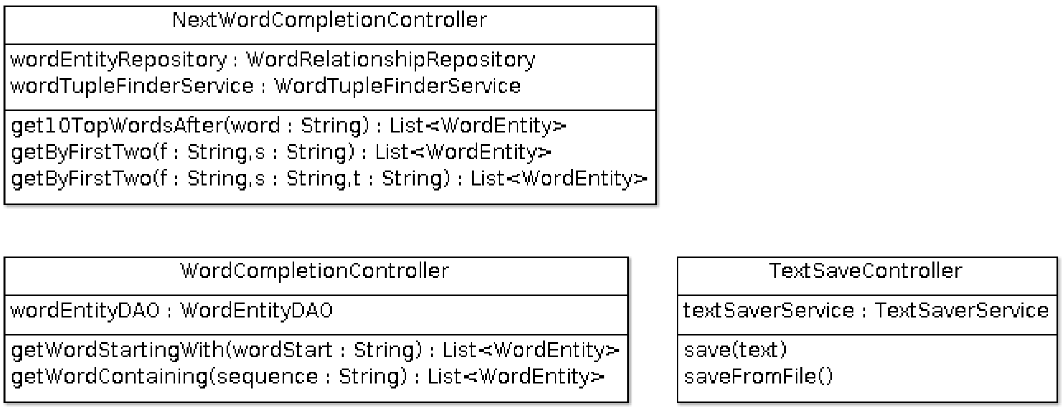
NextWordCompletionController
wordEntityRepository : WordRelationshipRepository
wordTupleFinderService : WordTupleFinderService
get10TopWordsAfter(word : String) : List<WordEntity>
getByFirstTwo(f : String,s : String) : List<WordEntity>
getByFirstTwo(f : String,s : String,t : String) : List<WordEntity>
WordCompletionController
wordEntityDAO : WordEntityDAO
getWordStartingWith(wordStart : String) : List<WordEntity>
getWordContaining(sequence : String) : List<WordEntity>
TextSaveController
textSaverService : TextSaverService
save(text)
saveFromFile()

Figure 12: Controller layer package

NextWordCompletionController

Table 16: NextWordCompletionController attributes

Attribute name	**Description**	**Example**
wordEntityRepository	Injected instance of WordEntityRepository	
wordTupleRepository	Injected instance of WordTupleRepository	

Table 17: NextWordCompletionController methods

Method name	**Notes**	**Parameters**
get10TopWordsAfter List<WordEntity> Public	Method get 10 top word results after the following word	word
getByFirstTwo List<WordEntity>	Method get 10 top word results after the following f and s	f s

Public	parameter.	
getByFirstTwo List<WordEntity> Public	Method get 10 top word results after the following f, s and t parameter.	f s t

WordCompletionController

Table 18: WordCompletionController attributes

Attribute name	**Description**	**Example**
wordEntityDAO	Injected instance of wordEntityDAO	

Table 19: WordCompletionController methods

Method name	**Notes**	**Parameters**
getWordStartingWith List<WordEntity> Public	Method gets 10 top WordEntity suggestions, where the word attribute starts with wordstart parameter. Example: you type “en”. Then you should get results starting with “en” like “enlarge”, “enforce”, “environment”.	wordStart
getWordContaining List<WordEntity> Public	Method gets 10 top WordEntity suggestions, where the word attribute contains a sequence parameter. Example: you type “mis”. Then you should get results “missile”, “transmission”.	sequence

TextSaveController

Table 20: TextSaveController attributes

Attribute name	**Description**	**Example**
textSaverService	Injected instance of **TextSaverService**	

Table 21: TextSaveController methods

Method name	**Notes**	**Parameters**
save void Public	Method intercepts text parameter and saves it to database.	text
save void Public	Method takes the hardcoded txt file and saves it to database.	

4. Conclusion

The main goal of the diploma was the implementing and designing the REST service which could predict next word depending on the previous given input.

After implementing the software, I could say that the primary goal was achieved. Software works as I expected. After measuring the response time the results varied from 20ms to 500ms. On average, the response time was more like 40ms. This result was achieved for 1.3 megabyte book with 230000+ words inside. To be honest, I was a little bit disappointed with the results, because sometimes next word suggestion takes 300ms, which is more or less okay, but definitely not instant.

As a result, I may say that it is possible to create this kind of service, but graph database is not the best choice for this goal, because at the end of the design it looks more like implementing key value store over the graph database. In addition, the performance results are not superb.

To improve the performance of my service, I could cache the repository layer or change the database to relational or key-value store. This may be the goal for the future.

After implementing the software the following goals achieved:

- The working REST suggestion service.
- The software almost fully covered with tests. (80% of code is covered with tests)
- The knowledge how to design those kinds of services.
- Fully documented process of implementing the design.
- The answer to the question: "Does the graph database suit for this goal?" – The answer is: "It depends.". However, I would suggest you to think twice before taking the graph database for the same goal. In next paragraph, I will describe why.
- The question about the usefulness of graphs in linguistics is left open. It is too wide for me to give evaluation.

Despite the fact, that the service works under the stress, performs queries etc., I realized that I ended with model, where I put too much meta-info to relationships between words. Because of the optimization, the relationship contains also info about the third and fourth word, not only connecting first and second. This is the main flaw of the design. Maybe, the better idea would be switching back to relational database. One of the proposed relational database design models could be found in Appendix 1.

Kokkuvõte

Käesoleva lõputöö põhieesmärk oli näidata kuidas arendatakse teksti sisestamise soovitussüsteemi REST teenuste ja graafandmebaasi toel, mis võimaldab ennustada järgmist sõna, sõltuvalt eelnevalt sisestatud sõnadest.

