



# Stemmarest

## Documentation of the PSE2 Project 2015

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



Copyright © 2015 Team PSE2

PUBLISHED BY ...

....ORG

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*First printing, March 2013*



## Contents

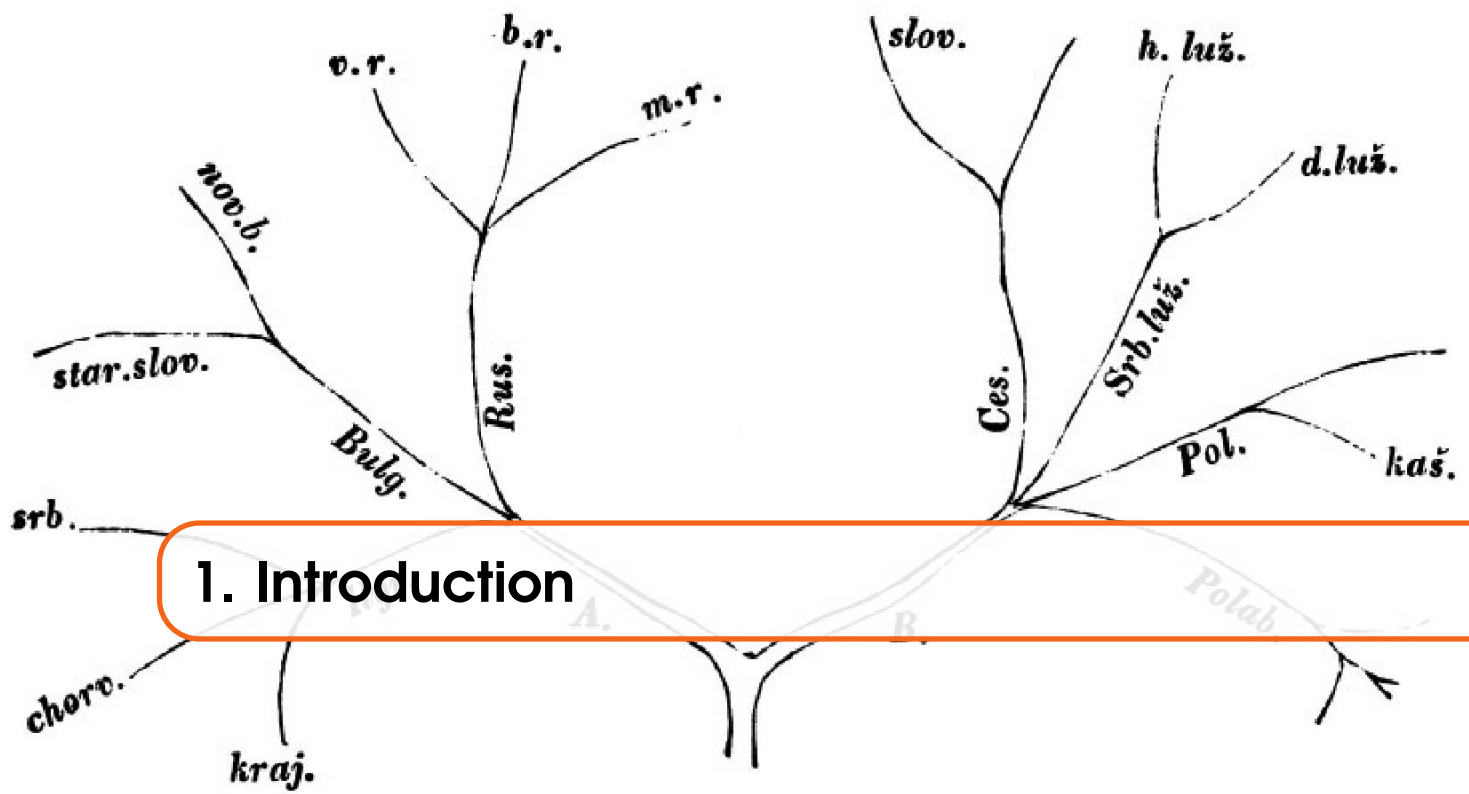
<b>I</b>	<b>Project</b>	
1	Introduction .....	6
2	Setup .....	7
3	Database (neo4j) .....	8
4	Jersey .....	10
5	Performance .....	11
<b>II</b>	<b>Testing</b>	
6	Concept .....	15
7	Configure the Test-Database .....	17
8	Unit Test .....	18
9	Integration Tests .....	19
10	Benchmark Testing .....	21

<b>11</b>	<b>Services</b> .....	<b>25</b>
11.1	Baseresource:	25
11.2	/stemma	25
11.3	/relation	26
11.4	/tradition	27
11.5	/reading	28
11.6	/witness	31
11.7	/user	31
<b>12</b>	<b>Models</b> .....	<b>33</b>
12.1	relationshipModel	33
12.2	readingModel	34
12.3	traditionModel	34
12.4	duplicateModel	34
12.5	graphModel	34
12.6	witnessModel	35
12.7	userModel	35
12.8	stemma	35
	<b>Bibliography</b> .....	<b>36</b>
	Books	36
	Articles	36

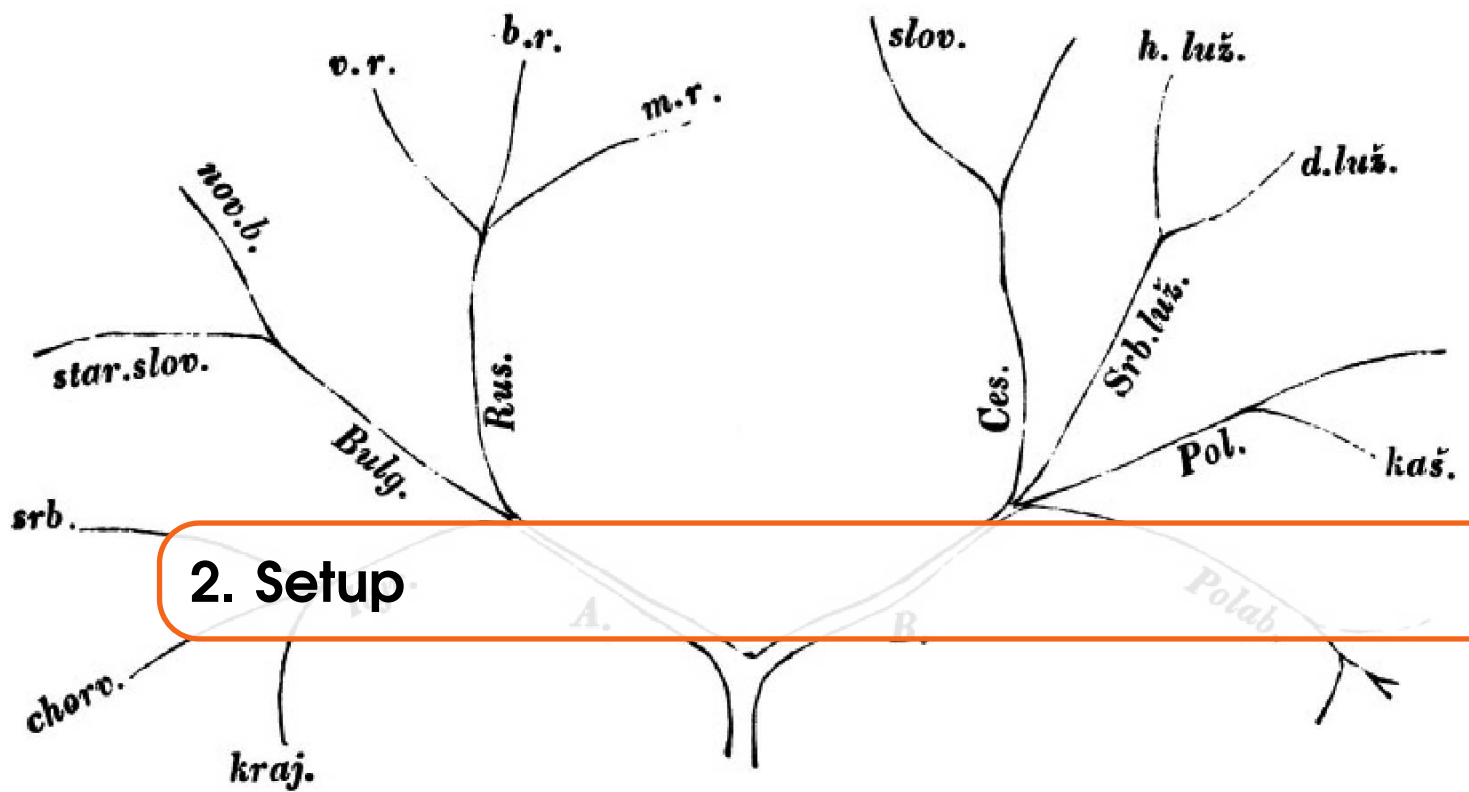


# Project

1	Introduction .....	6
2	Setup .....	7
3	Database (neo4j) .....	8
4	Jersey .....	10
5	Performance .....	11



Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.



