



# 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

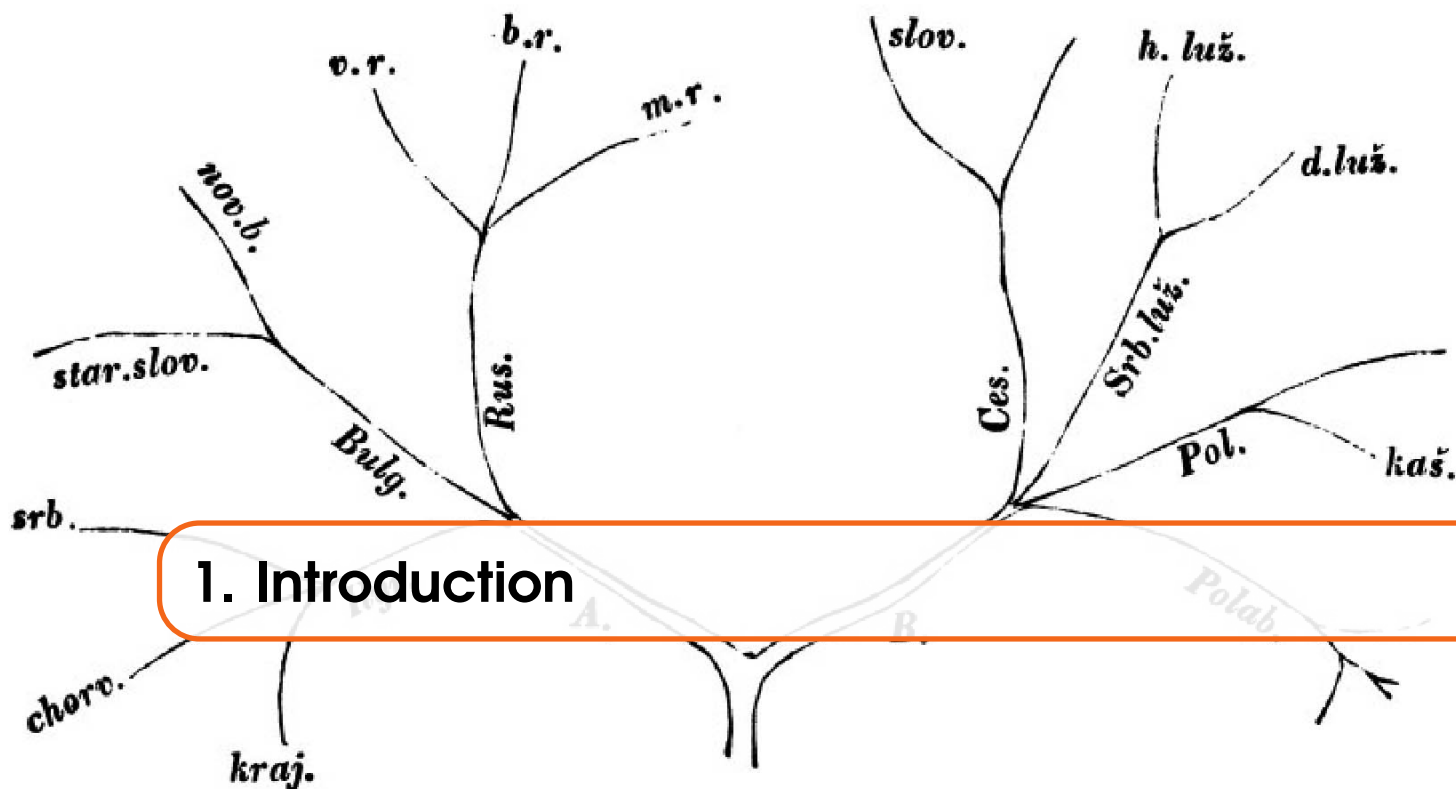
I	Project	
1	Introduction .....	6
2	Database (neo4j) .....	7
3	Jersey .....	9
4	Performance .....	10
II	Testing	
5	Concept .....	14
6	Configure the Test-Database .....	16
7	Unit Test .....	17
8	Integration Tests .....	18
9	Benchmark Testing .....	20

<b>10</b>	<b>Documentation .....</b>	<b>24</b>
10.1	/user	24
10.2	/user/create	24
10.3	/user/{id}	24
10.4	/user/traditions/{userId}	25
10.5	/textinfo/{textId}	25
10.6	/tradition/witness/{tradId}	26
10.7	/tradition/relation/{tradId}/relationships	26
10.8	/tradition/new	26
10.9	/tradition/get/{tradId}	27
10.10	/tradition/reading/{tradId}/{readId}	27
10.11	/tradition/duplicate/{tradId}	28
<b>11</b>	<b>Services .....</b>	<b>29</b>
11.1	Baseresource:	29
11.2	/user	29
11.3	/relation	30
11.4	/witness	30
11.5	/tradition	32
11.6	/myresource	34
<b>12</b>	<b>Models .....</b>	<b>35</b>
12.1	userModel	35
12.2	relationshipModel	35
12.3	duplicateModel	36
12.4	textInfoModel	36
	<b>Bibliography .....</b>	<b>37</b>
	Books	37
	Articles	37

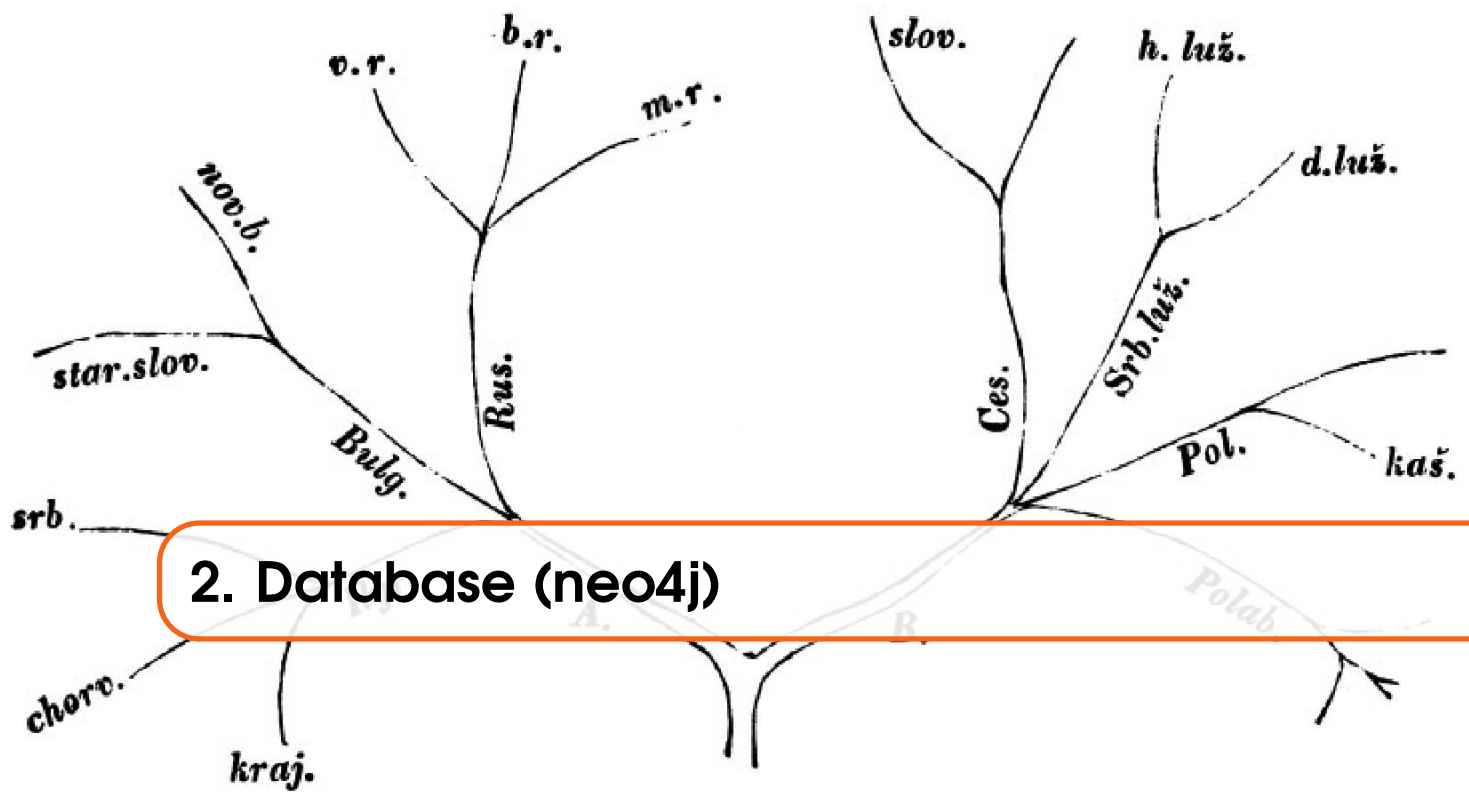


# Project

<b>1</b>	<b>Introduction .....</b>	<b>6</b>
<b>2</b>	<b>Database (neo4j) .....</b>	<b>7</b>
<b>3</b>	<b>Jersey .....</b>	<b>9</b>
<b>4</b>	<b>Performance .....</b>	<b>10</b>



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. Database (neo4j)

In our project we used a graph database instead of a relational database.

This was due to the fact, that relational databases are much faster when it comes down to look for objects in a list that fulfill some constraints to other objects. Normally in a relational database you would use multiple joins to get your desired result. In the graph way it's much easier because you traverse the graph (previously mentioned as list) from top down using either breadth- or depth-first algorithm and look for a specific relation between two nodes (previously objects). In respect of that we save a lot of time traversing instead of writing complicated queries which run mostly slow.

Therefore we use Neo4J (Additional info can be found under <http://neo4j.com/developer/graph-db-vs-rdbms/>). Neo4J is a graph database that is capable of managing millions of nodes and relationships and returning or changing them within logarithmic or even constant time. That means that it won't make a big difference whether you use 10 nodes, or 1 million. You can find additional information on benchmarking in the related chapter of this documentation.

