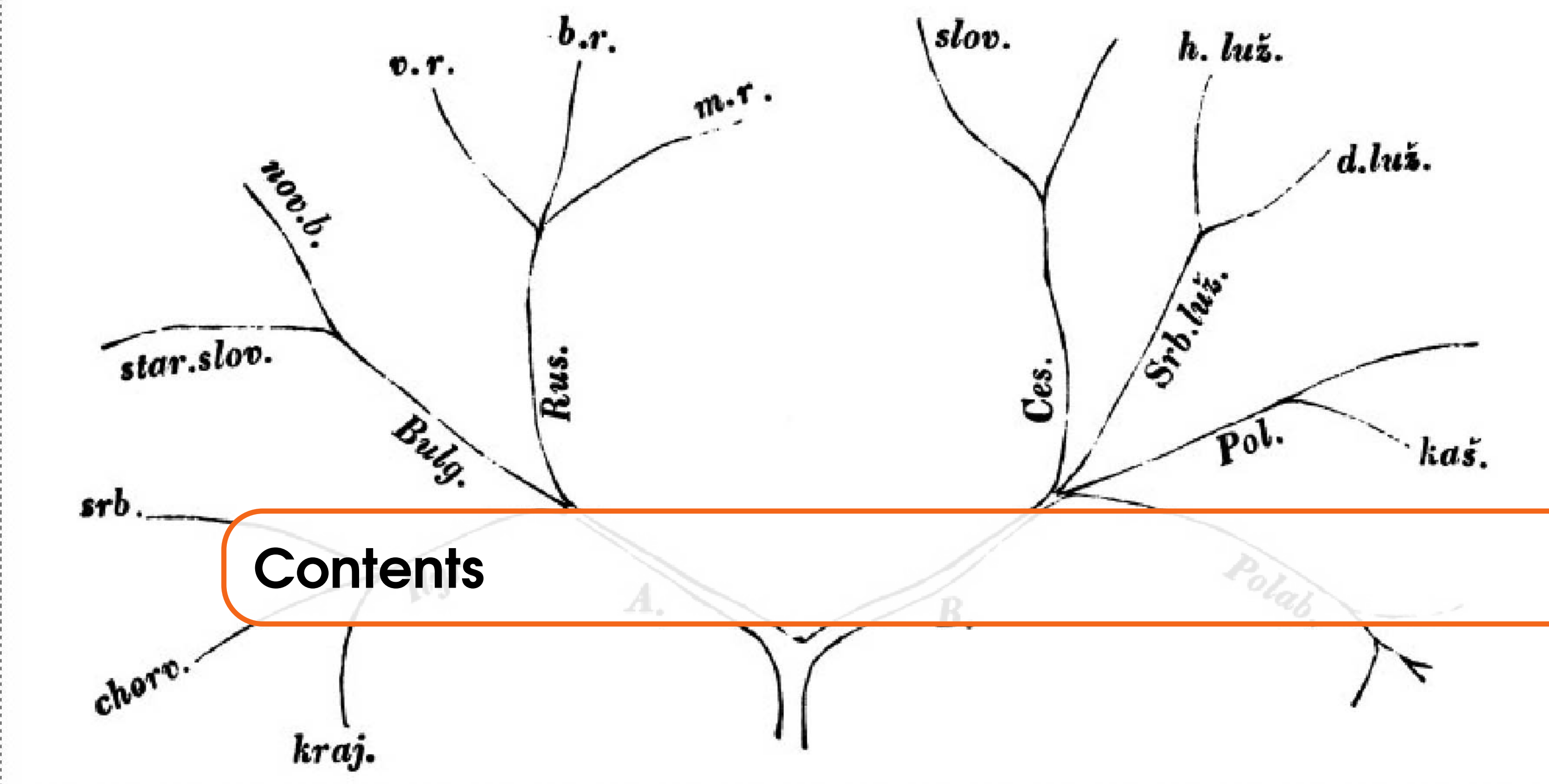
Stemmarest

Documentation of the PSE2 Project 2015

Ido Gershoni, Ramona Imhof, Joel Niklaus, Jakob Schaerer, Severin Zumbrunn

HTTPS://GITHUB.COM/TOHOTFORICE/PSE2_DH

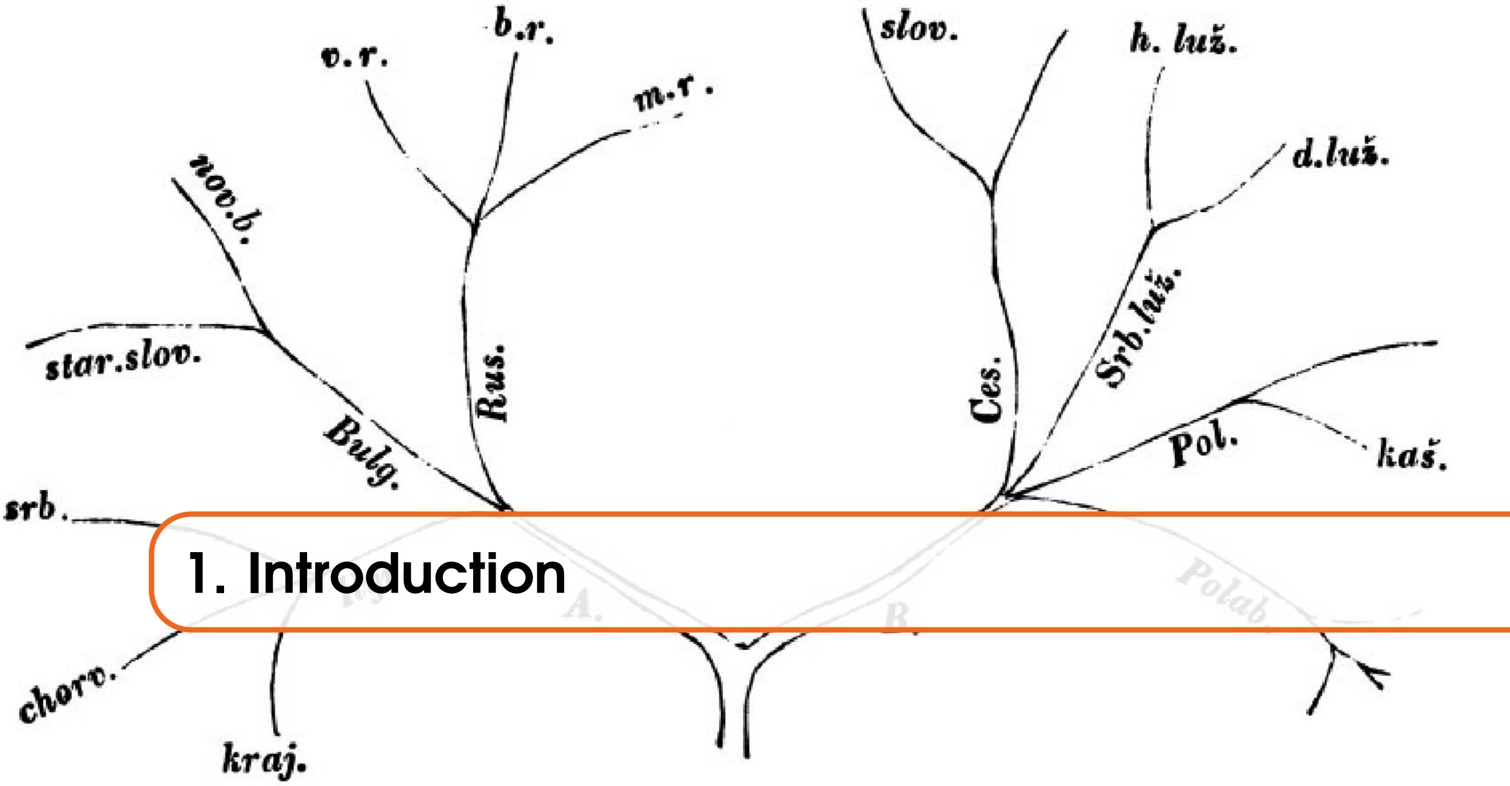
First printing, Mai 2015


b.r.
v.r.
m.r.
slov.
h. luž.
d.luž.
nov.b.
star.slov.
Rus.
Ces.
Srb.luž.
Bulg.
Pol.
kaš.
srb.
chorv.
kraj.

Contents

III RESTful API

Project



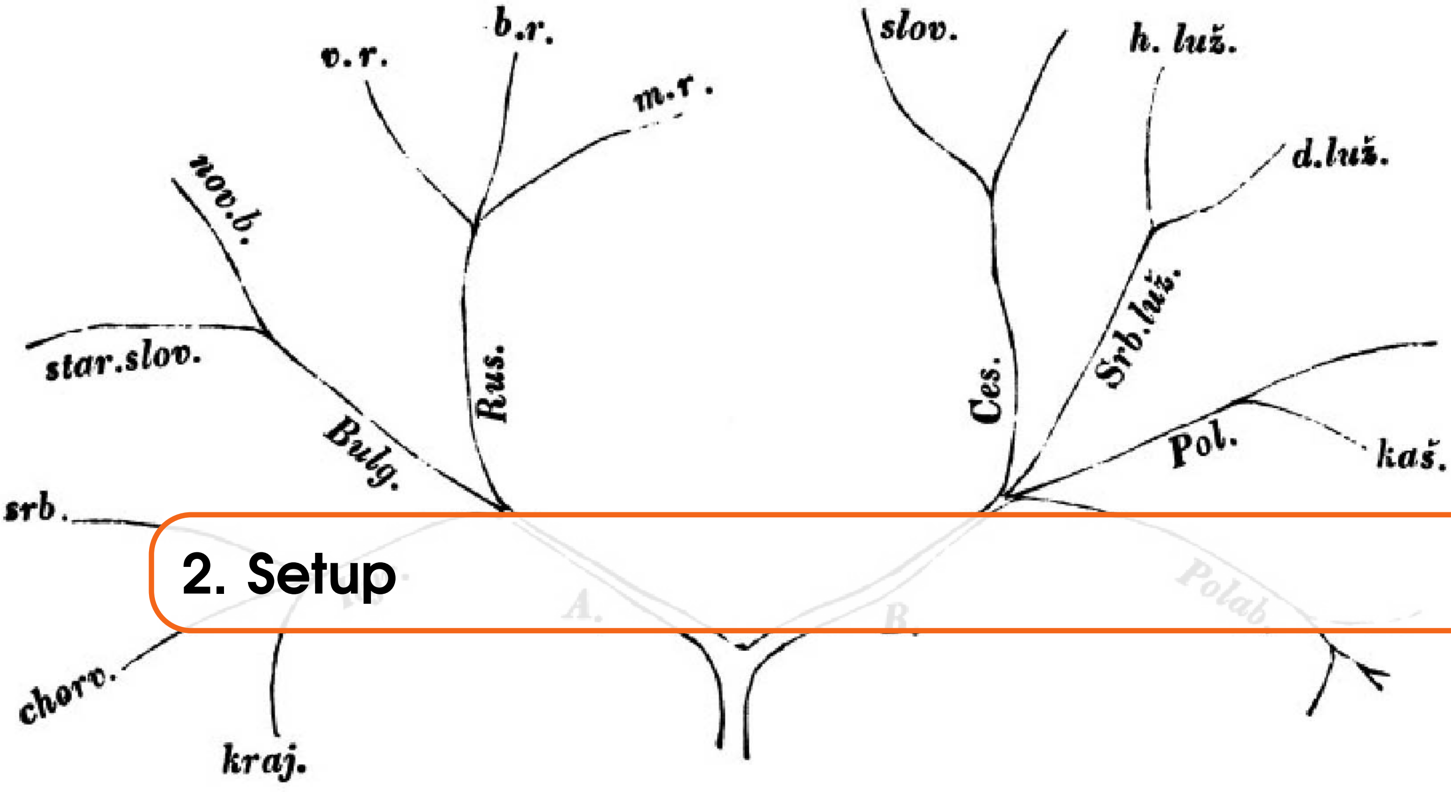
1. Introduction

We are a team of 5 students from the University of Bern. In the course of the Practical Software Engineering lecture we have been assigned this digital humanities software development project. The goal of this lecture is to experience the development process of a software while working in a team.

