D5.1 – Specification and Design of Conversation Data Repository

Document Number	D5.1
Document Title	Specification and Design of Conversation Data Repository
Version	1.5
Status	Final
Workpackage	WP5
Deliverable Type	Report
Contractual Date of Delivery	31.10.2014
Actual Date of Delivery	31.10.2014
Responsible Unit	USFD
Keyword List	conversational repository, REST, JSON, document format, data interchange
Dissemination level	PU

Editor

Adam Funk (USFD)

Contributors

Ahmet Aker (USFD)
Benoit Favre (AMU)
Carmelo Ferrante (UNITN)
Adam Funk (USFD)
Robert Gaizauskas (USFD)

SENSEI Coordinator

Prof. Giuseppe Riccardi
Department of Information Engineering and Computer Science
University of Trento, Italy
riccardi@disi.unitn.it

Document change history

Version	Date	Status	Author (Unit)	Description
0.1	23/07/2014	Draft	A. Funk (USFD)	Outline.
0.2	24/07/2014	Draft	A. Funk (USFD)	Requirements, interaction.
0.3	29/08/2014	Draft	A. Funk (USFD)	More requirements, interaction; implementation.
0.4	31/08/2014	Draft	A. Funk (USFD)	More details on API and internals; threading.
0.5	18/09/2014	Draft	B. Favre (AUX)	Role in prototype development.
0.6	20/09/2014	Draft	A. Funk (USFD)	REST API, sample document.
0.7	22/09/2014	Draft	A. Funk (USFD) B. Favre (AUX)	Testing plan.
0.8	23/09/2014	Draft	A. Funk (USFD)	Formatting; more about data.
0.9	29/09/2014	Draft	A. Funk (USFD)	Revised API; more about data, metadata, features, annotations.
1.0	30/09/2014	Draft	A. Funk (USFD)	Revised requirements, model, metadata. features, annotations.
1.1	02/10/2014	Draft	A. Funk (USFD)	Revised repository content, document example, REST API.
1.2	03/10/2014	Draft	A. Funk (USFD)	Revised metadata, implementation.
1.3	10/10/2014	Draft	E. Chiarani (UNITN) V. Giliberti (TP)	Quality checks.
1.4	14/10/2014	Draft	A. Funk (USFD)	Modifications in response to quality checks; other minor corrections.
1.5	17/10/2014	Final	E. Chiarani (UNITN) A. Funk (USFD)	Final quality check.
1.6	28/10/2014	Final	A. Funk (USFD)	Minor font size changes.

Executive summary

This document describes the requirements, specifications, and design of the repository which other components in the SENSEI prototype will use to store and access the conversational data. The repository will be a REST service provided by a Java servlet in a Tomcat container and will expose a range of HTTP methods that other prototype components will use to store, modify, delete, and retrieve documents as well as to run queries to find documents matching the query criteria. Documents will follow a flexible and easily adaptable model. Data interchange in and out of the repository will be in the JSON format as PUT and POST data (input) and HTTP response bodies (output).

Contents

1. Overview

The components of the SENSEI prototype need a shared repository for storing and accessing conversational data, including the texts themselves as well as the results of processing. Some other on-line and off-line components will need to store new documents; most will need to query the repository, read documents, and add additional information to them; and the user interface will need to query the repository and read documents quickly.

2. Requirements

2.1. Role of the repository in the context of prototype develpment

The SENSEI project will run an extrinsic evaluation of the approaches developed in the project. This evaluation is supported by a prototype implementing the evaluation scenarios for both the social media and speech use cases. As specified in D6.1, the prototype will be web-based, with a backend running on a server and a frontend running in the client's browser. While the frontend will a rich user interface based on latest web technologies, the server side will be embodied by a set of modules communicating through REST[1] services. These backend modules will implement SENSEI approaches to conversation analysis and summarization. In particular, some modules will run offline and have to store their result for later exposition to the user, while some other modules will run online under a latency constraint.