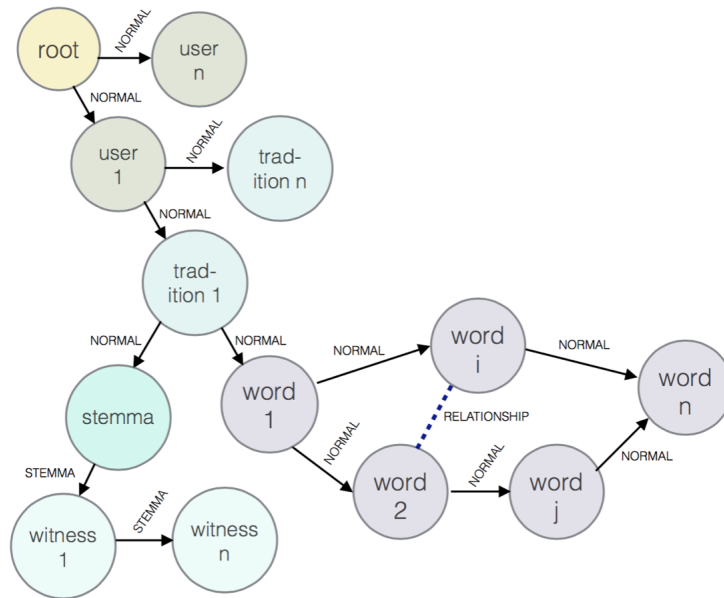
Our database is mainly one big graph. We use different Labels for Nodes and Relationships to increase the search speed.

In the database we use only a few labels:

Nodes	Relationships
ROOT	RELATIONSHIP
STEMMA	STEMMA
WITNESS	NORMAL
TRADITION	
USER	
WORD	

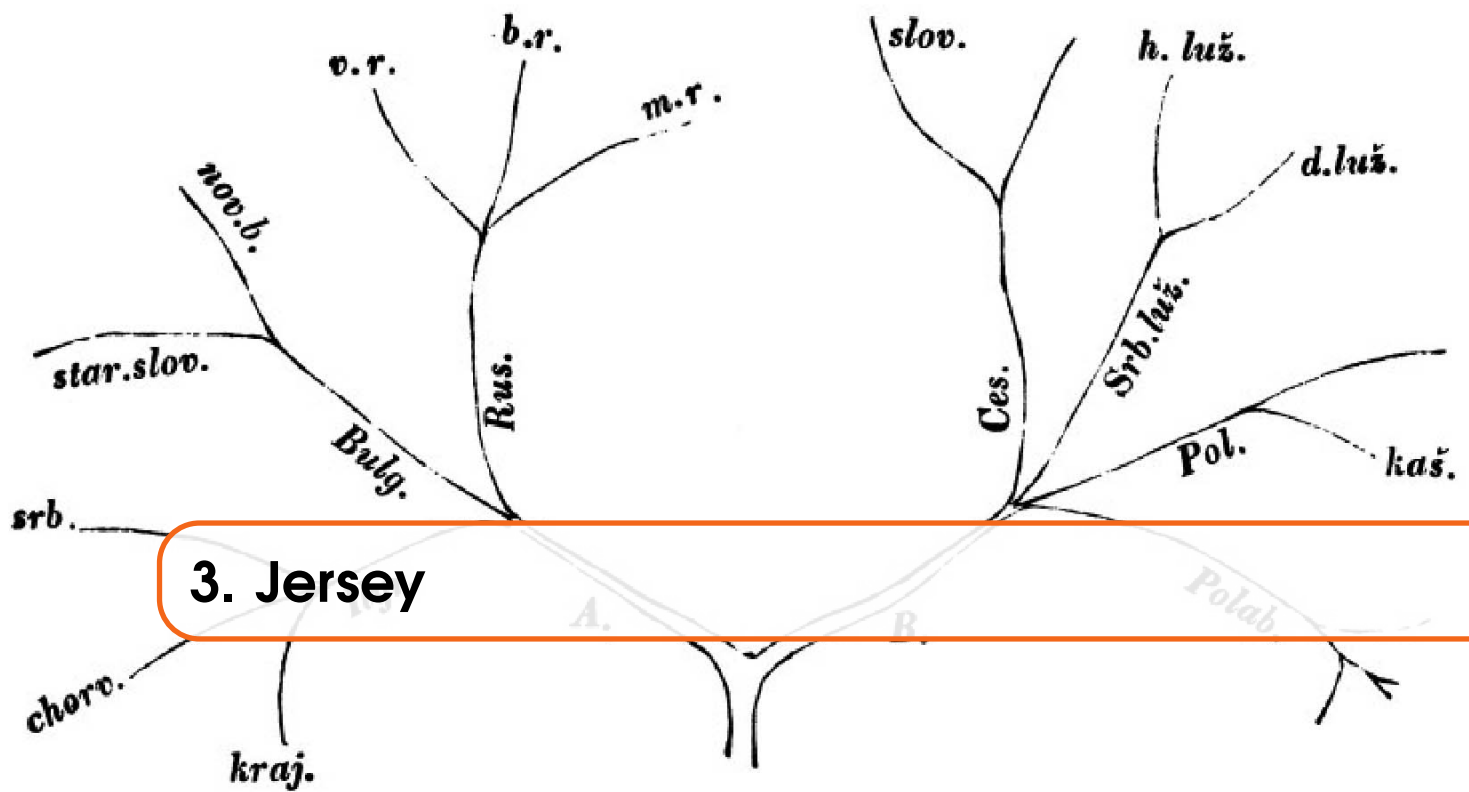
Since each label is stored in another file, searching or traversing the graph is stunningly fast.

The database structure is as follows:



Neo4J is delivered with 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 you can write simple queries that return nodes or traverse graphs. Cypher queries can be sent to the database and will then be interpreted and translated into an execution plan by the ExecutionEngine. This takes some time and therefore cypher can not compete with the native java traversal API.





### 3. Jersey

In our project we built a REST-API for stemmaweb an online tool for textual scholars willing to explore their texts. For this REST-API we needed a simple and easy to use Java framework and chose Jersey.

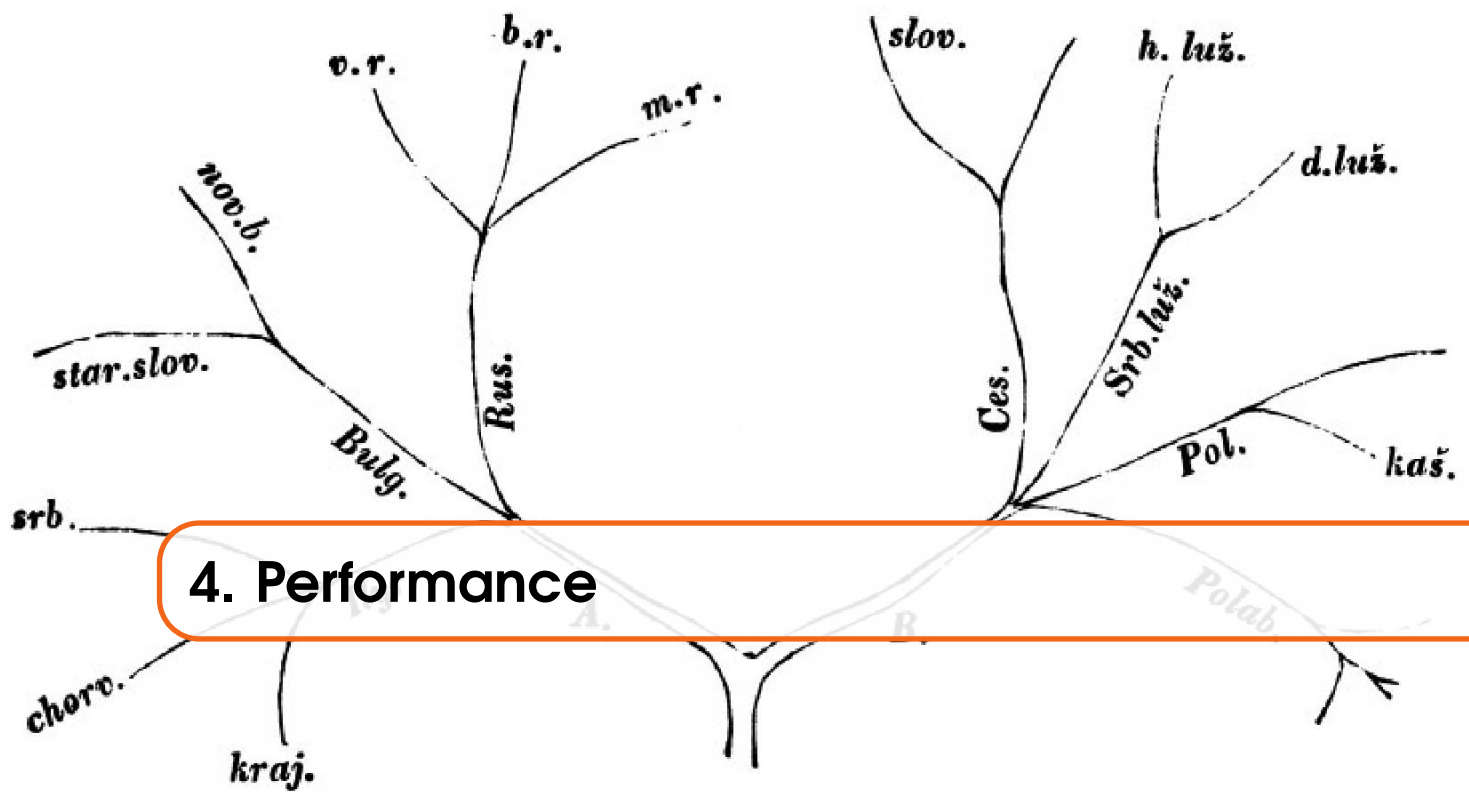
Jersey is an open source framework for developing RESTful Web Services in Java that is built upon JAX-RS and serves as a JAX-RS Reference Implementation. It adds additional features and utilities in order to further simplify development of REST-APIs. It abstracts away low-level details of the client-server communication which made it more easy for us to concentrate on the actual implementation of the user stories. In our project it is deployed with GlassFish, a Java EE Application Server. Jersey can help support exposing the data in very different media types, including JSON, which we used very frequently.

Jersey uses Annotations which made it very easy to use for us.

Here as an example the method declaration of duplicateReading:

```
@POST
@Path("duplicatereading/fromtradition/{tradId}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response duplicateReading(DuplicateModel duplicateModel)
```

At the top the „POST“ annotation states the http method. Then the „Path“ annotation lets us set the url path with parameters in curly braces. Furthermore the method can „consume“ data, in this case JSON. The „DuplicateModel“ is passed with the call as a JSON object and then gets parsed in a POJO. Last but not least the method „produces“ a response, in this case also in JSON.



## 4. Performance

One of the main goals of the redesign is to create a faster RESTful service. To assure the speed of the service some performance tests are done.

The aim of the performance tests is to show that the response time of the service is limited and within a usable range. So the time to execute all operations for a certain request is measured. This contains 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 are transmitted over the local loop interface, so the network speed is not measured.