The development process included the collection of the requirements through interviews with the customer, implementing the user-stories using agile programming techniques, constant dialog with the customer to ensure the software fulfills her requirements and finally delivering the project to the costumer so it can be put into use.

Our customers goal is to create a tool to analyze old texts by comparing different versions of them. The prototype of this web tool, Stemmaweb, focused on functionality and not on performance, which led to very slow loading times due to the complexity of the different connections between elements in different texts. We were asked to evaluate and develop a more efficient system which would significantly improve the performance of Stemmaweb, purposely by using a graph database over a standard relational database.

To achieve this we used the graph database Neo4. We programmed the software in the programming language Java and worked with Jersey, a Java RESTful framework. In four iterations during 12 weeks we implemented the user-stories given to us and defined a unique API call for each implemented function. This will allow the customer to easily connect her existing Graphical User Interface to the new software we have created.



2. Setup

Downloading

git clone https://github.com/tohotforice/PSE2_DH.git

Building

Stemmarest needs to be built using Maven. This can be done using a java IDE (e.g Eclipse) and a Maven plugin

Running

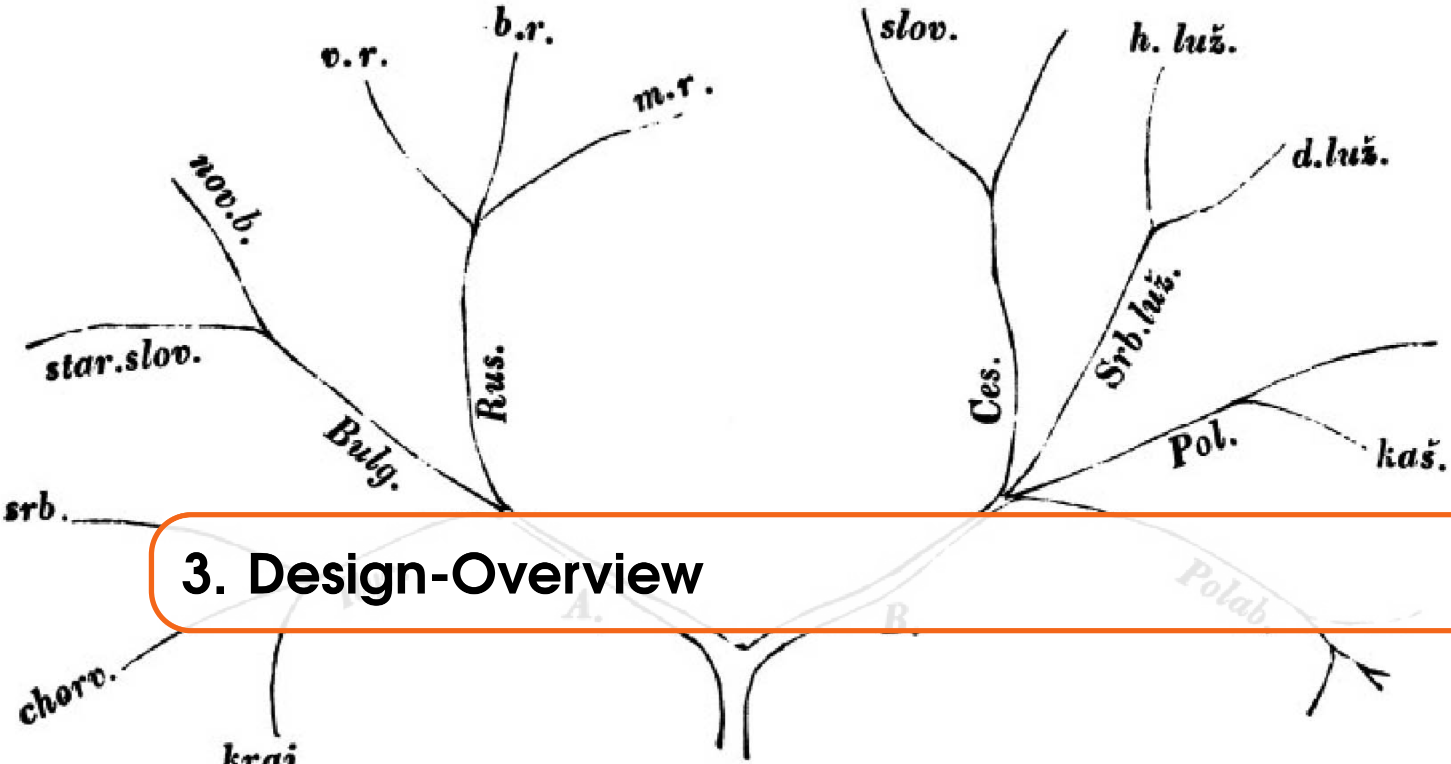
As this application represents a server side only, there is no full GUI included. It is possible though to test it by using the test interface testGui.html which is located at StemmaClient.

Using the test interface

- Create a user and give it an id (this is necessary as every graph needs to be owned by a user)
- Import an GraphML file using the id of the user you have created. The generated id of the tradition will be returned
- Use the custom request by typing in the API call you want (all calls are listed in the documentation)

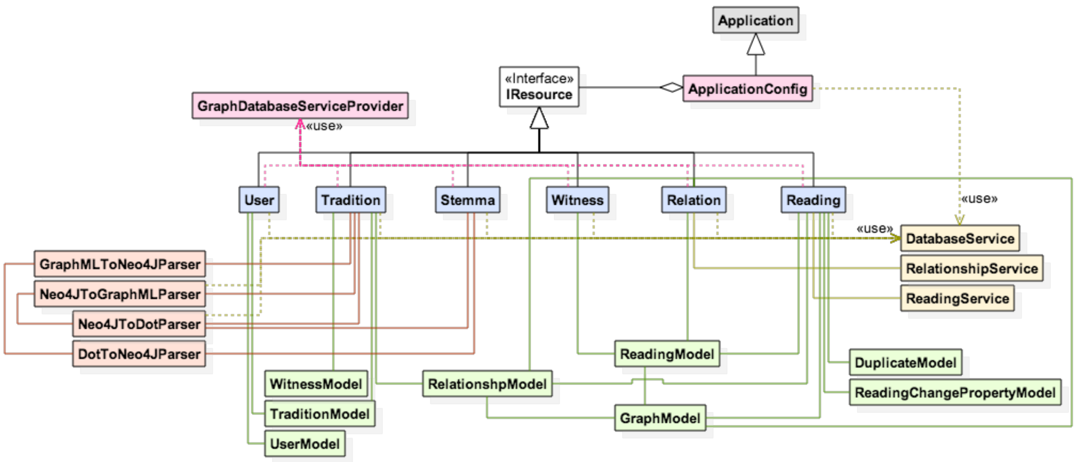
A word about node id's: when a graph is being imported each node gets from Neo4j a unique id-number. In order to use an id in an API call (e.g. reading-id) it is necessary to explicitly get it from the data base. This can be done by using the getAllReadings method (getallreadings/fromtradition/traditionId) or by actually going into the data base

More information in the README file on GitHib: *https://github.com/tohotforice/PSE2_DH*



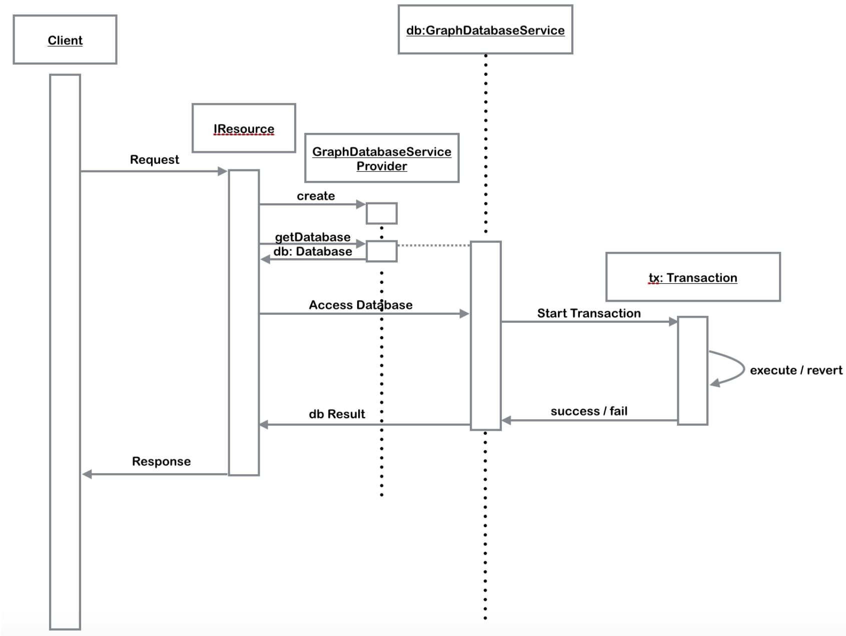
3. Design-Overview

Figure 3.1: Class-Overview



This overview shows the relation between the classes of the StemmaREST service. As StemmaREST is based on the Jersey Framework all the provided resources need to be registered in this framework. This is done in the ApplicationConfig class, like it is specified in the chapter Jersey. All classes which should be registered as a 'resource' need to implement the IResource interface. All IResources which need access to the database use the GraphDatabaseServiceProvider. The GraphDatabaseServiceProvider contains a static GraphDatabaseServiceObject which can be requested by the IResource and used to acces the database.
Several IResources need the service classes to share common functionality.
The model classes are dataclasses which contain the datamodels. They are also used for the serialisation and deserialisation of xml and JSON strings. For this serialisation the jackson package is used.
The parser classes are used to import graphml and dot files into the database.