## 2. Setup

### Downloading

git clone [https://github.com/tohotforice/PSE2\\_DH.git](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

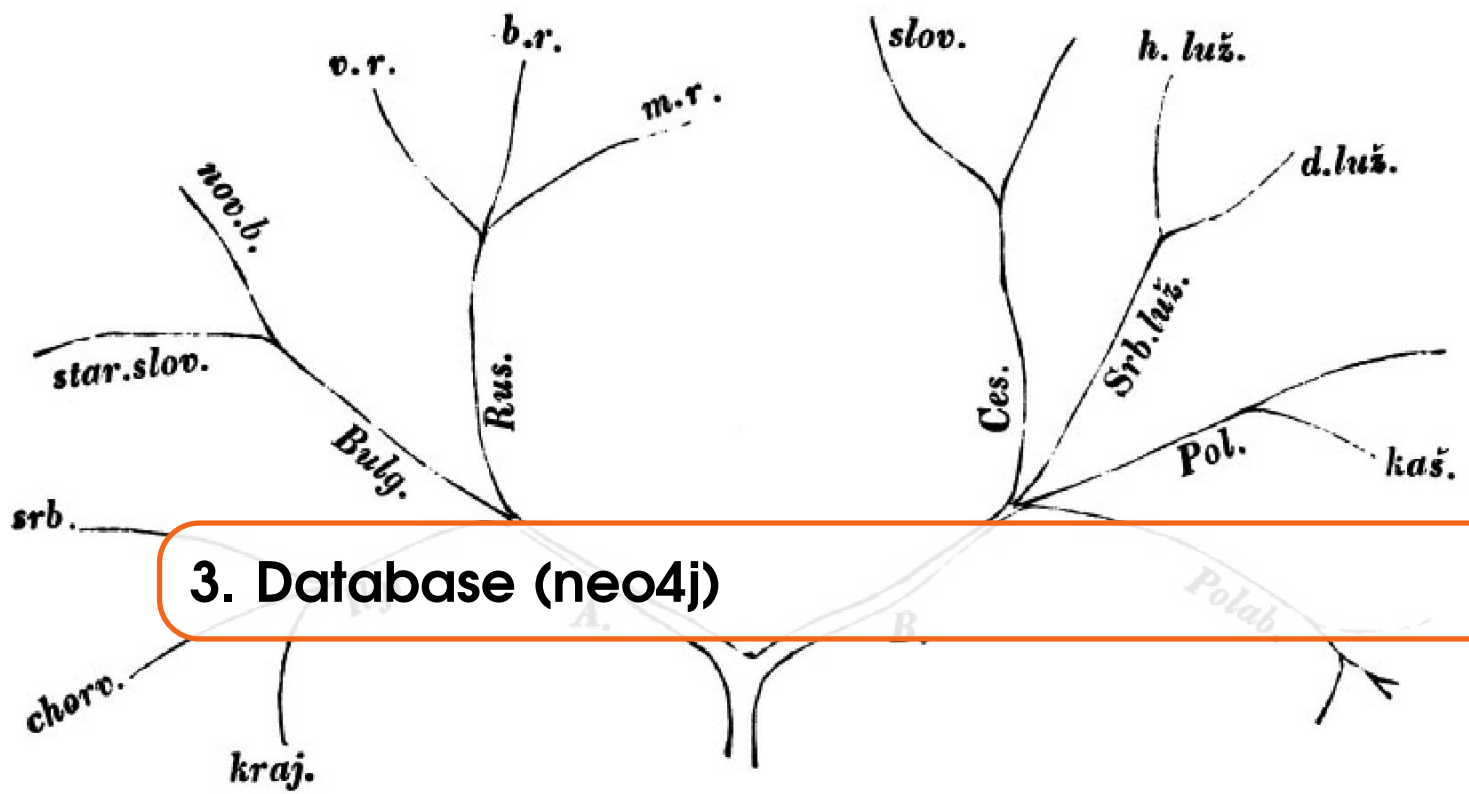
### Running

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/from-tradition/traditionId) or by actually going into the data base (see Neo4j visualization in section Database)



### 3. Database (neo4j)

In this project a graph database is used to store the data instead of a relational database. This was due to the fact that graph databases are much faster when it comes down to look for objects in a list that fulfil some constraints to other objects. In a relational database one would normally use multiple joins to get the desired result. 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 algorithm and look for a specific relation between two nodes, which represent objects. This could make a search of objects, which are connected to each other, very efficient.

We have chosen to use Neo4J (Additional info can be found under <http://neo4j.com/developer/graph-db-vs-rdbms/>) which is a graph database capable of very efficiently finding and managing nodes and relationship even in a large scale graph.

Additional information regarding benchmarking 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 to increase the search speed.

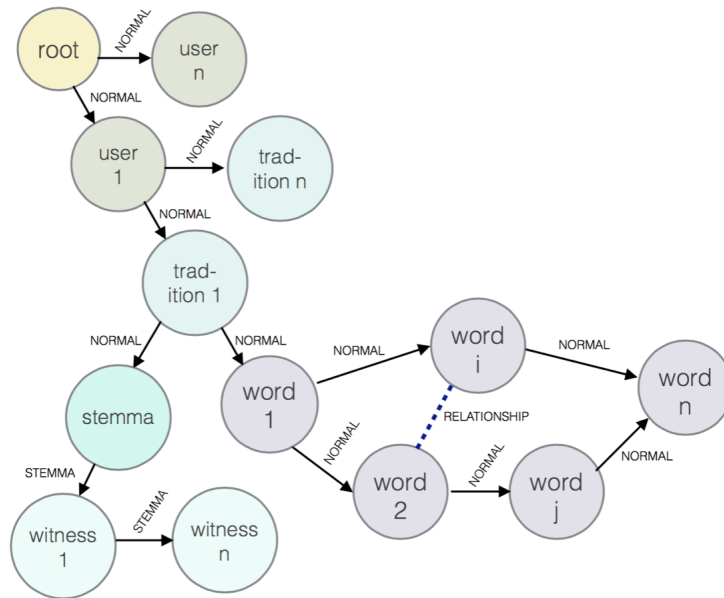
Only a few labels are used in the database:

Nodes	Relationships
ROOT	RELATIONSHIP
STEMMA	STEMMA
WITNESS	NORMAL
TRADITION	
USER	
WORD	

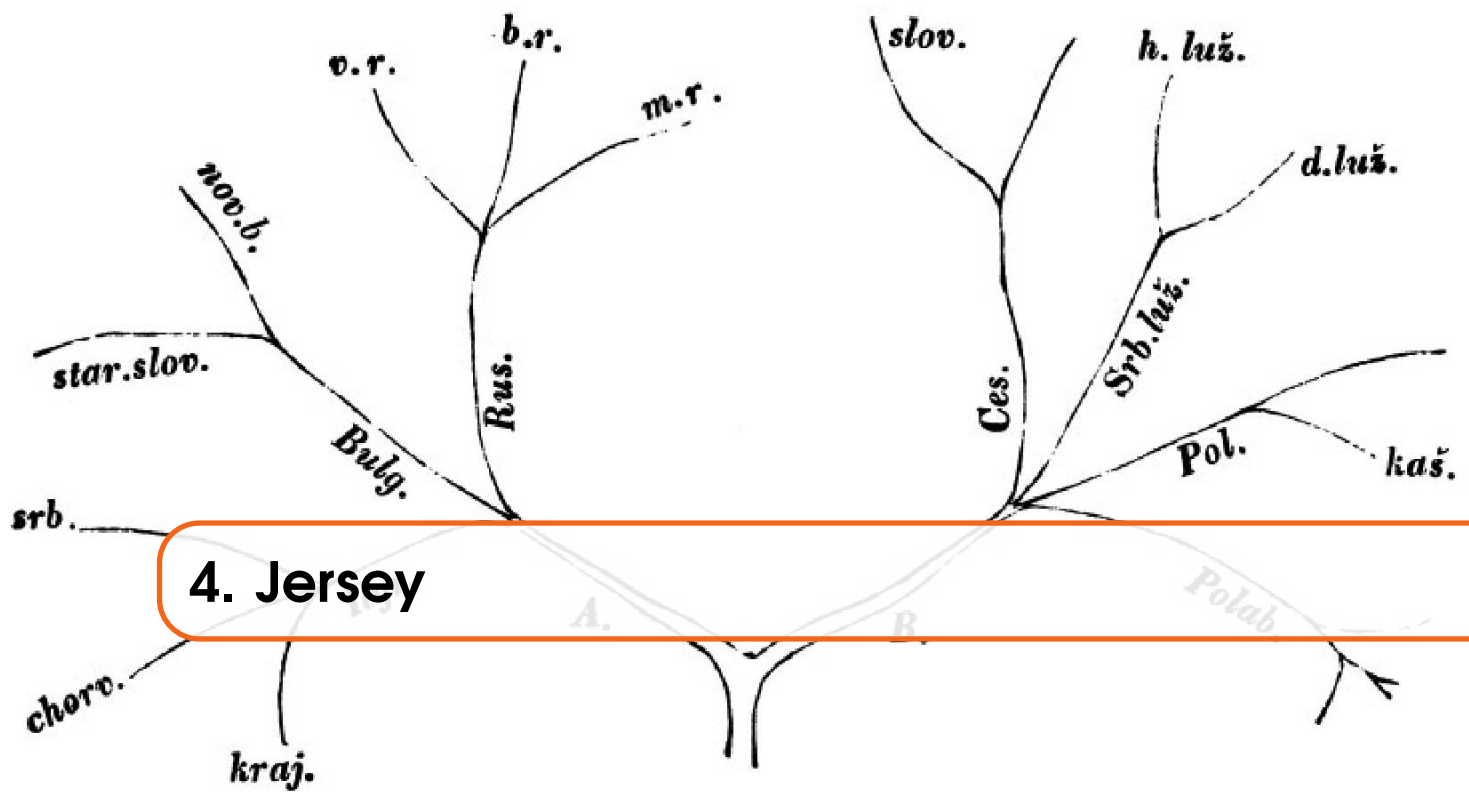
Since each label is stored in another file, searching or traversing the graph is stunningly fast.