To do the tests the database is populated by a random graph which contains several valid traditions on which the REST requests can be executed. Several tests with databases of a different size are done to show that the response time does not change with an increasing database size. (Actually there are several search by ID requests 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 in the size of database expected for an operational database.)

In the following measurement series the dimension of the database size is tested. This 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 which allows to work on a subgraph without filtering the whole database.

- R** The implementation of stemma rest uses some search node by id methods which search over the complete database but those are in  $O(\log n)$  and are not seen in the noise of the other operations. In a much bigger Database those methods will slow down the REST requests. But it is not expected that the database will grow that big, that this operations will have any impact.

The previous measurement series tested the dimension of the database size and could show that the response time is almost independent of it for the expected database size of something between

Figure 4.1: Database with 1000 nodes, working tradition with 100 nodes

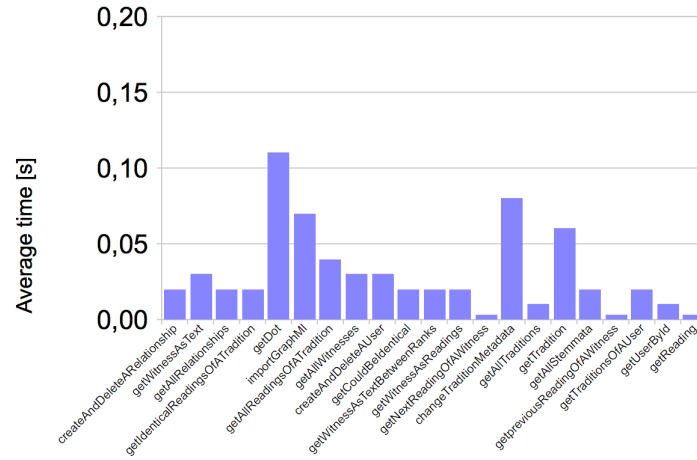


Figure 4.2: Database with 100000 nodes, working tradition with 100 nodes

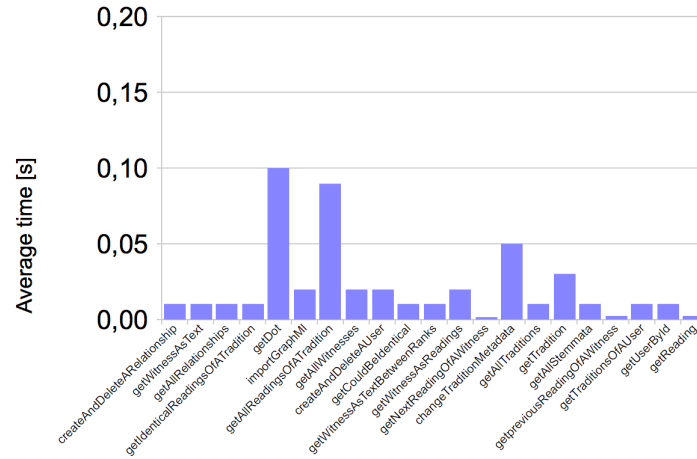
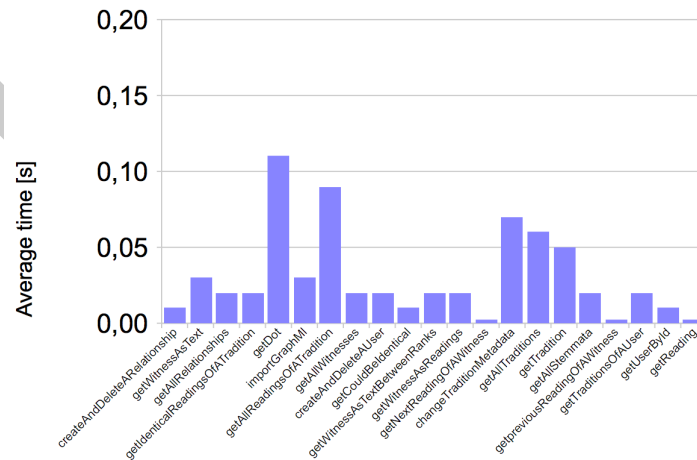


Figure 4.3: Database with 1000000 nodes, working tradition with 100 nodes



1000 and 1 Million Nodes (Readings). This independence results out 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 obviously that those algorithms are not independent of the tradition size as the working subset grows with a bigger tradition. Most of the algorithms to work on a tradition are in  $O(\log n)$  but 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 test series the dimension of the tradition size is varied. This measurement

Figure 4.4: Database with 10000 nodes, working tradition with 1000 nodes

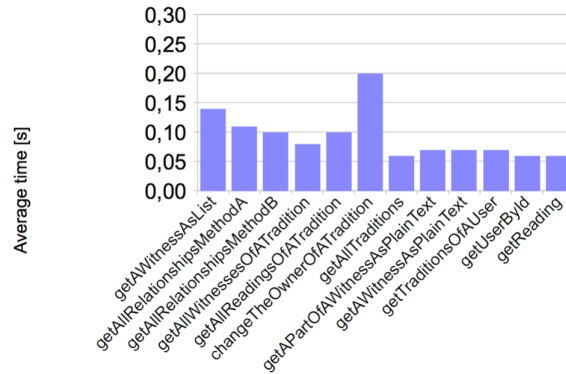
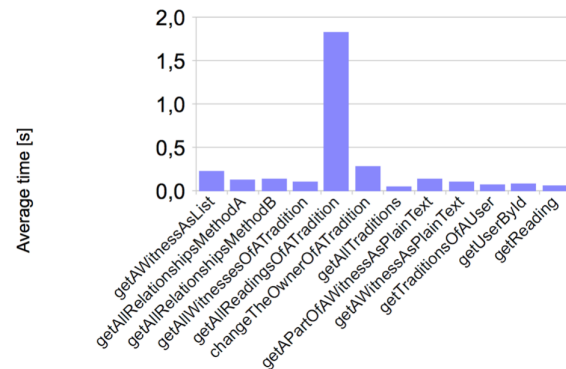
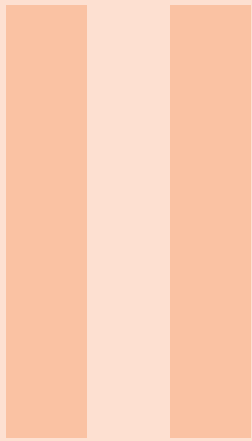


Figure 4.5: Database with 10000 nodes, working tradition with 10000 nodes

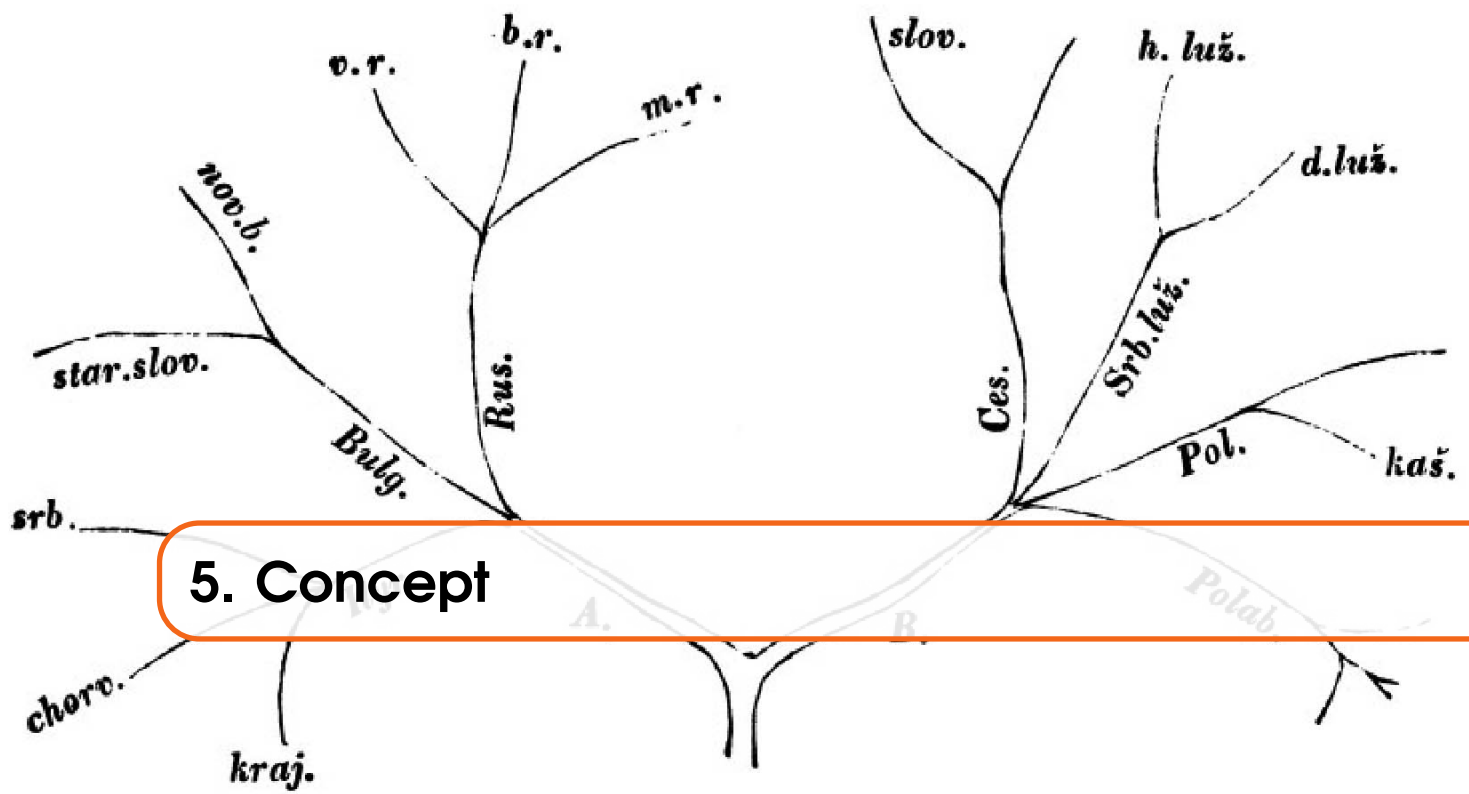


shows that the execution time depend on the size of the working tradition. In this test result the `getAllReadingsFromATradition` each reading of the tradition is parsed to a JSON Object and all are returned as an List. The parsing of those nodes executes in  $O(n)$  and the downloading of the JSON file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



# Testing

5	Concept .....	14
6	Configure the Test-Database .....	16
7	Unit Test .....	17
8	Integration Tests .....	18
9	Benchmark Testing .....	20



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 a manner they don't have any impact on the architecture of the project.

