Technical documentation

Defuze.me – EIP 2012

Information

Name of the project	Defuze.me
Type of document	Technical documentation
Date	19/01/2012
Version	1.8
Keywords	Architecture – Mechanism - Technologies – Diagrams – Bugs - Interactions
Authors	Adrien Jarthon – jartho_d Arnaud Sellier – sellie_a Alexandre Moore – moore_a Jocelyn De La Rosa – de-la-_o Athéna Calmettes – calmet_b Luc Pérès – peres_a François Gaillard - gailla_f

Redaction and modifications

Version	Date	Names	Description
1.0	14/04/2011	Adrien Jarthon	◆ Document creation ◆ Some translations from French version
1.1	15/04/2011	Adrien Jarthon Alexandre Moore	◆ Mobile application translation ◆ Translation
1.2	27/04/2011	Adrien Jarthon	◆ Web service API details
1.3	07/05/2011	Adrien Jarthon	◆ Diagrams update
1.4	08/05/2011	Adrien Jarthon	◆ Formatting
1.5	17/05/2011	Athéna Calmettes	◆ Table of figures added
1.6	08/06/2011	Jocelyn De La Rosa	◆ Document summary added
1.7	10/06/2011	Athéna Calmettes	◆ Keywords added
1.8	19/01/2012	Adrien Jarthon	◆ Formatage

Contents

1 - Document Summary

This document is the official technical documentation of the software suite defuze.me. There are describes :

- The technical documentation of the client application : the desktop application
- The technical documentation of the server: website and APIs
- The technical documentation of the mobile application running on Android and iOS
- Known bugs on the software suite

2 - Recall of the software's functional

2.1 - Software's description

Our EIP's goal is to create a radio broadcasting software to be used by professionals.

More than music broadcasting, it will allow radio station to manage most of their daily duties, especially advertisement contracts, jingles managements, recovery and exports of data.

The software shall have a modern and ergonomic interface, allowing a simple and effective broadcasting, and at the same time being able of advanced manipulation. It will be usable on touch screens as well as on multi-screens, and thus no matter the number of additional screens.

In the aspect of being used easily, some parts of the software will be usable on external devices, such as touch pads, communicating with the main software. Thus our software shall be usable over the network, even when far from the workstation.

At last, a particular effort shall be made on the interaction with the web server, which will be conceived by us, allowing listeners to interact with the animators through the Internet.

2.2 – Project's composition

Our project is composed by four different parts:

- The client software, which is a desktop application running on Windows, Linux and MacOS, allowing sound diffusion, audio library management and events planning.
- The light client, a lightweight application connected to the client software designed to run on small devices like tablets.
- Two mobile applications running on Android and iOS, connected to the client software in order to take the control of it. They allow a small subset of the client software's functionalities to be achieved remotely.
- Finally, a website and a web-service connected to the client software, allowing the user to control his planing or playing queue remotely. Also used to broadcast radio's informations on internet (playing queue, next track, announcement, …)

In the following document, each of these parts will be explained separately.

