Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

REST SERVER IN JOLIE
a Practical Case

Relatore:
Chiar.mo Prof.
DAVIDE SANGIORGI

Presentata da:
RICCARDO SIBANI

Correlatore:
Dott.
SAVERIO GIALLORENZO

Sessione
Anno Accademico 2015/2016

This thesis is dedicated to my parents.

For their endless love, support and encouragement ...

Introduzione

In questa tesi si è cercato di unire due dei paradigmi emergenti nell'ambito web: RESTful web development e Service-Oriented Programming.

Da un lato, REST è il principale paradigma architetturale delle applicazioni web, le applicazioni REST hanno una struttura procedurale il che evita l'utilizzo di meccanismi di handshaking.

Benchè REST dia una struttura standard per l'accesso alle risorse delle applicazioni web, la programmazione web lato server è spesso poco modulare e per questo complicata.

Il Service-Oriented Programming invece ha come uno dei principi fondamentali la modularizzazione dei componenti.

Applicazioni Service-Oriented sono caratterizzate da moduli indipendenti che consentono di semplificare sensibilmente lo sviluppo di applicazioni web.

Purtroppo, ad oggi, ci sono pochi esempi di integrazione delle due tecnologie: pare pertanto sensato provare ad unirle.

In questa tesi si esplorano i metodi per conseguire tale risultato con un'applicazione di un server che permetta la gestione di documenti e di note da parte di differenti utenti registrati, chiamato MergeFly.

La autenticazione viene implementata attraverso la gestione di token di accesso e scartato un meccanismo di gestione delle sessioni che sarebbe stato fuori dalla "pure RESTness", anzichè molto spesso utilizzato, mediante login.

MegeFly, una volta definitene le caratteristiche, sarà preso come base per la realizzazione dei verbi HTTP attraverso una programmazione SOA.

Nel documento verranno innanzitutto definiti 1. i limiti e le caratteristiche dell'applicazione, 2. la tecnologia SOA, con nello specifico 3. le caratteristiche di Jolie, 4. la tecnologia REST ed infine verrà proposta una 5. implementazione Jolie-REST attraverso l'applicazione MergeFly. Nella conclusione si valuterà l'effettiva validità e funzionalità dei risultati ottenuti dall'ipotesi riguardo il binomio Jolie-REST.

Introduction

The aim of this thesis is to merge two of the emerging paradigms about web programming: RESTful Web Development and Service-Oriented Programming.

REST is the main architectural paradigm about web applications, they are characterised by procedural structure which avoid the use of handshaking mechanisms.

Even though REST has a standard structure to access the resources of the web applications, the backend side is usually not very modular if not complicated.

Service-Oriented Programming, instead, has as one of the fundamental principles, the modularisation of the components. Service-Oriented Applications are characterised by separate modules that allow to simplify the development of the web applications.

There are very few example of integration between these two technologies: it seems therefore reasonable to merge them.

In this thesis the methodologies studied to reach this results are explored through an application that helps to handle documents and notes among several users, called MergeFly.

The MergeFly practical case, once that all the specifics had been set, will

be utilised in order to develop and handle HTTP requests through SOAP.

In this document will be first defined the 1. characteristics of the application, 2. SOAP technology, partially introduced the 3. Jolie Language, 4. REST and finally a 5. Jolie-REST implementation will be offered through the MergeFly case.

It is indeed implemented a token mechanism for authentication: it has been first discarded sessions and cookies algorithm of authentication in so far not into the pure RESTness theory, even if often used).

In the final part the functionality and effectiveness of the results will be evaluated, judging the Jolie-REST duo.

Contents

List of Figures

List of Tables

Chapter 1

MergeFly, what is it?

MergeFly is a web application that allows people to create and dynamically compose their documents during an event.

1.1 The problem

Many times students cannot attend lessons because they are ill, working or they just need to go out of the class for a short time; for sure many of them just do not pay attention at an important concept. When they take notes they might miss some important parts.

1.2 The idea

The idea is to develop a platform to manage events. Like lessons, working meetings or any kind of occurrence with a group of people and some notes that must be taken. In such a platform, joining an event gives the possibility to create the related document and to start taking notes. Whenever another user writes a note in the same event, we import it in our document in the

correct chronological position.

To guarantee an easy and dynamic experience to the user, the platform structures the import process in this way: the user sees the document as a page at the center of the window and the notes ready to be imported as blocks on one side of the layout. In this way it is easy to read and import them with a single click (or bt drop-dragging them inside the document).

It is possible to import notes from different users and create a documents were to study once the lessons are finished and students need to revise the lectures, it will indeed be possibile to download the written document into a PDF file and print it or use it in many different ways.

The platform is also useful for those people who cannot attend the lesson or to participate to a meeting.

Figure 1.1: MergeFly's Logo

1.3 Concept of Document

The document belongs to an event and, of course, every note is displayed in chronological order. This fact is important in order to understand the concept of document flow: when somebody imports notes from others, these pieces of document will be shown one after the other based on thei creation date.

The total flow of an event is respected by notes taken by the users. Indeed, the information comes from a physical speaker in the room where the event takes place. We can consider the speaker as the trigger of all the notes, when he will explore a topic, this will be quoted by users generating a certain amount of notes. These notes are going to be more recent than the previous topic and older than the next topic that the speaker discussed about. This method guarantees time-flow/accuracy into the document.

1.4 Concept of nodes and notes

Since this is a platform that allows users to write, read, share and import notes, it seems obvious to optimise the space into both the database that stores them and the server that handles them. The concept of note as described above is a container of a single type of information and with only one information in it. We can image the document as a flow of information chronologically ordered, each information is a piece of document and it can be independent. Examples of type of information are: text (usually a paragraph), snippets of code, images, and links. Each of this piece can be *stand alone* (even though it is meaningless without a context). This gives us the opportunity to store the information (which is characterised by a large amount of data) in different records. When someone imports one specific note, we can avoid to copy all the information contained into it and copy only the reference (node) of that note.

The document is a list of ordered nodes, referenced to the notes stored into the database.

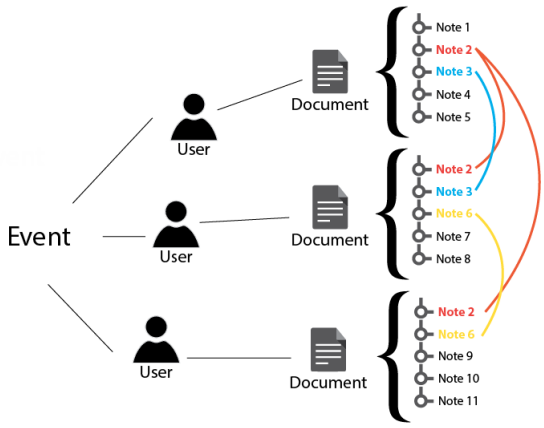

Figure 1.2: Concept's Diagram

1.5 The Platform

MergeFly should be available through a web platform as well as a desktop and tablet application. Event organisers are allowed to post events (e.g. from Facebook or Eventbrite) and invite participants. Businesses and organisations have to pay a fee that could vary depending on the number of participants and the budget.

Attendants, after a registration, can create their *event-related* documents and see what other participants are writing. If someone fails to catch a part or just wants to compare what has been typed, she can import notes from other documents. The user interface is a critical part of the design: it needs to be simple to understand and must advise the user if he is missing some important arguments. Furthermore, it is essential to offer a desktop application for the simple reason that a writer usually feels the document as an intellectual property and wants to keep it secure inside a computer. The user should think of the application as a simple text editor with few additional and extremely useful features.

1.6 Monetisation

Fee on events the platform is completely free for event attendants and would be accessible after a registration which would offer a series of services. Companies and those who want to organise an event (e.g a class or a workshop) have to pay a fee for each occurrence. This fee can vary depending on the number of participants, the budget and the number of required services.

Affiliate Marketing this source of income will come in a second moment, after the number of registered users has reached a critical mass. The platform will sponsors a series of event- related products and services and retain a percentage for each purchase made through the sponsorship.