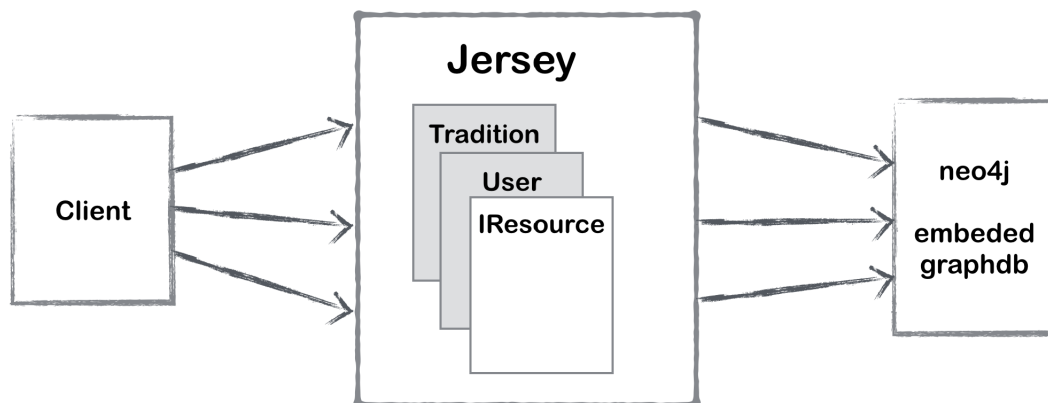
### Integration Tests

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

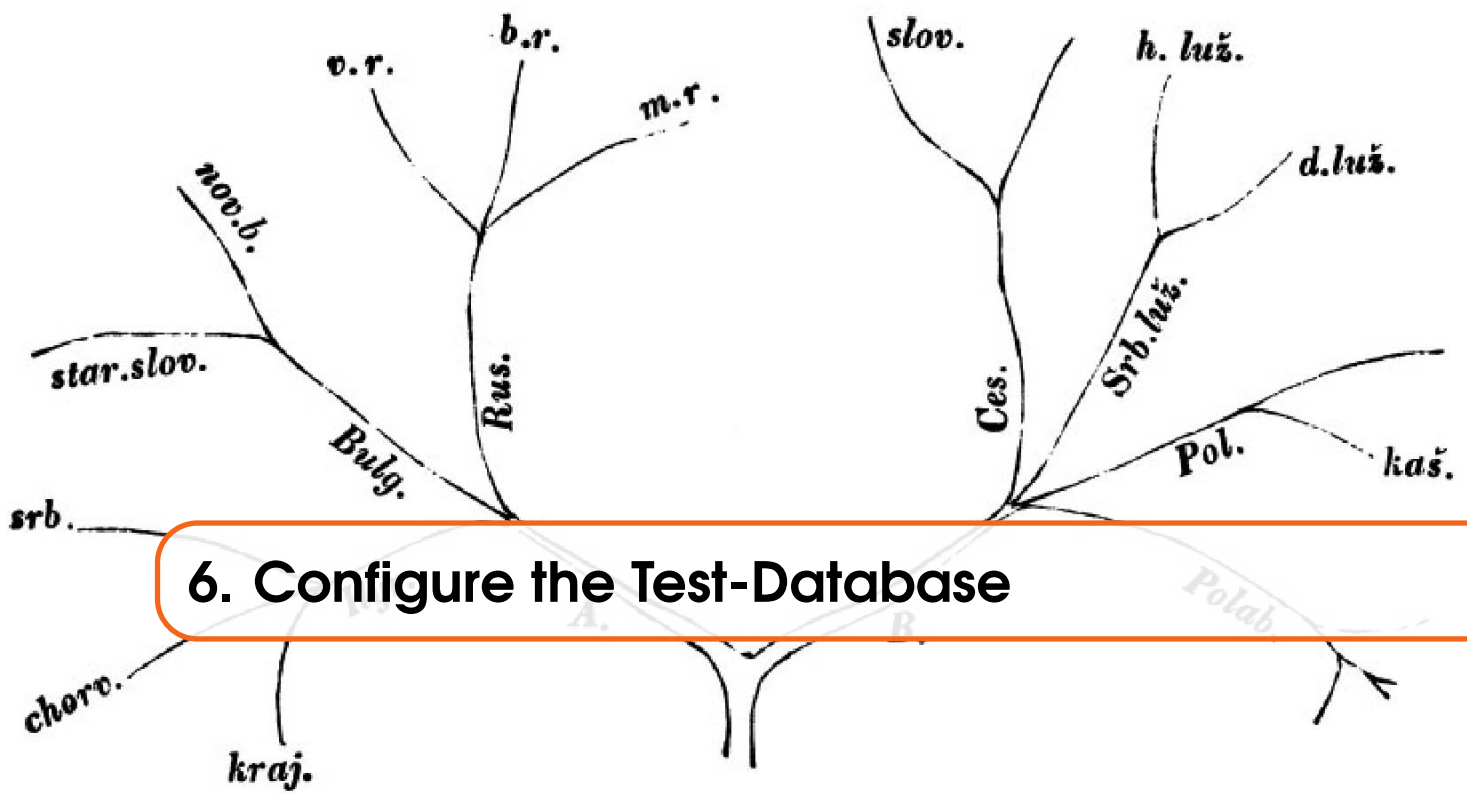
### Unit Tests

Unit tests are used for test driven development and are defined by the developer. They are only for the development process and are not referenced in quality audits.

## Jersey Overview



In the productive system for every REST call jersey is instantiating the requested resource and providing the service. Each resource object has its own database service, which is closed after each call. In production a embedded neo4j GraphDatabase is used. 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 can be configured to return an impermanent Database.



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:

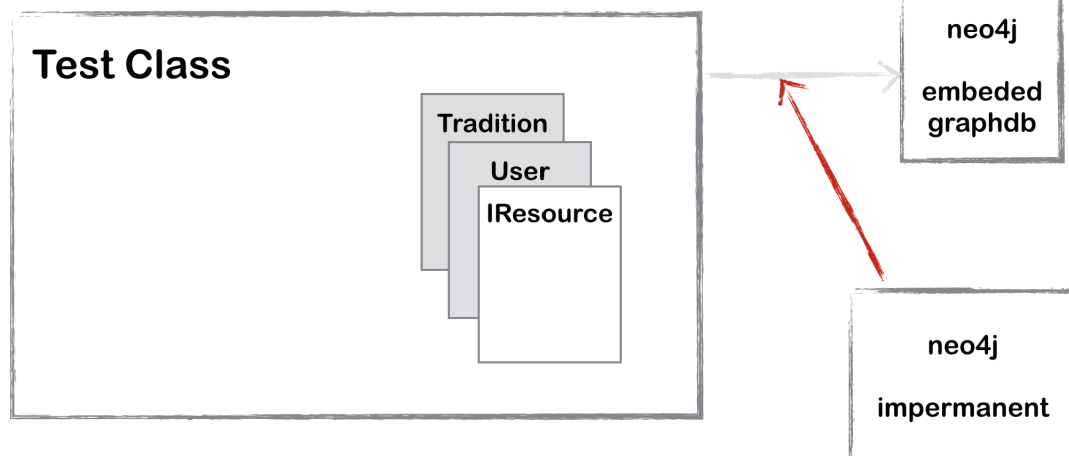
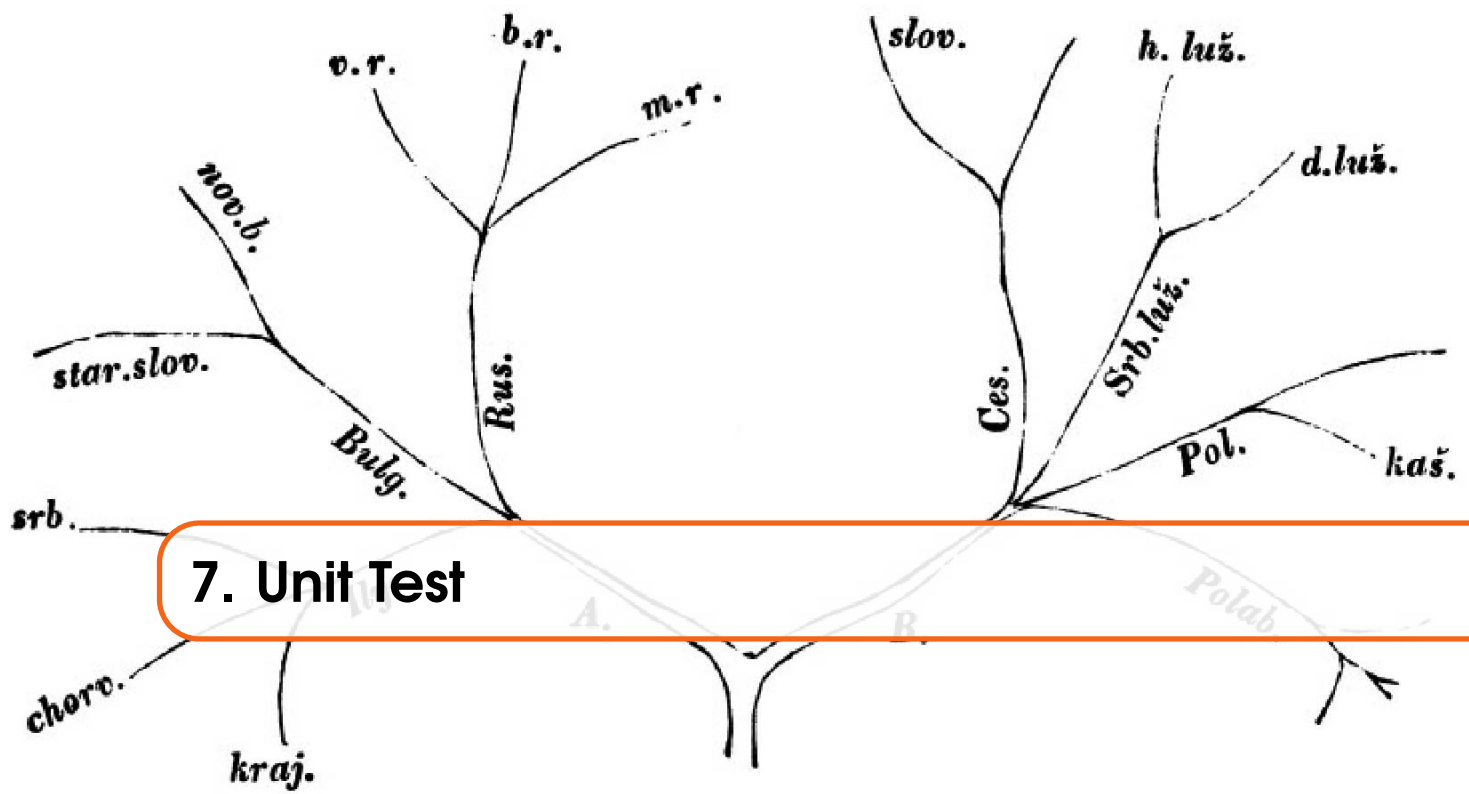
```
db = new GraphDatabaseServiceProvider().getDatabase();
```

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.