2.3 – Global architecture

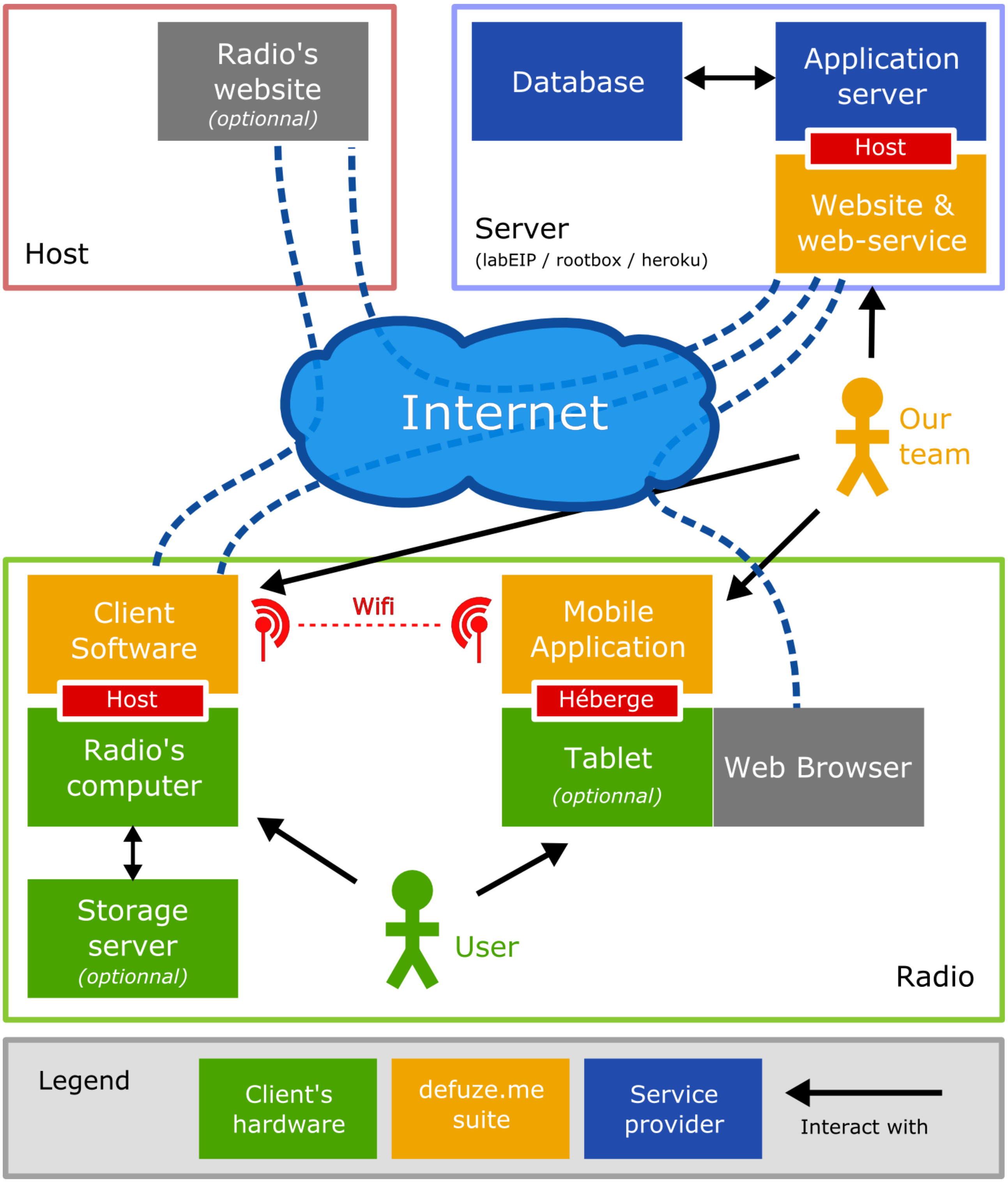

Illustration 1 : Global architecture

3 – Client

3.1 – Architecture

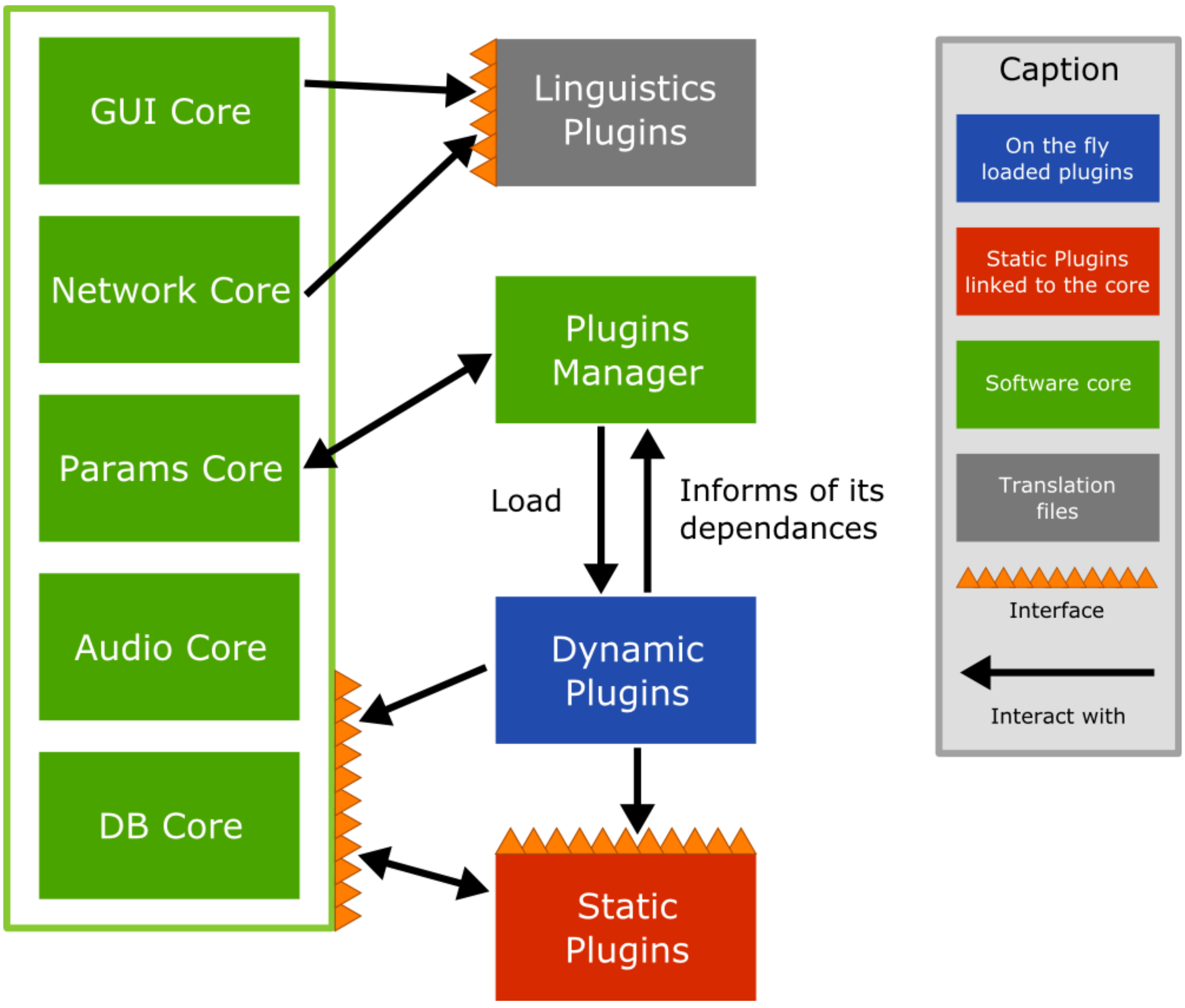

Illustration 2 : Client - Architecture

The core of the software, divided between multiple modules, doesn't directly provide functionalities to the user but services to the plugins.

Some of these plugins are static, they will provide, to the user, the base functionalities of the software (playlists and player for exemple). They are bonded to the software core.

On the other hand, dynamic plugins provide additional functionnalities which are not always needed for every user. So they can be (un)loaded on the fly is necessary, through the plugin manager.

Finally, linguistic plugins are binary files containing the translations.

3.2 – Technologies

The use is written in C++ and use the Qt framework from Nokia.

Qt provide all the necessary elements for the creation of our software, and brings a few essential caracteritics :

- Portability : A Qt Code can be compiled indifferently on the three main operating system, Windows, Linux and OS X.
- Supported by Nokia, Qt is used in major professional projects (such as KDE and Meego). Qt is in
- continuous development and gather a large and active community. These facts will ensure the viability of Qt of the futur.
- Nokia and Qt propose since a short time new Qt modules targeting the usage of mobile and touch-screen device. So the lite version use bleeding edge technologies.

From the numerous elements of the Qt framework, some are extremely important in the current architecture :

- Qt Plugins

  We use the Qt low-level API allowing us to add functionalities to the software. These plugins are either static or dynamics (under the form of share libraries .so or .dll). These plugins interact with the software through a C++ interface.
- Qt Linguist

  Qt Linguist is an integrated tool from the Qt framework which allow us to easily create multilingual software. It separates text from code (through the usage of keys). The translation can then be done separately from the development with a graphical tool then be released as binary files .gm. These .gm files can be load on the fly into the software.

  Furthermore, Qt can detect the user's operating system language and load the most adapted .gm file.
- Qt Mobility - Multimedia

  Qt Mobility – Multimedia is an add-on to Qt which provides a set of APIs to play and record audio media, and manage a collection of media content (playlists). It allows a low level access to the audio stream to modify it and apply effects and filters.

Technologies used outside Qt are related to the database.

- SQLite (through QtSql).