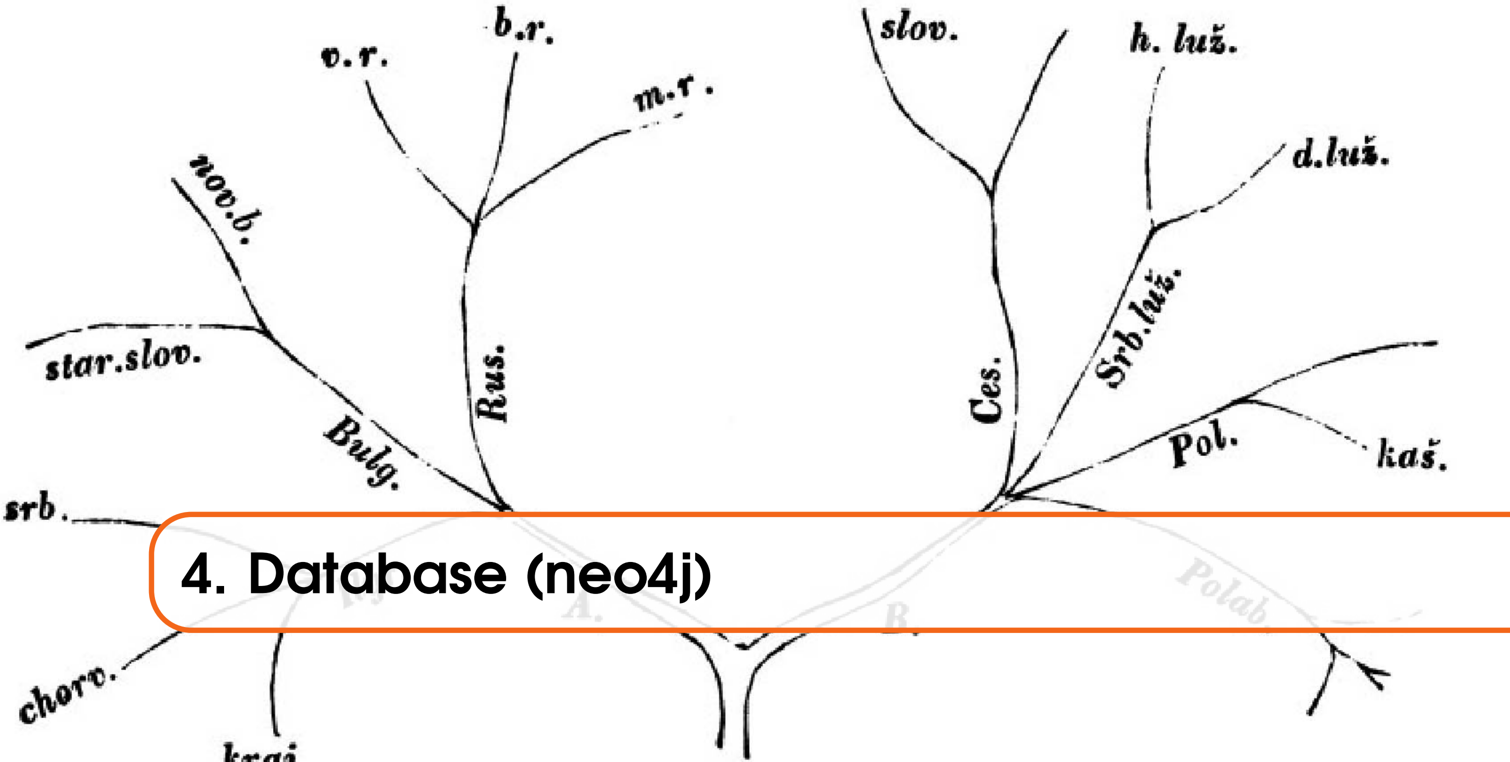
Figure 3.2: Request Sequence



When a client sends a request to the stemmaREST service. Jersey instantiates the requested IResource. GraphDatabaseServiceProvided is instantiated to request the singleton database. All requests require access to the database. The transaction is executed in the manner the request requires and a response is sent to the client.

More information about the database design and the Jersey integration can be found in the chapters Database and Jersey.

A more detailed Class-Overview can be found: `https://github.com/tohotforice/PSE2_DH/blob/master/Dokumentation/DetailedClassOverview.svg`



4. Database (neo4j)

In this project a graph database is used to store the data. This was done as graph databases are much faster when it comes down to look for objects in a list that contain some constraints to other objects. In a relational database one would normally use multiple joins to get the desired result, but by using a graph this task becomes much easier since it is possible to traverse the graph from node to node using either breadth- or depth-first algorithms for a specific relation between nodes. This method makes a search for nodes, representing objects, which are connected to each other, very efficient.

Neo4J was chosen(more information can be found at http://neo4j.com/developer/graph-db-vs-rdbms/), which is a graph database capable of very efficiently managing nodes and relationship even in a large scale graph.
Additional information regarding performance could be found in the related chapter of this documentation.

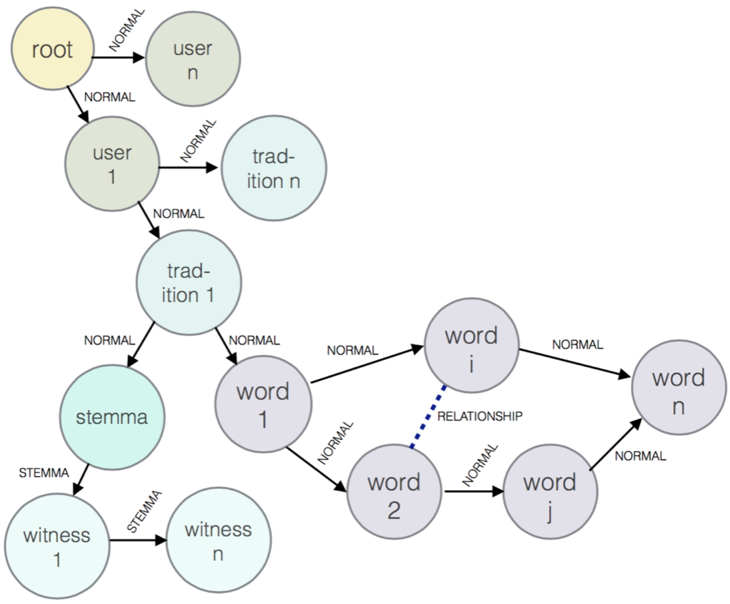
The stemmaweb database is basically a one big graph, with different labels marking different nodes and relationships.

Those labels are used in the database:

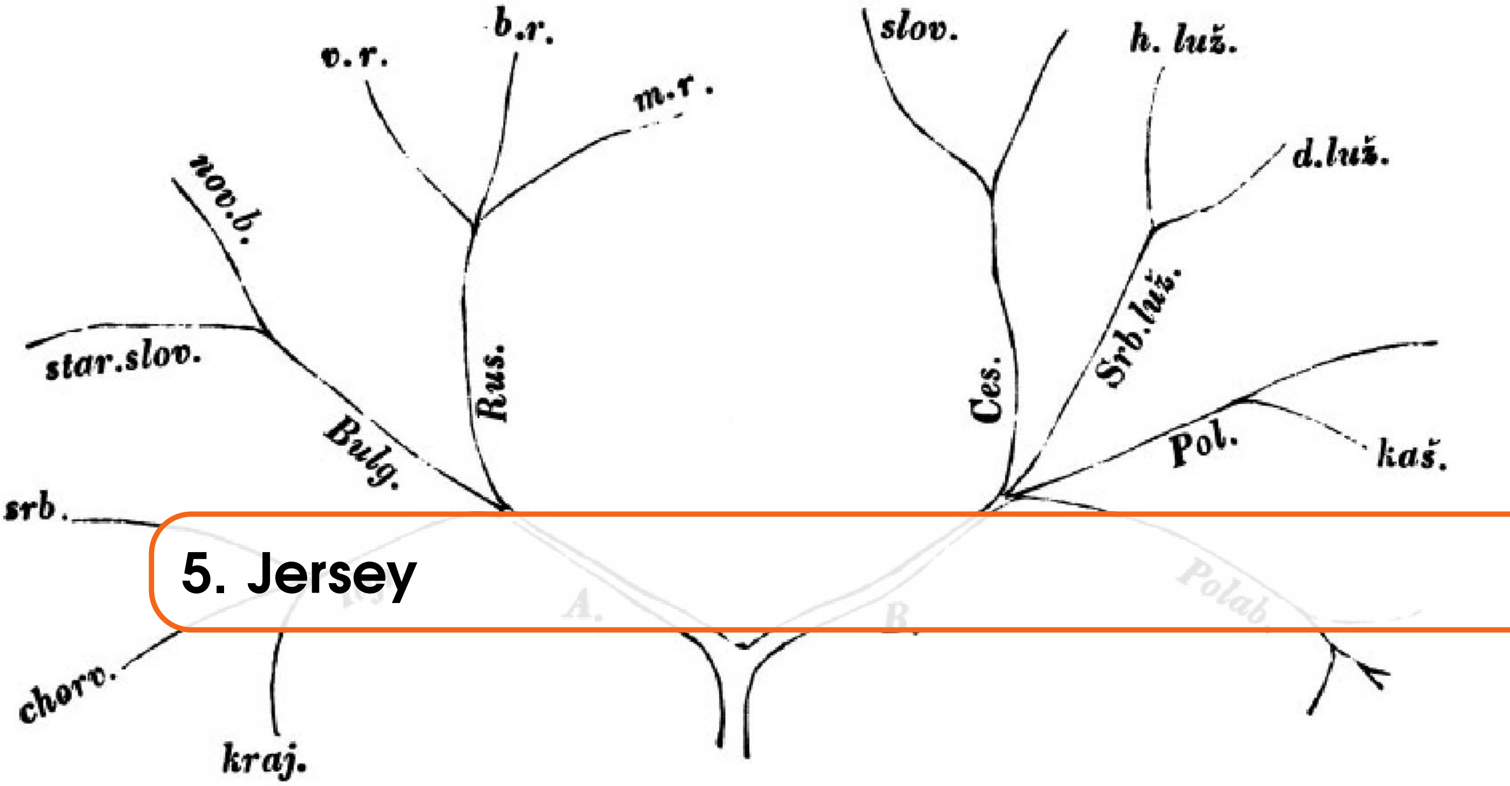
Nodes	Relationships
ROOT	RELATIONSHIP
STEMMA	STEMMA
WITNESS	NORMAL
TRADITION	
USER	
WORD	

Since each label is stored in another file, searching or traversing the graph is highly efficient.

The database structure is as follows:



Neo4J uses a script language called cypher. Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of the graph store. Cypher queries, though, need to be interpreted and translated into an execution plan. This is the reason why they are not always as fast as the native java traversal API, which has therefore become the common query tool used in the project.



5. Jersey

Introduction

Jersey is an open source java framework for developing RESTful Web Services in Java that is built upon JAX-RS and serves as a JAX-RS Reference Implementation. It adds additional features and utilities in order to further simplify development of REST-APIs. Jersey helps support exposing the data in different media types, including JSON, which is very frequently used in this project.

Method Declarations

An example of the method declaration of duplicateReading in the reading class:

```
@POST
@Path("duplicatereading")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response duplicateReading(DuplicateModel duplicateModel)
```

The @POST annotation states the http method.
The @Path annotation sets the url path.
The method "consumes" (i.e. gets from the client side) data sent by the client, in this case a java object of a "DuplicateModel" which is passed with the call as a JSON object and then gets parsed by the server into a POJO. The method "produces" (returns) a response, which is the method's return value, in this case also in JSON.

Another example from the witness class:

```
@GET
@Path("gettext/fromtradition/{tradId}/ofwitness/{witnessId}")
@Produces(MediaType.APPLICATION_JSON)
public Response getWitnessAsText(@PathParam("tradId") String tradId,
        @PathParam("witnessId") String witnessId) {
```

The values in curly braces in the path (tradId and witnessId) are path parameters, which are used in the method. In this example they are given in the URL and not as JSON. In the method declaration they are annotated with @PathParam.

