Documentation of rddrssr

rddrssr - An address distribution network

Hochschule der Medien, Stuttgart
Studiengang Medieninformatik

by

Sven Pfleiderer
(sven@roothausen.de)

October 25, 2010

Abstract

We all live in times of digital communication: Almost everybody is reachable via cellphone, instant messaging or email. Not only the communication itself evolved but also the communication channels increased dramatically. Some people have multiple email addresses, instant messaging accounts and profiles on several social networks. It is virtually impossible to keep track of all the information available.

To address these problems, Moritz Haarmann came up with an idea of a system which is able to manage contact data in a distributed and convenient way. The result of this idea was a protocol proposal which enables users to manage their address data so they can just stop worrying about it.

This documentation describes the implementation details and design decisions made to create an usable software which uses the protocol defined by Moritz.

Contents

1 Motivation

1.1 Current Situation

1.1.1 Digital Communication

We all live in times of digital communication: Almost everybody is reachable via cellphone, instant messaging or email. Not only the communication itself evolved but also the communication channels increased dramatically. Some people have multiple email addresses, instant messaging accounts and profiles on several social networks. It is virtually impossible to keep track of all the information available.

The systems which are used today can be divided into three categories:

- Groupware
- Cloud based Solutions
- Social Networks

1.1.2 Groupware

Groupware, or Collaborative Software, are systems which enable persons or companies to communicate and organize their work. Most groupware solutions provide at least email and calendar access and some form of rudimentary directory to access contact data.

Today, a lot of big software companies provide some kind of groupware suite and most middle sized and big companies rely very strongly on these solutions like Microsoft Exchange or Lotus Notes. These products are distributed under proprietary licenses, use closed non standardized protocols and are only available on specific platforms. Furthermore the contact management is mostly really basic.

Most of the systems are capable of supporting at least two kinds of directories: A global one storing data of the organization which is centrally maintained and a private one which is freely editable by the user.

There are also some solutions which are available as free software and use open standards, like LDAP [Wahl et al., 1997] or SyncML, to distribute contact information. But the problem with this solutions stays basically the same: We have centralized topologies which are single points of failure that are not aware of frequently occurring changes of contact data.

Both – open and closed solutions – may work in companies where contact data is limited to some crucial information, like phone numbers or email addresses, and is mostly provided by the company itself. When it comes to private relationships, they all lack the possibility to support the latest and greatest Web 2.0 service because the data fields they use are mostly limited. Additionally they are not able to identify when contacts change their personal data because they only represent a set of data which is maintained by the owner. If this owner does not know about changed information or just does not care to update it, it is outdated and, in the worst case, the user has no way to stay in touch with this particular contact.

1.1.3 Cloud based Solutions

Cloud Computing is certainly one of the most dropped buzzwords in the last years. The term describes a system, where services are provided on demand by a bunch of servers which are accessible over the Internet.

A lot of cloud based applications are accessible via interfaces which are based on web technologies. To access these services the only thing a user needs is a web browser capable to render HTML and CSS and able to execute JavaScript. Most of these services provide additional web based APIs to be accessible by desktop or mobile applications.

One well known provider of cloud services is Google. Google offers search, email services, personal calendars, mailing lists and also a contact management interface called Google Contacts. There are some other providers which offer similar products but to illustrate the nature of these services, only Google Contacts is covered.

Google Contacts is a cloud based application which allows users to import, store and manage the information of their contacts. It also allows users to synchronize this data to mobile phones, address books or connect to groupware clients via outlook connectors. It also supports SyncML to connect to mobile phones which does not implement the Google API.

1.1.4 Social Networks

Though social networks could be seen as a subset of cloud based applications, the way they handle address information differs from “normal” cloud based applications. They are, generally speaking, platforms which invite people to create a “User Profile” for themselves and to connect with other users of this platform. This profile often contains personal data like pictures, personal interests and address information. The selection of the given data strongly depends on the network used. Some well known examples are Facebook, Linkedin or Xing.

A big benefit of social networks is that data available to other users mostly keeps up to date and that personal relationships are represented in the graph of the network.

The big drawback of this approach is that almost none of these networks allows data exchange through standardized protocols like SyncML or LDAP. So they are only usable on smartphones or desktop computers capable to handle the network's APIs. Another problem of social networks is that there are many of them and they lack cross network accessibility. So users of different networks are not able to connect beyond network boundaries.

1.2 Where Is The Innovation?

While the communication technology itself moved on, the technology to distribute our contact information stayed mostly the same: There are digital business card formats like VCard, directory services like LDAP and even digital address books. But they are all nothing more than electronic counterparts of a technology which we had for a really long time – written text on business cards, telephone books or address books. In short: Ancient technology converted into digital form. Sure, the electronic representations have some benefits compared to their ancient counterparts, but the key problems stay the same: Once information is stored, it takes a lot of work to update it. Most people have no desire to invest that time so, after a short time, the information stored is outdated and stays that way.

Cloud based solutions provide a more convenient way to access and sync your data but still lack features like push update and relationship management. Social networks seem to be better suited for the job but the drawbacks are user lock-in into one network and centralized infrastructures which are owned by a single company. Additionally the user has no clue what happens to the data provided: Is it stored in a secure way and handled in a privacy preserving manner [Jones and Soltren, 2005]? Who has the authority to access it? What happens to the data if the company responsible for the system is bought? What happens if the company cease to exist?

To address all the issues with current systems in a modern way, the system was completely built from scratch. The goal of this project was to implement simple but powerful software which provides an interface based on well understood standards. It's only purpose is to manage and distribute contact data in a modern and platform independent way.

2 Protocol Design

2.1 Design Goals

2.1.1 Introduction

The final goal of this project is to provide a robust solution which allows people to just stay in touch with their contacts without having to worry if the contact data is still valid. The purpose of the protocol is to enable nodes of a network to exchange their contact data. It has been developed by Moritz Haarmann [Haarmann, 2010] and follows some simple design goals.