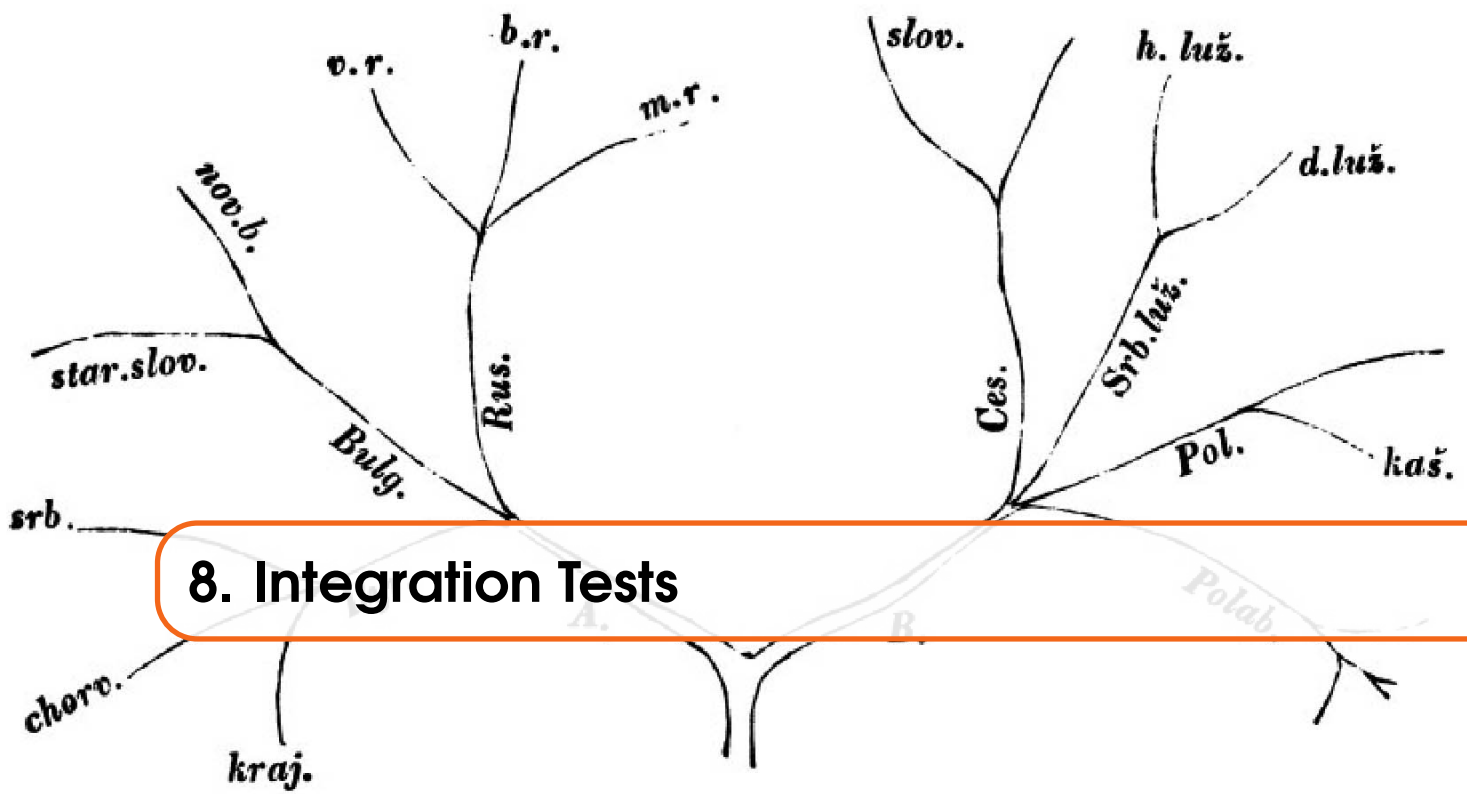


For Unit Tests the methods of the resource are called directly.

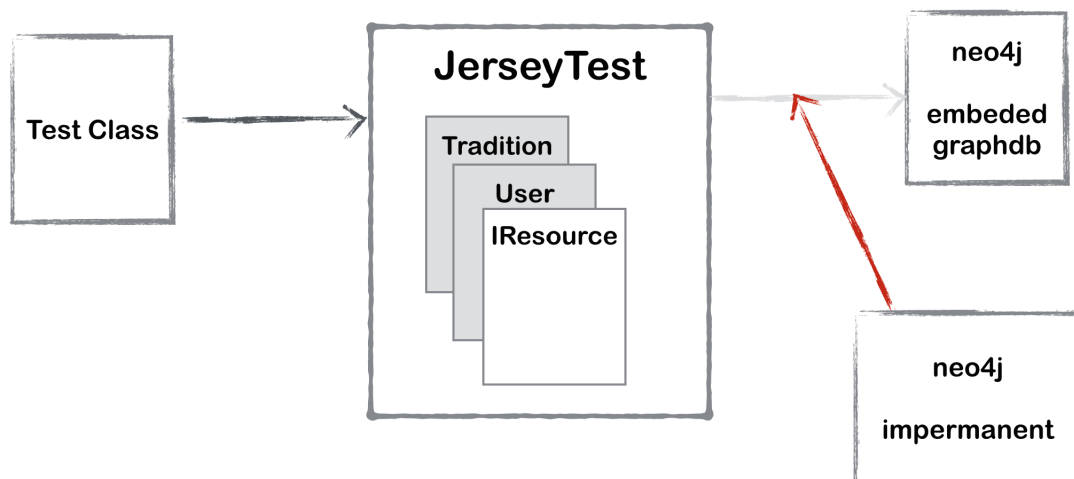
```
@Test
public void SimpleTest(){
    String actualResponse = userResource.getIt();
    assertEquals(actualResponse, "User!");
}
```