Lead generation is quite similar to affiliate marketing, in this case it is not necessary that the user buys something, what is necessary is to create a list of users potentially interested in one product and sell this list to businesses that require this service.

Analysis is the generalisation of the latter two points as the platform tracks every user. Some personal data, such as private documents, will be protected. In any case, the major part of data could be sold and, of course, we have the opportunity to offer these data to the companies which could be interested to know where and how to develop their business.

1.7 Competitors

The market is quite empty, but there are possible competitors:

Dropbox has recently introduced a service of annotations potentially similar, to some extents, to our service.

Evernote allows you to create documents and access to them from desktop application, tablet and smartphone.

Google Docs similar to standards graphical word processing program online with few features in order to share and modify other users' documents.

1.8 Specifications

MergeFly is a platform that allows users to create and import notes of users who join the same event.

Users are characterised by:

- Id
- First name
- Last name
- Date of Birth
- Subscription Date
- Type
  - Administrator
  - Premium
  - Basic
- Profile Picture

- Last Latitude
- Last Longitude
- Password
- Mail
- Token

The user can create different events where he has the possibility to add other users who might accept or refuse the invitation.

Events are characterised by:

- Name
- Place
- Creation Date
- Start Date
- End Date
- Creator Id
- Description
- Category Id

The relation between user and event is defined by the id of the event, the id of the user and by the acceptance status (accepted, declined, not defined).

Each event is characterised by a physical place which will be shown on map. About the place, these are the known informations:

- Latitude

- Longitude
- Name
- Address
- Cap
- City
- Country

To each event is correlated one category, defined by:

- Name
- Description
- Colour

Users can register into several groups, each Group is characterised by:

- Id
- Name
- Creation Date
- Image
- Description

About the users subscribed into a Group it is required to know:

- User id
- Group id

- accepted status to an event
- role
  - Admin
  - Normal
- Subscription date

Each user can create one document per joined event. Within the document it is required to know:

- Creator Id
- Name
- Event Id
- Creation Date
- Public / Private Document

Only premium users can create private documents. Basic users can only create public documents.

Every user can create notes to insert inside documents. The note will be responsible to memorise information such as

- Id
- Type
  - Image
  - Text
  - Code

  - Link

- Title
- Content
- Description

Each note might be one image, snippet of code, text (usually one paragraph) or link:*It is the sum of notes that creates the Document.*

Each note can be imported by several users. In order to perform it, it is necessary to use the concept of *node* that allows to link one note to many documents.

Each node is characterised by

- Document Id
- Note Id
- Creation Date
- Id

If, after having imported one note, the user decides to modify it, it will be created a copy of it to preserve the original.

Chapter 2

Service Oriented Architecture and Microservices

2.1 Service Oriented Architecture

A Service Oriented Architecture (SOA) is an architectural pattern that provides functionalities of applications (called services) via a shared communication protocol and usually available over the network.

Service Oriented Architectures try to ease the development and maintenance of big structures and services by logically separating them into smaller pieces.

One single piece represents one problem which might be separated again and again: each piece of the problem is solved through a single service. This paradigm has been used over the years in order to solve problems not only in IT fields, it can be considered as a reworked version of *Divide et Impera* approach.

Service-orientation is not related to a specific language since it is a paradigm and can be implemented in several ways.

One service can be connected to other services through protocols. SOAP (Simple Object Access Protocol) is a recent technology for supporting the creation of SOAs in charge to defining and communicating the parameters, the modality, and the protocol which is going to be used during the communication.

2.2 Web Service Approach

According to OASIS standards [1] Web Services can implement service-oriented architectures. They can collaborate between them like bricks in a wall, independently from the used platforms and the programming languages. These services can collaborate among different platforms with different paradigms and can be extended or extend other web applications or any kind of services.

Each Service-Oriented Architecture can be implement as server provider, consumer or both, generating a sequence of services that communicates one with the other.

2.2.1 Service Provider

It is Service Provider when a service is placed (delivered) on a Service Consumer request. The Service provider, if able, takes in charge the request and computes a response, possibly using other services. The Service Provider, in addition, guarantees its services and their description.

2.2.2 Service Consumer

It is Service Consumer when the Service sends one request to a Service Provider. When the request is satisfied (*Request Response*) or just sent (*One*

Way), the result can be computed.

2.3 WSDL

2.4 Characteristics

Service Discovery Services are designed in order to be reached by external services through some discovery mechanisms. It can happen through a service registry/catalogue or via universal resource locators (URL) and be moved without conditioning any system.

Self Contained each service must be independent: it has to work alone and it can be part of one or many systems. This, in fact, is the plus of the service oriented architecture: one service does not have to be implemented every time a platform needs it, it can be just reused since it is modular. We can distinguish into several types of modularity:

Modular Decomposability breaking the application in different modules, creates the possibility to manage many little services that are responsibile for one or a few operations. This is extremely smart because it allows the programmer to reuse the code in multiple situations.

Modular Composability it is possible to combine and join several services and to compose new and more complex services. This function requires the single services to be little pieces of independent and reusable software, usable for completely different applications.

Modular Understandability lets the developer or the users under-

stand the service without knowing how other services or the platform are implemented.

Modular Continuity the changes applied into a service (or replace a service with another one) should not influence the other components of the system.

Modular Protection is the module which is in charge of the security of the other services and / or the whole application. It has to handle cyber attacks and handle exceptional events unleashed voluntarily or involuntarily.

Interoperability it is important that every service is able to communicate with other services.These services can use different formats of communication and are differently implemented, this is why protocols and standardised languages, such as XML or JSON are used.

Loosely coupled the ability of the service to work knowing just the interfaces of the other services and to have very few dependencies between the modules of the application.

Contract Based interfaces, polices, and contracts are described and accepted thanks to a Protocol that defines them.

2.5 Pros

Service Oriented Architecture is not related to any specific platforms (it could be implemented with Java, .NET, Jolie, etc.) and, at the same time, it is easily reusable since every component might stand alone.

It is easy to change the order and the parameters of one or several services, since there can be services which coordinate other services: making it easy

to modify the interaction processes.

Each service can be easily replaced, since they are independent entities. To replace a service it is enough to reconfigure the protocol.

Chapter 3

Jolie

It is an orchestration language

3.1 From a simple service

Jolie is a general purpose programming language. Here we consider the characteristics that are relevant to the reader.

3.2 Behaviour

In Jolie, communications are possible thanks to inputPorts and outputPorts. The first ones, when initialised, open a socket and waits for incoming communications. OutputPorts instead are responsible for the delivery of the communications to the selected server which is listening.

The syntax of outputPorts and inputPorts is showed below:

```
1 inputPort portName {
2     Location:
3     Protocol:
4     Interfaces:
```

```
5  }
6
7  outputPort portName {
8      Location:
9      Protocol:
10     Interfaces:
11 }
```

The reader might notice that the syntax is similar. Indeed they have to establish some common configuration to use in order to speak to each other.

Location: is the medium where the service is going to call or receive the communication.

The syntax for TCP/IP sockets, used here to develop web services, is:

```
1  socket://localhost:portNumber
```

The protocol is identified by the keyword *socket*. Jolie supports other parameters like Bluetooth and local memory.

Protocol: it is the protocol used to format the messages. Jolie supported protocols are:

- HTTP
- HTTPS
- JSON/RPC
- XML/RPC
- SOAP
- SODEP
- SODEPS

Interfaces: are the contract which the two services must attend. In Service Oriented Architecture there is no handshake since both players know how to send and receive data. An example of interface will be shown further in this document.

In order to allow the exchange of information between services, we consider two kinds of communication: *Request Response* and *One Way*.

One Way: it waits for the Service Consumer communication and compute the service. It does not return anything, so we can consider it as an utility to call when no results are expected.

Request Response: it waits for a message, it computes a response and returns it. It is clearly possible that this operation requires time especially when several services are called, this means, strictly speaking, that if only one service fails or has communication problems and the fault is not handled properly: all the *request responses* will fail and return a wrong result (or, dramatically, not even return anything).