IResources

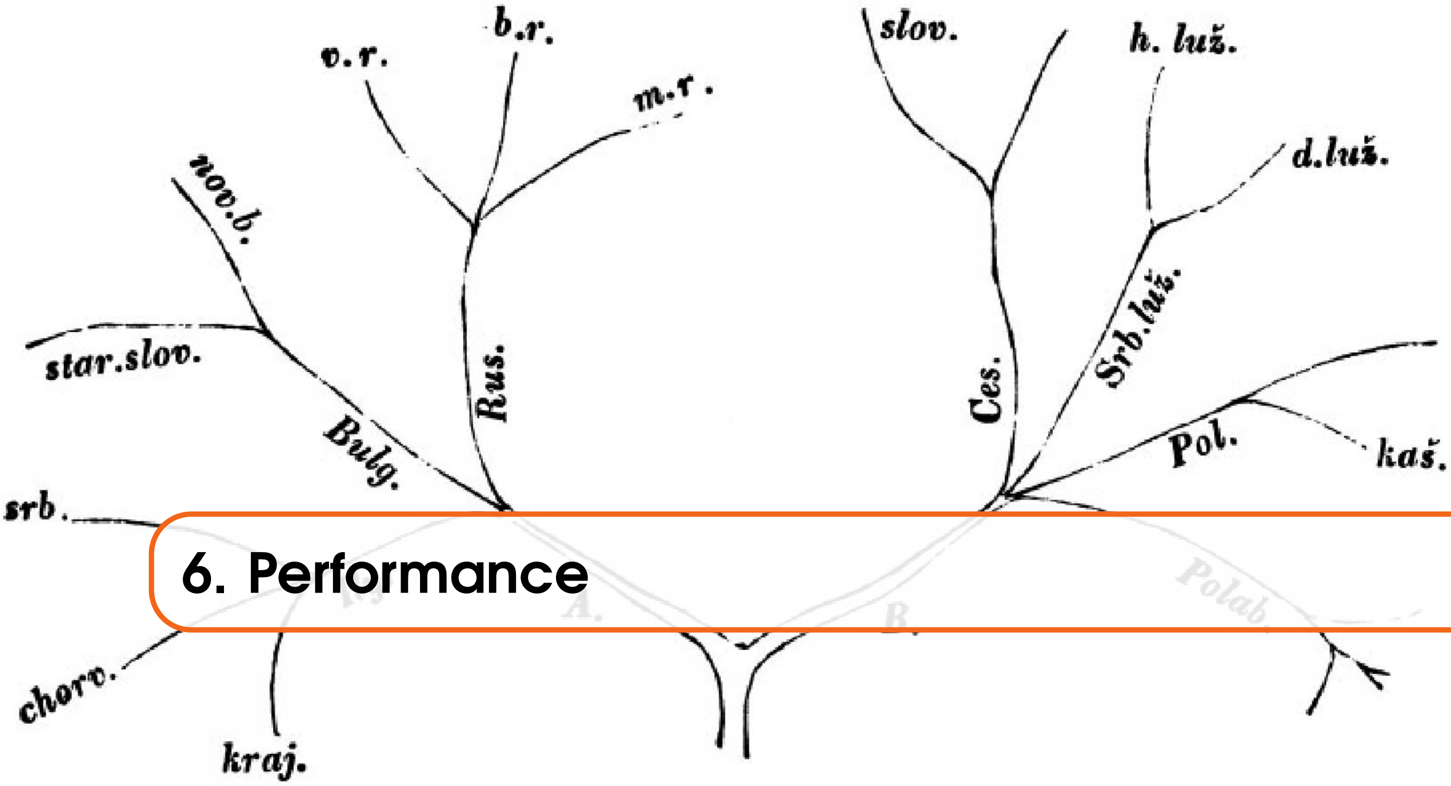
In the ApplicationConfig class all the IResources (the objects) are loaded in the following method:

```
@Override
public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
        s.add(Witness.class);
        s.add(User.class);
        s.add(Tradition.class);
        s.add(Relation.class);
        s.add(Stemma.class);
        s.add(Reading.class);

        return s;
}
```

An 'IResource' is a class that provides methods annotated with @GET, @POST, @PUT or @DELETE. All the API calls that can be made using stemmarest are defined in those six classes: Witness, User, Tradition, Relation, Stemma and Reading.

```
@Path("/reading")
public class Reading implements IResource {
```



6. Performance

One of the main goals of this project was to create a RESTful service which is significantly faster then the existing one. To verify the speed of the service some performance tests are done.

The goal of the performance tests is to show that the response time of the service is limited and within a usable range. The performance (benchmark) tests therefore measure the time needed to execute all operations for a certain request. This includes the time to transmit the data over HTTP, the time to execute the internal algorithms and the time to access the database. All the Data is transmitted over the local loop interface. The network speed is therefore not measured.

For the purpose of the tests the database is being populated by a random graph which contains several valid traditions on which the REST requests can be executed. Several tests with databases of different sizes are done to show that the response time does not change as the size of the database increases.

The first set of diagrams show the result of tests with different database sizes. Those tests show that the RESTful service response time is not influenced by the size of the database in a significant way. This is related to the use of the Graphdatabase in which a query can search a subgraph without filtering the whole database.

R The implementation of stemmarest uses some search-node-by-id methods (a part of Neo4j framework) which search over the complete database. It is important to realize that those quarries are done in $O(logn)$ time and are not seen in the noise of the other operations during the tests. This can be seen in the diagrams in such methods as *getReading*, *getNextReading* etc, which use the search-node-by-id method and their execution time hardly change even in a very big data base. In a much larger Database those methods will slow down the REST requests, though it is not expected that the database will grow so big that such operations will have any impact.

In those diagrams it is possible to see that the response time is almost independent to the database size between 1000 and 1 Million Nodes (Readings). As explained before, this is the result

Figure 6.1: Database with 1000 nodes, working tradition with 100 nodes

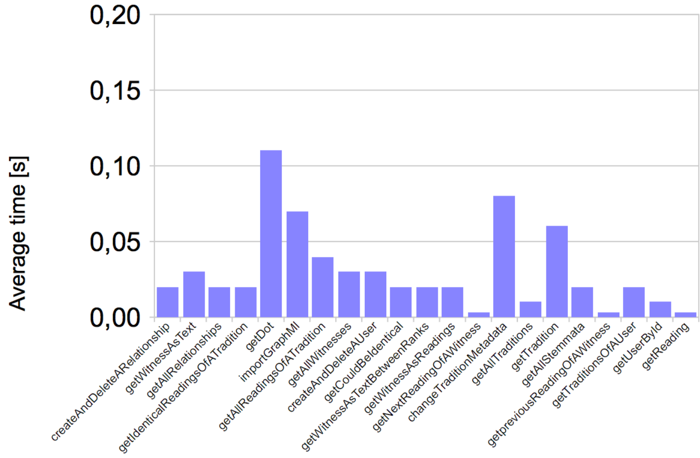


Figure 6.2: Database with 100000 nodes, working tradition with 100 nodes

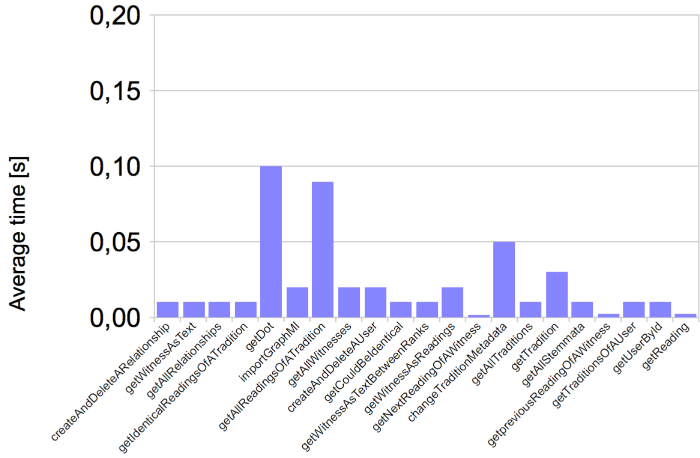
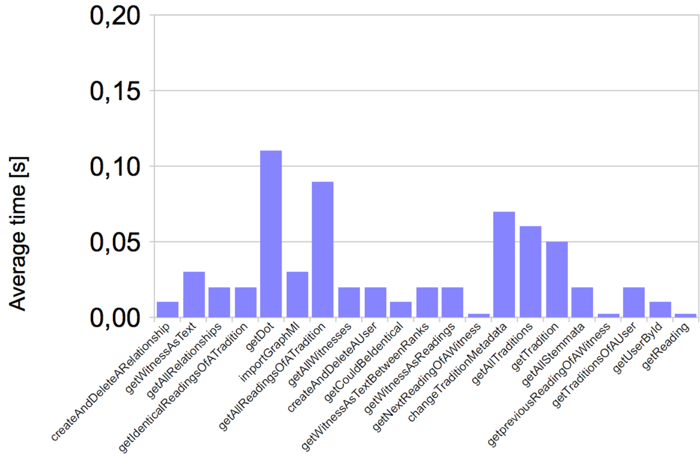


Figure 6.3: Database with 1000000 nodes, working tradition with 100 nodes



of the fact that each Tradition can be selected as a subgraph and the algorithms only have to search it, rather than the whole database. It is obvious, though, that the tradition size have an influence on the speed of the implemented algorithms as the working subset, which is in most cases a tradition, grows with a bigger tradition. Most of the algorithms which work on a tradition are in $O(logn)$, though there are also some export and import functions which have to handle each node and relation of the tradition and run in $O(n)$.

The following diagrams show the results of the tests in which the dimension of the tradition is varied while the size of the database stays the same. According to the diagrams, the execution