The database structure is as follows:



Neo4J uses a powerful script language called cypher. It's the equivalent to SQL in relational databases. Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of the graph store. Using cypher one can write simple queries that return nodes or traverse graphs. But as Cypher queries need to be interpreted and translated into an execution plan they are still less fast than the native java traversal API which is therefore the common query tool used in the project.

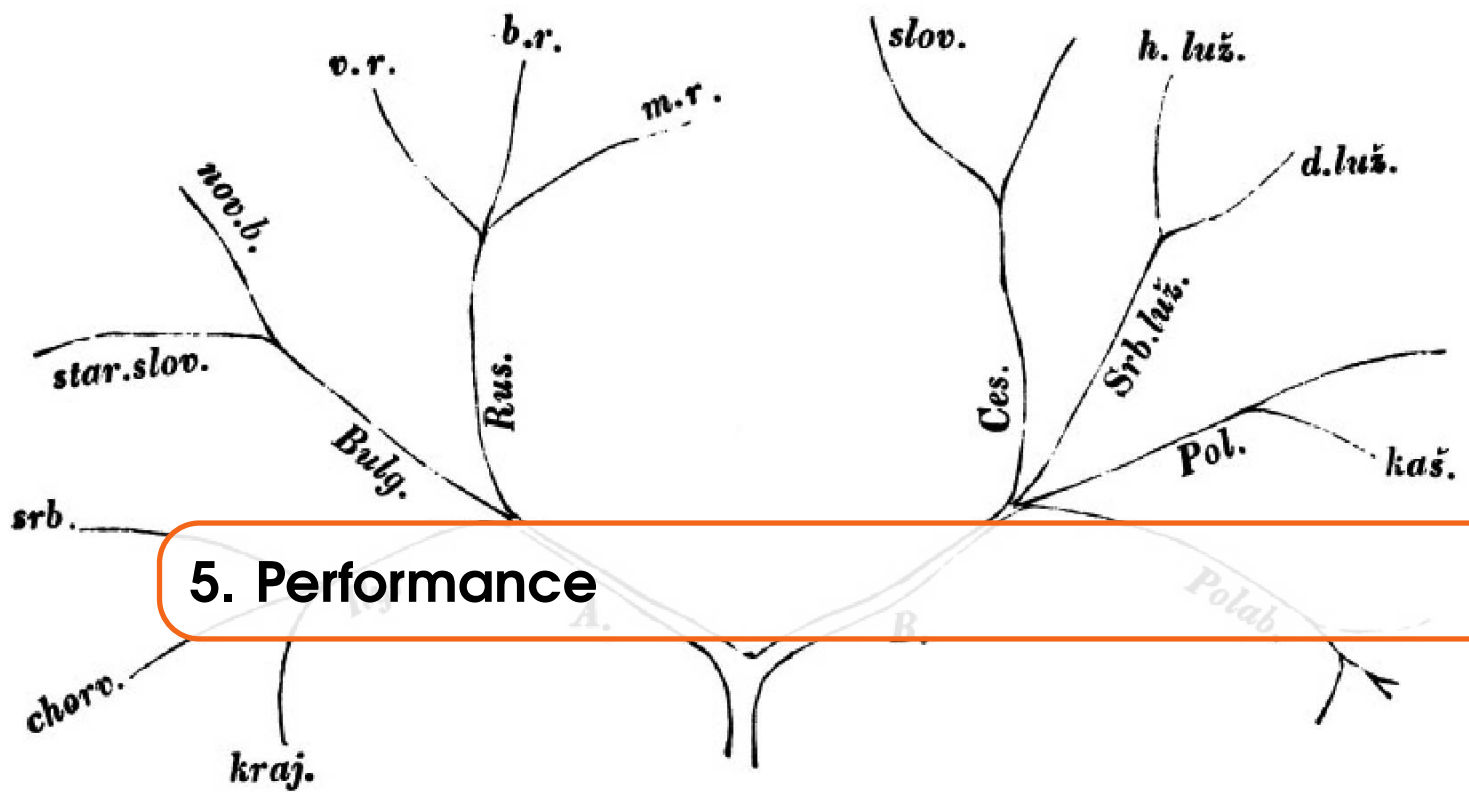


Jersey is an open source java framework for developing RESTful Web Services in Java that is built upon JAX-RS and serves as a JAX-RS Reference Implementation. It adds additional features and utilities in order to further simplify development of REST-APIs. Jersey helps support exposing the data in different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