Syntax declaration (in inputPort)

Request Response

```
requestResponseName( request )( response ){
// CODE
}
```

One Way

```
oneWayName( request ){
    //CODE
}
```

Call (in outputPort)

Solicit Response

```
requestResponseName@OutputPortName( request )( response )
```

It is a notification when a request response has been called.

```
onewayName@OutputPortName( request )
```

3.3 To a link of services

According to the thesis of Fabrizio Montesi on the Jolie Grammar [2] each output and input Port needs interfaces that declare the supported Request Response and One Way operations.

Below is reported an example of interface in Jolie.

```
interface interfaceName {
  OneWay: onewayName( requestType )
 RequestResponse: requestresponseName( requestType )( responseType )
}
```

Each parameter specified (both in the request or response parenthesis) must have a type. The default types are:

- bool
- int
- long
- double
- string

- raw
- void

All of these can be combined and create other types in a tree.

Syntax:

```
type typeName: basicType\\
```

example of combined types are:

```
type Vote:void {
    .name:string
    .choice:string
}

type Poll:void {
    .options*:string
    .votes:void {
        .href:int
        .vote*:Vote
}
```

3.4 Basic Structure & Error Handling

We refer the interested reader to [3]. However, the previous introduction on the language should give the necessary knowledge to the reader to understand and apply the examples showed forward.

3.5 Sessions

Jolie provides an automatised system to handle sessions through the *cset* keyword, with the following syntax:

```
cset {
  nameVariable: typeName.anotherVariable
}
```

Where nameVariable is the name of the variable to initialise and typeName is the type of the variable itself.

nameVariable will reference to anotherVariable parameters which is stored inside the type typeName, here an example.

```
execution{ concurrent }

type MessageType : void {
    .sid : string
    .message : string
}

type DeleteChat : void {
    .sid: string
}

type setID : void {
    .sid? : string
  .data? : string
}

interface Message {
    RequestResponse: start( MessageType )( setID )
}

inputPort Server {
  Location: "socket://localhost:8123"
    Protocol: http
    Interfaces: Message
```

```
25 }
26
27 cset {
28   session: MessageType.sid DeleteChat.sid
29 }
30
31 main
32 {
33   [start( request )( response ){
34     csets.session = response.sid = new
35     println@Console( request.message )()
36   }]
37 }
```

In this short example, when the *start* service is called by the client, it sets the *cset.session variable* equal to *response.sid* with a unique id (session key) so, supposing that our server is executing concurrently, it will handle different sessions from different clients.

Jolie also implements the session automatically, giving back the session key within *response.sid* in the setID type and it stores it locally in the browser of the user.

To unset the session id, it is sufficient to overwrite it with an empty value

```
1 [deleteChat(x)(response){
2         response.sid = ""
3 }] { nullProcess }
```

DeleteChart sets the .sid into the response variable to "" (or null), cset will change the stored variable on the client to "".

3.6 Cookies

Using cookies is similar to using sessions, it is enough to set properly few parameters within the http protocol:

```
inputPort Server {
Location: "socket://localhost:nnnn"
Protocol: http{
.cookies.session = "sid";
// take all the values inside variable "sid"
.cookies.session.cookieConfig.expires -> date
}
Interfaces: SomeInterfaces
}
```

In this example, we are setting inside the http protocol the .cookies node with the name of the cookie we are going to use (session) and then we assign the name of the variable we want to store on the client inside a string ("sid").

Of course, the type of the request should contain the .sid field as a parameter: and it must be into the response variable if we want to set it.

Chapter 4

REST

4.1 What is REST

REST [4] stands for *REpresentational State Transfer*. It relies on stateless, client - server cacheable communication protocols which usually is the HTTP protocol. The outstanding feature of REST is that every machine and every human being can reach all the contents without knowing anything beforehand about the resources the server is hosting. By contacting the server it is possible to receive the specific location of the resources and the required files.

> "A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media type."
>
> Roy Fielding [5]

In fact, a simple get request (without any parameter specified in the URI) should return the specification implementations and the available operations.

REST is composed of verbs and resources which are determined a priori, the sum of the operations which can be executed from both other servers and users, and create the so called *Hypermedia Controls* which allows other services to know the possible commands they can run. In these case the REST service becomes RESTful because every machine can explore it discovering his structure through some commands.

4.2 Principles

There are basically 5 fundamental principles to follow in order to create a REST server:

Principle one: Everything is a Resource in REST architectural style, they are accessed through URI (Uniform Resources Identifiers) which is a string of characters used to uniquely identify a resource which is allocated into a server. Resources are typically links on the web as files, images, videos, webpages, etc.

Principle two: Every resource is identified by a unique id, this means that the same resource might be hosted on different URIs but different information cannot be hosted in the same URI. URI is composed by URL and URN.

URL (Uniform Resource Locator) is the template address of the resource of the server and it is in charge to properly represent every source data which will be asked (e.g., images, HTML sources, JSON datas, XML datas, music files).

URN, instead, is the name of the variable that changes in the URI depending on the specific item requested.

Principle three: Interfaces should be as easy and simple as possible in order to be machine readable. Many times it is possible to access the same resource in different formats such as XML, JSON, and most common humane readable protocol like HTML: which shows results graphically and not only as a string. This can be done using HTTP protocol methods with verbs as GET, POST, PUT, and DELETE. Combining HTTP methods and resource names it is possible to create a uniform access to modify, create, update, and delete contents. An example of request to a REST server could be:

```
1 The user wants to get all the information about user called
2
3 Philip: http://serverhost.dom/Users/Philip
4
5 Users is part of the URL
6 Philip is part of the URN
```

Principle four: Communication is done by representation. Every request and response we send or receive is done through a representation of the sent object (or objects). When an object is sent in XML or JSON format, the parameters should be written inside a string: that string is a representation of the objects.

Principle five: Be Stateless. Every request the client makes to the server has to be stateless, which means it has to be unique and independent from each others.

If, for instance, an application needs to login in order to run a GET request, first it sends the first HTTP request with the login parameters,

then if the credentials are correct and a success message is received (with a key in order to validate the get request), it is possible to go ahead with requests which require authentication.

4.3 CRUD with REST

CRUD stands for *Create, Remove, Update and Delete*; these are the operations (or verbs) that MergeFly allows the client to use and it is possible thanks to the HTTP protocol and its implemented verbs.

According to W3C there are 7 HTTP verbs [6].

OPTIONS

GET

HEAD

POST

PUT

DELETE

TRACE

CONNECT

To the purposes of MergeFly, only 4 of them are used: 1. GET 2. POST 3. PUT 4. DELETE,

Here below is the description of each verb:

Create: in order to create, the HTTP POST verb is needed and, by sending the proper information, it is possible to create new records of a particular resource. In the URI it is specified the type of resource to create

and in the body all the properties and the information are written in order to assign them to that specific object.

```
1 POST http://www.serverhost.dom/users
```

Read: in order to read, the HTTP GET verb is needed, it is probably the most used method since it allows clients to receive any resource information. When the GET succeed, it returns information with XML or JSON representation and code 200 (OK). According to HTTP specification GET requests are used only to send and display data and any update or delete operation is not possible.

```
1 GET http://www.serverhost.dom/users
```

Update: in order to update, the HTTP PUT verb is needed, knowing the resource URI we want to update, giving in the body the updated representation of it, in order to proceed. This is usually used to replace a specific resource rather than modify one single characterisation because in the body all the information of the resource are needed.

```
1 PUT http://www.serverhost.dom/users/1234
```

Another verb used to update is PATCH which work pretty similar to PUT, but in the body is specified the single information to update rather than all the content.

```
1 PATCH http://www.serverhost.dom/users/1234
```

Delete: in order to delete, the HTTP DELETE verb is needed, with the URI of the specific resource to delete permanently.

```
1 DELETE http://www.serverhost.dom/user/1234
```

Chapter 5

MergeFly - REST

5.1 Why Jolie

Jolie eases the use of microservices and distributed computing. These are key features in the designing of a REST server: merging these two technologies seems an interesting challenge, which is one of the contributions of this thesis.