Figure 6.4: Database with 10000 nodes, working tradition with 1000 nodes

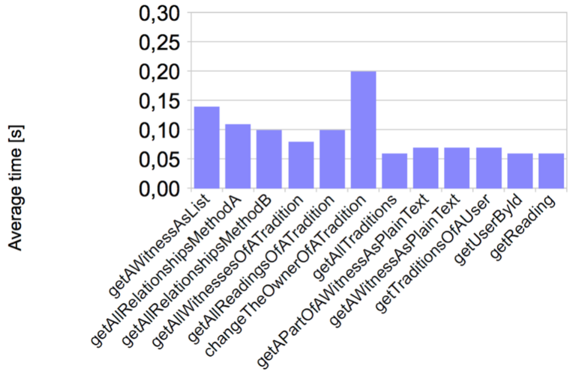
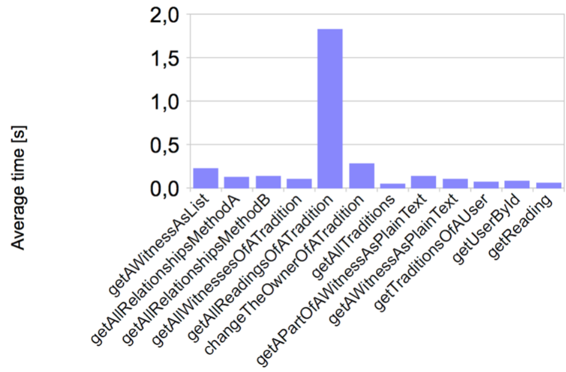
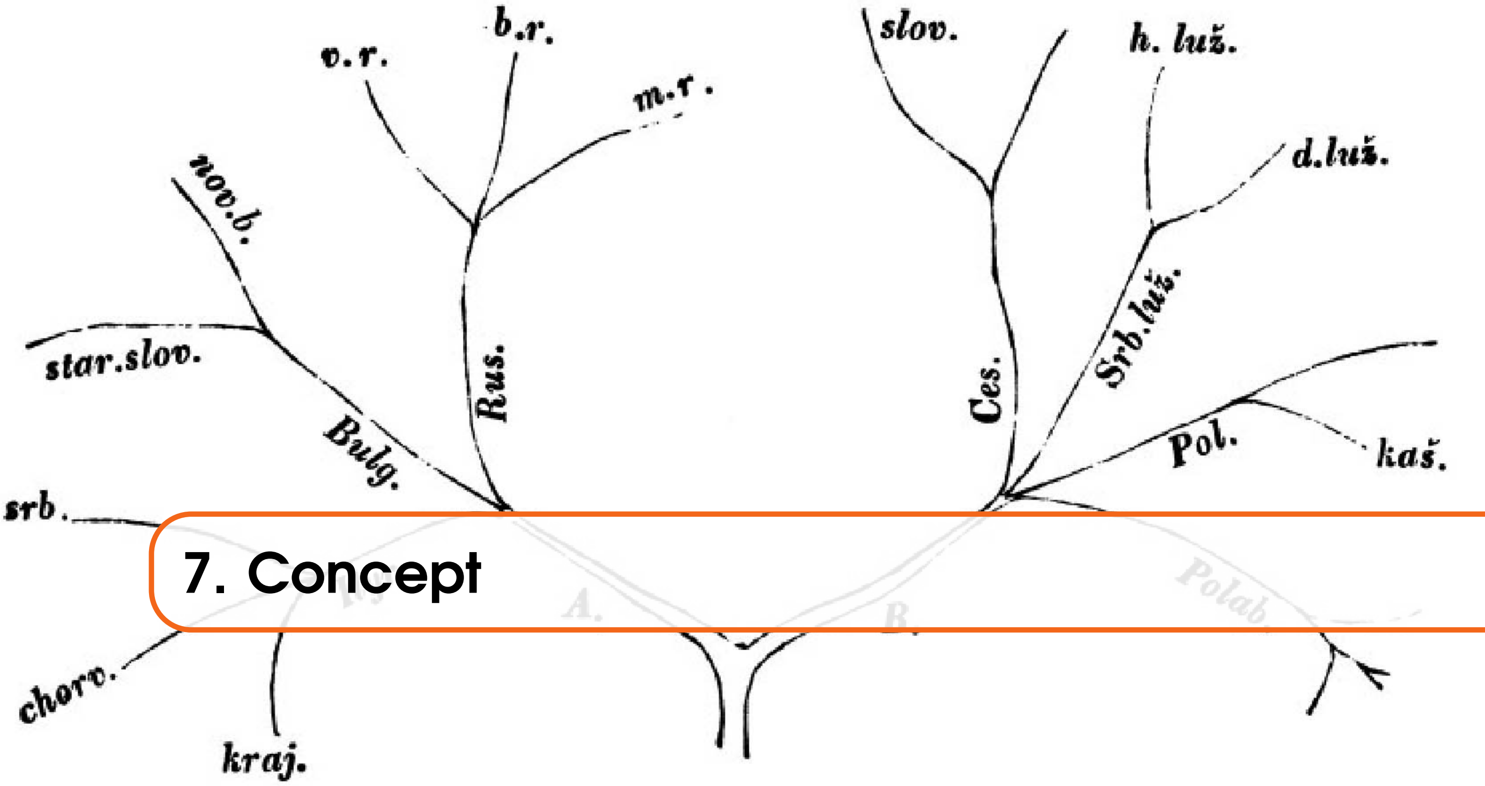


Figure 6.5: Database with 10000 nodes, working tradition with 10000 nodes



time depends on the size of the working tradition, as can be seen in the results for the getAllReadingsFromATradition method, in which each reading of the tradition is parsed to a JSON Object and all are returned as an List. The parsing of those nodes executes in O(n) time and the downloading of the JSON file takes about the same. As larger traditions are not expected the execution time of those methods stays in the accepted range.

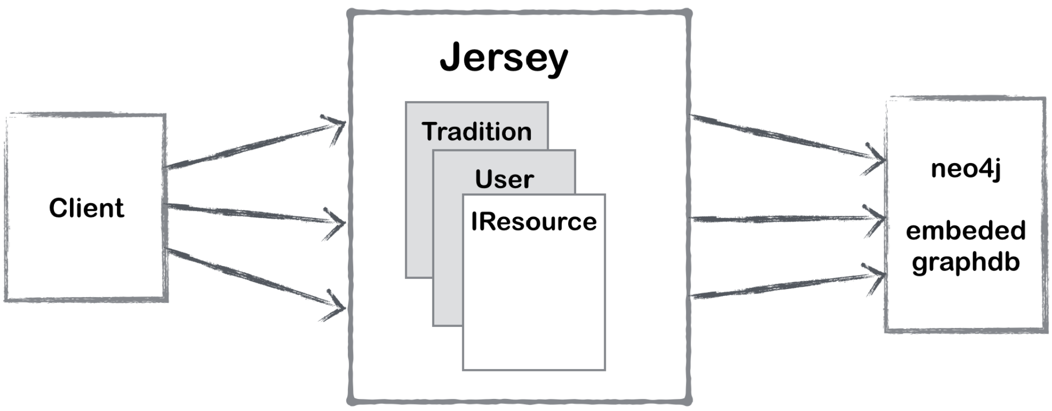
II Testing



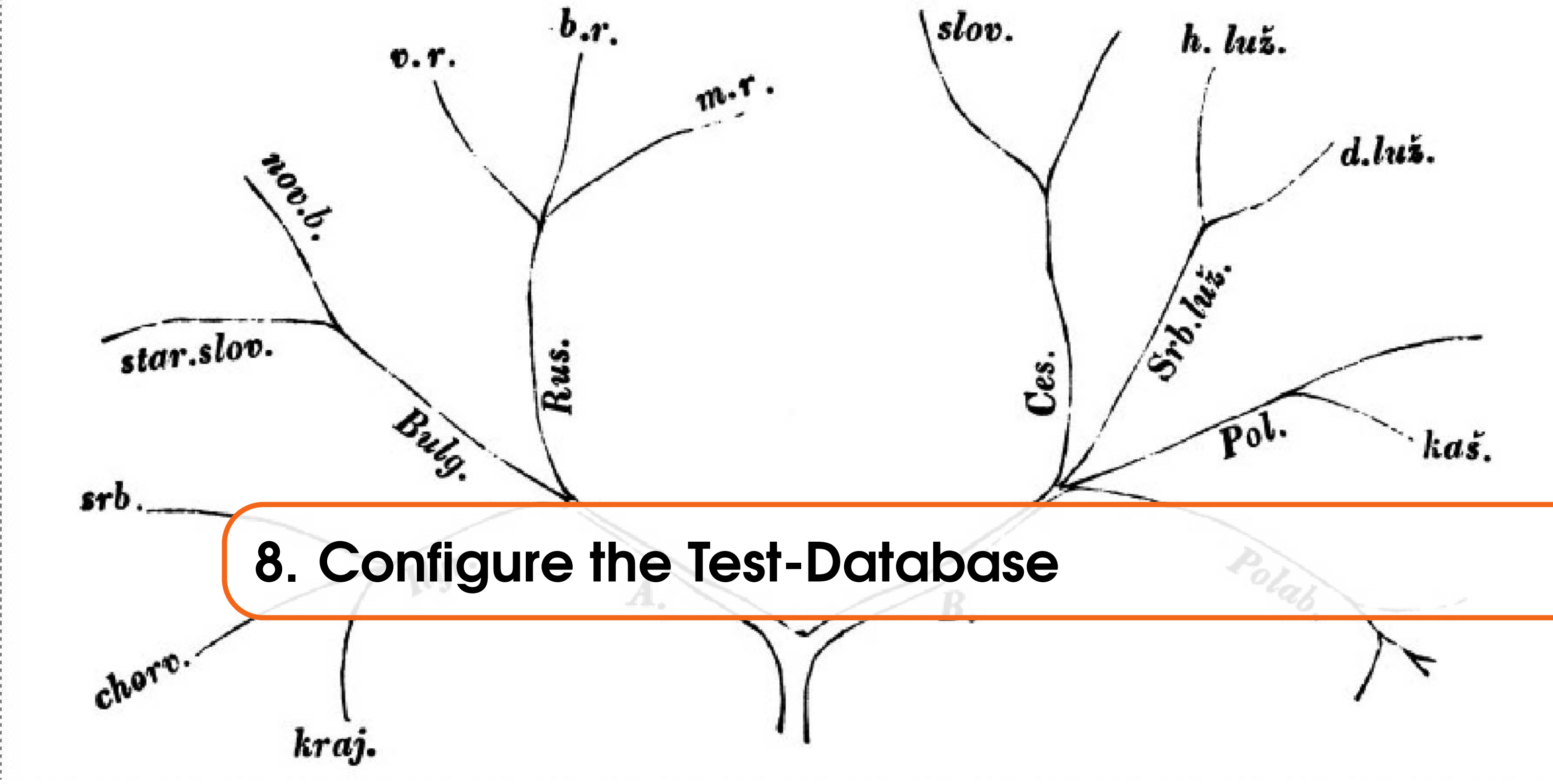
7. Concept

This chapter describes the test-concept of the Project. The tests were used for test driven development and to assure the quality of the product. All tests are written in such a manner that they do not have any impact on the architecture of the project.

Jersey Overview



For every REST call in the system a jersey instantiate the requested IResource and provides the service. A global singleton GraphDatabaseService object, which is is an embedded neo4j GraphDatabase, is used to provide the service. To achieve a minimal invasive test system the actual database is being replaced with a test database. To change the database with minimal test related code in the project, the GraphDatabaseServiceProvider is configured to return an impermanent Database. As the GraphDatabaseService is a singleton object this configuration is done before the start-up of the Jersey Testserver.



8. Configure the Test-Database

To use an impermanent Testdatabase in the Tests the following initialization steps are done: A Claswide GraphDatabaseService object db is registered:

```
GraphDatabaseService db;
```

In the @Before method the singleton GraphDatabaseService provided by the GraphDatabaseServiceProvider is overwritten by the following line:

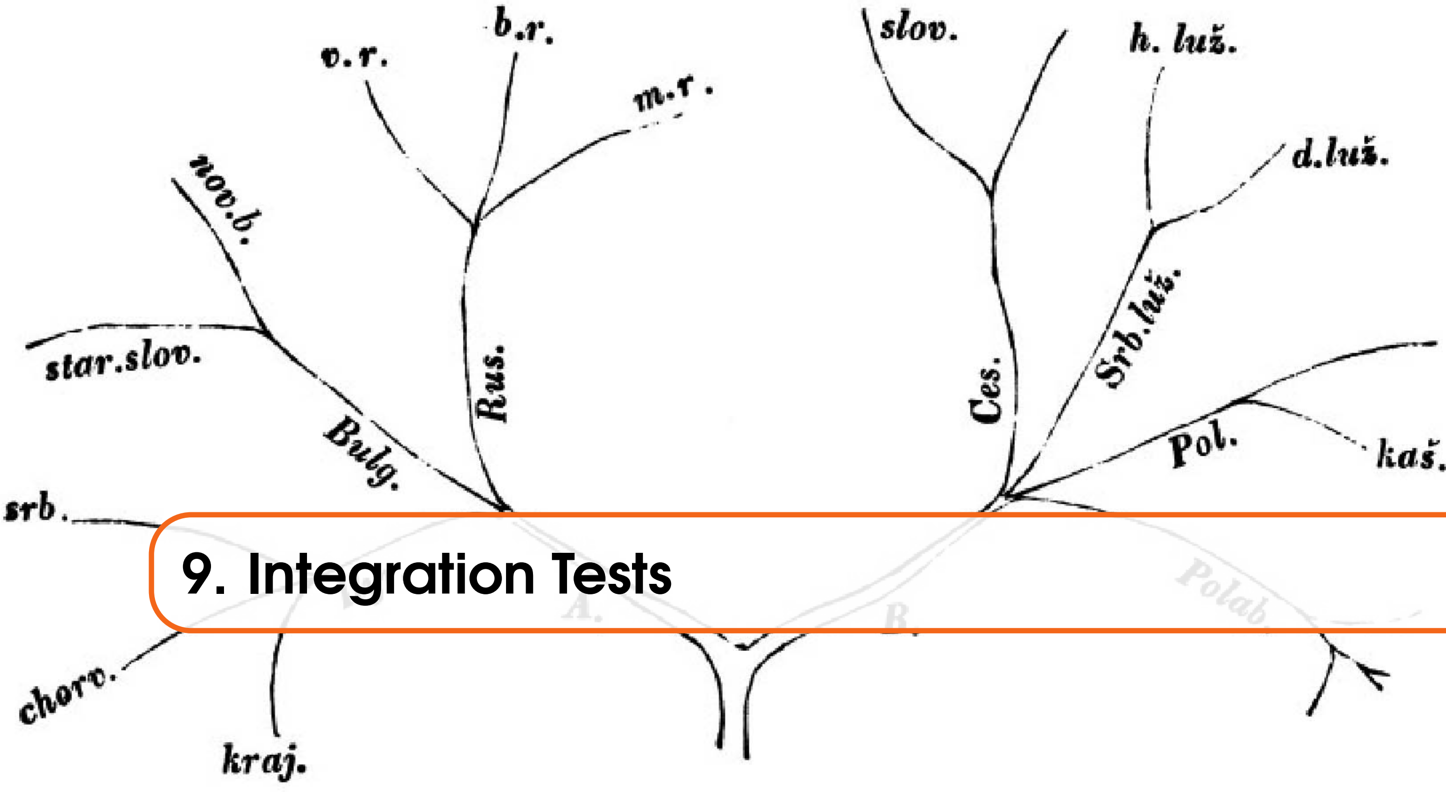
```
GraphDatabaseServiceProvider.setImpermanentDatabase();
```