  Sqlite was chosen as the user software SGDB for it simplicity. Indeed, this software require very few resources and no complex operation.

  The use of a complex SGDB (with the usage of a client/server architecture) is not necessary.

The software access the database through the Qt framework's module QtSQL which provide a simple object interface.

3.3 – Class diagram

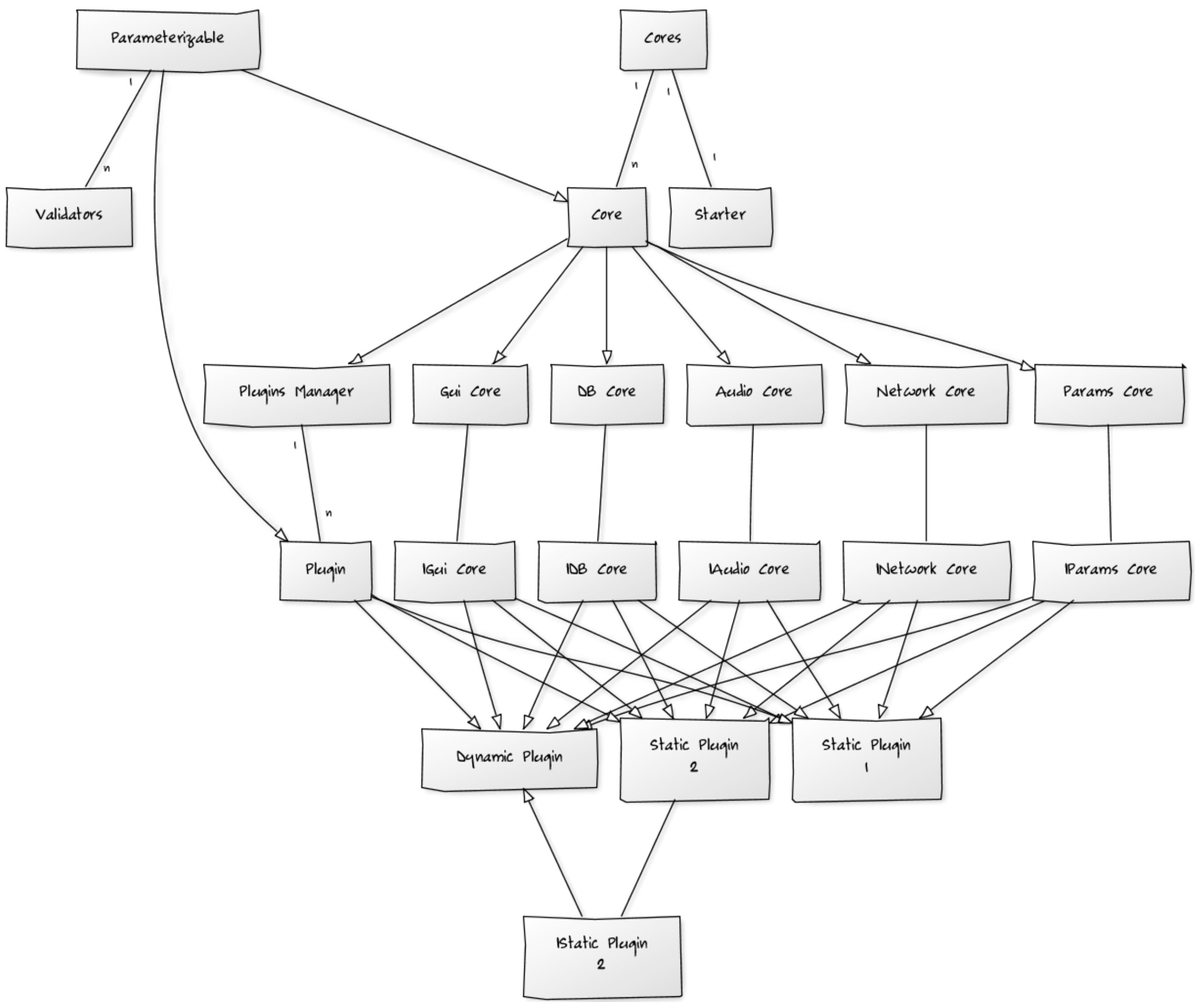

Illustration 3 : Client - Class diagram

- The Cores provides the most basic functionnalities using an interface (IGuiCore for the GuiCore for instance). Every cores or plugins can implement one or several Core interfaces in order to use the Cores' functionalities.
- The Static Plugins provide interfaces too, which can be implemented by (dynamic or static) plugins only.