The role of the conversation repository is to feed all the modules with conversational data and store the output of offline modules, while providing fast access in a convenient interface. The prototype will provide a collection browser, which shall display a list of conversations and list of speakers/commenters in the conversations; a conversation browser which will display all comments/turns of a conversation, as well as associated annotations (topical, semantic, argumentative, emotional); a speaker/commenter view which will show all interventions of a given user, with statistics related to associated annotations (such as the emotional trends in replies to this speaker/commenter); an advanced search function which can tackle all annotation types and display relevant results at the conversation or sub-conversation level; and an evaluation view which is used by subjects for inputting the result of the task they are conducting (such as AOFs (Agent Observation Forms) or THM (Town Hall Meeting) summaries[2]

2.2. Content

The repository needs to store the plain-text content of documents as well as metadata (such as the types listed in Appendix A); some metadata items refer to whole documents, whereas some refer to

[1] Representational State Transfer (REST) is an architectural style for web services. There is no official standard for REST itself, although it adheres to the standards [8; 9] for HTTP, over which it runs. REST services use URLs and standard HTTP verbs (with additional data in the body of PUT and POST calls) to send and receive data in any convenient or agreed format.

[2] See Sections 2.1 and 3.2 of deliverable D1.2 for detailed explanations of Agent Observation Forms (AOFs) and Town Hall Meeting (THM) summaries, respectively.

specific spans of text within documents. A suitable model therefore divides documents into the following constituents:

- textual data (document content);
- metadata about each document, such as the types listed in Appendix A.1;
- stand-off annotations (indexed with respect to the characters in the document content) containing morphosyntactic and semantic annotation, named-entity recognition, sentiment analysis, and other linguistic and quasi-linguistic information, as well as original mark-up (e.g., HTML tags), as listed in Appendix A.2;

Audio documents are a special case; the repository will store the following for each one:

- a link or pointer to the audio data stored elsewhere;
- stand-off annotations indexed in milliseconds;
- a link to a separate repository document containing the transcription, if it exists (the transcription document may have its own character-based stand-off annotations).

Other components of the prototype must be able to add metadata and annotations to existing documents, or modify them, as needed. The repository must also provide a flexible system for queries, so that other components can find documents according to their metadata.

2.3. Front-end requirements

The repository needs to respond to queries from the prototype's WWW front-end in real time (i.e., as on-line processing). All other functions (storing documents, adding metadata and annotations to documents, etc.) will be regarded as off-line processing, for which speed is not essential.

2.4. Prototype server characteristics

The prototype server has the following software.

- Ubuntu 14.04
- Python 2.7
- Python 3
- Apache2 PHP 5.3.2 with `mod_rewrite` enabled
- Java 7
- Perl
- C/C++ with gcc and g++ compilers
- MySQL (SQL database)
- MongoDB (NoSQL database)

- Apache Tomcat 6.0.39

The prototype server has 200 GB of disk space and 8 GB of RAM, but the repository must not hog resources because other components will be running at the same time, especially when the web front-end (user interface) is in use.

3. Interaction

This section describes how the repository interacts with other components.

3.1. Document model

The main unit of data storage in the repository is a *document* object similar to the GATE or TIPSTER model [4; 15], which allows arbitrary data to be stored as stand-off annotations, annotation features, and document-wide features. Each *document* object contains the following objects.

- *content*: the plain text of the document.
- *features*: a map of key-value pairs of data relating to the whole document, such as the source URL, the external document ID, the parent document ID, a link or pointer to an audio file, etc. The keys must be strings, but the values can be any standard JSON[3] types, including lists and maps (which can be nested).
- *id*: a unique integer identifying the document in the repository, assigned by the repository when the document is first stored.
- *annotationMap*: a map with string keys for the annotation set names and *annotationSet* values. Each *annotationSet* contains zero or more instances of *annotation*, each of which contains the following.
  - *id*: a unique integer assigned by the repository when the annotation is added (initially stored with the document).
  - *startOffset*: an integer representing the start of the annotation as a character offset in the textual content or as a time offset in the relevant audio file.
  - *endOffset*: an integer similarly representing the end of the annotation.
  - *type*: a string identifying the nature of the annotation, e.g., Token, Person, Location, Sentence, Opinion.
  - *features*: a map of key-value pairs, like the document-level feature map, to store additional data about each annotation, such as POS tags and other morphosyntactic information for tokens, co-referencing information for named entities, and the target and polarity of opinions.