Later the db object is initialized by:

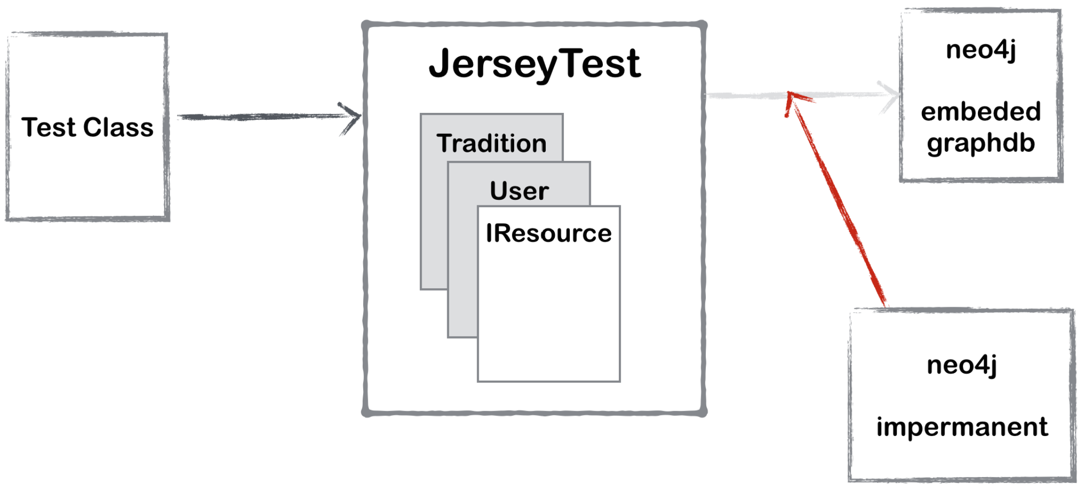
```
db = new GraphDatabaseServiceProvider().getDatabase();
```

In the @After method the database is being closed:

```
db.shutdown();
```



9. Integration Tests



Every user story is tested with integration tests. Those tests are the mean of assuring the quality of the project.

To inject objects into a resource the resource must be created statically. This is not possible when the resources are instantiated only when a REST call occurs. To solve this problem JerseyTestServerFactory creates a server where instantiated resources can be registered.
To start a JerseyTestServer a global JerseyTest is created:

```
private JerseyTest jerseyTest;
```

The JerseyTestServerFactory creates a JerseyTest with already instantiated resources. This is used to inject the mock objects.
Multiple resources can be added by chaining: .addResource(..).addResource()

```
jerseyTest = JerseyTestServerFactory.newJerseyTestServer()
        .addResource(userResource).create();
```

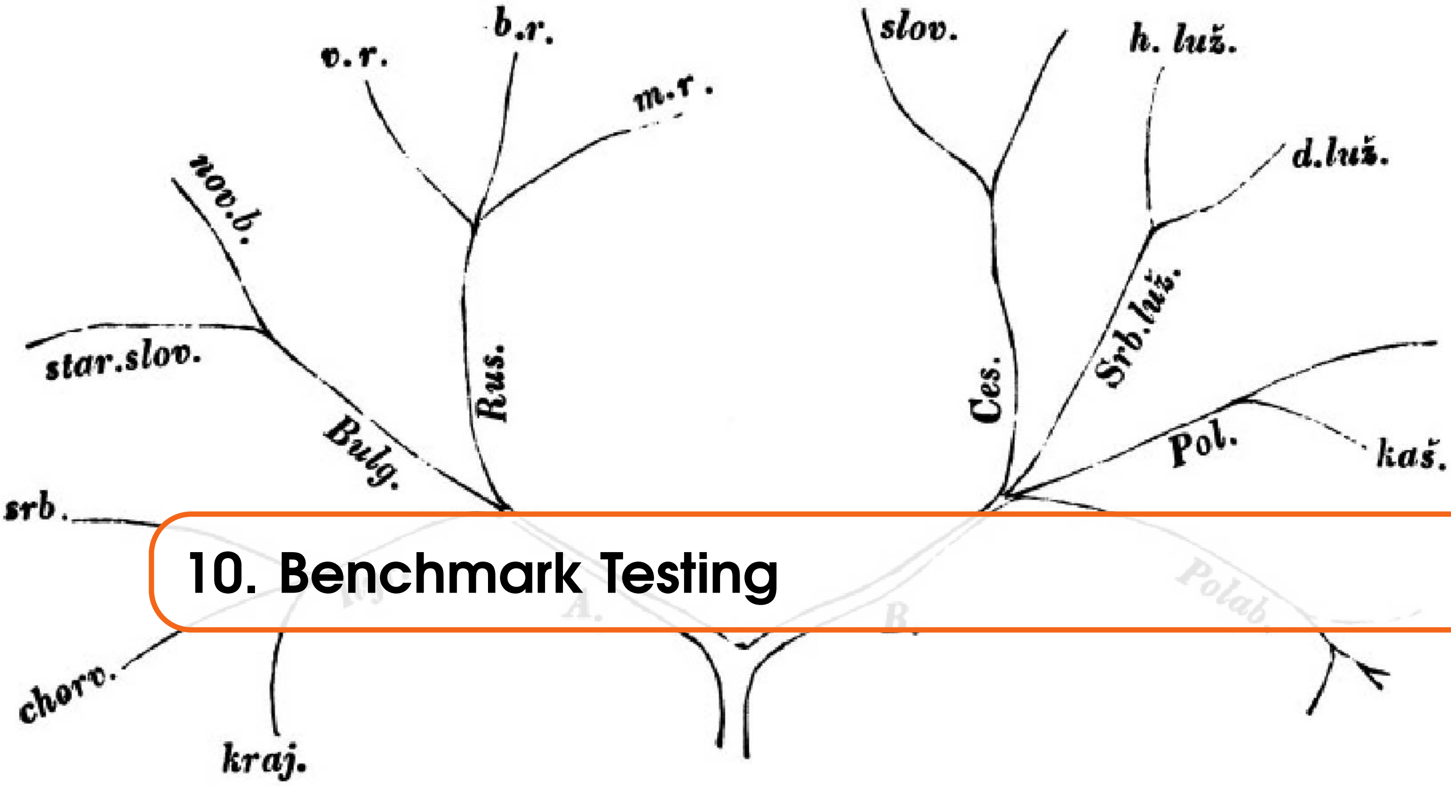
```
jerseyTest.setUp();
```

The test is done by calling a webresource of jerseyTest:

```
@Test
public void SimpleTest(){
  String actualResponse = jerseyTest.resource()
       .path("/user").get(String.class);
  assertEquals(actualResponse, "User!");
}
```

Example

https://github.com/tohotforice/PSE2_DH/blob/e364fcb0c164981281c5799a6bf9f9f9ea5eb503/stemmarest/src/test/java/net/stemmaweb/stemmaserver/UserTest.java



10. Benchmark Testing

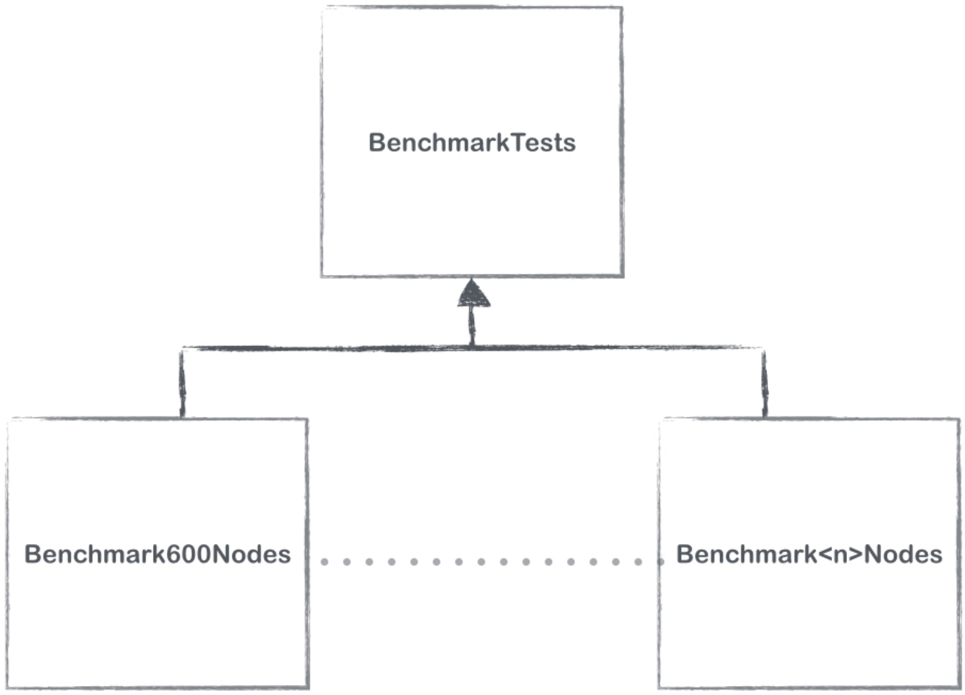
The main goal of the stemmarest project was to achieve better performance then the previous service. To measure the performance of the system benchmark testing was needed. A benchmark test basically calls the RESTful service multiple times and measure the response time. To achieve this a JUnit benchmark test suite is used. The JUnitbenchmarks measures the time used to execute a test and can generate visual representations of the measurement.
For effective benchmark testing it is important to have a variety of different databases. Those databases should differ in size from small to very large. To generate valid graphs only limited by space the class *RandomGraphGenerator* is being used. By calling this static method a graph is generated according to the given parameters.

> R Please note that the response time depends highly on the hardware the tests are running on and the actual state of Javas virtual machine.

To reduce the influence of the virtual machine before the measurements 5 warm-up calls are done. The hardware which was used for testing is represented in the report.

Setup

All the classes related to the Benchmark Tests can be found in the package *net.stemmaweb.stemmaserver.benachmarktests*. The class BenchmarkTests contains all the Tests. The classes Benchmark<n>Nodes contain the database generator. In it the number of nodes in the current test-database is being configured. BenchmarkTests cant be run as a JUnit test.