It is indeed clear that if it is required to add one resource to a REST server with Jolie (thanks to its modularity) it will be enough to call a proper service developed *ad hoc*. With a single interface, it is possible to overcome the limits of the server.

Supposing that it is required to change the server or to implement new distributed systems, it is enough to change the port of the interface and the system can easily scale.

5.2 Architecture

The main problem is that Jolie is a Service Oriented programming language and in order to user REST we need to accept the communication and *translate* the HTTP request in Jolie.

It has indeed been possible thank to routing mechanisms which allow to read the header and body of the HTTP communication and, translating it in a Service Oriented Paradigm: the message is sent to the main service which will handle and send it to the others appropriate Services.

Therefore in the next pages it will be implemented a *Jolie - REST Server*.

5.3 Structure of the Server

5.3.1 Routing

The routing service developed by Fabrizio Montesi for REST Servers [7] is a good example of the power and the versatility of Jolie Language. It is indeed extremely easy to set up a routing mechanism based on the url that has been given by the client.

Here below, the interface that will be introduced shortly (In this paper I will refer to "main service" or main.ol as the service which is launched from terminal, and that will embed the router):

```
1 type MakeLinkRequest:void {
2     .operation:string
3     .params:undefined
4     .method?:string // default: get
5 }
6
7 type Route:void {
```

```
8       .method:string
9       .template:string
10      .operation:string
11  }
12
13  type Resource:void {
14      .name:string
15      .id:string
16      .template:string
17  }
18
19  type Config:void {
20      .host:string
21      .routes*:Route
22      .resources*:Resource
23  }
24
25  interface RouterIface {
26  RequestResponse:
27      config(Config)(void),
28      makeLink(MakeLinkRequest)(string) throws BindingNotFound(
            void)
29  }
```

The two operations that are going to be implemented are config and makeLink: they are both Request-Responses.

The first one `Config` has the following parameters:

host: with the URI path information

routes: an array (*) with all the information about the paths that the router is going to handle: one for every http verb to implement. In this server the verbs will be GET, POST, PUT and DELETE.

resources: it is the link between the routes and the host. It has all the necessary information in order to build the router. Fetching the uri we can find the objects to represent and their variables surrounded by brackets {}.

Below, is reported the logic of the router:

```
1  define addResourceRoutes
2  {
3      routes[#routes] << {
4          .method = "get",
5          .template = resource.template,
6          .operation = resource.name + "_index"
7      };
8      routes[#routes] << {
9          .method = "get",
10         .template = resource.template + "/{" + resource.id + "}"
               ,
11         .operation = resource.name + "_show"
12     };
13     routes[#routes] << {
14         .method = "post",
15         .template = resource.template,
16         .operation = resource.name + "_create"
17     };
18     routes[#routes] << {
19         .method = "put",
20         .template = resource.template + "/{" + resource.id + "}"
               ,
21         .operation = resource.name + "_update"
22     };
23     routes[#routes] << {
24         .method = "delete",
```

```
25          .template = resource.template + "/{" + resource.id + "}"
                ,
26          .operation = resource.name + "_destroy"
27      }
28  }
29
30  init
31  {
32      config( config )() {
33          routes -> config.routes;
34          resource -> config.resources[i];
35          for( i = 0, i < #config.resources, i++ ) {
36              addResourceRoutes
37          }
38      }
39  }
```

Jolie provides a procedure, called `init`, that encloses instructions that are going to be executed when the service starts. This is particularly useful in Servers that are listening for connections because it allows to first compute some instructions, set all the variables and load all the other services, and external procedures that will be required by the invoked service. When the `init` procedure `config` is called, it builds the paths with the resources set by the invoker and it sets the host address through config.host.

The cycle will call addResourceRoutes for every defined resource and it memorises for each one all the routes for the verbs GET, POST, PUT and DELETE. In order to call the right procedure from the service which actually created them and runs the router, we need a standard and specific nomenclature.

In the following example, is important to highlight that the name in typeResource must be unique for all the verbs implemented as services since

is required in the config procedure.

GET: get method can be distinguished into 2 particular species: with data and without data. We refer to data as the params of the URI (*URN*). (e.g. 2 in /document/2)

If the data is set, then we are looking for a specific record, otherwise we just want to send back a bouquet of possible options of the specific object requested. The procedure name for the first will be `someName_show()()` and for the latter `someName_index()()`

POST: post method is in charge to create a new instance of the object so it will be characterised by `someName_create()()`.

PUT: put method is the same to POST but a semantics and logic distinguo is needed, since the PUT modifies a record and post creates it. The put is characterised by `someName_update()()`.

DELETE: delete method is in charge to delete one instance (from the database). It is characterised by `someName_delete()()`.

Another procedure defined in router service is the Request-Response makeLink()()

```
1 [ makeLink( request )( response ) {
2     if ( !is_defined( request.method ) ) {
3         request.method = "get"
4     };
5     makeLink
6 } ]
```

If the request method is not set, it proceeds with the standard GET method, let us say this is the most conservative behaviour since it does not create nor change or delete any record in our database.

Then the makeLink procedure is called

```
1  define findRoute
2  {
3      for( i = 0, i < #routes && !found, i++ ) {
4          if ( routes[i].method == method ) {
5              match@UriTemplates( {
6                  .uri = request.requestUri,
7                  .template = routes[i].template
8              } )( found );
9              op = routes[i].operation
10         }
11     }
12 }
13
14 define route
15 {
16     findRoute;
17     if ( !found ) {
18         statusCode = 404
19     } else {
20         statusCode = 200;
21         with( invokeReq ) {
22             .operation = op;
23             .outputPort = "App"
24         };
25         foreach( n : found ) {
26             invokeReq.data.(n) << found.(n)
27         };
28         foreach( n : request.data ) {
29             invokeReq.data.(n) << request.data.(n)
30         };
31         invoke@Reflection( invokeReq )( response )
32     }
```

```
33 }
34
35 define makeLink
36 {
37     for( i = 0, i < #routes && !found, i++ ) {
38         if ( routes[i].method == request.method && routes[i].
               operation == request.operation ) {
39             with( expand ) {
40                 .template = routes[i].template;
41                 .params -> request.params
42             };
43             expand@UriTemplates( expand )( response );
44             response = "http://" + config.host + response
45         }
46     }
47 }
```

What the procedure does, is to find if the URI route is set by the client in the HTTP request and to compare the method request itself with the ones defined in its variables. If the URI matches one of the routes saved in the router, it will set the status code to 200 (OK) and proceed otherwise to 404 (Not Found) and quit. Now that the request can be accommodated, it looks for data both in the header and the body of our http request and it calls the appropriate service hosted in our main service.

5.3.2 Call and Set Up the Routing Service

The first thing to do is to include the router file (router.iol) and create the InputPort.

```
1 include "router.iol"
2
3 outputPort Router {
```

```
4     Interfaces: RouterIface
5 }
```

And then embed the router in main.ol file:

```
1 embedded {
2 Jolie:
3     "Router/router.ol" in Router
4 }
```

Embedding the service Jolie simply starts to run the router.ol file from Jolie (the same way it is possible from command line).

The first thing we have to do in our server is to configure the host that will be used in order to listen and catch new communications. Then the array where to save all the URI to implement can be configured.

```
1 config.resources[0] << {
2     .name = "user",
3     .id = "id",
4     .template = "/user"
5 };
```

The array resource is used to define every representational object in our server: if the client wants to call something it has to be defined it here. Inside the variable `config.resource` there is the parameter .name that sets the name of the services the router will call depending on the HTTP request's method invoked. In the example the assigned value for *name* is *user* therefore the callable procedures will be:

- user_index
- user_show
- user_create

- user_update
- user_destroy

It is important to emphasise that none of the procedures must be implemented if, for instance, during the process of requirements analysis it has been defined to not let the client to create users: it is enough to not declare the service user_create inside the protocol and go ahead with all the others. Template parameter is the identifier of the URI: the URI "/user" will be caught and handled according to the proper service. Id is the value that will follow the template, it must be defined but it might be optionally assigned by the client (e.g. general GET).