### 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)



## 8. Integration Tests



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

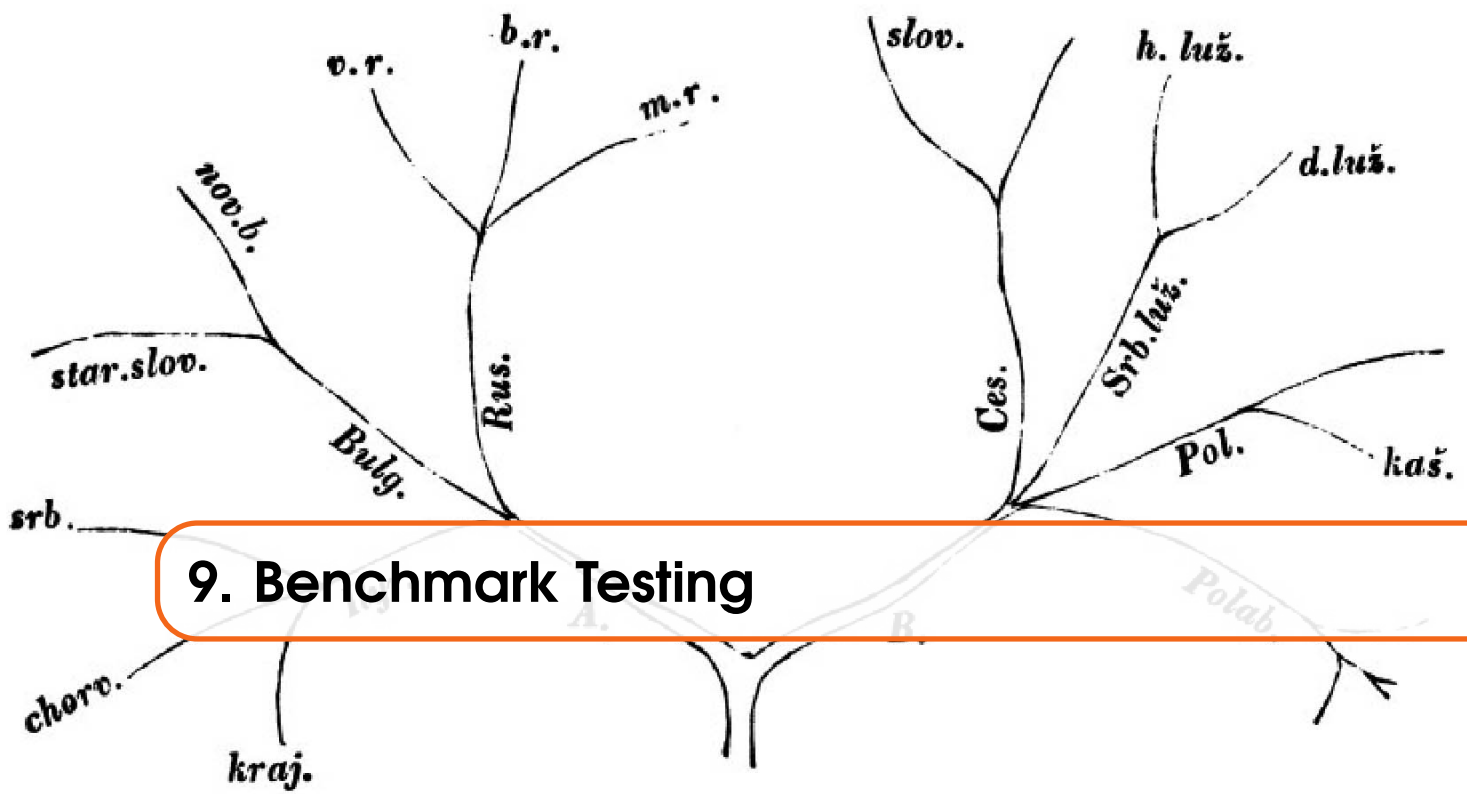
```
@Test
```

```
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



## 9. Benchmark Testing

A main goal of the PSE2 stemmarest project is a good performance compared to the previous RESTful service. To measure the performance benchmark testing is needed. A benchmark test basically calls the RESTful service multiple times and measure 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 the benchmark testing it is of interest to have a variety of different databases. Those databases should differ in their size from small to very huge. This allows to measure the algorithms in extreme situations. To generate valid graphs only limited by disk space the class *RandomGraphGenerator* can be used. By calling the static method *role* a graph is generated according to the parameters.

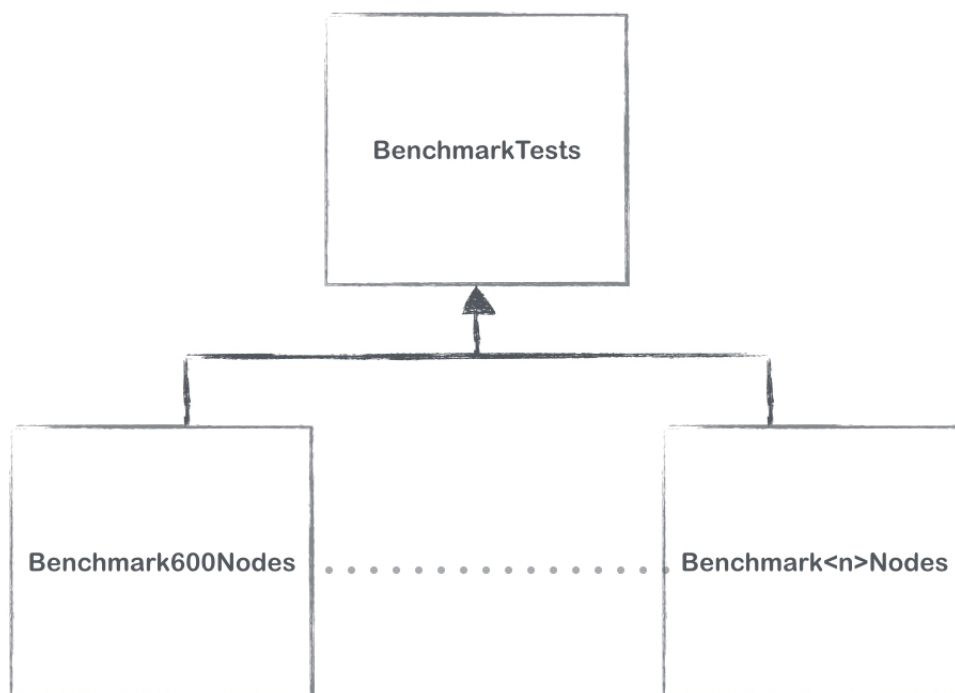


Please note that the response time highly depends on the hardware the tests are running on and the actual state of Java's virtual machine.

To reduce the influence of the virtual machine before the measurements 5 warm-up calls are done to bring the virtual machine to live. 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 generation. Here is configured how many nodes the database has. These are also the classes which are run with JUnit test. *BenchmarkTests* can't be run as a JUnit test as it is an abstract class. s



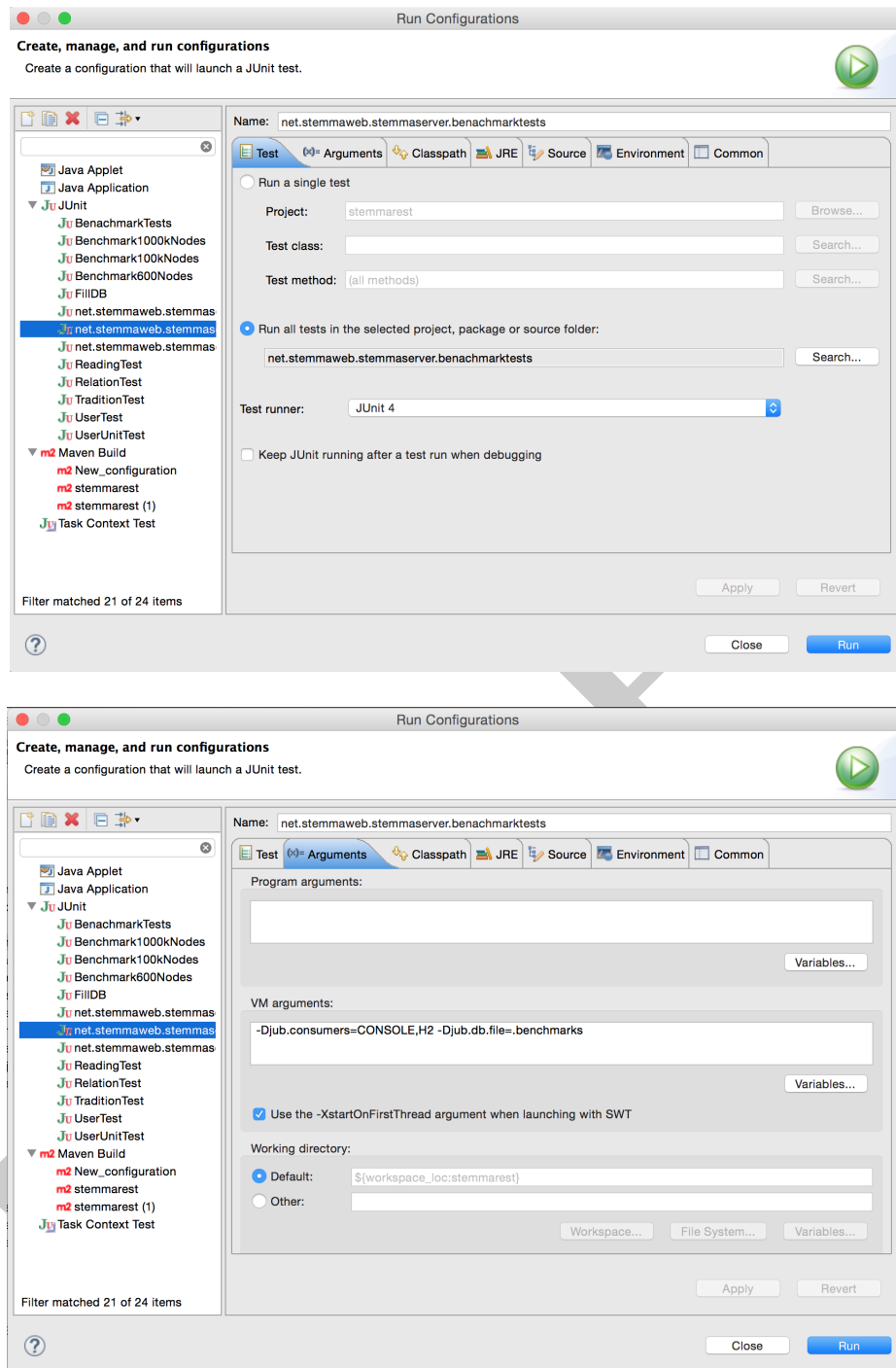
Tests are simply implemented in the `BenchmarkTests` class with the `@Test` annotation. It is best practice only to implement the restcall in this method and only test if `Response.Status` is OK. This assures that as less as possible overhead time is measured. And the integration- and JUnit tests should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

**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. But to generate the report an argument needs to be passed by. 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 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 generation of huge graphs. Its recommended not to use the computer during this tests.

# RESTful API

## 10 Documentation ..... 24

- 10.1 /user
- 10.2 /user/create
- 10.3 /user/{id}
- 10.4 /user/traditions/{userId}
- 10.5 /textinfo/{textId}
- 10.6 /tradition/witness/{tradId}
- 10.7 /tradition/relation/{tradId}/relationships
- 10.8 /tradition/new
- 10.9 /tradition/get/{tradId}
- 10.10 /tradition/reading/{tradId}/{readId}
- 10.11 /tradition/duplicate/{tradId}

## 11 Services ..... 29

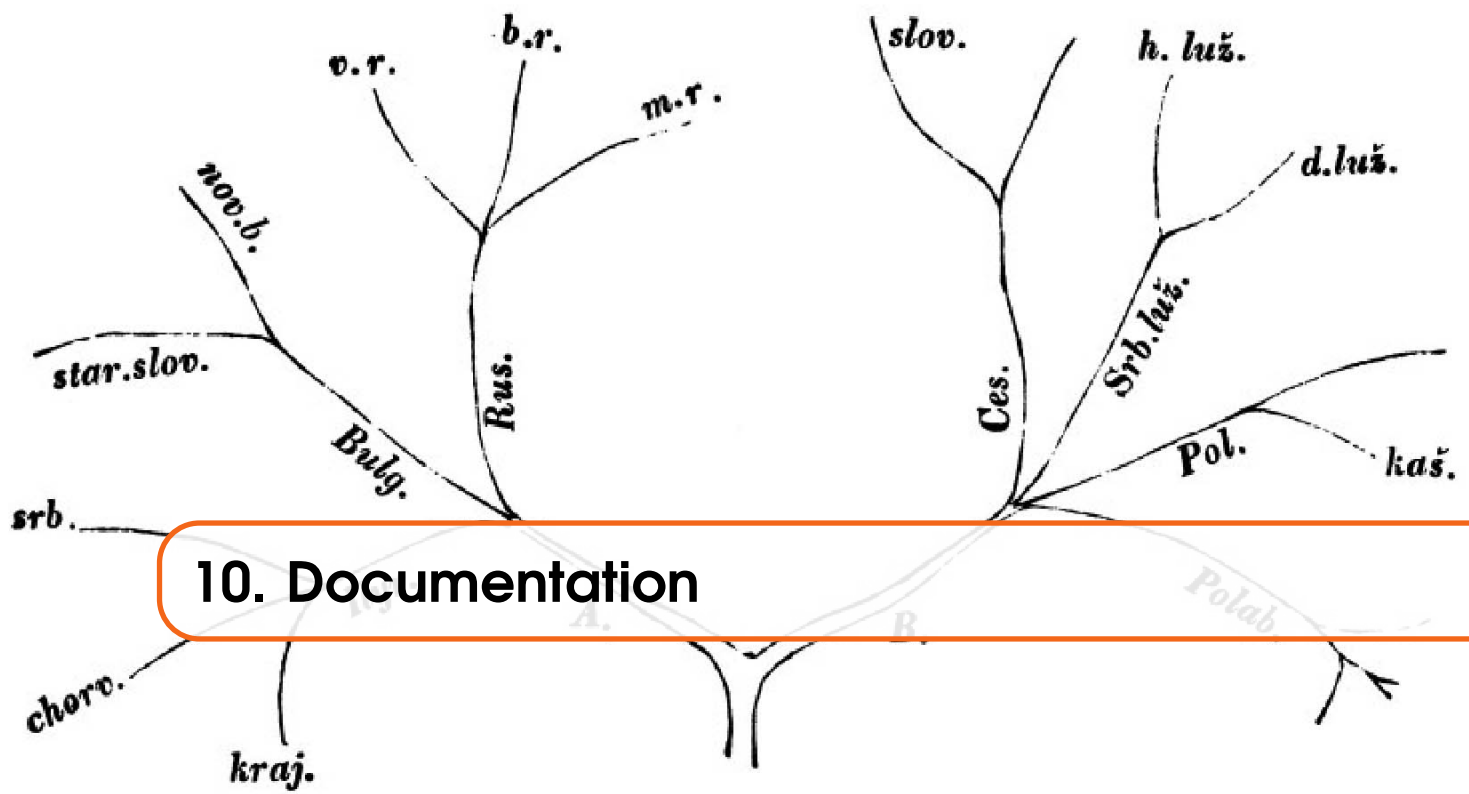
- 11.1 Baseresource:
- 11.2 /user
- 11.3 /relation
- 11.4 /witness
- 11.5 /tradition
- 11.6 /myresource

## 12 Models ..... 35

- 12.1 userModel
- 12.2 relationshipModel
- 12.3 duplicateModel
- 12.4 textInfoModel

## Bibliography ..... 37

- Books
- Articles



## 10. Documentation

### 10.1 /user

**GET** /user

#### Summary

Returns a welcome message

#### Parameter

**Return** — SUCCESS. text/plain

"User!"

### 10.2 /user/create

**POST** /user/create

#### Summary

Creates a user.

#### Parameter application/json

{ "userId":<userId>, "isPublic":<isAdmin> }

**Return** — CREATED. application/json

{ "userId":<userId>, "isPublic":<isAdmin> }

**Return** — CONFLICT. application/json

Error: A user with this id already exists

### 10.3 /user/{id}



**GET** /user/{id}

### Summary

Returns the user as JSON Object

**Parameter URL**

Id: the user id

**Return — OK. application/json**