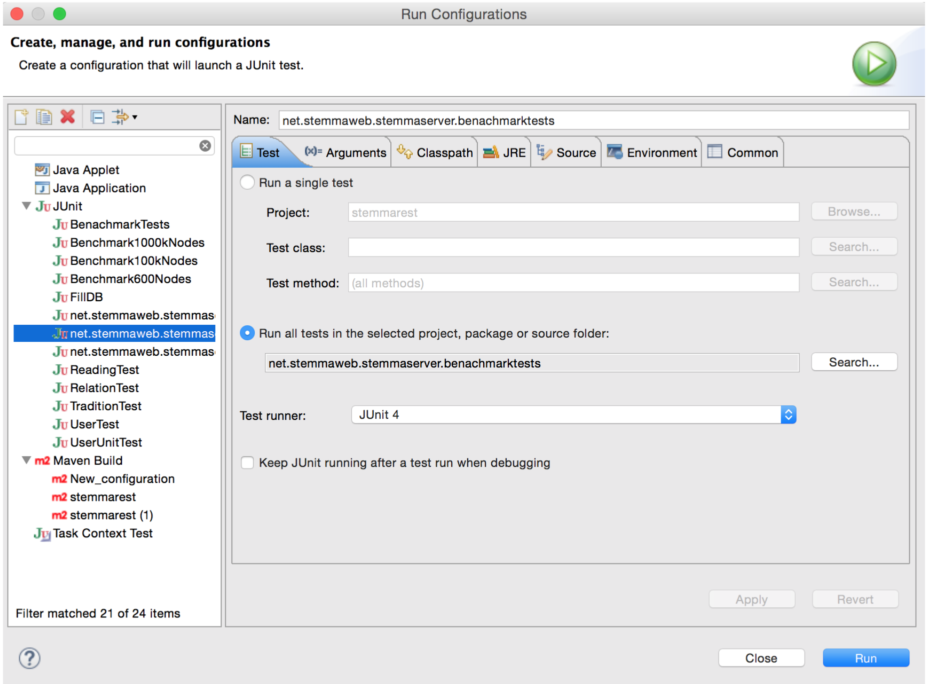
Tests are implemented in the BenchmarkTests class with the @Test annotation. Only restcall is implemented in the methods and they only test if the Response.Status is OK. This assures that as low as possible overhead time will be generated and measured.

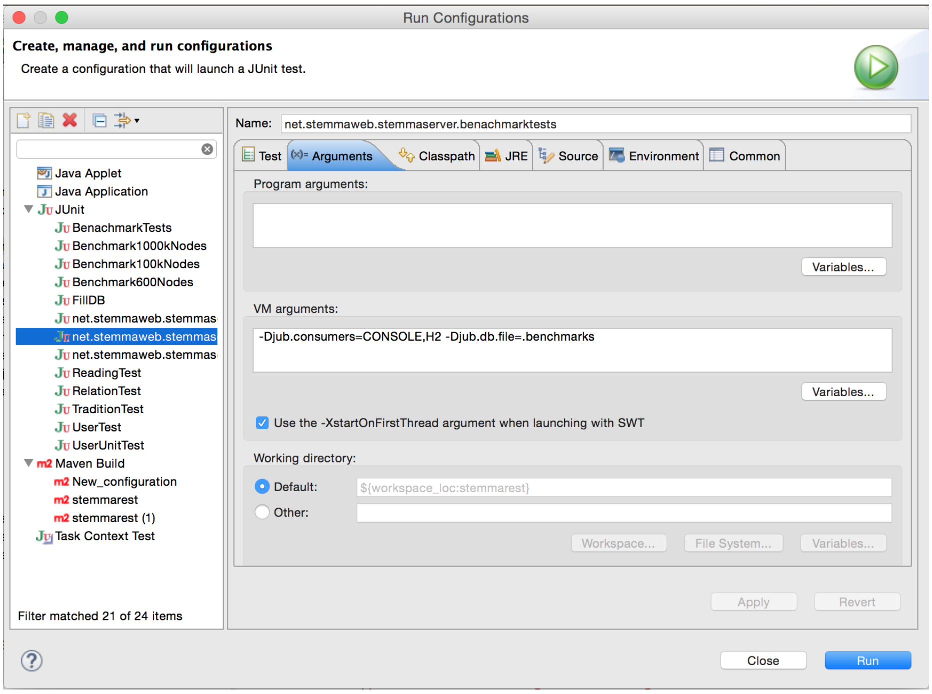
R JUnitBenchmarks measures the time to execute (@Before, @Test, @After). Heavy operations which should not be measured can be done in @BeforeClass and @AfterClass.

To create a new database, the test-environment copies the class Benchmark600Nodes and rename it to the count of Nodes that are being inserted. In the class itself only two small adjustments are done: the name of the report file is being changed *@BenchmarkMethodChart(filePrefix = "benchmark/benchmark-600Nodes")* and the properties of the database which should be generated are being adjusted *rgg.role(db, 2, 1, 3, 100);*. role(databaseService, cardinalityOfUsers, cardinalityOfTraditionsPerUser, cardinalityOfWitnessesPerTradition, degreeOfTheTraditionGraphs)

Run Benchmarktests

The Benchmarktests can be run as every JUnit test. To generate the report, though, an argument needs to be passed. Create a JUnit Test as follows:



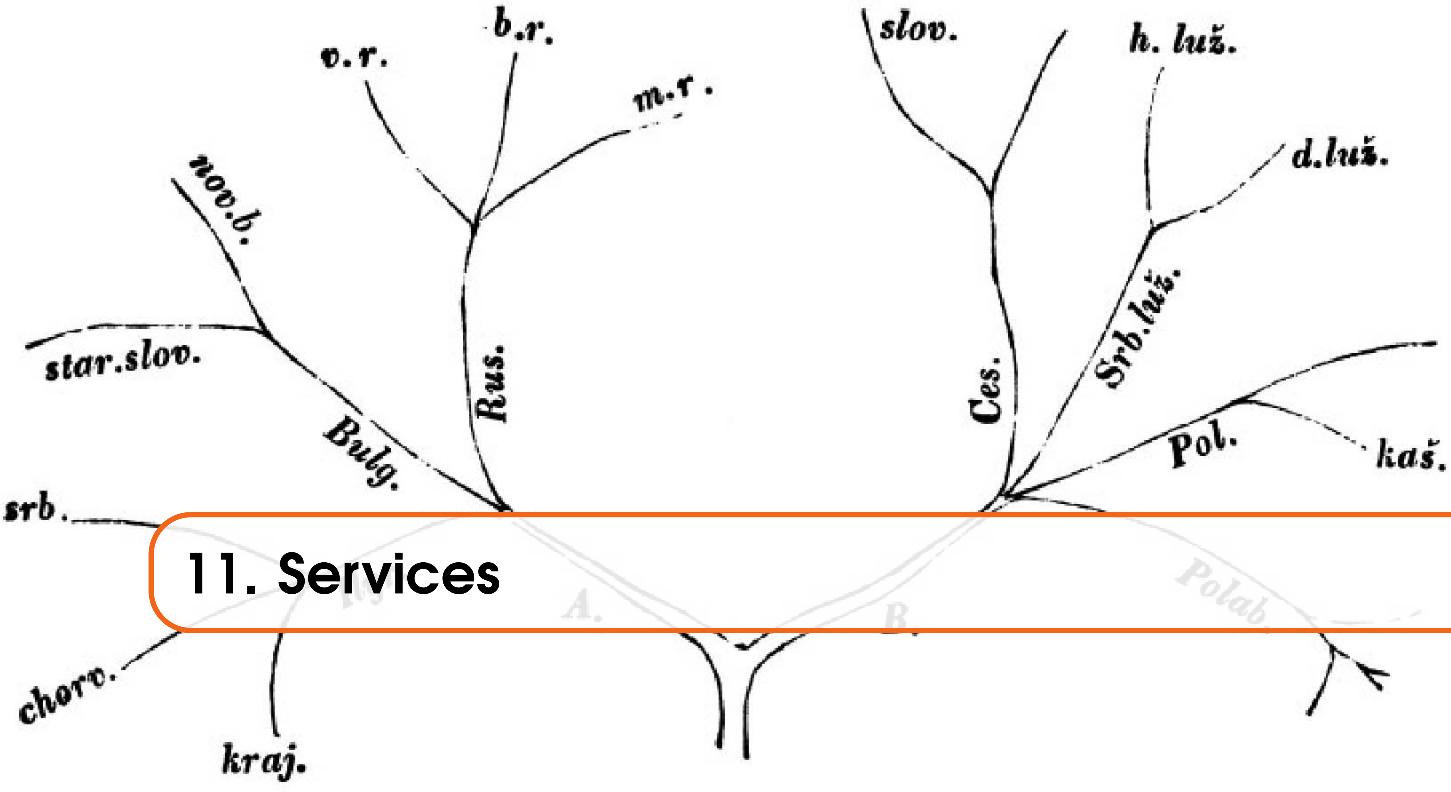


On the tab Arguments *-Djub.consumers=CONSOLE,H2 -Djub.db.file=.benchmarks* has to be inserted into the VM Arguments input. After that the test can be executed as usual. After the execution of the tests the reports will be stored at *benchmark/*.

The execution of the tests will take some time because of the need to generate huge graphs. Its recommended not to use the computer for other assignments during the tests.

III

RESTful API



11. Services

11.1 Baseresource:

http://localhost:8080/

11.2 /stemma

Method Name: getAllStemmata
Gets a list of all Stemmata available, as dot format

GET /getallstemmata/fromtradition/{tradId}

Response list of stemmata as dot

Parameter tradId as string

Method Name: setStemma
Puts the Stemma of a DOT file in the database

POST /newstemma/intradition/{tradId}

Response stemma as dot

Parameter tradId as string

Method Name: reorientStemma
Reorients a stemma tree with a given new root node

POST /reorientstemma/fromtradition/{tradId}/withtitle/{stemmaTitle}/withnewrootnode/{nodeId}

Response stemma as dot

Parameter tradId as string

Parameter stemmaTitle as string

Parameter nodeId as string

Method Name: getStemma
Returns JSON string with a Stemma of a tradition in DOT format

GET /getstemma/fromtradition/{tradId}/withtitle/{stemmaTitle}

Response stemma as dot

Parameter tradId as string

Parameter stemmaTitle as string

11.3 /relation

Method Name: delete
Remove all relationships, as it is done in https://github.com/tla/stemmaweb/blob/master/lib/stemmaweb/Controller/Relat
line 271) in Relationships of type RELATIONSHIP between the two nodes.

POST /deleterelationship/fromtradition/{tradId}

Request relationshipModel as application/json

Response as text/plain: HTTP Response 404 when no node was found, 200 When relationships where removed

Parameter tradId as string

Method Name: create
Creates a new relationship between the two nodes specified.

POST /createrelationship

Request relationshipModel as application/json

Response graphModel as application/json

Method Name: getAllRelationships
Get a list of all relationships from a given tradition.

GET /getallrelationships/fromtradition/{tradId}

Response list of relationshipModel as application/json

Parameter tradId as string

Method Name: deleteById
Removes a relationship by ID.

DELETE /deleterelationshipbyid/withrelationship/{relationshipId}

- **Response** relationshipModel as application/json
- **Parameter** relationshipId as string

11.4 /tradition

Method Name: getAllRelationships
Gets a list of all relationships of a tradition with the given id.