Once that all the resources are defined, the config procedure might be called giving all the parameters stored in config.

```
1 config@Router( config )();
```

5.3.3 Services Implementation

After having set up the router, it is now possible to start developing the applications itself.

Here, it is presented the standard method that has been used by the writer using the example of document to perform the following operations: 1. general show 2. item show 3. create 4. update 5. delete.

General Show

Assuming that the resource has been configured in init with something similar to:

```
1 config.resources[3] << {
2     .name = "document",
```

```
3     .id = "did",
4     .template = "/document"
5 };
```

It is possible to create the *_index()()* service: since it is a *Request Response* and it will be characterised by the following syntax:

```
1 [ document_index( )( response ) {
2         nullProcess
3 }]
```

Inside the service there must be provided the list of all the available documents, in order to perform it, once service `Document.ol` has been created.

Document.iol interface:

```
1 type DocumentsList: void {
2   .docs* : int
3 }
4
5 interface DocumentIface {
6   RequestResponse:
7     getAllDocuments( void )( undefined ),
8 }
```

Document.ol file:

```
1 include "console.iol"
2 include "document.iol"
3 include "../Database/DatabaseService.iol"
4 include "string_utils.iol"
5
6 inputPort Main {
7     Location: "local"
8     Interfaces: DocumentIface
9 }
10
```

```
11 outputPort DatabasePort {
12     Location: "socket://localhost:8887"
13     Protocol: http
14     Interfaces: DatabaseIface
15 }
16
17 execution{ concurrent }
18
19 main
20 {
21   [ getAllDocuments( request )( response ) {
22         q = "SELECT id FROM Documents";
23         query@DatabasePort( q )( list );
24
25         for (i=0, i<#list.row, i++) {
26         response.docs[i] = int(list.row[i].id[0])
27         }
28     } ]
29 }
```

About the use of DatabasePort, please read below in the Database Section of this Chapter. Once the call procedure is set and the Service Provider is called, *list* variable should contain a list of id showed with the following format:

```
1 type returnFromDB: void {
2     .row* : void {
3         .paramName: undefined
4     }
5 }
```

Where paramName is the name of the field (or fields) of all the columns returned and row is the number of the record[i].

The result is handled and passed back to the Service Consumer.

Main.ol document_index()()

```
[ document_index( )( response ) {
    getAllDocuments@Document( )( documents );
    for( i = 0, i < #documents.docs, i++ ){
        makeLink@Router( {
            .operation = "document_show",
            .params.did = documents.docs[i]
        })( response.href[i])
    }
} ]
```

Main.ol file will compute the result and show it within the href json object:

When the URL requested is *"http://localhost:8080/document"* the output is the following

```
/* Example */
{
  "href": [
    "http://localhost:8080/document/1",
    "http://localhost:8080/document/9",
    "http://localhost:8080/document/11",
    "http://localhost:8080/document/13",
    "http://localhost:8080/document/14",
    "http://localhost:8080/document/17",
    "http://localhost:8080/document/2",
    "http://localhost:8080/document/3",
    "http://localhost:8080/document/6",
    "http://localhost:8080/document/7",
    "http://localhost:8080/document/10",
    "http://localhost:8080/document/12",
    "http://localhost:8080/document/16",
    "http://localhost:8080/document/15",
    "http://localhost:8080/document/8",
```

```
19      "http://localhost:8080/document/19",
20      "http://localhost:8080/document/20",
21      "http://localhost:8080/document/21"
22    ]
23  }
```

Show

When a HTTP GET request is performed, there might be one parameter specified in the URL which is memorized into the *did* variable. What the service *document_show()()* does, is to show all the information about that specific document.

main.ol

```
1  [ document_show( request )( response ){
2      getDocument@Document( int(request.did) )( response.doc );
3      makeLink@Router( {
4          .operation = "documentNote_index",
5          .params.did = request.did
6      })( response.href)
7  } ]
```

document.ol

```
1  [ getDocument( request )( response ) {
2      q = "getDoc( :doc_id )";
3      q.doc_id = request;
4      call@DatabasePort( q )( doc );
5      handleDoc << doc.row[0];
6      doc.row[0].event_id = int(handleDoc.event_id[0]);
7      doc.row[0].public = int(handleDoc.public[0]);
8      doc.row[0].creator_id = int(handleDoc.creator_id[0]);
9      doc.row[0].id = int(handleDoc.id[0]);
10
```

```
11      response << doc.row[0]
12  } ]
```

First the *getDocument()()* service is called, passing the *did* variable. Once the request to the database has been accomplished and the response sent to main.ol, the service recreates the path for the next available options (supposing to have the REST object Note):

For example, when the URL requested is `"http://localhost:8080/document/20"`, the output is the following:

```
1  {
2    "doc": {
3      "creator_lastname": "Sibani",
4      "creationdate": "2016-06-12 23:26:57.0",
5      "event_id": 13,
6      "public": 0,
7      "creator_id": 54,
8      "name": "Doc Title",
9      "creator_name": "Riccardo",
10     "id": 20
11   },
12   "info": "http://localhost:8080/document/20/note"
13 }
```

Create

When a HTTP POST request is performed, there might be parameters specified into the body of the HTTP Request. What the service *document_create()()* does, is to insert a new document inside the MySQL Database.

HTTP Request

```
1  POST /document HTTP/1.1
2  Host: localhost:8080
```

```
3 Cache-Control: no-cache
4 Postman-Token: 41ec1529-5fc1-f0c0-0cd7-909e5b6cf831
5 Content-Type: application/x-www-form-urlencoded
6
7 creator_id=54&name=docProva2&event_id=12&visibility_type=1&token
    =10
```

main.ol

```
1  define loginProcedure
2  {
3      q = "SELECT id FROM USERS WHERE token = :token";
4      q.token = int(token);
5      query@DatabasePortToCall( q )( loginResponse );
6      if(is_defined( loginResponse.row[0].id[0] )) {
7          login = loginResponse.row[0].id[0]
8      } else {
9          login = false
10     };
11     println@Console( "login " + login )()
12 }
13
14 /*
15  * // CODE
16  */
17
18 [ document_create( request )( response ) {
19     token = request.token;
20     loginProcedure;
21     if(login!= false && login == request.creator_id) {
22         undef( request.token );
23         createDocument@Document( request )( response.doc )
24     } else {
25         response.document = "You are not allowed"
```

```
26      };
27      makeLink@Router( {
28          .operation = "document_index",
29          .params.id = request.id
30      })( response.info)
31
32  } ]
```

document.ol

```
1  [ createDocument( q )( response ) {
2      q.creator_id[0] = int(q.creator_id[0]);
3      q.event_id[0] = int(q.event_id[0]);
4      q.visibility_type[0] = int(q.visibility_type[0]);
5      q = "createDoc( :creator_id, :name, :event_id, :
           visibility_type ) ";
6      call@DatabasePort( q )( doc );
7     response = doc.row[0].returned_id[0]
8  } ]
```

Since the creation of a new document requires some authentication mechanisms, loginProcedure has been defined. What this procedure does is to simply check the accuracy of the token with the id of the creator (of course all the application is supposed to run under SSL [11]).

Once the identity of the client has been approved, the token is unset and the document is created.

All the parameters which are specified into the body, are now available as variables inside the request. If for instance, the server needs the token; it will be reachable through response.token.

MakeLink, instead, refers to `document_index()()` since it is reasonable to return the list of all the documents (the document is empty and it would not return an empty list of notes).