- Open
- Complete
- Secure
- Distributed
- Easily understandable

2.1.2 Open

The protocol should be free to use for everyone. Everyone must have the possibility to look at a freely available specification and implement a software which uses it – without any license or patent restrictions.

Additionally the whole development process of the protocol should be community driven and open to new ideas and feedback from users or other developers. Users should be encouraged to participate with the goal of making the system better.

2.1.3 Complete

To support all the needs users have, the system should be able to handle all kinds of contact exchange in an intuitive way. It should be as close as possible to the reality of human relationships and handle the data exchange as gracefully as possible.

2.1.4 Secure

Security is an important factor when handling private contact data. Not only because users demand security but also because a lot of the requirements depend on secure transport. There are basically three security matters that should be addressed:

- Transported user data should only be visible to the entities entitled to see them.

- Harmful modification of data should be prevented.
- The protocol should be robust enough to handle potential attacks.

2.1.5 Distributed

To be able to compete with big players like Google or Facebook, which provide centralized systems with enormous resources, the system has to act as a distributed network. It should be possible for everyone to run software which supports the protocol.

That software should have the possibility to connect to every other node in the network to exchange contact data. Examples of someone who could run such nodes would be companies that provide services to their users, online communities or private users who want to provide such services to themselves and their friends.

2.1.6 Easily understandable

To attract a significant amount of users, the protocol has to be simple to understand and to implement. In addition, simple protocols have proven to be more robust and secure because the simpler a protocol design is, the lower is the possibility of errors or needless attack vectors.

2.2 Technical Decisions

2.2.1 Brief Overview

The protocol is designed to transport contact data over a HTTP connection using REST Semantics. The data itself is encapsulated within a JSON container. The APIs are divided into two parts: The Server-Server API and the Client-Server API. Authentication between multiple servers is provided by asymmetric encryption. Between clients and servers, the clients authenticate themselves using OAuth.

2.2.2 HTTP Transport

HTTP [Fielding et al., 1999] is likely to be the most used networking protocol worldwide. All websites are served via HTTP and a lot of applications use HTTP to transport their data. Using HTTP to transport data has some really great advantages:

- Great library and tooling support: Because of the widespread use of HTTP there are libraries for almost every programming language and a lot of frameworks which use them.
- Firewall friendly: As a result of the great distribution of HTTP, it is not blocked on most firewalls.

- Well understood semantics: A lot of application developers are familiar with HTTP.
- Well documented error scenarios and status codes: There is a whole set of predefined status codes to handle errors and corner cases which can be used inside the application logic.
- Accessible through browsers: Modern web browsers are not only able to act as a simple client and access HTTP based APIs, they are also capable to run whole applications written in JavaScript to interact with them.

2.2.3 REST Semantics

REST, or Representational State Transfer, is a style of software architecture for web based APIs. It was defined by Roy Fielding [Fielding, 2000] and represents HTTPs ability to address resources via URL patterns and operate on them with HTTP protocol methods which are normally called HTTP verbs. In this protocol version only the four most known verbs are used: GET, PUT, POST and DELETE.

HTTP Verb	Usage
GET	Retrieve data
PUT	Update existing data
POST	Create new data
DELETE	Delete existing data

Table 1: Used HTTP Verbs

These semantics provide a simple CRUD[1]-interface which allows users to create, view and alter data addressed by its URL pattern. To make such an API usable these patterns have to be consistent.

For example, a URL pattern to identify a collection of resources would be *http://example.com/resources/*. A HTTP GET request would return a list of all stored resources. If one wanted to access the resource identified by the ID 1337, the URL to access this resource would be *http://example.com/resources/1337*. To delete this resource, a user would only need to send a HTTP-DELETE request to this URL.

2.2.4 JSON Data Format

JSON, the JavaScript Object Notation, is a lightweight data exchange format which has gained a lot of attention from developers since it was first discovered and named by Douglas Crockford [Crockford, 2006]. It is a simple subset of ECMAScript/Javascript

[1]Create, Read, Update, Delete

[Ecma International, 1999] and easy to parse. There are many libraries for a lot of programming languages and it is used as data exchange format for a lot of big web applications.

```
{ "object": { "title": "Hello", "value": "World" } }
```

Listing 1: Simple JSON Example

2.2.5 RSA Encryption/Signatures

To protect transport data against hostile attacks, it should be encrypted to prevent others from sniffing the contents. Furthermore it should be signed to prevent harmful alternation. These services are both provided by the widely used RSA algorithm. It is well supported by crypto suites like OpenSSL or the Java Security API and it is well tested against all sorts of attacks.

It would be possible to use HTTPS, the secure and encrypted version of HTTP, to provide secure transport. But a lot of hosting environments do not support HTTPS. To allow a secure transport in such an environment, it was decided to use RSA encrypted content on top of unencrypted HTTP instead of using HTTPS.

2.3 Metadata Transport

2.3.1 Why metadata?

To communicate with each other, servers need to exchange some important information. Some of this information would be:

- Public RSA keys
- A list of supported data types and their definitions
- A base URL which is called on callback actions

If these items are not provided, the communication will fail. To avoid this problem, a read only interface for metadata is provided. It basically consists of two parts which are enclosed under the */meta/* namespace. These parts are the Server Info and the available types.

2.3.2 Server Info

To have an entry point for all crucial information, the protocol defines a “Server Info” artifact, as seen as in Listing 2, which provides this information.

```
{
  "name":"rddrssr.io",
  "baseUrl":"http://contacta.de:8080/",
  "software":"scala foobar",
  "types":["core.im.xmpp","core.person.date_of_birth","core.person.
      email","core.person.first_name","core.person.gender","core.
      person.last_name","core.person.phone"],
  "version":1,
  "key":"MII...=="
}
```

Listing 2: Server Info of example server

2.3.3 Types

People tend to change not only their existing data like email addresses or telephone numbers but also start using new communication services like social networks or instant messaging. In the future, they may also use services which are not even invented. So extensibility is a really important point which has to be covered.

To allow developer to extend the system with their own types, a metadata format in JSON was defined which covers the most important features of transport data: Validation and Recognition.