[3]JavaScript Object Notation (JSON) is a lightweight data format which is relatively human-readable and supported by libraries in most modern programming languages. A JSON object is a map of key-value pairs, where the keys are strings and the values can be strings, numbers, booleans, `null`, arrays (lists), or objects (maps); arrays and objects can be nested. [7]

Allowing arbitrary data types for the feature values and arbitrary names for annotation sets and types allows the flexibility to store unforeseen kinds of data in the future without changing the code or adversely affecting interoperability. Other components are encouraged to obtain their input and write their output using annotation set names and annotation types consistently; the conventions can be agreed as the project develops. Appendix A illustrates types of metadata at the document level (to be stored as document features) and at the annotation level (to be stored as annotations and their features); the annotation-level metadata refer to specific spans of the document.

Annotations within the same set (as well as across sets) can overlap, so the formal representation of this system is an annotation graph. [3]

The standard data interchange format between components will be JSON, as agreed by the project partners for interoperability of components coded in various languages. Appendix B gives a simple example.

3.2. Meta-documents

Meta-documents, such as THM summaries generated by off-line processing, clustering output, etc., will be stored in the repository as additional documents, using the content, annotations, and features as appropriate for each case. For example, a multi-document summary could be stored as follows: with the summary text itself in the meta-document's content and an array of document IDs in a specified feature to list the documents that have been summarized, whereas a clustering result could contains a document feature listing the clustered document IDs, another containing the label of the cluster, and an empty document content.

3.3. API

Appendix C lists the provisional (draft) API for the repository service. The first running implementation (similar to a proof of concept) will provide the following basic methods for testing.

- *GET* `http://localhost:port/documents` lists all document ids.
- *GET* `http://localhost:port/document/<doc_id>` returns the specified document.
- *PUT* `http://localhost:port/document/replace/<doc_id>` replaces the document with the JSON document in the PUT data.
- *DELETE* `http://localhost:port/document/<doc_id>` deletes the document.
- *POST* `http://localhost:port/document` stores the new document and returns its unique identifier (generated by the repository).

As explained in Section 4.3.1 (*Gateway*) of D6.1, the prototype components are internally (i.e., on the prototype server) distinguished by port numbers; the port for the repository service will be determined by a global configuration file.

Once the repository is running successfully with those methods, the rest of the desired API will be added incrementally as the project progresses.

4. Implementation

4.1. Front-end

The “front end” of the repository—i.e., the part that other prototype components communicate with—is implemented as a REST service running in Apache Tomcat. The service is being developed with the Apache CXF [1; 2] library, which (in USFD’s experience) is very reliable and stable and produces consistent results, as well as the Spring Framwork [10] configuration system. CXF provides an implementation of JAX-RS (the Java API for RESTful Services)

The REST API will provide methods for adding, retrieving, modifying, and deleting documents, as well for carrying out queries based on document features. Section 3.3 describes the basic methods to be implemented in the first version of the repository, and Appendix C lists a fuller draft API to be implemented as the project progresses.

4.2. Internals and back-end

Documents and their constituents (as described in Section 3.1) are represented in the repository software itself as instances of Java classes, which are mapped to and from JSON using the Jackson [13] library for consistent input and output.

The objects are stored in the MongoDB [11] server using using the Spring Data MongoDB [12] library, which provides a JSON-based query language that will be used to find matching documents in respose to queries from other components.

4.3. Additional tools

The following additional tools will be provided with the repository software:

- a Java client for testing;
- command-line tools for testing and pre-loading the repository, based on the Java client (the initial testing tool will be quite simple, to support the basic test case of storing a test document, asking for it back using the document ID, and checking to make sure it matches; future tools will be more sophisticated).

4.4. Threading

Threading (determining the tree structures of conversations) of a set of related documents can be derived from their external IDs and their parent documents’ external IDs using standard algorithms (as used in mail and news clients). An off-line process integrated into the repository will generate the threading information, probably using the efficient algorithm that Zawinski [16] provides, and store it in document features.

4.5. Testing plan

The initial testing will consist of using the command-line tools mentioned in Section 4.3 to add some documents to the repository, retrieve them, and verify that they match.

A more sophisticated testing system will follow, consisting of unit tests using curl [14] to test the outpt of REST URLs. These tests will be added to AUX's GitLab Continuous Integration (gitlab-ci) system to run automatically.

5. Conclusions

This document presents the specifications for the conversational repository which will be used by various components of the prototype for document storage, querying, and information interchange. The repository will consist of a Java Servlet running in the Apache Tomcat web container and will provide a REST service which other components interact with over HTTP. Most of the data will be exchanged in JSON.