Update

When a HTTP PUT request is performed, there might be parameters specified both into the URL and into the body of the HTTP Request. What the service *document_update()()* does, is to insert a new document inside the MySQL Database.

main.ol

```
1    [ document_update( request )( response ) {
2        token = request.token;
3        loginProcedure;
4        if(login!= false && login == request.user_id) {
5            undef( request.token );
6            request.doc_id = int( request.doc_id );
7            request.user_id = int( request.user_id );
8
9            updateDocument@Document( request )( response.doc )
10       } else {
11           response.document = "You are not allowed"
12       };
13       makeLink@Router( {
14           .operation = "document_index",
15           .params.id = request.id
16       })( response.info)
```

```
1  document.iol
2  [ updateDocument( )( response ) {
3      q = "call updateDoc( :doc_id, :user_id, :name, :public )";
4      call@DatabasePort( q )( doc )
5    } ]
```

Update document is similar to the creation of a new document, inside the document.ol file the service just calls the appropriate procedure.

MakeLink refers to document_index()() since it is reasonable to return

the list of all the documents.

Delete

When an HTTP DELETE request is performed, there shouldn't be any parameters specified into the body of the HTTP Request but only into the URL. Token might be inserted into the header, so it should be captured from the http request which arrives to the router and append it to the response to send it to the main.ol.

WebInputPort in router changes:

```
inputPort WebInput {
Location: "socket://localhost:8080"
Protocol: http {
    .default.get = "get";
    .default.post = "post";
    .default.put = "put";
    .default.delete = "delete";
    .method -> method;
    .statusCode -> statusCode;
    .headers.token = "token";
    .format = "json"
}
Interfaces: WebIface
}
```

Delete (and all the verbs that need token):

```
[ delete( request )( response ) {
        method = "post";
        route;
        response.token = request.token;
} ]
```

What the service `document_create()()` does, is to delete a specific document from the MySQL Database.

```
1  [ document_delete( request )( response ) {
2      token = request.token;
3      loginProcedure;
4      if(login!= false && login == request.user_id) {
5          undef( request.token );
6          deleteDocument@Document( request.did )( response )
7      } else {
8          response.document = "You are not allowed"
9      };
10     makeLink@Router( {
11         .operation = "document_index",
12         .params.id = request.id
13     })( response.info)
14 }]
```

document.ol will delete the record through a stored procedure.

MakeLink refer to document_index()() since it is reasonable to return the list of all the documents.

Nested Requests

The content of a note inside a document can be displayed, for instance, the GET request "*http://localhost:8080/document/7/note/18*" works perfectly with the only precaution to declare the resource properly:

```
1 config.resources[4] << {
2     .name = "documentNote",
3     .id = "nid",
4     .template = "/document/{did}/note"
5 };
```

Here the document is still reachable through *.did* thanks to the two brackets {} in the URL. The note id, instead, will be *nid*.

Example of show is here available

```
[ documentNote_show( request )( response ) {
    getDocumentNotes@Document( int(request.did) )( notes );
    for( i = 0, i < #notes.row, i++ ){
        if( notes.row[i].note_id == request.nid) {
            response.note << notes.row[i]
        } else {
            response.note = "Not Found"
        }
    }
} ]
```

An output exemple for the previous HTTP request could be:

```
{
  "note": {
    "note_id": "18",
    "creationdate": "2016-04-19 13:13:49.0",
    "description": "Finals 2016",
    "id": "18",
    "type": "text",
    "title": "Intro",
    "document_id": "7",
    "content": "We are LIVE with Cleveland Cavaliers Coach
       Tyronn Lue, Kyrie Irving and LeBron James at NBAFinals
       Media Day"
  }
}
```

5.4 Database

5.4.1 Jolie and MySQL

MergeFly has been designed to work with MySQL [9] and, of course, Jolie allows to use MySQL Database.

It is enough to download the appropriate library [10] and insert it in *lib* folder in the root of the program: Jolie will automatically import it.

In order to use this library, it must be included the Jolie interface in our main.ol file or embed the database service provided by Jolie within another service.

```
1  include "../Database/DatabaseService.iol"
2
3  /*
4   * // CODE
5   */
6
7  outputPort DatabasePort { // DatabasePort to Embed
8      Location: "socket://localhost:8887"
9      Protocol: http
10     Interfaces: DatabaseIface
11 }
12
13 outputPort DatabasePortToCall { // Call DatabaseService within
       main.ol
14     Location: "socket://localhost:8887"
15     Protocol: http
16     Interfaces: DatabaseIface
17 }
18
19 embedded {
```

```
20    Jolie: "Database/database.ol" in DatabasePort
21 }
```

DatabaseService.ol

```
1  include "database.iol"
2  include "databaseService.iol"
3
4
5  inputPort Main {
6      Location: "socket://localhost:8887"
7      Protocol: http
8      Interfaces: DatabaseIface
9  }
10
11 execution{ sequential }
12
13 init {
14     with ( connectionInfo ){
15         .host = "localhost";
16         .driver = "mysql";
17         .port = 8889;
18         .database = "polleg_it";
19         .username = "root";
20         .password = "root"
21     };
22
23     connect@Database(connectionInfo )();
24     println@Console("Connected to database .")()
25 }
26
27 main
28 {
29     [ call( request )( response ) {
```

```
30          request = "call " + request;
31          query@Database( request )( response )
32      } ]
33
34      [ query( request )( response ) {
35          query@Database( request )( response )
36      } ]
37  }
```

Since the service is embedded in *main.ol*, call and query services can be invoked. The first one allow the Service Consumer to perform SQL Store Procedures calls, the latter to run simple queries.

The structure of the request in the call should be

```
1 q = "procedureName( :param1, :param2, :param3, :param4 ) ";
2 q.param1 = value;
3 q.param2 = "value";
4 q.param3 = 4;
5 q.param4 = null;
6 call@DatabasePort( q )( doc );
```

The structure of the query in the call should be

```
1 q = "INSERT INTO TABLENAME VALUES ( :param1, :param2, :param3, :
    param4 ) ";
2 q.param1 = value;
3 q.param2 = "value";
4 q.param3 = 4;
5 q.param4 = null;
6 call@DatabasePort( q )( doc );
```

Jolie, indeed, provides an automatised binding system.

5.4.2 Database Structure

E-R Diagram (*Entity - Relationship Diagram*) is a graphical and conceptual representation of a database.

There might be two kind of instances:

Entity: Is the instance which might be independent and exist even if is only *standing alone*.

Relationship : Relationships instead need to refer to two or more entities.

For sure the argument deserves to be studied more in deep [8].

Entity - Relationship Dictionary

Below the Entity - Relationship Dictionary

Entity	Description	Attributes	Id
Users (E)	Informations about registered users	id, name, lastname, born, subscriptiondate, type(premium, basic, admin), image_profile, latitude, longitude, password, mail, deleted	id
Events (E)	Informations about events	id, name, price_id, creationdate, startdate, stopdate, creator_id, type(public, private), description, category_name	id
Categories (E)	Informations about categories	name, description, colour	name
Participations (R)	Users who join an event	event_id, user_id, status	event_id, user_id
Places (E)	Informations about registered places	id, latitude, longitude, name, address, cap, city, country	id
Groups (E)	Informations about groups	id, name, creationdate, image, description	id
Members (R)	Users inscribed to one group	user_id, group_id, accepted, role, joindate	user_id, group_id
Documents (E)	Informations about documents	id, creator_id, name, event_id, creationdate, public	id
Notes (R)	Informations about notes and their contents	id, type(image, text, code, link), title, content, description, creation	id
Nodes (E)	informations about notes (pieces of document)	id, document_id, note_id, creationdate	id