To add a new type which can be interpreted and validated by another system, all a

URL	HTTP Method	Description
/meta/index.json	GET	Server Info: Data needed to communicate with the server.
/meta/types/index.json	GET	List of all supported types.
/meta/types/[typename].json	GET	Type definition only for the type typename.

Table 2: Items of the meta namespace

Attribute	Description
name	Human readable name of the server.
baseUrl	A base URL used for callback actions.
software	The software which is running on the server.
version	The running version of the software.
types	An array containing the names of the available types.
key	The public key of the server which is needed for encryption and signature validation.

Table 3: Available attributes of the “Server Info”

developer has to do is to implement such a definition and announce it to other servers. An example of such a definition is shown in Listing 3.

```
{
  "cname":"XMPP",
  "information":"http://momo.brauchtman.net/types/",
  "name":"core.im.xmpp",
  "occurences":[0,0],
  "length":255,
  "revision":1,
  "format":"/\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b/"
}
```

Listing 3: Data type definition for XMPP

2.4 Payload Transport

2.4.1 Supported actions

Between two servers of a given network, there is only a limited amount of actions which have to be supported.

- Request a user relation
- Accept a user relation
- Update contact data
- Delete user relations

There are some actions which are planned for future versions of this protocol but are not specified at the moment.

- Migration of users between servers

Attribute	Description
name	Unique name of the type
cname	Canonical name of the type
information	A URL where a human readable explanation can be found
occurences	An array with two integer values: The minimal and the maximal occurrence of this type. If both have the value 0, there is no restriction.
length	The length of the content in chars.
revision	The revision of this type to distinguish versions.
format	A regular expression to validate the content.

Table 4: Available attributes of a type

- Announcement of new types
- Public key revocation

2.4.2 User Data

The actual user data is also transported inside a JSON format which contains the user's URL and an array of data. An example is shown in Listing 4.

Attribute	Description
userurl	The URL which identifies the user with its server.
data	An array of data containing key-value pairs of data. A key is identified by the attribute type, the value is identified by the attribute data.

Table 5: Attributes of the user data

```
{
  "userurl":"http://contacta.de:8080/pfleidi",
  "data":[{
    "type":"core.person.first_name",
    "value":"Sven"
  },{
    "type":"core.person.last_name",
    "value":"Norris"
  },{
    "type":"core.person.email",
    "value":"sven@roothausen.de"
  },{
    "type":"core.im.xmpp",
    "value":"a@aaaa.dd"
  }]
}
```

Listing 4: User data

2.4.3 Request Cycle

Before the actual payload can be transported, the metadata has to be fetched if it is not stored already. There may be some other actions, like asking the user for permission, a system has to finish until it is able to react to incoming request. For this reason, all requests which transport a payload are designed to be asynchronous.

As seen in Figure 1, the initialisation process needs one request per server to complete: A request to create a new relation and a request to accept this relation and transport the data needed.

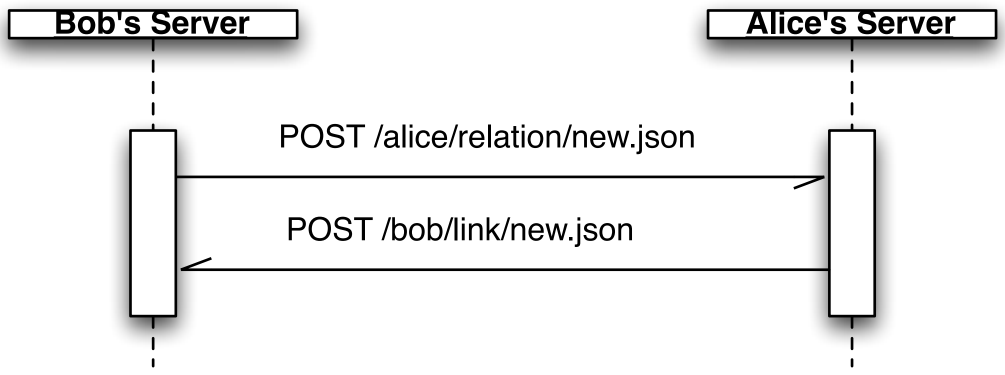
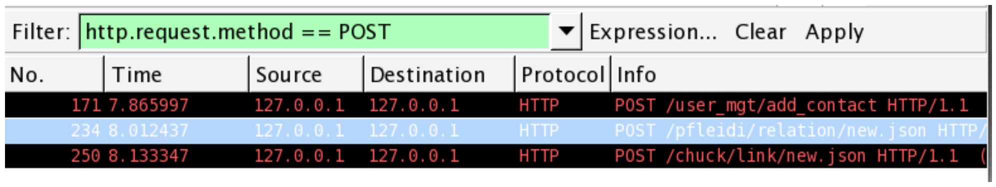


Figure 1: Initialisation of a relation

The first request contains the requesting user's URL and a generated token. This token is needed to verify the validity of the corresponding callback request.

After Bob's server sent a relation request to Alice's server, Alice's server fetches the needed metadata and asks Alice for permission to forward her contact information to Bob. If this permission is granted, Alice's server sends all contact data of Alice together with her URL and the token received before to Bob's server. After that, Bob's server is able to validate the token, parse the data and validate it, too. If the data passes all validations it gets stored in the database of Bob's server. As seen in Listing 5, all data is encapsulated in JSON.

Because only Bob's server generates and validates the token, this mechanism is completely implementation agnostic and doesn't need any support on the side of Alice's server.