A. Metadata

A.1. Metadata at the document level

We envisage storing the following metadata items (particularly relevant to workpackage 2) as document features. Most of these apply only to certain types of posts or articles, and most documents will have only some of these features.

externalID

parentID externalID of parent

postID externalID of the article to which a comment is attached

version version number of an edited post

domain domain of the post's URL

APIObjectID id of the post on the original media platform

APIAuthorID the author ID in the native API where this comes from (e.g., Facebook user id)

APIToAuthorID ID in the native API of the author targeted by the message (e.g., in a comment)

PostType article, comment, status update, reply, repost, etc.

AuthorType single user, fanpage, group, registered, anonymous, etc.

PageSuperType social, news, blog, forum, video, other

SourceType guardian, corriere, metronews.fr, facebook, twitter, g+, social-other; news-other, etc.

Title title of the post, if available

Keywords keywords of the post, if available

Author author of the post, surface username

AuthorProfilePictureURL url of the user profile picture

pageNumOfComments number of comments on the page

pageNumOfLikes

pageNumOfDislikes

pageNumOfViews

pageNumOfFavorites

pageNumOfReTweets

pageNumOfShares Shares in Facebook or Google+

userNumOfFollowers

userNumOfFollowing

userNumOfFriends

Mood mood type in corriere.it

MoodStrength mood strength in corriere.it

inReplyTo addressee of comments

embeddedMediaType text, text+photo, text+video, text+link, photo, video, link

authorsMentioned other authors mentioned in the post

authorSource from re-tweets, etc.

tags

isBestComment label for "Guardian picks" and best comments

PictureURLs list of URLs of pictures in an article or post

MediaURLs list of URLs of video or other media included in an article or post

date clipping date (real, extracted or guessed)

indexingTimestamp

crawlQueryMatch text that matched the query that triggered crawling this clipping

crawlQueryID id of queries that triggered crawling this clipping

authorLocation geolocation of the author

langDetected language of document as automatically detected

langReported language of the social interface displayed to the user

WebsaysPolarity sentiment analysis

clusters list of IDs of documents clustered together based on matchKeywords, matchAuthorNames or Polarity

namedEntities a list of named entity recognition results for the document (individual spans and coreferencing can be marked with annotations, as shown in Appendix A.2.

A.2. Metadata at the annotation level

The following metadata items refer to specific spans of text rather than to whole documents so they are stored as annotations and features of annotations.

- *Sentence* annotations will indicate sentence segmentation.
- *Token* annotations reflect not only tokenization but also (using features) the output of POS tagging and morphological analysis, as illustrated in the "`nlp`" annotation sets in Appendix B and Figure 3.
- Dependency parsing output can be represented by additional features on *Token* annotations to indicate the relations and the annotation IDs of the arguments of the relations, or by additional annotations to carry the same information.
- Chunking output is probably best represented by additional annotations spanning the chunks,

with relevant annotation types such as *NP*, *VP*, and *PP* (noun phrase, verb phrase, prepositional phrase). The annotation graph model (see Section 3.1 and Bird and Liberman [3]) allows nested and overlapping annotations of any type.

- For named entity recognition (NER), annotations can mark the spans of the following types: person, person descriptor, organization, organization descriptor, location, location descriptor, miscellaneous. Other arbitrary specific classes can be created using dictionaries. Either of the two conventional systems for marking named entities can be used: according to type (annotations of type *Person*, *Location*, etc.) or specified with features (NER annotations all of type *Mention* with a feature `"kind":"Person"`, `"kind":"Location"`, etc.)

  Coreferencing chains can be represented with features on named entity annotations whose values are lists of IDs of matching annotations.
- The semantic parsing output from the SEMAFOR tool [5; 6] is in the JSON format by default, as illustrated in Figure 1, and each unit of output refers to one sentence, so the obvious and easy way to store this in the document model is simply to add each sentence's SEMAFOR to a dedicated feature of the *Sentence* annotation, either in the existing "`nlp`" annotation set or to a duplicate annotation in another set, as shown in Figure 2.
- Sentiment analysis results can be stored as annotations whose features indicate polarity, numeric score, target (what the opinion is about), and holder (who has the opinion).

```
{"tokens": ["The", "cat", "sat", "on", "the", "mat", "."],
 "frames": [{"target": {"start": 2,
                        "end": 3,
                        "name": "Being_located",
                        "text": "sat"},
             "annotationSets": [{"frameElements": [{"start": 0,
                                                     "end": 2,
                                                     "name": "Theme",
                                                     "text": "The cat"},
                                                    {"start": 3,
                                                     "end": 6,
                                                     "name": "Location",
                                                     "text": "on the mat"}],
                                 "score": 28.53643260661777,
                                 "rank": 0 } ]
            }
           ]
}
```