An example of the method declaration of duplicateReading:

```
@POST
@Path("duplicatereading/fromtradition/{tradId}")
@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 with parameters in curly brackets. The method can „consume“ data sent by the client, which is 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“ a response, which is the method's return value, in this case also in JSON.



## 5. Performance

One of the main goals of this project is it to create a RESTful service which is significantly faster then the existing one. To assure 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, therefore the tests measure the time 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, so the network speed is 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 while the size of the database increases. (There are several Node-ID queries in the database which execute under  $O(\log n)$ ). But as the database access time takes such a small percentage of the measurement scope, this does not show up as the size of the database grows).

In the following measurements are a result of tests which test the dimension of the database. 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 on 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 then important to realize that those queries are done on  $O(\log n)$  time and are not seen in the noise of the other operations during the tests. In a much bigger 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.

The previous measurement test the dimension of the database and could show that the response time is almost independent to its size between 1000 and 1 Million Nodes (Readings). This indepen-

Figure 5.1: Database with 1000 nodes, working tradition with 100 nodes

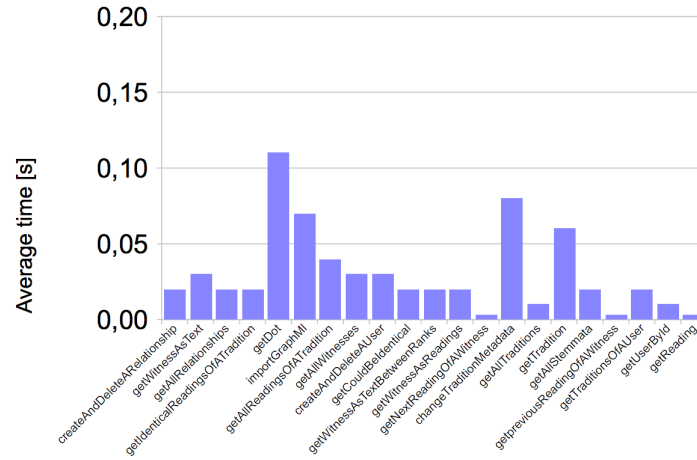


Figure 5.2: Database with 100000 nodes, working tradition with 100 nodes

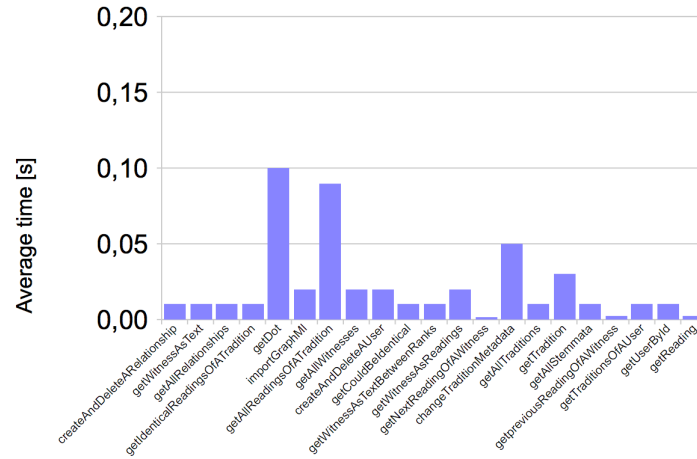
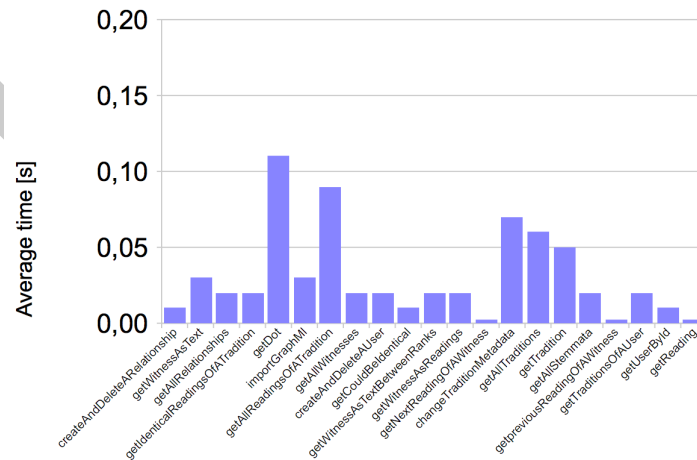


Figure 5.3: Database with 1000000 nodes, working tradition with 100 nodes



dence is a results of the fact that each Tradition can be selected as a subgraph and the algorithms only have a subset of the whole database to search trough. It is obvious the tradition size have an

influence on the speed of those algorithms, as the working subset grows with a bigger tradition. Most of the algorithms which work on a tradition are in  $O(\log n)$ , though there are also some export and import functions which have to handle each node and relation of the tradition and run in  $O(n)$ .