```
{ 'userId': <userId>, 'isAdmin': <isAdmin> }
```

*The information about the user*

**Return — NOT\_FOUND. application/json**

*The information about the user*

## 10.4 /user/traditions/{userId}

**GET** /user/traditions/{userId}

### Summary

List all Traditions of a user

**Parameter URL**

userId: the id of the user

**Return — OK. application/json**

```
{ "traditions": [ { "name": <traditionName> } ] }
```

**Return — NOT\_FOUND. application/json**

Error: A user with this id does not exist!

## 10.5 /textinfo/{textId}

**POST** /textinfo/{textId}

### Summary

Update the textInfo of a tradition.

**Parameter URL**

textId: the id of the tradition

**Parameter application/json**

```
{ 'name': <new_name>, 'language': <new_language>, 'isPublic': <is_public>, 'ownerId':  
<new_ownerId> }
```

**Return — SUCCESS. application/json**

```
{ 'name': <new_name>, 'language': <new_language>, 'isPublic': <is_public>, 'ownerId':  
<new_ownerId> }
```

*The new information of the tradition.*

**Return** — **CONFLICT**. **application/json**  
"Error: A user with this id does not exist"  
*If the user does not exist.*

**Return** — **NOT\_FOUND**. **application/json**  
*If the tradition was not found.*

## 10.6 /tradition/witness/{tradId}

**GET** /tradition/witness/{tradId}

### Summary

List all Witness of a tradition

**Parameter URL**  
tradId: the id of the tradition

**Return** — **OK**. **application/json**

**Return** — **NOT\_FOUND**. **application/json**  
Error: A tradition with this id does not exist!

## 10.7 /tradition/relation/{tradId}/relationships

**GET** /tradition/relation/{tradId}/relationships

### Summary

List all Relationships of a tradition

**Parameter URL**  
tradId: the id of the tradition

**Return** — **OK**. **application/json**

## 10.8 /tradition/new

**POST** /tradition/new

### Summary

Create a new tradition.

**Parameter text/plain**  
name: The name of the tradition

**Parameter multipart/form-data**  
language: The language of the tradition  
public: 0 if the tradition is not public 1 if the tradition is public  
name: The name of the tradition

file: multipart file input stream

**Return — CONFLICT. application/json**

"Error: No user with this id exists"

**Return — INTERNAL\_SERVER\_ERROR. application/json**

"Error: Tradition could not be imported!" *If the server was not able to parse the input file*

**Return — OK. application/json**

"Tradition imported successfully"

## 10.9 /tradition/get/{tradId}

**GET** /tradition/get/{tradId}

### Summary

Get a graphml of a tradition

**Parameter URL**

tradId: the id of the tradition

**Return — OK. application/xml**

**Return — NOT\_FOUND. application/json**

Error: A tradition with this id does not exist!

## 10.10 /tradition/reading/{tradId}/{readId}

**GET** /tradition/reading/{tradId}/{readId}

### Summary

Get a specific reading of a tradition

**Parameter URL**

tradId: the id of the tradition

**Parameter URL**

readId: the id of the reading

**Return — OK. application/json** The reading in json

**Return — NOT\_FOUND. application/json**

Error: A tradition with this id does not exist!

**Return — NOT\_FOUND. application/json**

Error: A reading with this id does not exist!

### 10.11 /tradition/duplicate/{tradId}

**POST** /tradition/duplicate/{tradId}

#### Summary

Duplicates readings in a tradition. Is the opposite method of merge.

#### Parameter URL

tradId: the id of the tradition

#### Parameter application/json