Filter: http.request.method == POST

No.	Time	Source	Destination	Protocol	Info
171	7.865997	127.0.0.1	127.0.0.1	HTTP	POST /user_mgt/add_contact HTTP/1.1
234	8.012437	127.0.0.1	127.0.0.1	HTTP	POST /pfleidi/relation/new.json HTTP/
250	8.133347	127.0.0.1	127.0.0.1	HTTP	POST /chuck/link/new.json HTTP/1.1 (

Figure 2: Relation Request sniffed with Wireshark

```
{
  "userurl":"http://contacta.de:8080/pfleidi",
  "token":"EbjDfS6zNy389gdho5CpUzomlmI=",
  "data":[{
    "type":"core.person.first_name",
    "value":"Sven"
```

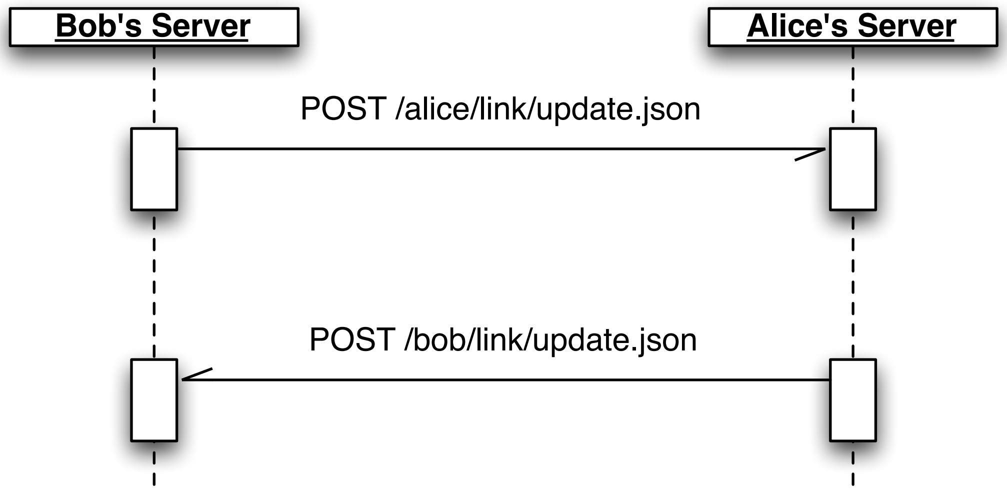


Figure 3: Update of a relation

```
  },{
    "type":"core.person.last_name",
    "value":"Pfleiderer"
  },{
    "type":"core.person.email",
    "value":"sven@roothausen.de"
  },{
    "type":"core.im.xmpp",
    "value":"pfleidi@jabber.roothausen.de"
  }]
}
```

Listing 5: User data encapsulated in link request

The update request shown in Figure 3 is actually really similar to the link request which forwards user data to other servers. With the only exception that no callback token is needed.

After a user updates its contact information, the server pushes the new information to all active links of this user. This happens usually without any requests from other servers.

Because not all links are needed forever, the protocol also provides a (not yet implemented) destroy request which is shown in Figure 4. After such a request is received, the receiving server deletes the semantic link of the corresponding user and the contact who sent the request.

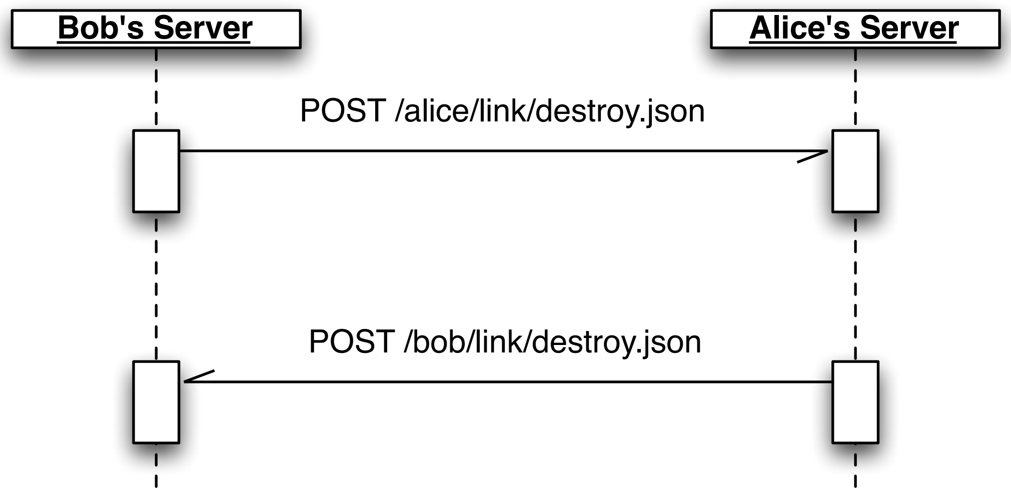


Figure 4: Destruction of a relation

2.4.4 Transport Container

Container Fundamentals All requests which are able to alter the state of data stored on other servers have to be signed to ensure the identity of the sender. They also have to be encrypted to protect the user's privacy. To allow that in a reusable manner, the protocol specifies a transport container in which data is encapsulated.

Name	Description
address	The base URL of the server.
checksum	A SHA-256 Hash of the encrypted payload.
signature	A signature over the encrypted payload.
payload	The encrypted payload itself.

Table 6: Items of the meta namespace

To make the encapsulation process as efficient as possible the first value to be validated has to be the checksum. After that, the signature has to be verified. If both tests pass, the payload itself can be decrypted. If one of the previous test fails, the payload can be discarded and no time is wasted for decryption.

Version 1 The first version of the container was implemented using JSON. For testing and debugging purposes the payload had to be transmitted decrypted. Most JSON parsers were parsing the whole content and it was not trivial to get an unparsed representation of the payload part inside the container. Because of that behavior this specification was soon discarded in favor of version 2 of the container.

Version 2 HTTP POST-requests support named query parameters which are separated by an ampersand (“&”). The second version of the container transports all attributes as POST query parameters. With this notion it is really simple to separate the payload and the other values to ensure it is valid.

2.4.5 Security

As explained before: The security of this protocol relies strongly on strong cryptography, namely RSA, AES and SHA1. All transported content is signed, hashed and encrypted and a proper implementation has to validate all of those properties. Additionally there is a mechanism to generate tokens which are only valid for certain requests. The possibility of a mechanism to revoke public keys was also discussed but never implemented, because such a mechanism would be very vulnerable to denial of service attacks.