In the following tests the dimension of the tradition is varied. Those measurements show

Figure 5.4: Database with 10000 nodes, working tradition with 1000 nodes

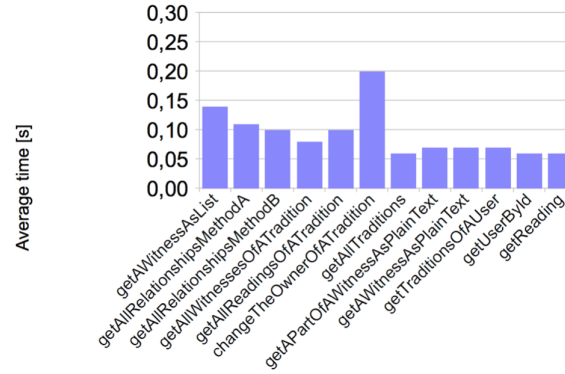
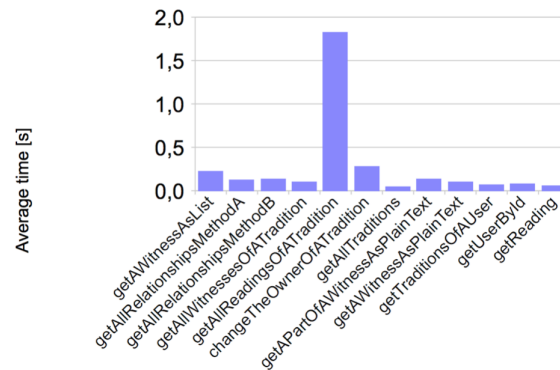
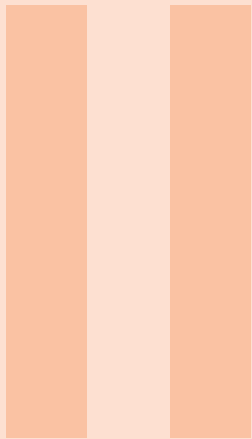


Figure 5.5: Database with 10000 nodes, working tradition with 10000 nodes

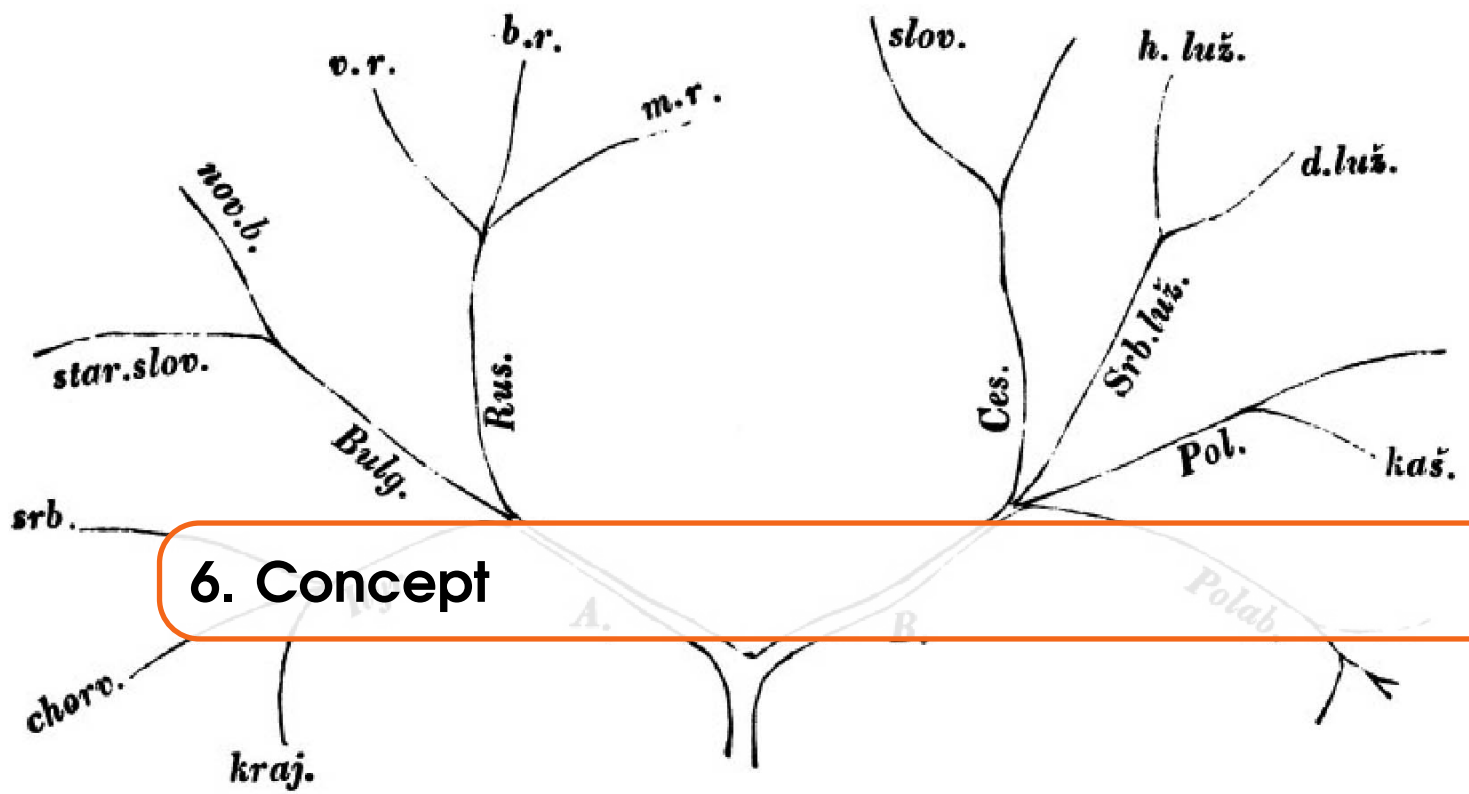


that the execution time depends on the size of the working tradition, as in the results for the `getAllReadingsFromATradition` method, in which each reading of the tradition is parsed to a JSON Object and all are returned as an List. The parsing of those nodes executes in  $O(n)$  and the downloading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



# Testing

6	Concept .....	15
7	Configure the Test-Database .....	17
8	Unit Test .....	18
9	Integration Tests .....	19
10	Benchmark Testing .....	21



This chapter describes the test-concept of the Digital Humanities PSE2 Project. The testing is used to assure the quality of the project and for test driven development. All tests are written in such a manner that they do not have any impact on the architecture of the project.

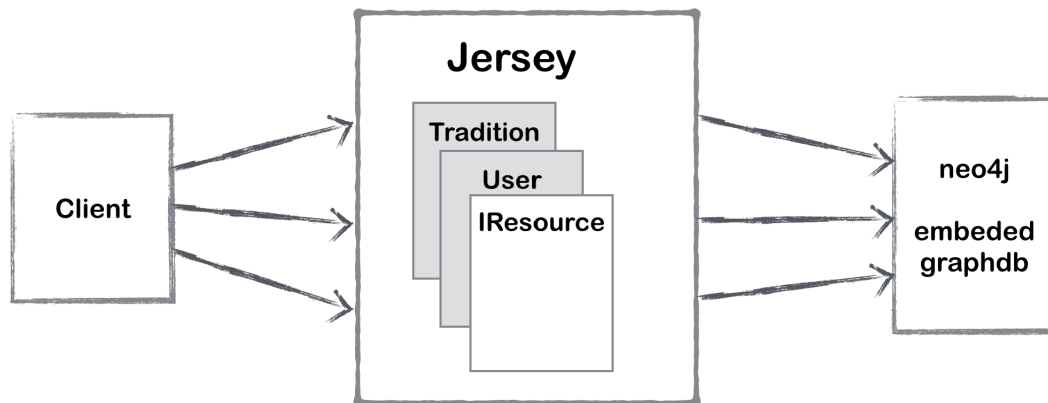
### Integration Tests

Every user story is tested by integration tests. Those tests assures the quality of the project. The technique of integration tests is described in the Integration test chapter.

### Unit Tests

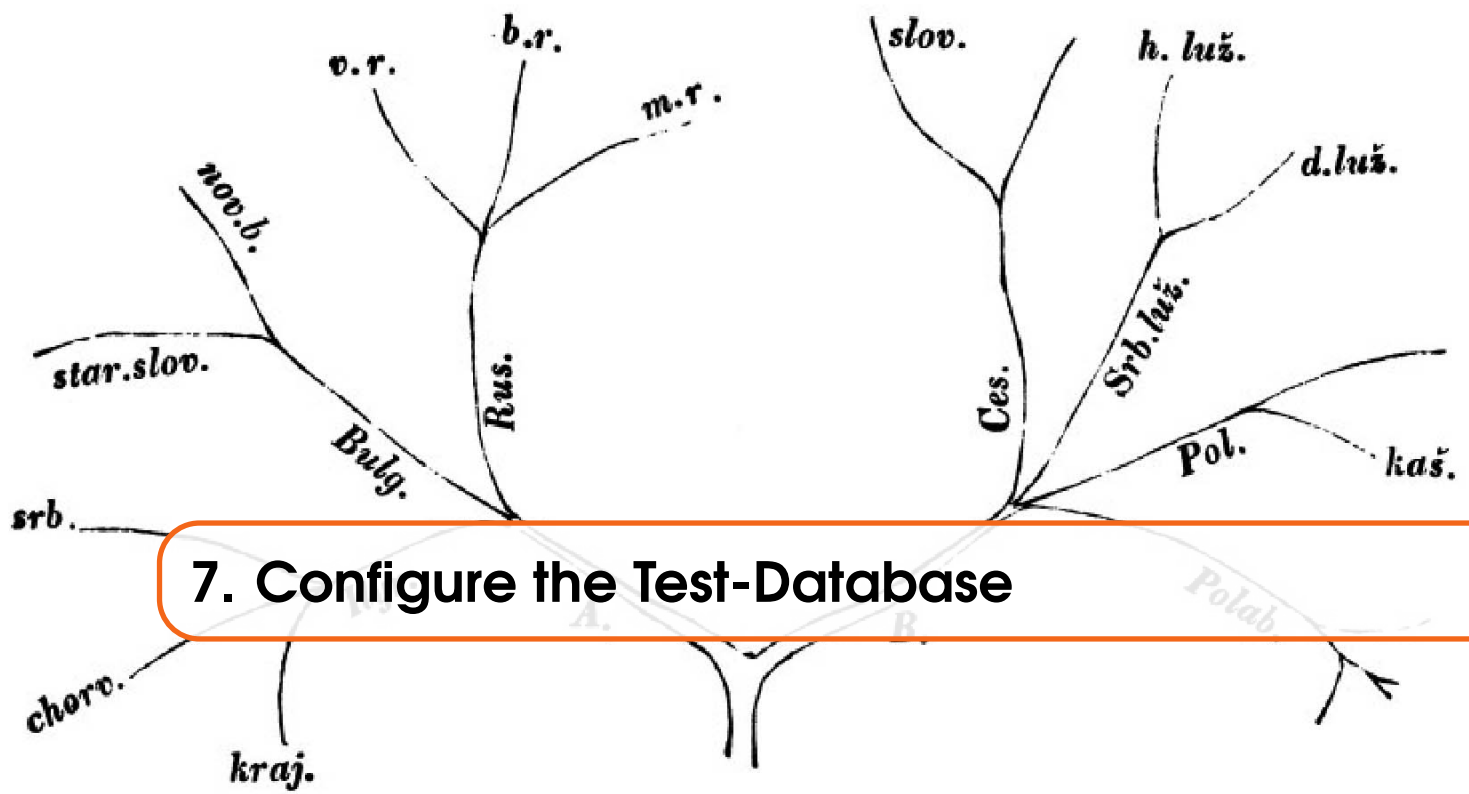
Unit tests are used for test-driven-development and are defined by the developer. They are only used for the development process and are not a referenced to the quality of the end product.

## Jersey Overview



In the productive system, for every REST call a jersey instantiate the requested IResource and provides the service. A global singleton GraphDatabaseService object is used to provide the GraphDatabaseService. In production this GraphDatabaseService is an embedded neo4j GraphDatabase. To achieve a minimal invasive test system the productive database needs to be 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 needs to be done before the start-up of the Jersey Testserver.





In this Project the `GraphDatabaseService` is a Singleton Object provided by the `GraphDatabaseServiceProvider`. To use a Testdatabase in the Tests the following steps have to be done. A Claswide `GraphDatabaseService` object `db` has to be registered.