3.4 – Data structure

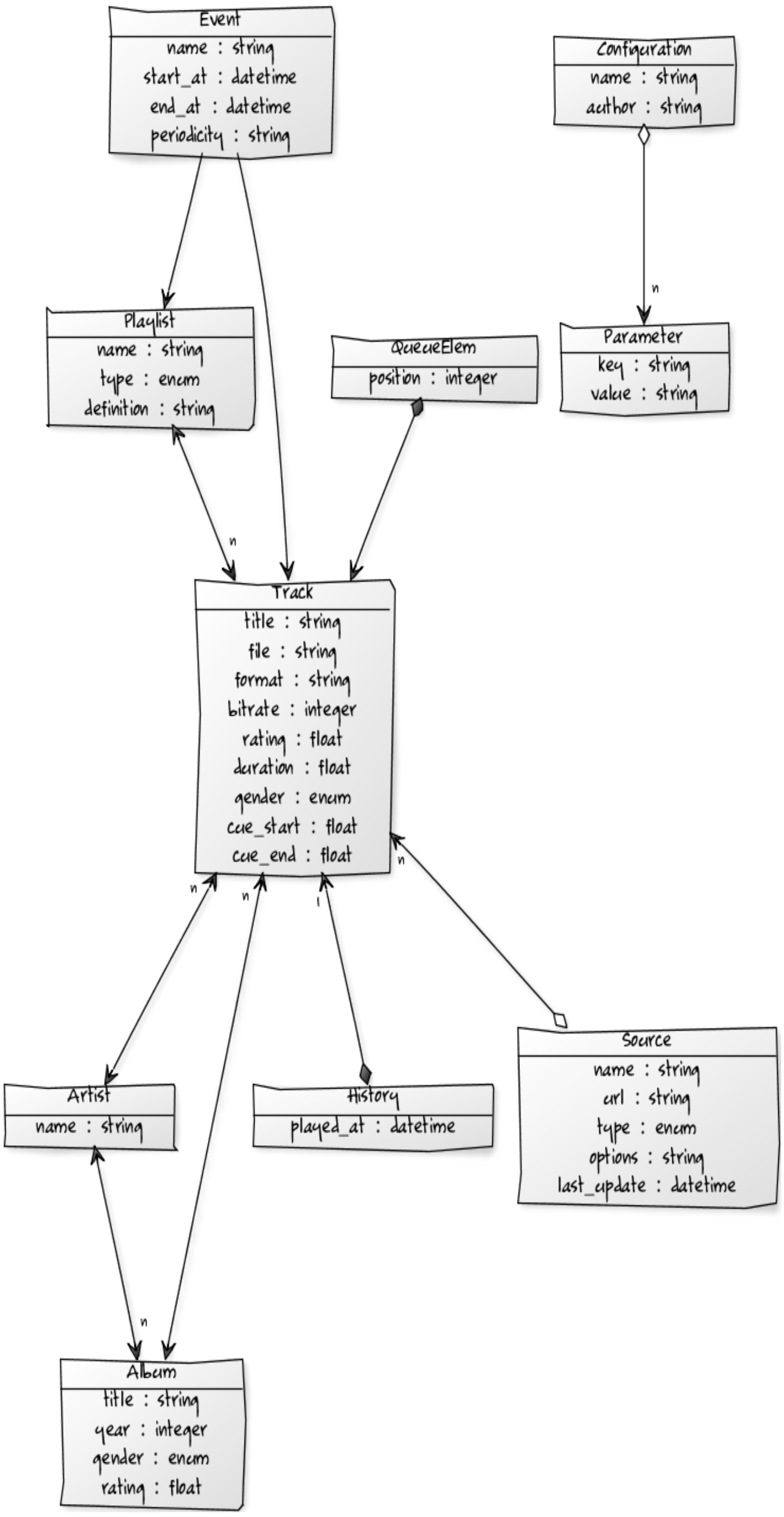

Illustration 4 : Database - Class diagram

3.5 – Light client

The light application is directly connected to the client application and give basics controls to the users. It acts like statics Plugins. It does not interact directly with CORES, but only with the TABLET API PLUGIN which acts like a gateway between the CORES and the light client.