2.5 Changes

Most proposals are not perfect and this proposal is no exception. Because of that, some parts of the specification written in Moritz’ thesis had to be slightly modified.

- Because of different character encodings, the length of a data type is defined in chars, not bytes.
- The options field was removed because it has not been used.
- The container format has been changed (as described in 2.4.4)
- The number of requests to establish a semantic link between two users of different servers was reduced to two.

3 Implementation

3.1 Frameworks and Middleware

3.1.1 Scala

Scala is a hybrid programming language designed to run on the Java-VM. It implements features from both, functional and object oriented programming paradigms, combined with some syntactic sugar and some concepts to make it simple to implement native looking domain specific languages.

It has a static type system, like Java, and uses exactly the same data types. Contrary to Java, Scala supports type inference so most of the type declarations, and also semicolons, are optional.

For concurrency Scala supports an abstraction called the Actor Model in addition to the Java concurrency APIs. The usage and purpose of this API is explained in [Ghosh, 2008].

Unlike Java, Scala also supports "Traits". These are language constructs which define, like Java Interfaces, the signature of supported methods. But contrary to Java Interfaces, methods contained in Traits are allowed to be partially of fully implemented. This way Scala provides a mechanism to add behavior to classes or objects, not only via class based inheritance but also through composition of traits.

Scala was chosen as an implementation language for this project because it is a fast and expressive language which has the ability to use Java-APIs because it is fully interoperable with the rest of the Java ecosystem.

```scala
// singleton object
object HelloWorld {
  // main method
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Listing 6: "Hello World" application in Scala

3.1.2 Lift

Lift was launched in 2007, so it is a relatively new web framework. It is an open source project licensed under the Apache 2.0 License. Despite it only reached version 1.0 in 2009, it is already used at big companies like Foursquare and Novell.

The Lift web application is framework written in Scala. It is known to be secure, elegant and scalable and uses a lot of Scala's more advanced features like pattern matching, type classes or the native support of XML. Lift also supports a specialized and lightweight kind of actor implementation to power real-time web-applications that use Comet to push content to the browser.

It also includes some database mapper libraries and a well fit template system. There are also some server side bindings to interact with JavaScript running on the client.

Because Lift 2.0 introduced a mapper library which can store persistent data inside of MongoDB and its good support of concurrent actions with help of actors, it was decided to use a preview version of Lift 2.0, which was still under heavy development at that time.

3.1.3 MongoDB

MongoDB is an open source database system which uses a document based, schema free data model. It is one of the new emerging noSQL data stores that don't rely on SQL as a query language and focus on special needs of application developers rather than providing a general purpose solution.

Data inside MongoDB is stored in a format called Binary JSON (BSON), which is a binary equivalent of JSON. This allows the data to be modeled more intuitively. During the development process, it was also possible to change the data model on the fly without having to alter some tables or migrate the data.

An other really useful feature of MongoDB was the ability to submit ad-hoc queries for debugging purposes. These queries are written in JavaScript and JSON and can be executed in an small shell environment called "mongo". In Listing 7 a sample ad-hoc query is shown as well as its result as a text representation.

```
db.users.find({"userName" : "pfleidi"});