GET /getallrelationships/fromtradition/{tradId}

- **Response** list of relationshipModels as application/json
- **Parameter** tradId as string

Method Name: changeTraditionMetadata
Changes the metadata of the tradition.

POST /changemetadata/fromtradition/{tradId}

- **Request** traditionModel as application/json
- **Response** traditionModel as application/json
- **Parameter** tradId as string

Method Name: getAllTraditions
Gets a list of all the complete traditions in the database.

GET /getalltraditions

- **Response** list of traditionModels as application/json

Method Name: getAllWitnesses
Gets a list of all the witnesses of a tradition with the given id.

GET /getallwitnesses/fromtradition/{tradId}

- **Response** list of witnessModels as application/json
- **Parameter** tradId as string

Method Name: getTradition
Returns GraphML file from specified tradition owned by user

GET /gettradition/withid/{tradId}

- **Response** tradition as application/xml (graphML)

Parameter tradId as string

Method Name: deleteTraditionById
Removes a complete tradition

DELETE /deletetradition/withid/{tradId}

Response as text/plain: http response

Parameter tradId as string

Method Name: importGraphMl
Imports a tradition by given GraphML file and meta data

POST //newtraditionwithgraphml

Request graphML as multipart/form-data

Response the id of the imported tradition as text/plain

Method Name: getDot
Returns DOT file from specified tradition owned by user

GET /getdot/fromtradition/{tradId}

Response as application/json: XML data

Parameter tradId as string

11.5 /reading

Method Name: changeReadingProperties
Changes properties of a reading according to its keys

POST /changeproperties/ofreading/{readId}

Request ReadingChangePropertyModel as application/json

Response readingModel as application/json

Parameter readId as long

Method Name: getReading
Returns a single reading by global neo4j id

GET /getreading/withreadingid/{readId}

Response readingModel as application/json

Parameter readId as long

Method Name: duplicateReading
Duplicates a reading in a specific tradition. Opposite of merge

POST /duplicatereading

Request duplicateModel as application/json

Response GraphModel as application/json

Method Name: mergeReadings
Merges two readings into one single reading in a specific tradition. Opposite of duplicate

POST /mergereadings/first/{firstReadId}/second/{secondReadId}

Response as application/json: Status.OK on success or Status.INTERNAL_SERVER_ERROR with a detailed message.

Parameter secondReadId as long

Parameter firstReadId as long

Method Name: splitReading
Splits up a single reading into several ones in a specific tradition. Opposite of compress

POST /splitreading/ofreading/{readId}/withsplitindex/{splitIndex}

Request as text/plain

Response GraphModel as application/json

Parameter readId as long

Parameter splitIndex as int

Method Name: getNextReadingInWitness
gets the next readings from a given readings in the same witness

GET /getnextreading/fromwitness/{witnessId}/ofreading/{readId}

Response readingModel as application/json

Parameter readId as long

Parameter witnessId as string

Method Name: getPreviousReadingInWitness
gets the previous readings from a given readings in the same witness

GET /getpreviousreading/fromwitness/{witnessId}/ofreading/{readId}

Response readingModel as application/json

Parameter readId as long

Parameter witnessId as string

Method Name: getAllReadings
Returns a list of all readings in a tradition

GET /getallreadings/fromtradition/{tradId}

Response list of readingModels as application/json

Parameter tradId as string

Method Name: getIdenticalReadings
Get all readings which have the same text and the same rank between given ranks

GET /getidenticalreadings/fromtradition/{tradId}/fromstartrank/{startRank}/toendrank/{endRank}

Response list of list of readingModels as application/json

Parameter endRank as long

Parameter tradId as string

Parameter startRank as long

Method Name: getCouldBeIdenticalReadings
Returns a list of a list of readingModels with could be one the same rank without problems

GET /couldbeidenticalreadings/fromtradition/{tradId}/fromstartrank/{startRank}/toendrank/{endRank}

Response list of readingModels as application/json

Parameter endRank as long

Parameter tradId as string

Parameter startRank as long

Method Name: compressReadings
Compress two readings into one. Texts will be concatenated together (with or without a space or extra text. The reading with the lower rank will be given first. Opposite of split

POST /compressreadings/read1id/{read1Id}/read2id/{read2Id}/concatenate/{con}

Request as text/plain

Response as application/json: status.ok if compress was successful. Status.INTERNAL_SERVER_ERROR with a detailed message if not concatenated

Parameter read1Id as long

Parameter read2Id as long

Parameter con as string

11.6 /witness

Method Name: getWitnessAsText
finds a witness in the database and returns it as a string

GET /gettext/fromtradition/{tradId}/ofwitness/{witnessId}

Response witness as string

Parameter tradId as string

Parameter witnessId as string

Method Name: getWitnessAsTextBetweenRanks
find a requested witness in the data base and return it as a string according to define start and end readings (including the readings in those ranks). if end-rank is too high or start-rank too low will return till the end/from the start of the witness

GET /gettext/fromtradition/{tradId}/ofwitness/{witnessId}/fromstartrank/{startRank}/toendrank/{endRank}

Response witness as string

Parameter endRank as string

Parameter tradId as string

Parameter startRank as string

Parameter witnessId as string

Method Name: getWitnessAsReadings
finds a witness in the database and returns it as a list of readings

GET /getreadinglist/fromtradition/{tradId}/ofwitness/{witnessId}

Response list of readingModels as application/json

Parameter tradId as string

Parameter witnessId as string

11.7 /user

Method Name: create
Creates a user based on the parameters submitted in JSON.

POST /createuser

Request userModel as application/json

Response userModel as application/json

Method Name: getUserById
Gets a user by the id.

GET /getuser/withid/{userId}

Response userModel as application/json

Parameter userId as string

Method Name: deleteUserById
Removes a user and all his traditions

DELETE /deleteuser/withid/{userId}

Response as text/plain: OK on success or an ERROR in JSON format

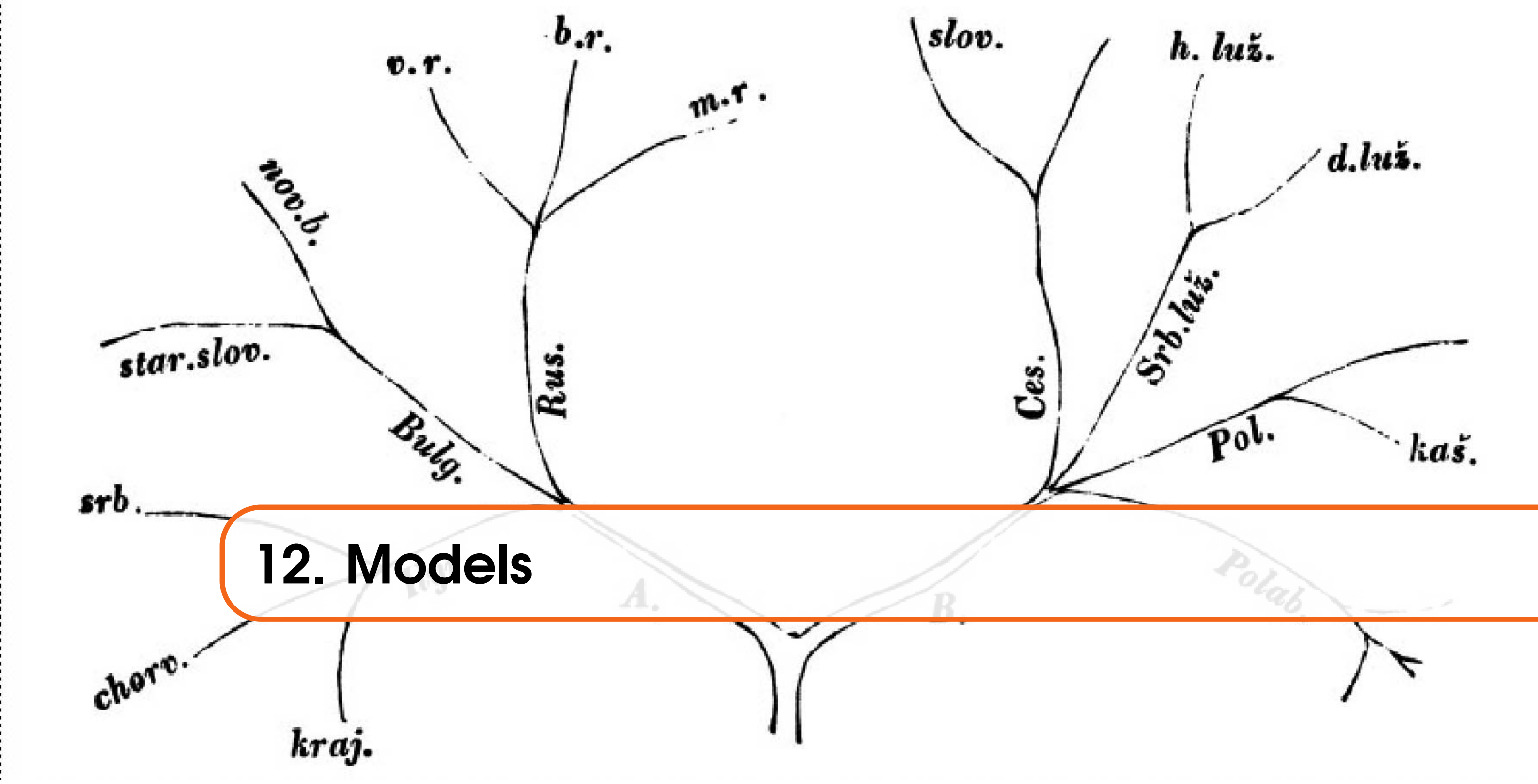
Parameter userId as string

Method Name: getTraditionsByUserId
Get all Traditions of a user

GET /gettraditions/ofuser/{userId}

Response as application/json: OK on success or an ERROR in JSON format

Parameter userId as string



12. Models

12.1 relationshipModel

- **Property** a_derivable_from_b as string
- **Property** alters_meaning as string
- **Property** annotation as string
- **Property** b_derivable_from_a as string
- **Property** displayform as string
- **Property** extra as string
- **Property** id as string
- **Property** is_significant as string
- **Property** non_independent as string
- **Property** reading_a as string
- **Property** reading_b as string
- **Property** scope as string
- **Property** source as string
- **Property** target as string
- **Property** type as string

- **Property** witness as string

12.2 readingModel

- **Property** grammar_invalid as string
- **Property** id as string
- **Property** is_common as string
- **Property** is_end as string
- **Property** is_lacuna as string
- **Property** is_lemma as string
- **Property** is_nonsense as string
- **Property** is_ph as string
- **Property** is_start as string
- **Property** join_next as string
- **Property** join_prior as string
- **Property** language as string
- **Property** lexemes as string
- **Property** normal_form as string
- **Property** rank as long
- **Property** text as string

12.3 traditionModel

- **Property** id as string
- **Property** isPublic as string
- **Property** language as string
- **Property** name as string
- **Property** ownerId as string

12.4 duplicateModel

- **Property** readings as long
- **Property** witnesses as string

12.5 graphModel

Property readings as long

Property witnesses as string

12.6 witnessModel

Property id as string

12.7 userModel

Property id as string

Property isAdmin as string

12.8 ReadingChangePropertyModel

Property properties as list of KeyPropertyModels

12.9 stemma

A tree that provides an overview over the witnesses of a tradition. The relations between the witnesses which are displayed as nodes is central. Here the stemmata are mostly returned in dot format.

12.10 GraphML

`http://de.wikipedia.org/wiki/GraphML`

12.11 KeyPropertyModel

Property key as string

Property property as string