Figure 1: Example of SEMAFOR output

```
{"type":"Sentence",
 "start":0,
 "end":23,
 "id":100,
 "features":{"SEMAFOR":{"tokens": [...],
                        "frames": [...]
                       }
            }
}
```

Figure 2: Abbreviated example of SEMAFOR output attached to an existing *Sentence* annotation

B. Document format

The following simple example illustrates the document model described in Section 3.1. The document and annotation "`id`" fields are added by the repository so they appear in its output but they not present in the input from other components.

```
{"content":"The cat sat on the mat.",
 "id":1,
 "features":{"externalID":"http://example.com/cat.html",
             "langDetected":"eng",
             "keywords":["cat", "mat"],
             "date":"2014-09-20T13:39:16+0100"},
 "annotationMap":{"nlp":[{"type":"Sentence", "start":0, "end":23,
                          "features":{}, "id":100},
                         {"type":"Token", "start":0, "end":3, "id":101,
                          "features":{"category":"DT", "kind":"word",
                                      "orth":"upperInitial"}}
                         {"type":"Token", "start":4, "end":7, "id":102,
                          "features":{"category":"NN", "kind":"word", "id"
                                      "orth":"lowercase"}}
                         {"type":"Token", "start":8, "end":11, "id":103,
                          "features":{"category":"VBD", "kind":"word",
                                      "orth":"lowercase"}}
                         {"type":"Token", "start":12, "end":14, "id":104,
                          "features":{"category":"IN", "kind":"word",
                                      "orth":"lowercase"}}
                         {"type":"Token", "start":15, "end":18, "id":105,
                          "features":{"category":"DT", "kind":"word",
                                      "orth":"lowercase"}}
                         {"type":"Token", "start":19, "end":22, "id":106,
                          "features":{"category":"NN", "kind":"word",
                                      "orth":"lowercase"}}
                         {"type":"Token", "start":22, "end":23, "id":107,
                          "features":{"category":".", "kind":"punctuation"}}],
                  "html":[{"type":"b", "start":4, "end":7, "id":108,
                           "features":{}}]}
}
```

C. Provisional REST API

We propose the following API methods for clients to add data to the repository and query it. This list is provisional and will be extended and modified over the course of the project as other components' needs change or become clearer.

- *GET* `http://localhost:port/documents` lists the ids of all documents in the repository.
- *GET* `http://localhost:port/document/<doc_id>` returns the specified document.
- *PUT* `http://localhost:port/document/replace/<doc_id>` replaces the document with the JSON document in the PUT data.
- *DELETE* `http://localhost:port/document/<doc_id>` deletes the specified document.
- *POST* `http://localhost:port/document` stores a new document and returns its unique id (generated by the repository); the POST data should be a JSON object representing a document.
- *POST* `http://localhost:port/document/content` stores a new document and returns its unique id; the POST data should be plain text, which will become the document's content; the document's feature and annotation maps will be empty at this point.
- *POST* `http://localhost:port/annotations/<doc_id>` adds annotations and (as needed) annotation sets to the document; this method creates a new annotation set (with the given annotations) for each set name that does not already exist, and simply adds the given annotations to sets that already exist; the POST data consists of a JSON object representing one or more annotation sets—Figure 3 shows an example.
- *POST* `http://localhost:port/features/<doc_id>` adds features to the document, creating new features for keys that are not already in the document's feature map and overwriting the values associated with existing keys; the POST data should be a JSON object containing the features—Figure 4 shows an example.
- *DELETE* `http://localhost:port/annotations/<doc_id>/<set_name>` deletes the named annotation set from the document.
- *DELETE* `http://localhost:port/annotations/<doc_id>/<set_name>/<ann_id>` deletes the specified annotation from the document.
- *DELETE* `http://localhost:port/features/<doc_id>/<feature_name>` deletes the specified feature from the document.
- *GET* `http://localhost:port/documents?<f0>=<v0>&<f1>=<v1>...` returns a list of identifiers of doc-

```
{"nlp":[{"start":5, "end":28, "type":"sentence",
          "features":{}},
         {"start":5, "end":8, "type":"token",
          "features":{"pos":"DET", "string":"The"}},
         {"start":9, "end":12, "type":"token",
          "features":{"pos":"NN", "string":"dog"}},
         ...]
}
```

Figure 3: Example of JSON data to add annotations to the `nlp` set

uments that have the specified feature-value pairs, where the values are simple types (numeric, boolean, string).