Table 5.1: Entity - Relationship Dictionary

Entity - Relationship diagram

Below the MergeFly E-R Diagram

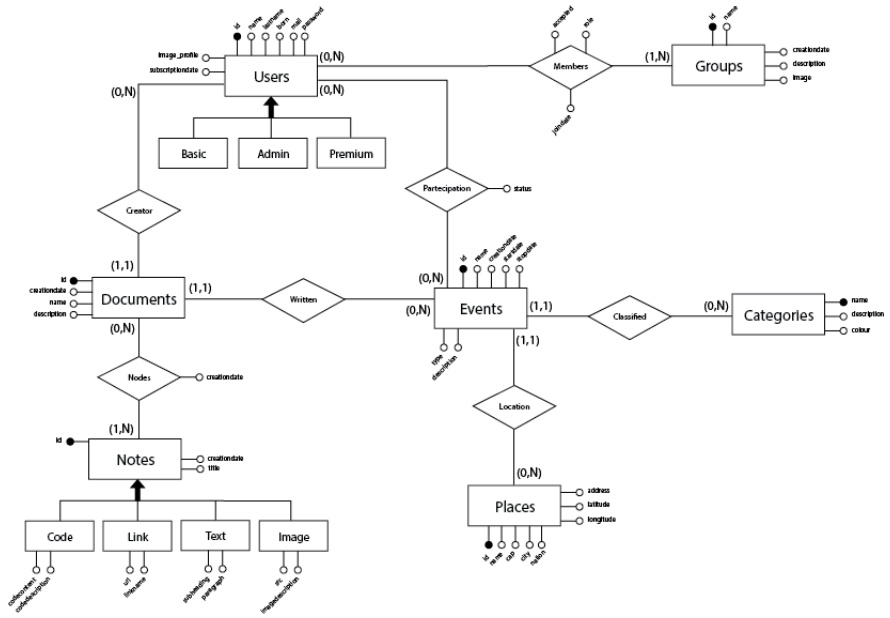

Figure 5.1: MergeFly E-R Diagram, first implementation

Generalizations

Generalizations are presented in notes and user.

Users: Have been joined and, in one single entity, one enum field has been added which specifies one of the following types:

- Basic
- Premium
- Admin

Notes: Notes have been joined as well, another field has been added which specifies one of the following types:

- Code
- Link
- Text
- Image

Here below the updated E-R Diagram

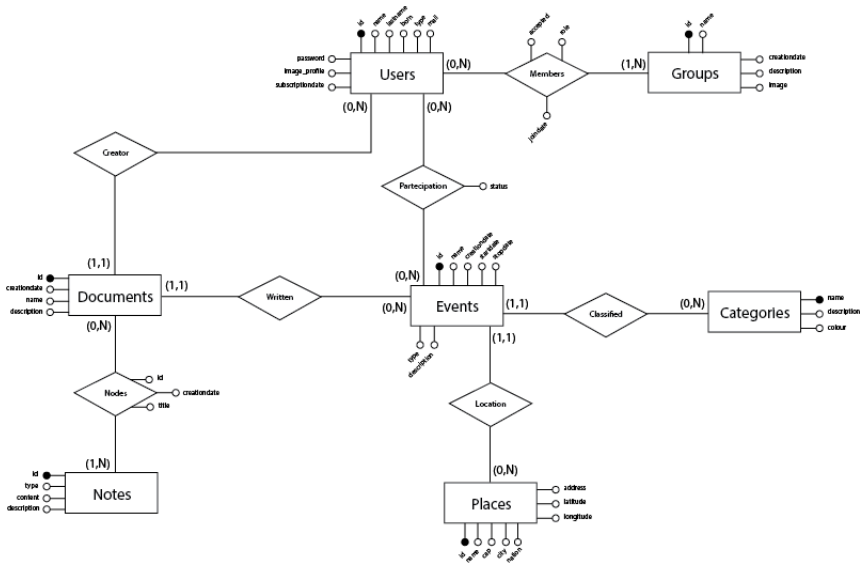

Figure 5.2: MergeFly E-R Diagram, final implementation

5.4.3 Database specifications

Business Rules

1. Password must contain more than 8 characters
2. Event Start date must be earlier then the Event Finish date
3. Information about groups can be modified only by Administrators
4. Users can join all the public events
5. Users can join private events only through an invitations
6. Only Premium users can create private events
7. There cannot be two places with same street, city, and Country
8. Users can delete their profile only after an explicit request

9. Private documents and their notes can be viewed only by its creator (and invited persons)

10. Private notes of a private documents can be imported only by authorized users

5.4.4 Application Functionalities

Functionalities are showed grouped by the *concept* where they will be inserted into:

Login/Registration :

1. Take data from users who performed the access
2. Create new user

Home :

1. Show list of the currently live documents
2. Show list of invitations to groups
3. Show list of invitations to events
4. Accept or refuse invitations to groups
5. Accept or refuse invitations to events

Live Document :

1. Show Document
2. Show list of notes within a document
3. Update one note
4. Delete one note

5. Add / Create one note
6. Show list of all the notes wrote by other users who participate to the same events (and have public profile)
7. Import notes wrote by other users

Documents :

1. Show the list of documents of a user

Document :

1. Show the document content
2. Show the list of documents of a user

Events :

1. Show the list of events joined by the user
2. Show the position of the events the user joined or has been invitated
3. Show list of the events inside an 100km radius
4. Show positions of the events inside an 100km radius

Event :

1. Show informations about one event
2. Show Event participatants
3. Create document about one event
4. Join an event
5. Change the participation status

Add Event :

1. Add one event
2. Add one place or select an existing one
3. Add participants inside one event

Groups :

1. Show list of groups
2. Create one group
3. Accept or refuse the participation to one event

Group :

1. Show list of memebers into one group
2. Show informations about one group
3. Update informations about one group
4. Leave one group
5. Show list of public events that have been created after the joining of the user and in where the user participates.
6. Show list of public documents that have been created after the joining of the user and in where the user participates.

Profile :

1. Show the data of the user
2. Update the data of the user
3. Show number of documents
4. Show number of groups

Implemented Views

userInfo : in order to not let the password exposed to attacks

eventsInfo : in order to facilitate nested queries

Stored Procedures

Here below the list of all the stored procedures implemented, more about their implementation can be found in Appendix A.

1. login
2. updatePosition
3. getUser
4. insertUser
5. userNearEvents
6. getUserEvents
7. searchEvents
8. searchPlaces
9. suggestedPlaces
10. updateUser
11. upgradeUser
12. changePassword
13. deleteUser

14. damnatioMemoriae
15. getEvents
16. getEvent
17. addPlace
18. similarPlaces
19. addEvent
20. updateEvent
21. createCategory
22. getCategories
23. updateCategory
24. getUserDocs
25. getUserDoc
26. createDoc
27. updateDocName
28. updateDocVisibility
29. getDoc
30. getDocContent
31. createNote
32. createNoteWithDate

33. createNode
34. importNote
35. addNoteToDoc
36. modifyNode
37. deleteNode
38. getEventNodes
39. getEventNotes
40. getGroupMembers
41. getEventPartecipants
42. getEventWaitingPartecipants
43. getEventDeclinedPartecipants
44. getPartecipationStatus
45. createGroup
46. addMember
47. removeMember
48. acceptMembership
49. refuseMembership
50. getUserGroups
51. updateGroup

52. getGroupInfo
53. getPlace
54. addPartecipant
55. updatePartecipationStatus
56. addNote
57. getNote
58. updateNote
59. searchUser
60. searchGroup
61. addGroupToEvent
62. getNotifications
63. getGroupsRequest
64. getEventsGroupByUserId
65. getDocumentsGroupByUserId
66. getUserCurrentlyLiveDocs

Triggers

Here below the list of all the Triggers implemented, more about their implementation can be found in Appendix B.

1. check_user

2. check_event
3. check_note
4. check_group

Conclusion

The aim of the thesis is to create a RESTful server with the Jolie programming language following the service oriented architecture standards.

REST, as used and tested architectural style was the perfect canditate for our application, which is confirmed by our analysis. Jolie, on the other hand, fully interpreted the REST principles and HTTP behaviours this laid the foundations for expanding MergeFly server thanks to the principle of modularity.

Indeed, it will be easy, once the router mechanisms has been set up as previously demonstrated, to add and modify services.

This is a distinguishing feature of the Jolie Language.

The router mechanism proposed by Fabrizio Montesi has been explained in deep and developed giving to the reader a practical case about how to built a REST server without even knowing anything about REST architectures: it is simply necessary to follow the instructions provided, and the programmer will have a RESTful Server.

Jolie demonstrated to be a stable and mature language, able to support the development of the features of the application (even the integration with MySQL).