The light client use a specific functionality of the Qt framework : QML.

The Qt Modeling Language is a Qt technology which allow the creation of graphic interface adapted to touch screen. It separate the user interface from the rest of the code and is a must have for the light client which mainly use Static Plugin's Code but not their interface.

4 – Server

4.1 – Architecture

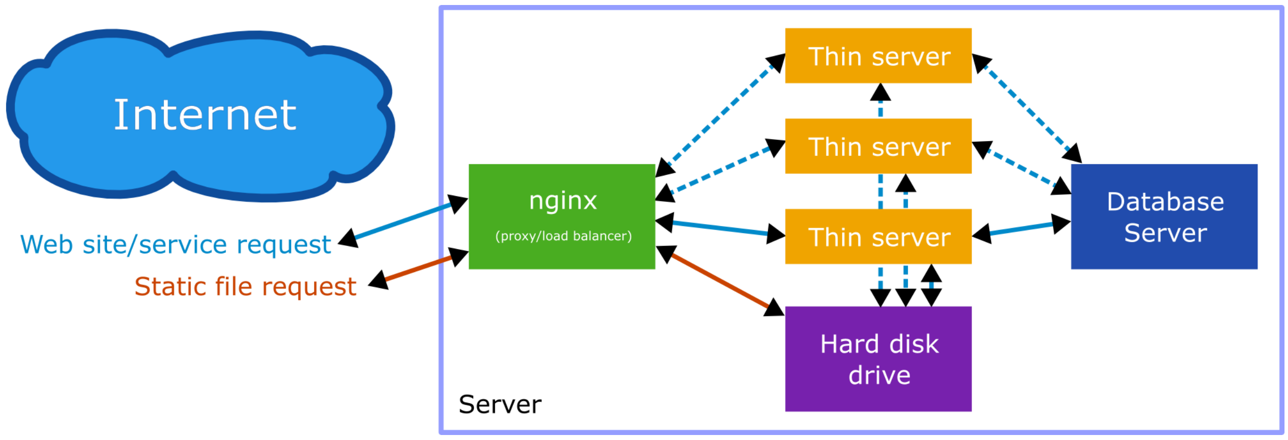

Illustration 5 : Server - Architecture

The server provide the following functionalities :

- The software's public web site.
- The online administration interface.
- The web-service communicating with the client software.

4.2 – Technologies

The server is development using the ruby language with the Ruby on Rails framework. It is based on the following components:

- Ruby (1.9+) ;
- Ruby on Rails (3.0+) ;
- Thin server (1.2+), the web server which run our application ;
- nginx (0.7+), proxy and load balancer ;
- PostgreSQL, the database server.

We chose the Ruby on Rails framework because it is a bleeding edge and flexible technology, which allow us to create web application rapidly and easy to maintain. Furthermore, it is a technology regularly updated.

Thin is one of the lighter and faster server compatible with Ruby on Rails, it perfectly match our needs.

Finally, we chose PostgreSQL as a database server. It is very fast and maintained regularly, however Ruby and Rails is supporting most on the database on the market, it can be easily changed if necessary.