```
{ 'readings': [readId1, readId2], 'witnesses': ["wit1", "wit2"] }
```

#### Return — OK. application/json

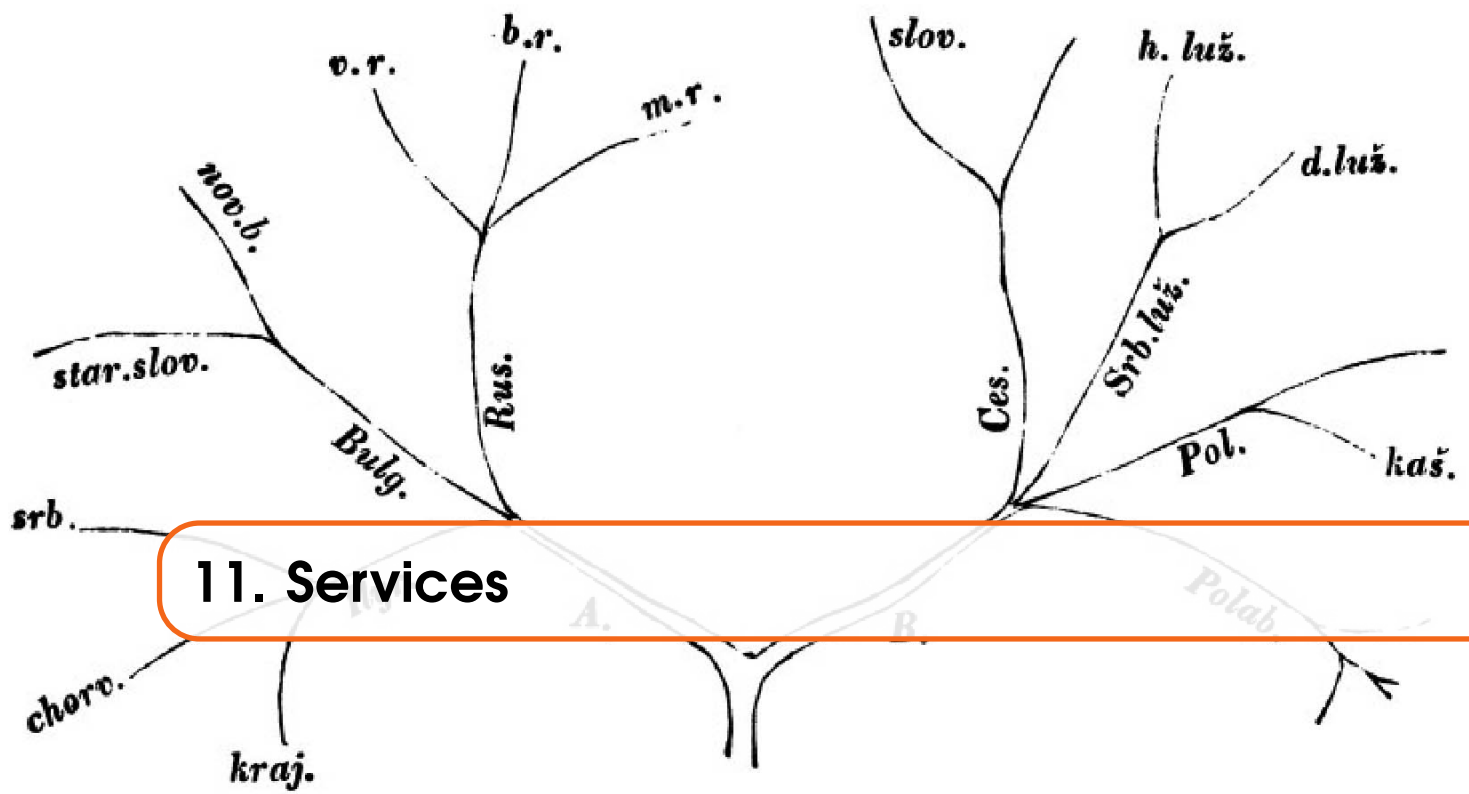
*Successfully duplicated readings*

#### Return — NOT\_FOUND. application/json

*If the tradition was not found.*

#### Return — NOT\_FOUND. application/json

*If one of the readings was not found.*



## 11. Services

### 11.1 Baseresource:

<http://localhost:8080/>

### 11.2 /user

Method Name: getIt

**GET** /

**Response** as text/plain

Method Name: create

**POST** /create

**Request** userModel as application/json

**Response** as application/json

Method Name: getUserById

**GET** /{userId}

**Response** as application/json

Method Name: deleteUserById

**DELETE** /{userId}

**Response** as text/plain

**Parameter** userId as string

**Method Name:** getTraditionsByUserId

**GET** /traditions/{userId}

**Response** as application/json

**Parameter** userId as string

### 11.3 /relation

**Method Name:** getIt

**GET** /

**Response** as text/plain

**Method Name:** delete

**POST** /{textId}/relationships/delete

**Request** relationshipModel as application/json

**Response** as text/plain

**Parameter** textId as string

**Method Name:** create

**POST** /{textId}/relationships

**Request** relationshipModel as application/json

**Response** as application/json

**Parameter** textId as string

**Method Name:** deleteById

**DELETE** /{textId}/relationships/{relationshipId}

**Response** as text/plain

**Parameter** textId as string

**Parameter** relationshipId as string

### 11.4 /witness

**Method Name:** getWitnssAsPlainText

**GET** /string/rank/{tradId}/{textId}/{startRank}/{endRank}

**Response** as application/json

**Parameter** textId as string

**Parameter** endRank as string

**Parameter** tradId as string

**Parameter** startRank as string

**Method Name:** getWitnessAsPlainText

**GET** /string/{tradId}/{textId}

**Response** as application/json

**Parameter** textId as string

**Parameter** tradId as string

**Method Name:** getWitnessAsReadings

**GET** /list/{tradId}/{textId}

**Response** as application/json

**Parameter** textId as string

**Parameter** tradId as string

**Method Name:** getNextReadingInWitness

**GET** /reading/next/{tradId}/{textId}/{readId}

**Response** as application/json

**Parameter** textId as string

**Parameter** readId as long

**Parameter** tradId as string

**Method Name:** getPreviousReadingInWitness

**GET** /reading/previous/{tradId}/{textId}/{readId}

**Response** as application/json

**Parameter** textId as string

**Parameter** readId as long

**Parameter** tradId as string

## 11.5 /tradition

**Method Name:** getAllRelationships

**GET** /relation/{tradId}/relationships

**Response** as application/json

**Parameter** tradId as string

**Method Name:** deleteUserById

**DELETE** /{tradId}

**Response** as text/plain

**Parameter** tradId as string

**Method Name:** changeOwnerOfATradition

**POST** /{textId}

**Request** textInfoModel as application/json

**Response** as application/json

**Parameter** textId as string

**Method Name:** getReading

**GET** /reading/{tradId}/{readId}

**Response** as application/json

**Parameter** readId as long

**Parameter** tradId as string

**Method Name:** getAllTraditions

**GET** /all

**Response** as application/json

**Method Name:** duplicateReading

**POST** /duplicate/{tradId}

**Request** duplicateModel as application/json

**Response** as application/json



■ **Parameter** tradId as string

**Method Name:** mergeReadings

**POST** /merge/{tradId}/{firstReadId}/{secondReadId}

■ **Response** as application/json

■ **Parameter** secondReadId as long

■ **Parameter** tradId as string

■ **Parameter** firstReadId as long

**Method Name:** splitReading

**POST** /split/{tradId}/{readId}

■ **Response** as application/json

■ **Parameter** readId as long

■ **Parameter** tradId as string

**Method Name:** getAllWitnesses

**GET** /witness/{tradId}

■ **Response** as application/json

■ **Parameter** tradId as string

**Method Name:** getTradition

**GET** /get/{tradId}

■ **Response** as application/json

■ **Parameter** tradId as string

**Method Name:** importGraphMl

**POST** /new

■ **Request** as multipart/form-data

■ **Response** as application/json

**Method Name:** getAllReadingsOfTradition

**GET** /readings/{tradId}

■ **Response** as application/json

■ **Parameter** tradId as string

**Method Name:** getDot

**GET** /getdot/{tradId}

■ **Response** as application/json

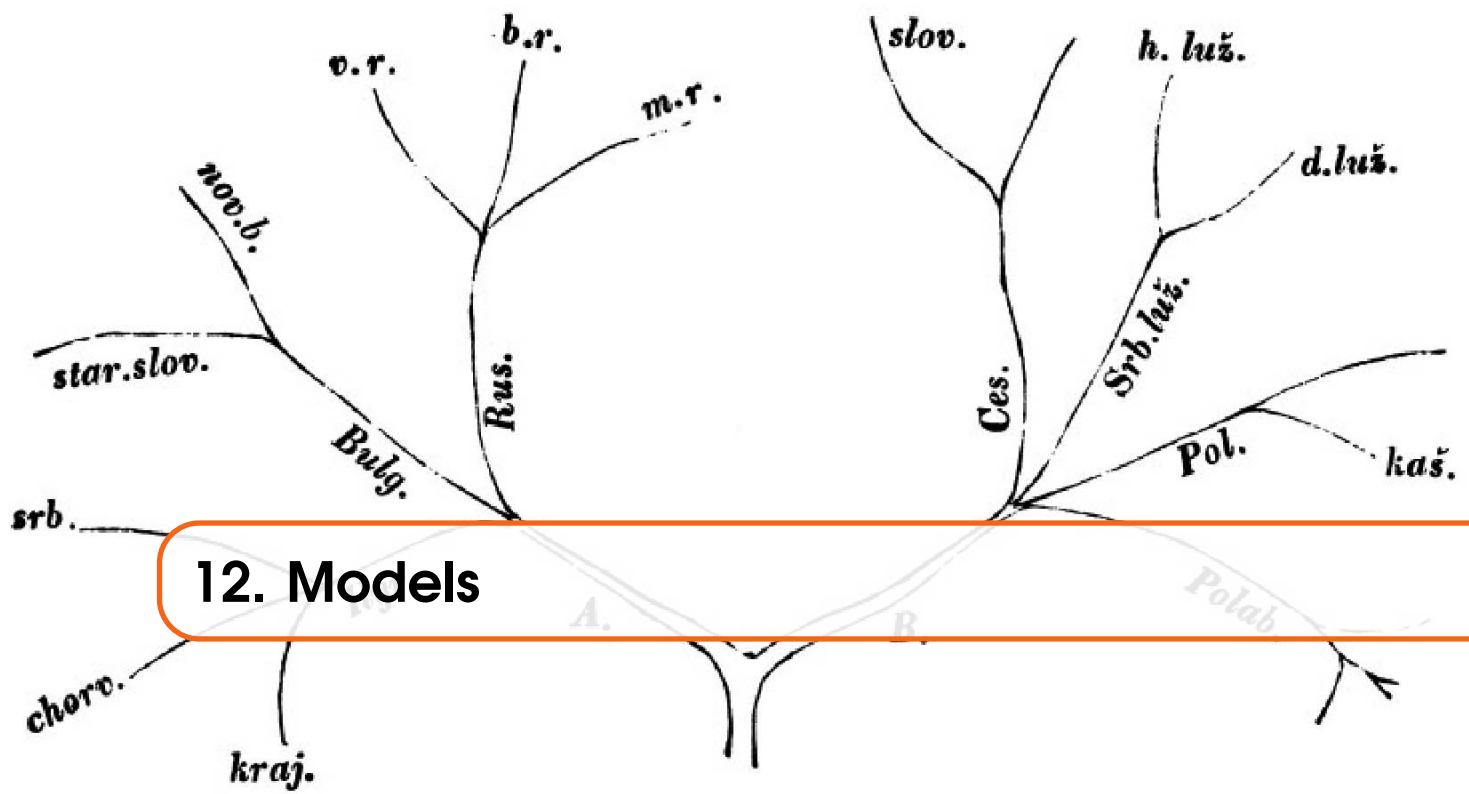
■ **Parameter** tradId as string

## 11.6 /myresource

**Method Name:** getIt

**GET** /

■ **Response** as text/plain



## 12. Models

### 12.1 userModel

- Property id as string
- Property isAdmin as string

### 12.2 relationshipModel

- Property de0 as string
- Property de1 as string
- Property de10 as string
- Property de11 as string
- Property de12 as string
- Property de2 as string
- Property de3 as string
- Property de4 as string
- Property de5 as string
- Property de6 as string
- Property de7 as string
- Property de8 as string

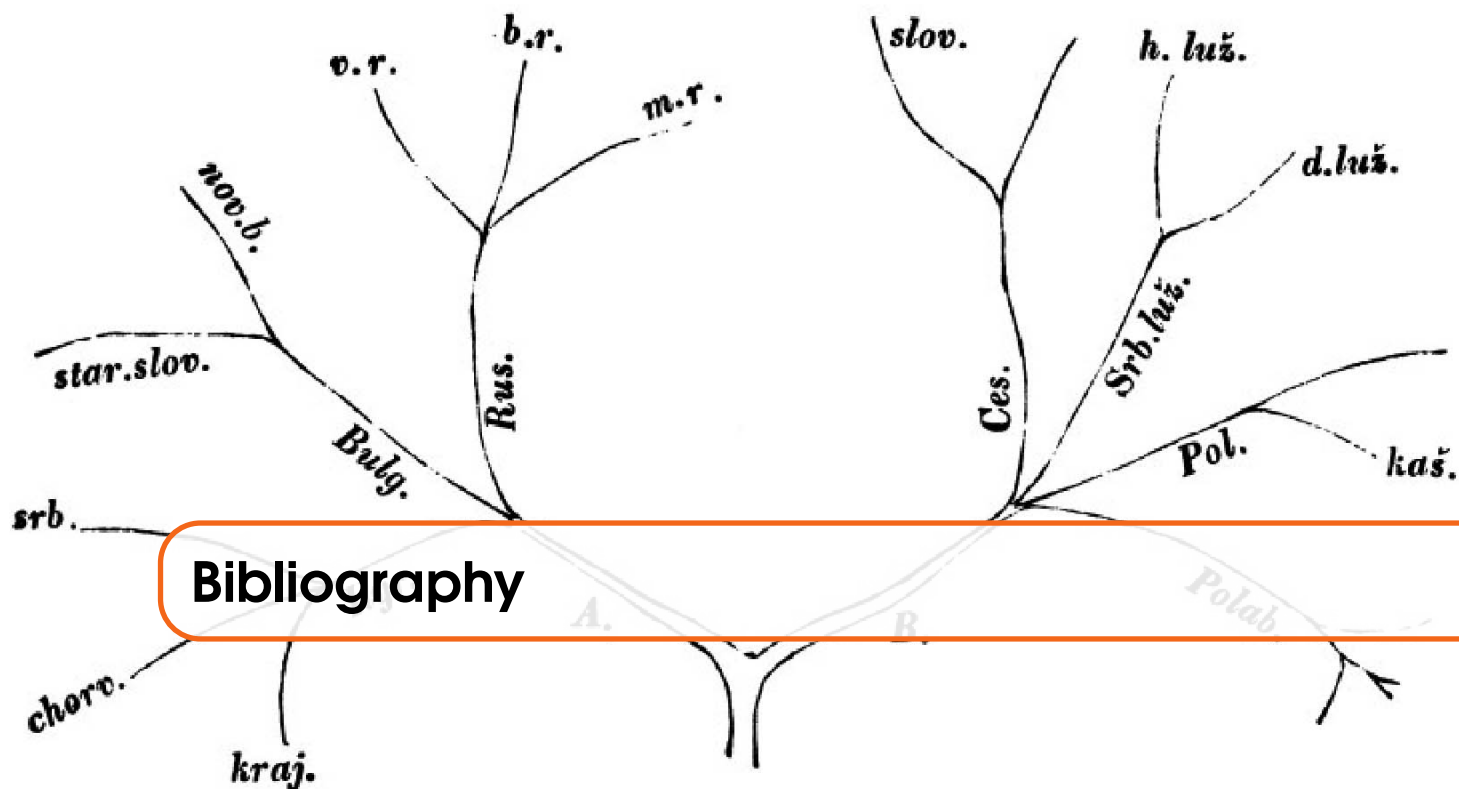
- **Property** de9 as string
- **Property** id as string
- **Property** source as string
- **Property** target as string

### 12.3 duplicateModel

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

### 12.4 textInfoModel

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



## Bibliography

Books  
Articles

DRAFT