It is indeed demonstrated that all the server can run with an intuitive language such as Jolie is, implementing a non-trivial RESTful Architecture.

Appendix A

Database MySQL

Tabels and Views of MergeFly Database:

```
1  /* Database creation */
2  CREATE DATABASE IF NOT EXISTS merge;
3  -- USE polleg_it;
4  USE merge;
5
6  /******************** TABLES **********************/
7
8  /* USERS */
9  CREATE TABLE IF NOT EXISTS users (
10   id INT(11) AUTO_INCREMENT,
11   name VARCHAR(100) NOT NULL,
12   lastname VARCHAR(100) NOT NULL,
13   born DATE,
14   subscriptiondate TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
15   type ENUM('basic','premium','admin') DEFAULT 'basic',
16   image_profile VARCHAR(300),
17   latitude DECIMAL(11,8), /* it must be defined */
18   longitude DECIMAL(11,8), /* it must be defined */
19   password VARCHAR(300) NOT NULL,
20   mail VARCHAR(150) NOT NULL,
21   deleted  ENUM('0','1') DEFAULT '0',
22   PRIMARY KEY (id)
23 ) engine=INNODB;
24
25 /* GROUPS */
26 CREATE TABLE IF NOT EXISTS groups(
27   id INT(11) AUTO_INCREMENT,
28   name VARCHAR(100) NOT NULL,
29   creationdate TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
30   image VARCHAR(300),
31   description VARCHAR(2000) NOT NULL DEFAULT "No description.",
32   PRIMARY KEY (id)
33 ) engine=INNODB;
34
35 /* PLACES */
36 CREATE TABLE IF NOT EXISTS places (
37   id INT(11) AUTO_INCREMENT,
38   latitude DECIMAL(11,8),
39   longitude DECIMAL(11,8),
40   name VARCHAR(100) NOT NULL,
41   address VARCHAR(200) NOT NULL,
42   cap VARCHAR(10),
43   city VARCHAR(50) NOT NULL,
44   nation VARCHAR(50) NOT NULL DEFAULT "Italy",
45   PRIMARY KEY (id)
46 ) engine=INNODB;
47
48 /* CATEGORIES */
49 CREATE TABLE IF NOT EXISTS categories (
50   name VARCHAR(100) NOT NULL,
51   description VARCHAR(3000),
52   colour VARCHAR(7),
53   PRIMARY KEY (name)
54 ) engine=INNODB;
```

```
55
56  /* NOTES */
57  CREATE TABLE IF NOT EXISTS notes (
58     id INT(11) AUTO_INCREMENT,
59     type ENUM('code','text', 'image', 'link') DEFAULT 'text',
60     title VARCHAR(300) DEFAULT "Note title",
61     content TEXT,
62     description VARCHAR(200),
63     creationdate timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
64     PRIMARY KEY(id)
65  ) engine=INNODB;
66
67  /* MEMBERS */
68  CREATE TABLE IF NOT EXISTS members(
69     user_id INT(11) NOT NULL,
70     group_id INT(11) NOT NULL,
71     accepted BOOLEAN DEFAULT 0,
72     role ENUM('admin','normal') DEFAULT 'normal',
73     joindate TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
74     PRIMARY KEY (user_id, group_id),
75     FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
76     FOREIGN KEY (group_id) REFERENCES groups(id) ON DELETE CASCADE
77  ) engine=INNODB;
78
79  /* EVENTS */
80  CREATE TABLE IF NOT EXISTS events (
81     id INT(11) AUTO_INCREMENT,
82     name VARCHAR(100) NOT NULL,
83     place_id INT,
84     creationdate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
85     startdate DATE,
86     stopdate DATE,
87     creator_id INT,
88     type ENUM('public','private') DEFAULT 'public',
89     description VARCHAR(2000),
90     category_name VARCHAR(100) default 'Meeting',
91     PRIMARY KEY (id),
92     FOREIGN KEY (place_id) REFERENCES places(id),
93     FOREIGN KEY (creator_id) REFERENCES users(id) ON DELETE SET NULL,
94     FOREIGN KEY (category_name) REFERENCES categories(name) ON DELETE SET NULL
95  ) engine=INNODB;
96
97  /* DOCUMENTS */
98  CREATE TABLE IF NOT EXISTS documents (
99     id INT(11) AUTO_INCREMENT,
100    creator_id INT,
101    name VARCHAR(100) DEFAULT "unknown document",
102    event_id INT,
103    creationdate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
104    public ENUM('0','1') DEFAULT '1',
105    PRIMARY KEY (id),
106    FOREIGN KEY (creator_id) REFERENCES users(id) ON DELETE CASCADE,
107    FOREIGN KEY (event_id) REFERENCES events(id) ON DELETE SET NULL
108
109 ) engine=INNODB;
110
```

```
111 /* NODES */
112 CREATE TABLE IF NOT EXISTS nodes (
113   document_id INT(11),
114   note_id INT(11),
115   creationdate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
116   PRIMARY KEY (note_id, document_id),
117   FOREIGN KEY (document_id) REFERENCES documents(id) ON DELETE CASCADE,
118   FOREIGN KEY (note_id) REFERENCES notes(id) ON DELETE CASCADE
119 ) engine=INNODB;
120
121 /* PARTECIPATIONS */
122 CREATE TABLE IF NOT EXISTS partecipations (
123   event_id INT(11) NOT NULL,
124   user_id INT(11) NOT NULL,
125   status ENUM('accepted','declined','waiting') DEFAULT 'waiting',
126   PRIMARY KEY (event_id, user_id),
127   FOREIGN KEY(event_id) REFERENCES events(id) ON DELETE CASCADE,
128   FOREIGN KEY(user_id) REFERENCES users(id) ON DELETE CASCADE
129 ) engine=INNODB;
130
131 /******************** VIEWS ************************/
132
133 CREATE VIEW usersInfo(id, name, lastname, born, subscriptiondate, type, image_profile,
        mail) AS
134   SELECT id, name, lastname, born, subscriptiondate, type, image_profile, mail FROM
          users WHERE deleted="0";
135
136 CREATE VIEW eventsInfo(event_id, event_name, type, creationdate, startdate, stopdate,
        event_description, creator_id,
137   creator_name, creator_lastname, place_id, place_name, address, cap, city, nation,
          latitude, longitude, category_name, category_description, category_colour) AS
138   SELECT evnt.id, evnt.name, evnt.type, evnt.creationdate, evnt.startdate, evnt.stopdate
          , evnt.description,
139           usr.id, usr.name, usr.lastname, plc.id, plc.name, plc.address, plc.cap, plc.
                city, plc.nation, plc.latitude, plc.longitude,
140           evnt.category_name, cat.description, cat.colour
141   FROM events AS evnt, places AS plc, usersInfo AS usr, categories AS cat
142   WHERE ( (evnt.place_id = plc.id) AND (evnt.creator_id = usr.id) AND (cat.name = evnt.
          category_name) );
```

Bibliography

[1] http://www.oasis-open.org/specs/#dpwsv1.1

[2] http://amslaurea.unibo.it/2372/1/pascali_stefano_tesi.pdf

[3] http://fabriziomontesi.com/files/mgz14.pdf

[4] https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf

[5] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[6] http://www.w3.org/Protocols/rfc2616/rfc2616.txt

[7] http://arxiv.org/pdf/1410.3712v3.pdf

[8] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.1085

[9] http://www.mysql.com

[10] https://dev.mysql.com/downloads/connector/j/5.0.html

[11] https://tools.ietf.org/html/rfc5246

Acknowledgement

I wish to express my sincere thanks to my parents for all the support during these 3 years of University, they always motivated me by hook or by crook.

I am also grateful to Prof. Sangiorgi for giving me the opportunity of this thesis and to Dott. Giallorenzo for standing me and always helping with all my doubts and my English "typos". I am also thankful to them both for all what they thought me in the stunning course about Operative Systems.

I take this opportunity to express my gratitude to Alisa for helping me and correct all these papers, *lo siento*, without any knowledge about IT but a great heart. And to Filippo, the one without I would still be stuck in some projects.