```
GraphDatabaseService db;
```

In the `@Before` method the singleton `GraphDatabaseService` provided by the `GraphDatabaseServiceProvider` has to be overwritten by the following line:

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

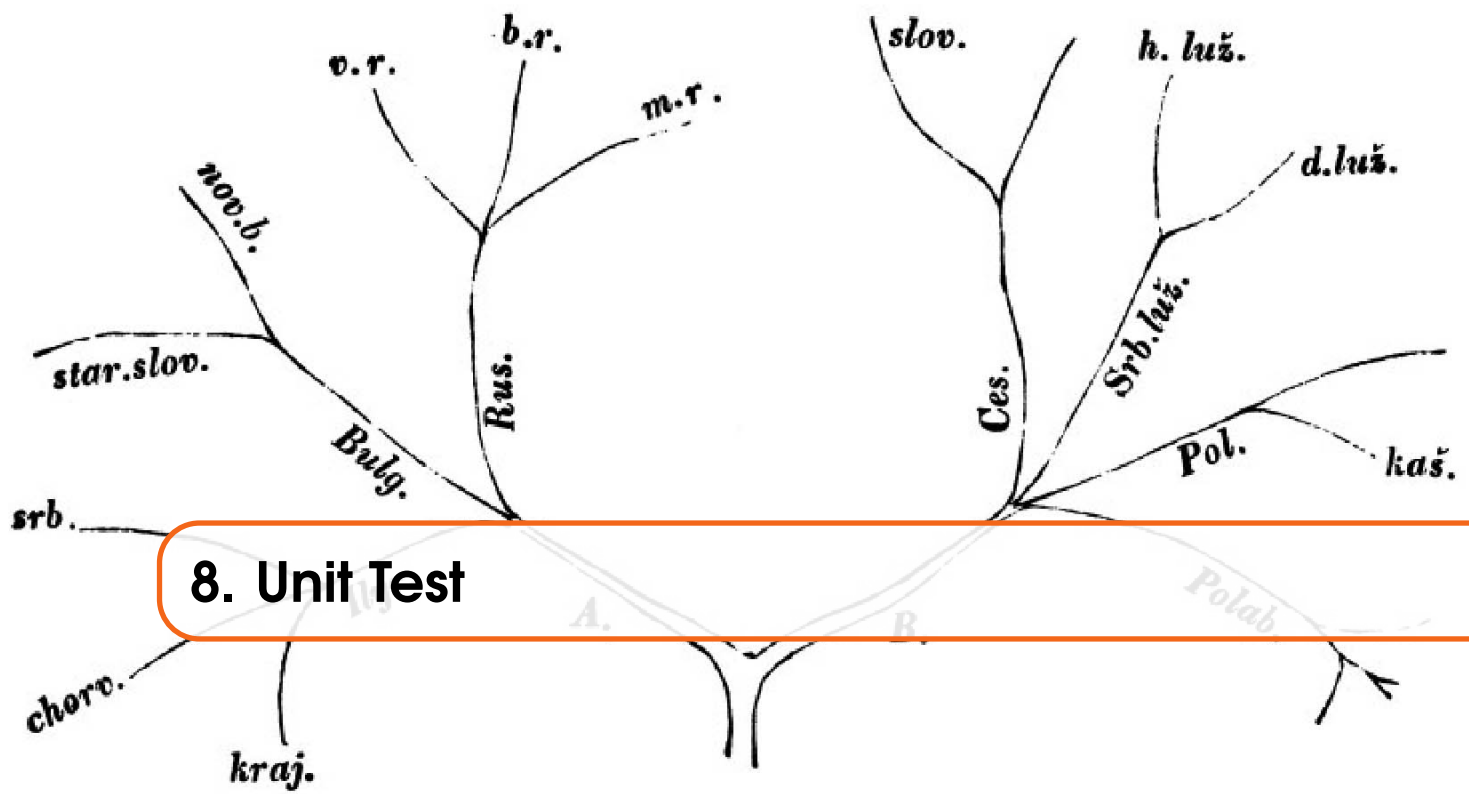
Later the `db` object can be initialized by:

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

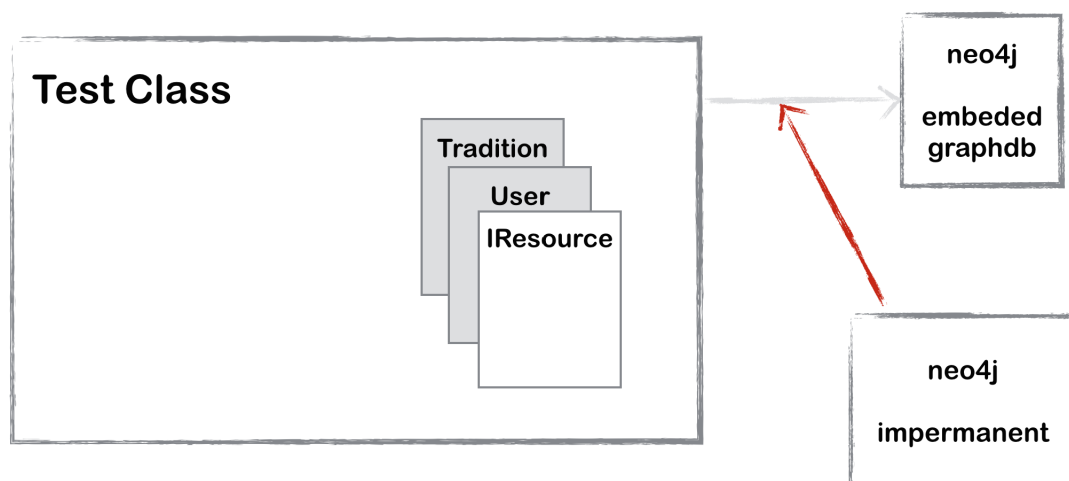
In the `@After` method the database has to be closed

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

With this configuration the impermanent Testdatabase of Neo4j is used during the tests.



## 8. Unit Test

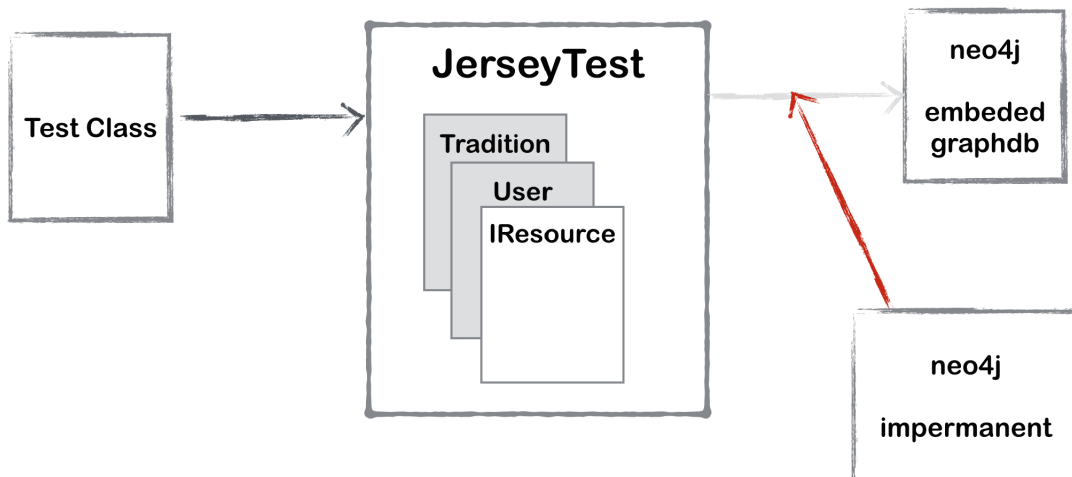
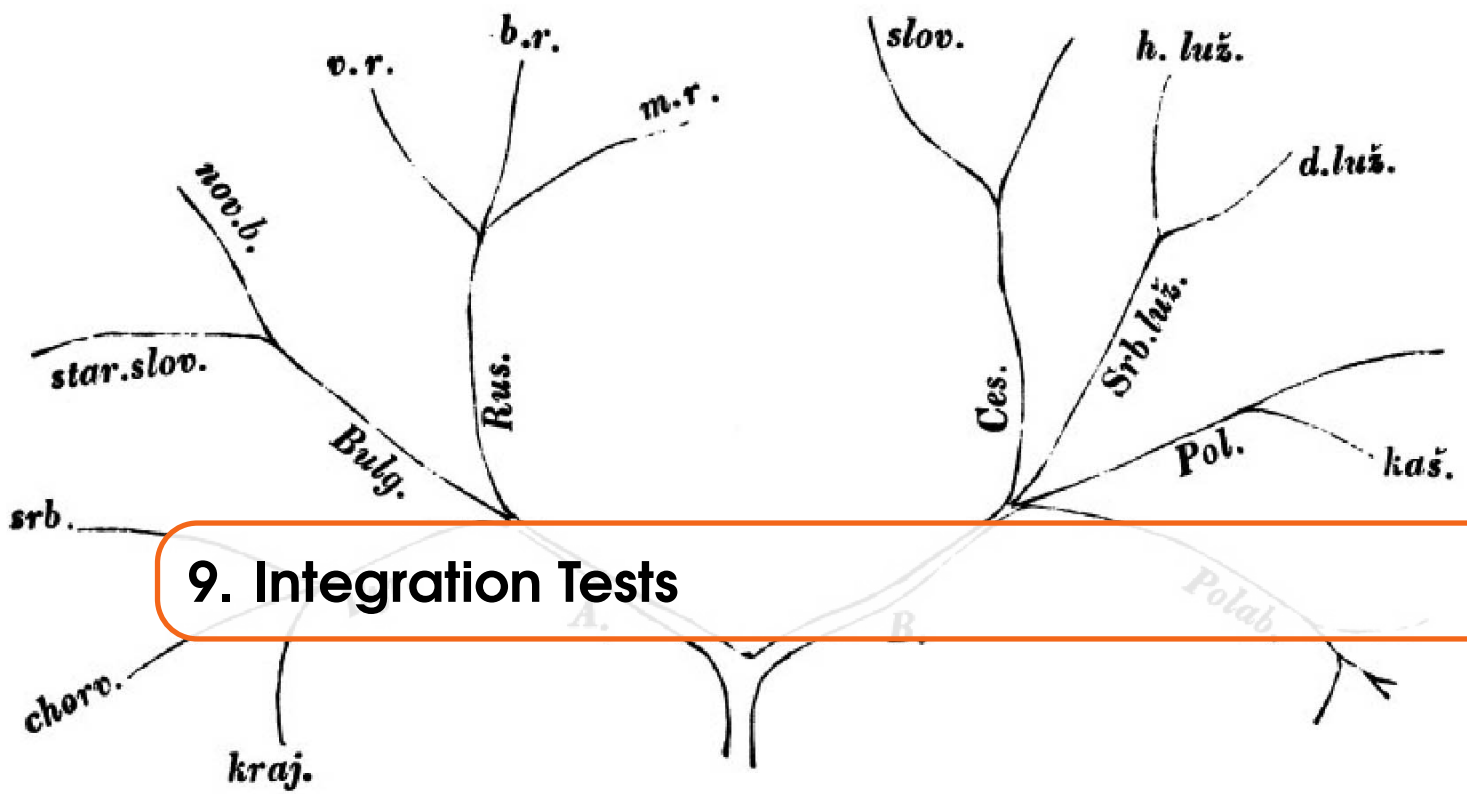


For Unit Tests the methods of the resource are called directly.