{
  "_id" : ObjectId("4c9b6dfec28b66d49fdba0ef"),
  "firstName" : "Sven",
  "lastName" : "Pfleiderer",
  "email" : "sven@roothausen.de",
  "locale" : "en_US",
  ...
}
```

Listing 7: Ad-Hoc query submitted via the MongoDB shell

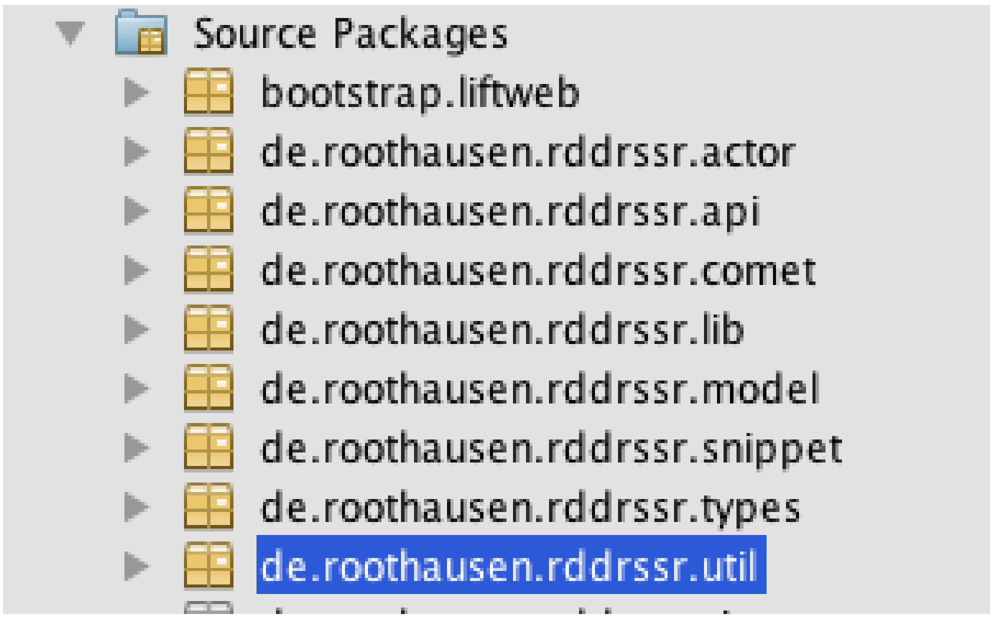
3.1.4 SBT

The Simple Build Tool (SBT) is used to package the application, manage dependencies and to automate the build process. At first Apache Maven was used to to that job but since SBT is the tool preferred in Lift 2.0, the project was migrated to use SBT instead of Maven.

3.2 Application architecture

3.2.1 Big picture

Packet structure The project is structured in two namespaces: "boot.liftweb" and "de.roothausen.*". While boot.liftweb only contains some configuration options for the Lift framework itself, the whole application code is located in de.roothausen.*. The current state of all used packages can be seen in Figure 5.



Source Packages
- bootstrap.liftweb
- de.roothausen.rddrssr.actor
- de.roothausen.rddrssr.api
- de.roothausen.rddrssr.comet
- de.roothausen.rddrssr.lib
- de.roothausen.rddrssr.model
- de.roothausen.rddrssr.snippet
- de.roothausen.rddrssr.types
- de.roothausen.rddrssr.util

Figure 5: Structure of project packets

API Data flow The most important part of the application is its ability to manage incoming requests over an API defined in Section 2. As seen in Figure 6, all API requests are routed through a central dispatcher. This dispatcher redirects the requests to utility methods of the corresponding objects: ServerInfo for the "/meta/"-namespace and User for "{userName}" where userName is the name of a valid registered user on this system.

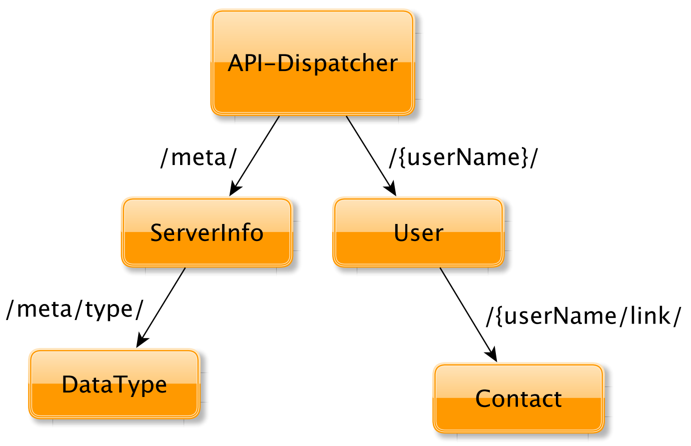


Figure 6: API dispatch overview

If a request needs to display a special data type or contact data has to be modified, the request will be delegated to the right object responsible for that particular request.

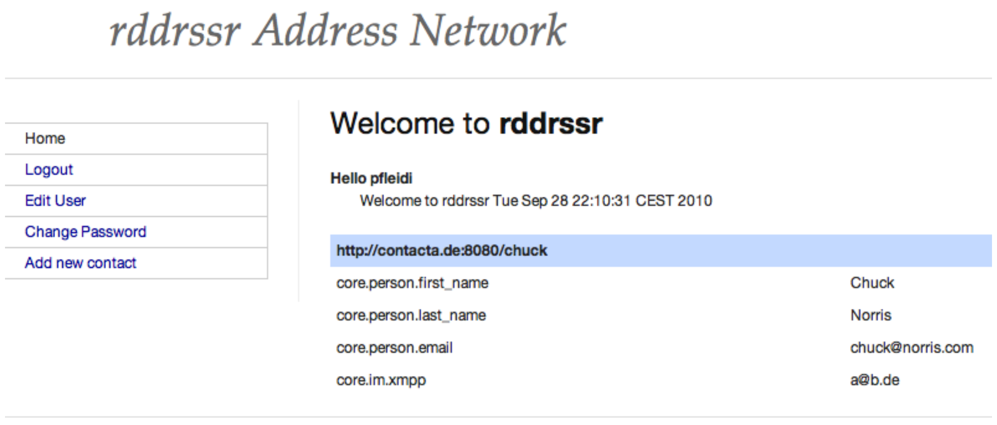
User-Interface The User-Interface shown in Figure 7 is clean, very simple and only offers elements for the most important actions to the user: There are some forms to sign up, to view and change your data and to add new contacts. If one is logged in, a list of all added contacts is shown to the user.

3.2.2 Configuration

Because the application uses as much default values as possible, not much configuration is needed to get started. If an additional email server is needed to send confirmation emails, this can be done in property files.

```
mail.smtp.auth=true
mail.smtp.host=localhost
mail.smtp.user=foo
mail.smtp.pass=bar
mail.smtp.starttls.enable=false
mail.smtp.ssl.trust=*

```


rddrssr Address Network

Welcome to **rddrssr**

Hello pfleidi
Welcome to rddrssr Tue Sep 28 22:10:31 CEST 2010

http://contacta.de:8080/chuck	
core.person.first_name	Chuck
core.person.last_name	Norris
core.person.email	chuck@norris.com
core.im.xmpp	a@b.de

Figure 7: Screenshot of the current UI

```
rddrssr.emailfrom=rddrssr@roothausen.de
rddrssr.serverinfo.name=rddrssr.io
rddrssr.serverinfo.software=scala foobar
rddrssr.serverinfo.baseurl=http://contacta.de:8080/
rddrssr.serverinfo.version=1
```

Listing 8: default.properties file for configuration

All configuration options are included in the sample file featured in Listing 8. All entries beginning with mail.* are actually only needed if confirmation emails should be used to confirm email addresses of users or if password reset messages have to be mailed to a user.

All options beginning with rddrssr.* are mostly responsible for server metadata. For testing purposes, the defaults will work fine.

3.2.3 Companion objects

Scala does not support static methods, like Java does. As an alternative, Scala supports singleton objects which are defined by the “object”-keyword which can be used instead of the “class”-keyword. These singleton objects only have one instance running and this instance can be accessed by its name. If one wanted to call a method bar() on an object Foo, all he or she had to do was to call Foo.bar().

Companion objects can have the same name but have other inheritance relationships than a corresponding class. If there is a class with an object of the same name, this object is called companion object.

All the data models are constructed after that principle: All database objects are instances of a class and all utility methods that are not bound to a instance of this class are located in the corresponding singleton companion.