- *POST* `http://localhost:port/documents` returns a list of identifiers of documents that have feature-value pairs specified in the POST data, where the values can be complex or nested types as well as well as simple ones.

```
{"source_url":"http://example.com/foo.html",
 "pos_tagged":true,
 "lang_detected":"eng",
 "named_entitities":["Rufus T Firefly", "Freedonia",
                     "Acme Corporation"],
 "similar_docs":["http://example.org/bar.html"]
}
```

Figure 4: Example of JSON data to add document features

Bibliography

[1] Apache Software Foundation. *Apache CXF: an Open-Source Services Framework*, 2014. URL `https://cxf.apache.org/`.

[2] Naveen Balani and Rajeev Hathi. *Apache CXF Web Service Development*. From Technologies to Solutions. Packt Publishing, December 2009.

[3] Steven Bird and Mark Liberman. A formal framework for linguistic annotation. *Speech communication*, 33 (1):23–60, 2001.

[4] Hamish Cunningham, Kevin Humphreys, and Robert Gaizauskas. GATE—a TIPSTER-based general architecture for text engineering. In *Proceedings of the TIPSTER Text Program (Phase III) 6 Month Workshop*. Morgan Kaufmann, 1997.

[5] Dipanjan Das and Noah A. Smith. Semi-supervised frame-semantic parsing for unknown predicates. In *Proceedings of Annual Meeting of the Association for Computational Linguistics*, Portland, OR, June 2011. ACL.

[6] Dipanjan Das, Nathan Schneider, Desai Chen, , and Noah A. Smith. SEMAFOR 1.0: A probabilistic frame-semantic parser. Technical Report CMU-LTI-10-001, Carnegie Mellon University, April 2010.

[7] Ecma International. The JSON data interchange format. Technical Report ECMA-404, Ecma International, October 2013. URL `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[8] R. Fielding and J. Reschke. Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. Technical Report RFC 7230, Internet Engineering Task Force, June 2014. URL `https://tools.ietf.org/html/rfc7230`.

[9] R. Fielding and J. Reschke. Hypertext transfer protocol (HTTP/1.1): Semantics and content. Technical Report RFC 7231, Internet Engineering Task Force, June 2014. URL `https://tools.ietf.org/html/rfc7231`.

[10] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, Phillip Webb, Rob Winch, Brian Clozel, Stephane Nicoll, and Sebastien Deleuze. *Spring Framework Reference Documentation*, 2014. URL `http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/`.

[11] MongoDB Community. *The MongoDB 2.6 Manual*. MongoDB, Inc., 2014. URL `http://docs.mongodb.org/manual/`.

[12] Mark Pollack, Thomas Risberg, Oliver Gierke, Costin Leau, Jon Brisbin, Thomas Darimont, and Christoph Strobl. *Spring Data MongoDB Reference Documentation*. Pivotal Software, 2014. URL `http://docs.spring.io/spring-data/mongodb/docs/current/reference/html/`.

[13] Tatu Saloranta. *Jackson JSON Processor Wiki*. FasterXML, LLC, 2013. URL `http://wiki.fasterxml.com/JacksonHome`.

[14] Daniel Stenberg. *curl.1 the man page*, 2014. URL `http://curl.haxx.se/docs/manpage.html`.

[15] Yorick Wilks, Robert Gaizauskas, Kevin Humphreys, and Hamish Cunningham. LaSIE jumps the GATE. Technical report, University of Sheffield, 2000. URL `http://staffwww.dcs.shef.ac.uk/people/Y.Wilks/papers/lasie/node3.html`.

[16] Jamie Zawinski. *Message Threading*, 2002. URL `http://www.jwz.org/doc/threading.html`.