```
@Test
public void SimpleTest(){
    String actualResponse = userResource.getIt();
    assertEquals(actualResponse, "User!");
}
```

### Example

[https://github.com/tohotforice/PSE2\\_DH/blob/e364fcb0c164981281c5799a6bf9f9f9ea5eb503/stemmarest/src/test/java/net/stemmaweb/stemmaserver/UserUnitTest.java](https://github.com/tohotforice/PSE2_DH/blob/e364fcb0c164981281c5799a6bf9f9f9ea5eb503/stemmarest/src/test/java/net/stemmaweb/stemmaserver/UserUnitTest.java)



To inject objects into a resource it is mandatory that the resource is created statically at the place the injection is done. This is not possible when the resources are instantiated when a REST call occurs. To solve this JerseyTestServerFactory creates a server where already instantiated resources can be registered. To start a JerseyTestServer a global JerseyTest has to be created.

```
private JerseyTest jerseyTest;
```

The JerseyTestServerFactory creates a JerseyTest with already instantiated resources. This is necessary 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

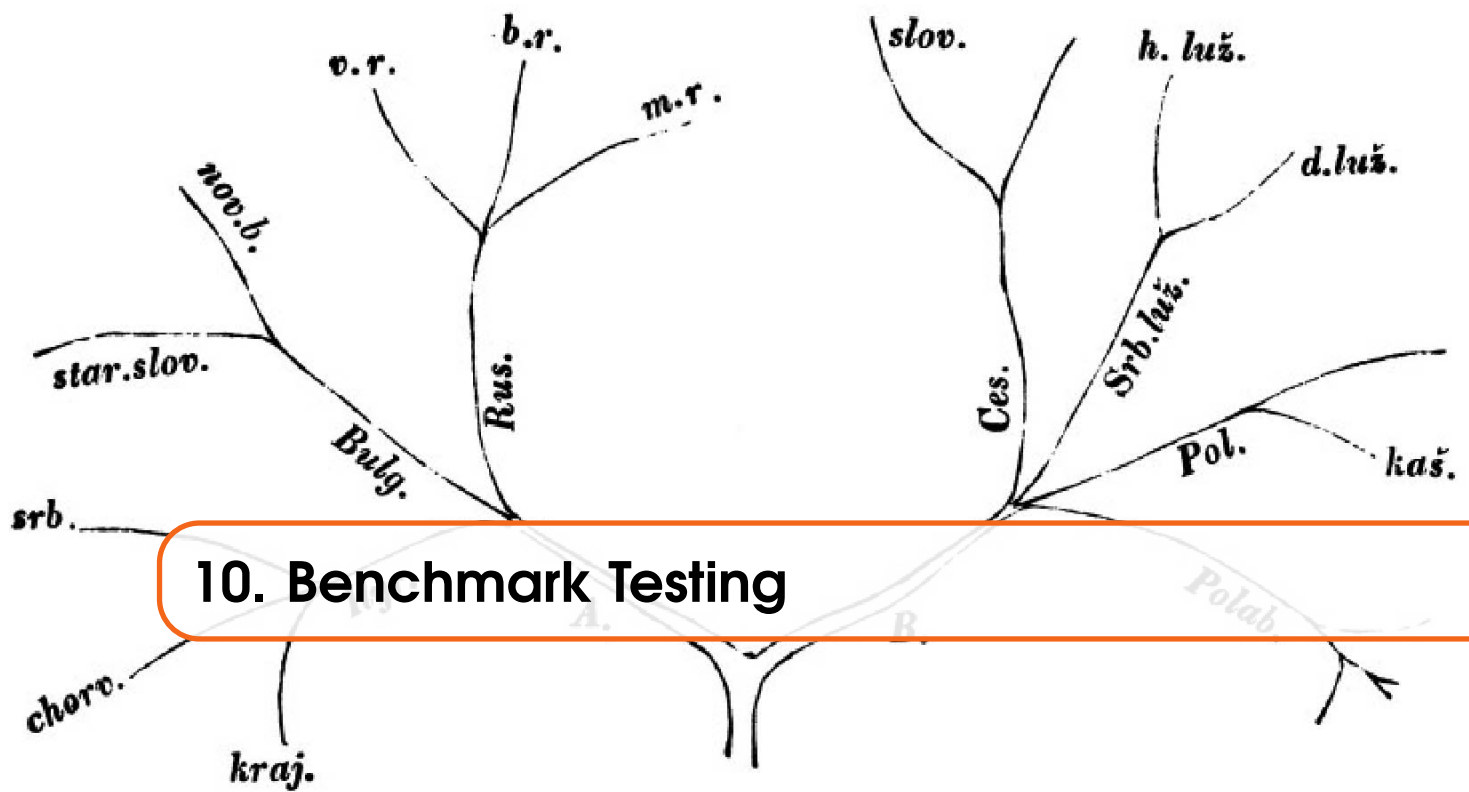
@Test

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

DRAFT



## 10. Benchmark Testing

The main goal of the stemmarest project is to achieve better performance than the previous service. To measure the performance of the system benchmark testing is needed. A benchmark test basically calls the RESTful service multiple times and measures the response time. To achieve this *com.carrotsearch.junitbenchmarks* a handy JUnit benchmark test suite is used. This JUnitbenchmarks measure the time which is used to execute a test and can generate visual representations of the measurement.

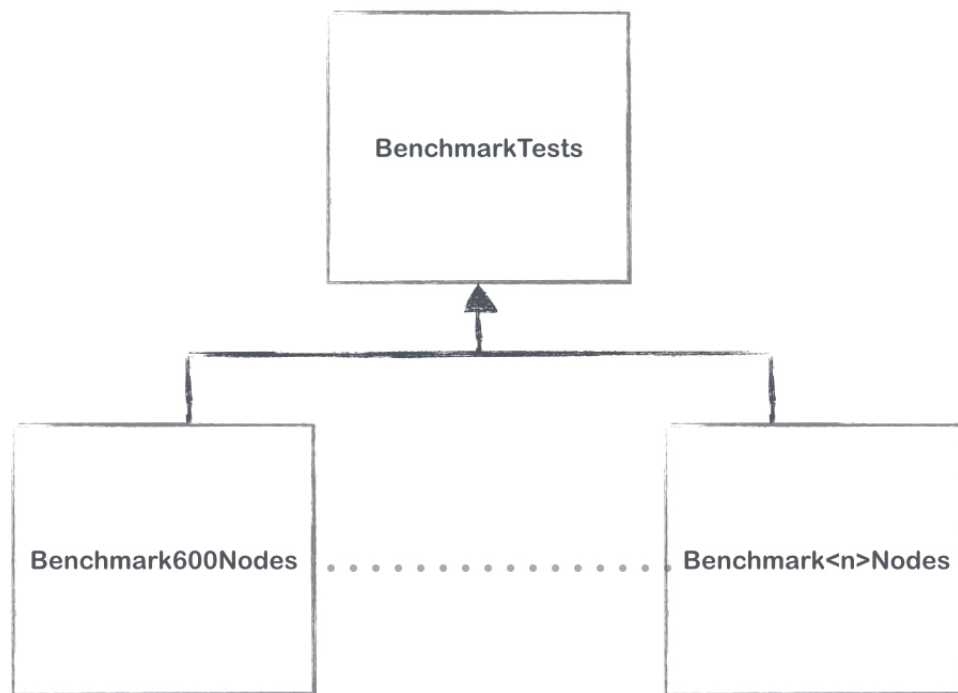
For effective benchmark testing it is important to have a variety of different databases. Those databases should differ in size from small to very huge. This allows to measure the algorithms in extreme situations. To generate valid graphs only limited by space the class *RandomGraphGenerator* can be used. By calling this static method a graph is generated according to the parameters.

**R** Please note that the response time highly depends on the hardware the tests are running on and the actual state of Java's 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.benchmarktests*. The class *BenchmarkTests* contains all Tests. The classes *Benchmark<n>Nodes* contain the database generator. In it it is being configured how many nodes the database has. Those are also the classes which run with the JUnit test. *BenchmarkTests* can't be run as a JUnit test as it is an abstract class.



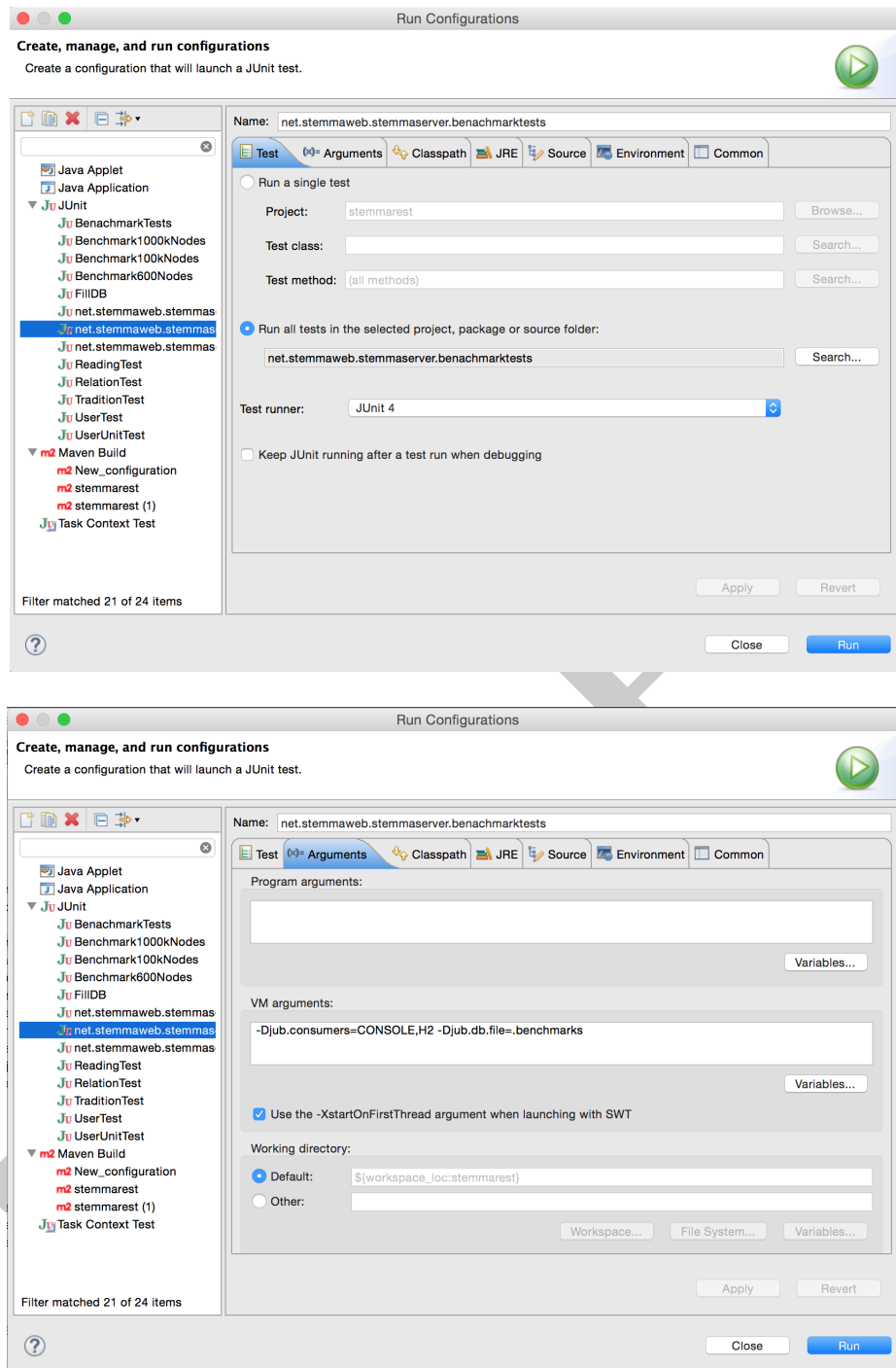