```scala
class DataType extends MongoRecord[DataType] with MongoId[DataType]
    {

  object cname extends StringField(this, 50)
  // more attributes of an instance

  def toJson: JObject = {
    // generate JSON representation of this instance
  }

}

object DataType extends DataType with MongoMetaRecord[DataType]
  with LiftActor with Loggable {

  def getJson(name: String): Box[JValue] = {
    /* get database object and return obj.toJson
       if that fails, return an error object */
  }


  ...
}
```

Listing 9: Usage of companion objects for data models

3.2.4 API-Dispatch

Lift already contains a trait called RestHelper. It provides a method called serve() which is the base building block of the whole API dispatch. With the help of pattern matching, all incoming requests can be routed to the responsible objects based on their URL patterns. The request are also filtered based on the HTTP-verb used to make this request. GetRequest means HTTP GET, PostRequest means HTTP POST and so on.

For example, the pattern Req(”meta” ::”index” :: Nil , ”json”, GetRequest) means it is is a request of type HTTP GET to the url */meta/index.json*.

```scala
serve {
  // metadata namespace
  case Req("meta" ::"index" :: Nil, "json", GetRequest) =>
```

```
    errorDispatch(ServerInfo.getJson)
    case Req("meta" :: "types" :: "index" :: Nil, "json", GetRequest
      ) =>
    errorDispatch(ServerInfo.getJsonTypes)
...
    // user namespace
 ...
  case r@Req(userName :: "relation" :: "new" :: Nil, "json",
      PostRequest) => {
        dispatchWithContainer(createContainer(r)) { (container) =>
          User.respondToRelationCall(userName, container)
        }
    }

}
```

Listing 10: Call of serve() inside RestHelper

The methods dispatchWithContainer and errorDispatch are utility methods which execute passed in functions and return either the expected result of the function or an error object. If an error object is returned, the framework is able to convert it to the matching HTTP status code and show a human readable error message.

3.2.5 Database objects

As mentioned before, all database classes are constructed to have companion objects. The classes inherit a lot of functionality from MongoRecord which is part of the MongoRecord library. The companions inherit from the classes themselves and mix-in a MongoMetaRecord trait. This trait contains methods used for this special kind of companion object.

Figure 8 shows how the data model is composed of collections of data and their attributes. These attributes contain the needed data. It is also possible to use references which refer to entries in other collections. The main collection in this application is the one where user data is stored. This user collection has references to contacts of a user.

```
class Attribute extends MongoRecord[Attribute]
  with MongoId[Attribute] {
    def meta = Attribute
    object value extends StringField(this, 255)
}

object Attribute extends Attribute
  with MongoMetaRecord[Attribute] {
```

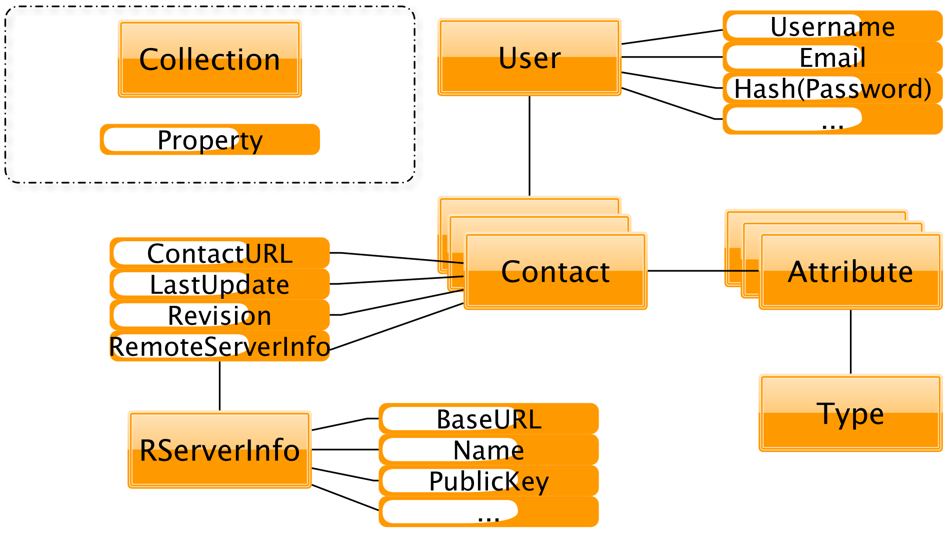


Figure 8: Draft of the database model

```
    def createRecord = new Attribute
    override def collectionName = "attributes"
}
```

Listing 11: Simplified database object

3.2.6 JSON-Parsing

JSON generation and parsing is a really important part of the application. Fortunately, the Lift framework already includes a really great module to work with JSON. There is an integrated Domain Specific Language (DSL) which allows to generate JSON in an intuitive way and there is also a mechanism which allows to map JSON code to Scala's Case-Classes. This mechanism is even powerful enough to check if the parsed input contains the right types and is able to map these values to attributes of Case-Classes.

As seen in Listing 12, there are some modules which need to be imported. After that, the generation of JSON is really simple and easy readable: The arrow (–>) indicates an attribute value relationship and the tilde (˜) indicates that at this point a new attribute is defined.

```
import _root_.net.liftweb.json._
import _root_.net.liftweb.json.JsonDSL._
import _root_.net.liftweb.json.JsonAST._

// method to generate JSON
def toJson: JObject = {
  (("name" -> name.value) ~
    ("cname" -> cname.value) ~
    ("information" -> information.value) ~
    ("revision" -> revision.value) ~
    ("format" -> format.value) ~
    ("occurences" -> occurences) ~
    ("length" -> length.value))
}
```

Listing 12: Generation of JSON code

Listing 13 shows the parsing of JSON read from a file. The case class contains all attributes of the format which should be parsed. If some attributes are missing, an exception is thrown. This allows a convenient validation of incoming data and a really simple way to map JSON content to objects.

```
// defining the structure
case class Type(name: String, cname: String,
  information: String, format: String,
  length: Int, revision: Int, occurences: List[Int])

def parseMethod(filenName: String): Type = {
  // read json from file
  val data = Source.fromFile(file, "utf-8").getLines.mkString
  val json = JsonParser.parse(data)
  // implicit return of the exctracted value
  json.extract[Type]
}
```

Listing 13: Parsing of JSON

3.2.7 Actor system

Incoming HTTP requests should complete as quickly as possible. To allow that, situations in which the system is blocked by some I/O-operation need to be avoided. As a result of this requirement I/O-operations are mostly encapsulated in some kind of actor which allows to run these operation in parallel.

Because the only HTTP-client included into Lift is the synchronous Apache HTTP-Client, it was chosen to encapsulate this library within an actor. To allow some methods to be executed asynchronously, some classes include the LiftActor trait to execute some methods in a non-blocking manner.