Käesoleva töö peamine tulemus on dokumentatsioon kuidas disainiti ja arendati rakenduse, ja rakendus ise.

Antud töö kirjeldab ja seletab kuidas teksti sisestamise soovitussüsteem võib olla tehtud, kui te kasutate graafandmebaasi. Lisaks eelnevalt mainitule, töö seletab ja analüüsib antud implementatsiooni ja annab talle hinnangu.

Summary

The main goal of the diploma was to show how to implement and design REST service which could predict next word depending on the previously given words.

Main result of this work is fully documented process of designing and implementation the application, and application itself.

This work describes and explains how the word suggestion service could be built, if you use graph database. In addition to above mentioned, the work describes and analyses cons and pros of our implementation and gives evaluation to it.

References

[1] https://developers.google.com/web-search/docs/

[2] http://lucene.apache.org/core/

http://www.lucenetutorial.com/lucene-in-5-minutes.html

[3] http://www.iqubemarketing.com/glossary-big-data-terminolgy/

[4] http://neo4j.com/product/

[5] Fielding, Roy T. (2000). "Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.

[6] http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

[7] “Principles Of OOD”, Robert C. Martin (“Uncle BOB”), butunclebob.com. (Note the reference to “the first five principles”, though the acronym is not used in this article.)

http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

[8] Beck, K. Test-Driven Development by Example, Addison Wesley - Vaseem, 2003

Koskela, L. "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007

[9] Introductory graph theory. New York: Dover. ISBN 0-486-24775-9.

[10] Introductory graph theory. New York: Dover. ISBN 0-486-24775-9.

[11] https://www.youtube.com/watch?v=IozhMc6lPTU&feature=youtube_gdata

[12] Ahmed Sabbir Arif, Wolfgang Stuerzlinger Analysis of Text Entry Performance Metrics Dept. of Computer Science & Engineering York University

[13] http://www.nngroup.com/articles/response-times-3-important-limits/

[14]http://www.oxforddictionaries.com/words/how-many-words-are-there-in-the-english-language

[15] http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=yarv&lang2=java

[16] https://docs.python.org/2/tutorial/

[17] https://www.ruby-lang.org/en/about/

[18] http://php.net/manual/en/intro-whatis.php

[19] http://neo4j.com/developer/language-guides/

[20]http://docs.spring.io/spring/docs/4.2.0.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#spring-introduction

[21] http://projects.spring.io/spring-boot/

[22] http://projects.spring.io/spring-data-neo4j/

[23] https://docs.angularjs.org/guide/introduction

[24] http://learn.jquery.com/javascript-101/getting-started/

[25] http://www.slideshare.net/jmusser/open-apis-state-of-the-market-2011

[26] Kent Beck, Erich Gamma. JUnit Cookbook.

[27] http://www.oracle.com/technetwork/java/javase/jdbc/index.html#corespec40

[28] http://en.wikipedia.org/wiki/Web_container

[29] http://msdn.microsoft.com/en-us/library/ff648105.aspx

Source code

Source code is available at: https://github.com/ilja903/wordservice

Appendix 1

Alternative relational database model:

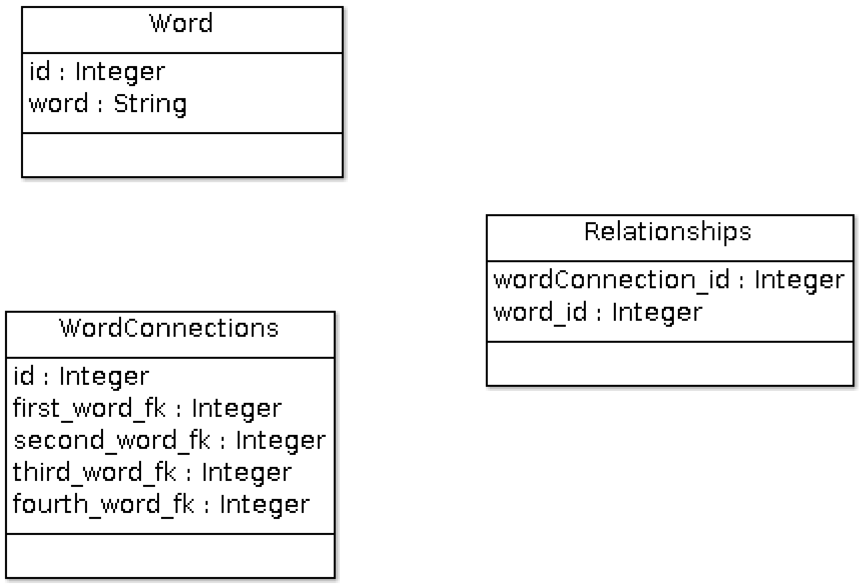

Figure 13: Alternative model in relational database

Word

Table 22: Word attributes

Attribute name	**Description**	**Example**
id	Primary key	1
word	Word of a word table	“Someword”

WordConnection

Table 23: WordConnection attributes

Attribute name	**Description**	**Example**
id	Primary key	1

first_word_fk	Foreign key to the first ford in Word table	1
second_word_fk	Foreign key to the second ford in Word table	2
third_word_fk	Foreign key to the third ford in Word table	3
fourth_word_fk	Foreign key to the fourth ford in Word table	4

Relationship

Table 24: Relationship attributes

Attribute name	**Description**	**Example**
wordConnection_id	Foreign key to the wordConnection table	1
word_id	Foreign key to the word, which comes right after the wordConnection	“Someword”
popularity	Number of how much this relationship was found during the text save.	123