Tests are simply implemented in the `BenchmarkTests` class with the `@Test` annotation. It is best practice only to implement the restcall in this method and only test if `Response.Status` is OK. This assures that as low as possible overhead time will be measured. The integration- and JUnit tests should be done in a different location.

**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, test-environment copy the class `Benchmark600Nodes` and rename it to the count of Nodes that should be inserted. In the class itself only two small adjustments need to be done. First: change the name of the report file `@BenchmarkMethodChart(filePrefix = "benchmark/benchmark-600Nodes")`. Second: adjust the properties of the database which should be generated `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 run as usual. After the test execution the reports are stored under `benchmark/`.

- R** 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 this tests.



# RESTful API

## 11 Services ..... 25

- 11.1 Baseresource:
- 11.2 /stemma
- 11.3 /relation
- 11.4 /tradition
- 11.5 /reading
- 11.6 /witness
- 11.7 /user

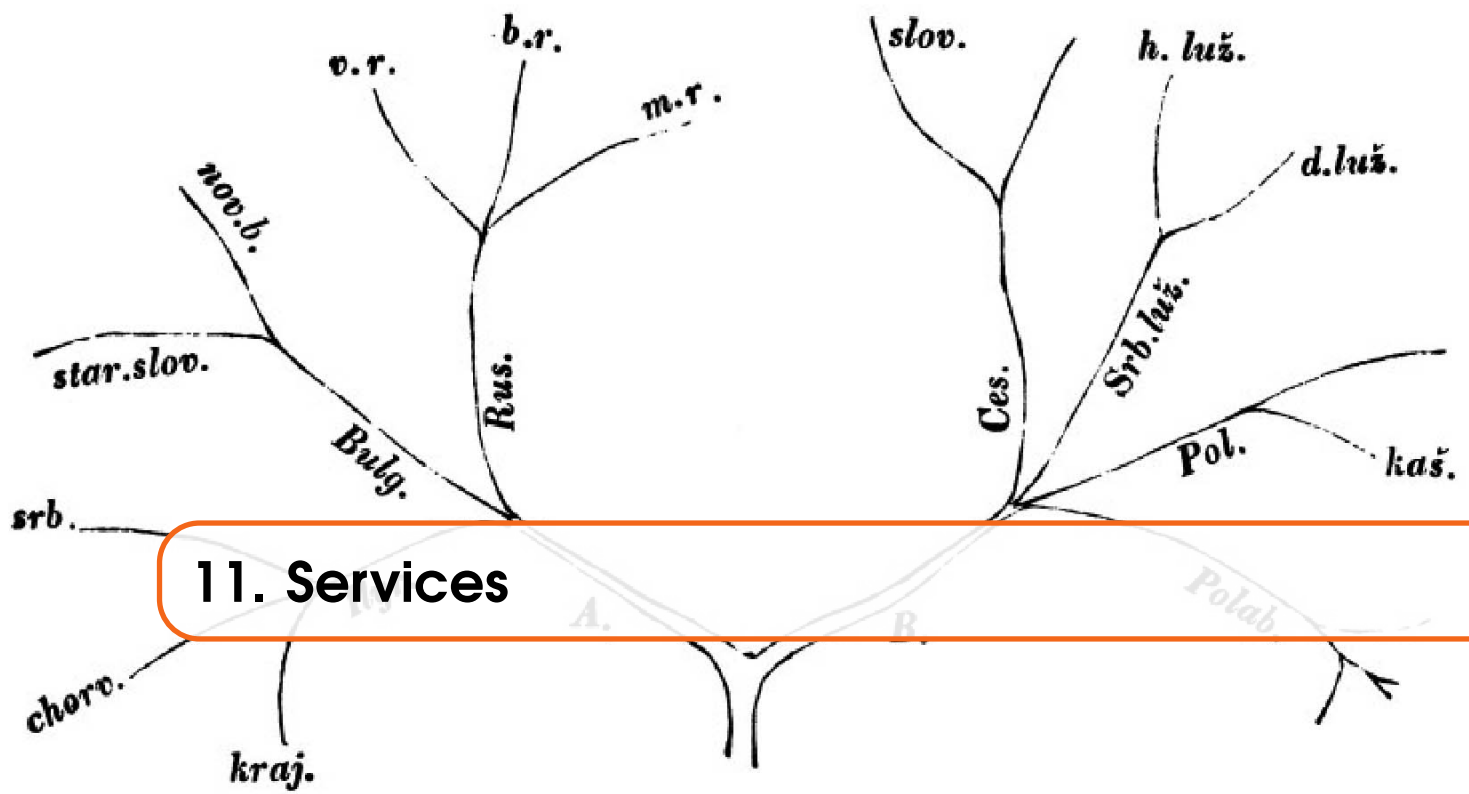
## 12 Models ..... 33

- 12.1 relationshipModel
- 12.2 readingModel
- 12.3 traditionModel
- 12.4 duplicateModel
- 12.5 graphModel
- 12.6 witnessModel
- 12.7 userModel
- 12.8 stemma

## Bibliography ..... 36

- Books
- Articles





## 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}

**Request** as application/json

**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/Relationships.php#L271> 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 were 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** 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 relationshipModel 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** as application/json: XML data

**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** 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** 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:** getIdentialReadings

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** a witness as a 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** a witness as a 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

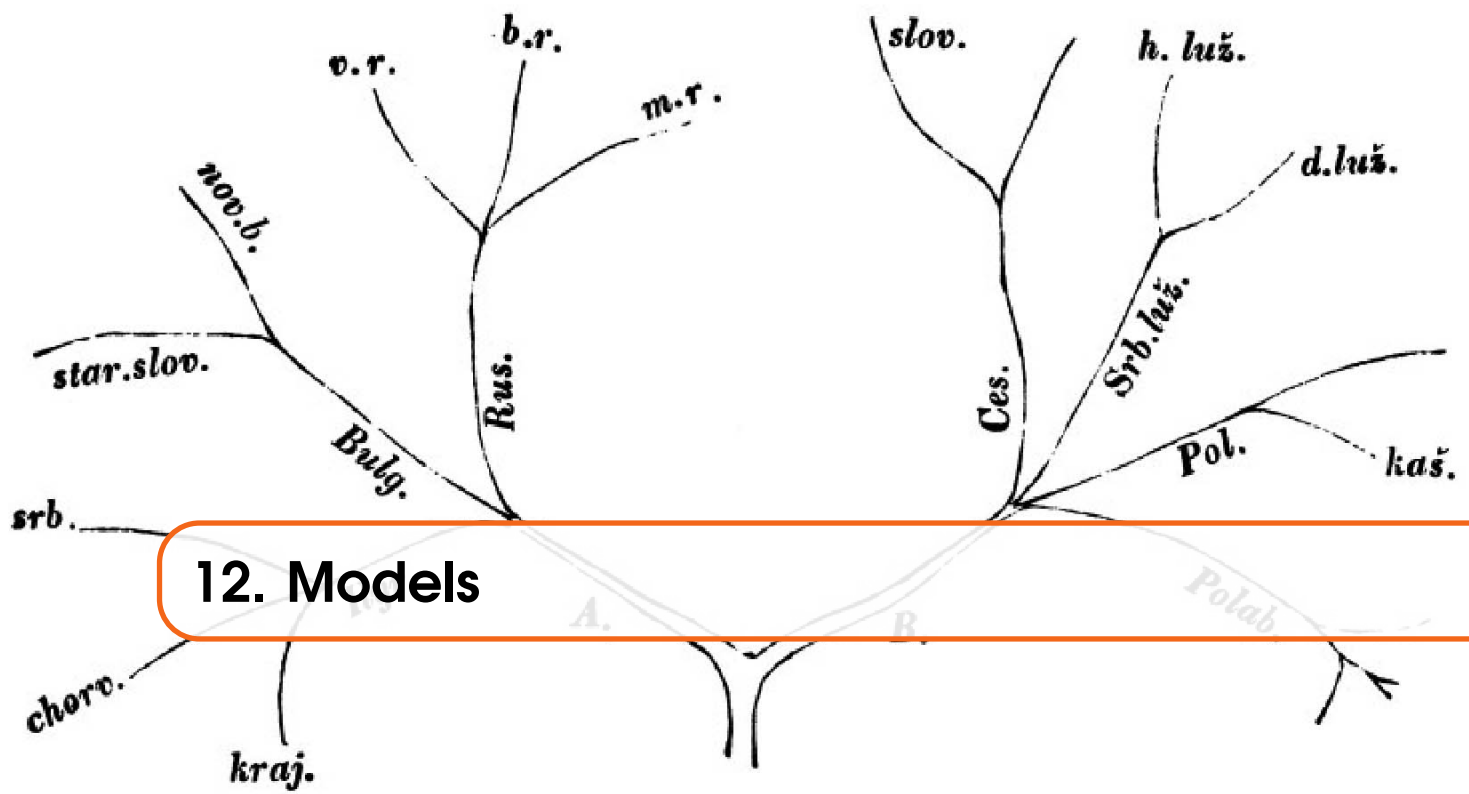
Get all Traditions of a user

**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

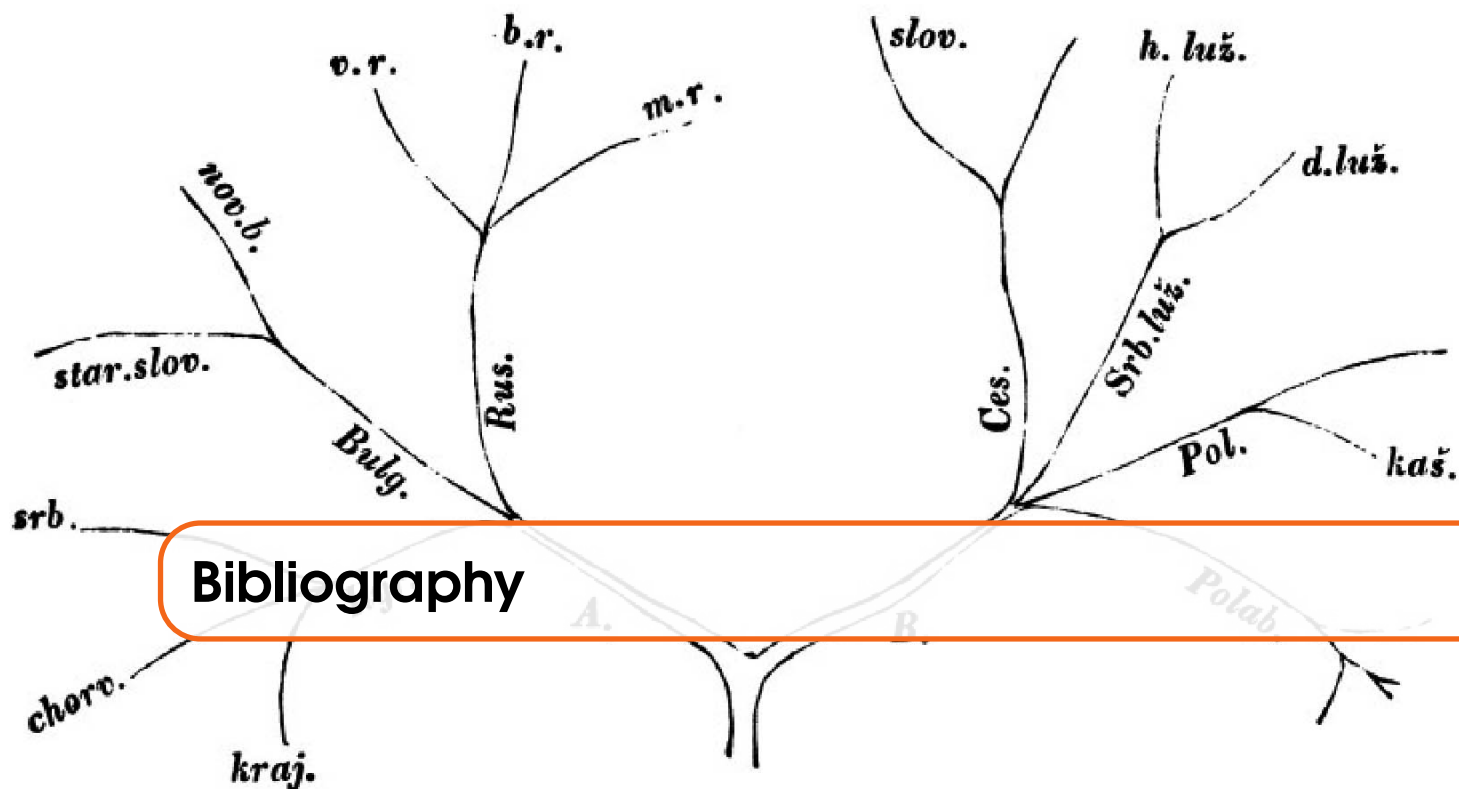
## 12.7 userModel

■ **Property** id as string

■ **Property** isAdmin as string

## 12.8 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.



## Bibliography

Books  
Articles