```
object HttpClientActor extends LiftActor
  with Loggable {

  protected def messageHandler = {
    case HttpGetRequest(url) => httpGet(url)
    case HttpGetRequestWithCallback(url, callback) => callback !
        httpGet(url)
  }


  private def httpGet(url: String): HttpResponse = {
    // implementation
  }

}
```

Listing 14: Simplified HTTP Actor

4 Lessons Learned

4.1 Problems == Opportunities

This project was intended to be a proof of concept for a newly invented system, implemented using bleeding edge technology. This normally means that it is likely that problems occur nobody has even thought of. Because of this uncertainty, it was accepted and sometimes even embraced that something unexpected would happen.

Dealing with new technology always means learning. In this project this meant looking into an implementation because some documentation didn't exist and trying to work around some errors nobody had foreseen. This section is intended to help others in a similar situation.

4.2 Read framework code

Lift 2.0 was, at the time the project was implemented, a moving target: There were changes in some APIs, new features were added and a lot of the documentation wasn't ready or didn't exist.

Sometimes the only way getting into some APIs, was to read their source code. In this project it was really helpful, both in understanding the framework and learning the language itself, to be able to read the, quite well written, code of the lift-mongodb-record module. Additionally this code provided a lot of ideas and patterns which helped implementing other parts of the application.

Retrospectively seen, reading code did take a lot of time but it was absolutely worth it.

4.3 Ask the community

A few times there were situations where reading source code didn't help – either because there was something wrong in the framework itself or because it just wasn't clear why an error did occur. In this case the Lift-community proved to be as "warm and welcoming" as described on the lift website. There were a lot of helpful comments and the questions asked were answered by someone who knew what was going on.

In the beginning of the project, a class called ProtoUser, which normally used the relational mapper of lift, was modified to use the MongoDB backend. After some problems with a PasswordField attribute, the description of that problem [Pfleiderer, 2010] was posted to the Lift mailinglist and the author of the MongoDB module for Lift helped to fix the problem. He didn't just point out were to look for: He introduced a new class called MongoPasswordField to the framework which fixed the error and then he created a sample project which was using the port of the ProtoUser class previously mentioned.

4.4 If nothing else works: println does

This project was mostly written in Netbeans and VIM with plugins for Scala. Even if netbeans was able to understand the folder structure, it didn't have working facilities do debug or profile running code.

Debugging hard problems without knowing what is going on is virtually impossible. So there was a need to be able to get informations without using a debugging facility. The way which always worked was to use the commandline-tool "curl" to simulate a client, to get information of the application via logging statements and to look at network traffic using sniffers like Wireshark or TCPDump.

4.5 Scala works well for web-development

Scala itself has proven, that it is elegant and expressive enough to provide a really good programming experience. Even though it is statically typed, it was possible to write small and concise code that did a lot of things. Using features like function literals, pattern matching and traits to generalize code fragments the author was also able to re-use a lot of the written code. This lead to shorter classes which were easier to review.

Even if the tooling for Scala isn't really good at the moment, the language itself provides a lot of features which make development of applications a lot more fun.

5 Perspective

5.1 Protocoll Specification Rework

5.1.1 Nobody is perfect

Nobody is perfect and the protocol proposal is no exception: There are some parts which could be improved.

Some parts have to be specified because there wasn't time to do so. Other parts work fine but have to be renamed because the name is not semantically clear.

Some of the minor mistakes were fixed during the development process. But, there's still some baggage which has to be addressed. This section addresses some of the ideas to improve the protocol proposal.

5.1.2 Get really RESTful

In it's current state, the protocol uses only HTTP GET and POST to do all communication between servers. This design was chosen because authentication information had to be transported somewhere within a request.

The goal is to get rid of some of the POST requests and replace them with PUT and DELETE where it is semantically advisable. To do that there has to be another way to transport authentication data.

At the moment the most appealing choice is to transport authentication data such as checksums or signatures via custom fields added to the HTTP header of a request. If this scheme works and can be easily implemented, update and delete requests do not need to be implemented using POST requests.

5.1.3 Model clear semantics

To readers, which are new to the protocol, it is often not clear what the difference between a relation and link request is and how to use them. To address this, the model of an initiation requests and it's URL-structures need some redefining to represent the underlying semantics in a clear and concise way.

5.2 Implementation Rework

Not only the protocol, but also the implementation itself has to be re-factored. The most probable action would be to take the most important parts and adapt them to new protocol specification. The parts which were written at the beginning, like the user management and the notification email support, are the ones which are most error-prone so it would be a good idea to write them from scratch.

Additionally, the current implementation does not contain any interfaces to support clients. This and many other features have to be implemented, as well.

6 References

References

Crockford, D. (2006). Rfc4627: Javascript object notation.

Ecma International (1999). Ecmascript language specification. Standard ECMA-262.

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, Citeseer.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Rfc 2616, hypertext transfer protocol – http/1.1.

Ghosh, D. (2008). Scala actors 101 - threadless and scalable. `http://java.dzone.com/articles/scala-threadless-concurrent`.

Haarmann, M. (2010). Distributed management of personal contact data.

Jones, H. and Soltren, J. (2005). Facebook: Threats to privacy. *Project MAC: MIT Project on Mathematics and Computing*.

Pfleiderer, S. (2010). Protouser with mongorecords - google-groups. `http://groups.google.com/group/liftweb/browse_thread/thread/add984be7d476a43/da71c2f2c349ec39l#da71c2f2c349ec39`.

Wahl, M., Howes, T., and Kille, S. (1997). Lightweight Directory Access Protocol (v3). RFC 2251 (Proposed Standard). Obsoleted by RFCs 4510, 4511, 4513, 4512, updated by RFCs 3377, 3771.