4.3 – Data structure

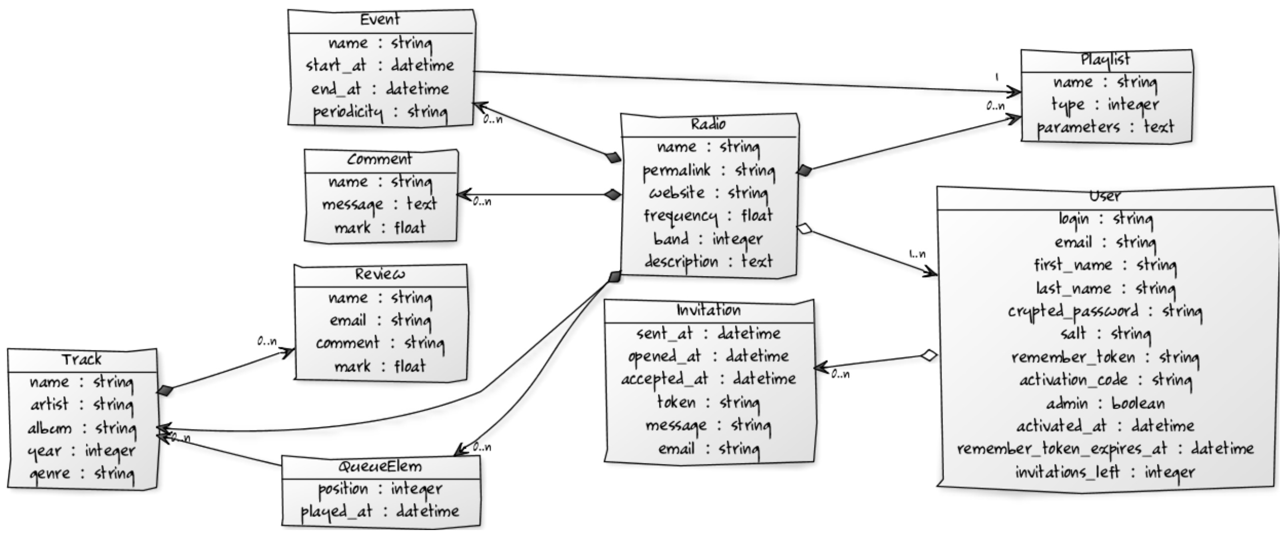

Illustration 6 : Server - Class diagram

The database server allow us to stock informations on the radio software (such as the queue list, playlist and events), thus can be displayed on the web site and allow the user to change these informations remotely. Which will be updated on the server then synchronized with the client software.

This database will contain user account and radio, allowing authentication and license control.

4.4 – External interactions

The server (in purple), interact with other components of our project through local or distant network using the following protocols.

The security will be adapted to each situation restrains.

The radio server (in blue) is the only one which we don't control the software part. It can be a custom server belonging to the radio on which we offer to retrieve broadcasting information of the client software.

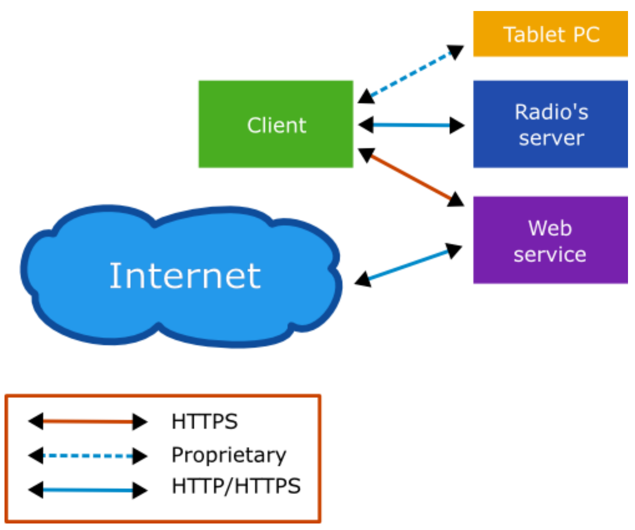

Illustration 7 : Server - Used protocols

4.5 – Client API

The client API is the one between the client software and the web service. It's main purpose is to update radio's live informations on the website, so the user can manage them remotely. The API is private and only accessible from authenticated clients.

It's a RESTfull HTML API using Json or XML indifferently as data format.

Here is a quick list of possible actions, with their RESTfull URL, available HTTP verbs and formats.

URL	Verbs	Formats
/radios/my-radio-name	GET POST	XML, JSON (Informations about the radio)
/radios/my-radio-name/playing	GET POST	XML, JSON (Play queue)
/radios/my-radio-name/history	POST	XML, JSON (Play historic)
/radios/my-radio-name/stats	GET POST	XML, JSON (Statistics of the radio), PNG (Charts)
/radios/my-radio-name/comments	GET	XML, JSON (Radio's reviews)
/radios/my-radio-name/tracks/3/rate	GET	XML, JSON (Track's marks)
/radios/my-radio-name/planning	GET POST	XML, JSON (Scheduling)
/radios/my-radio-name/license	GET POST	XML, JSON (License checking)

4.6 – Public API

The public API allows the radio's own website to fetch live informations directly from our web-service, which is a lot easier than getting the data directly from the software.

It's also a RESTfull HTML API very easy to use.

The access to this API can be public or restricted as chosen by the radio's administrator.

URL	Verbs	Formats
/radios/my-radio-name	GET	XML, JSON, HTML (Informations about the radio), PNG (QR code of radio)
/radios/my-radio-name/playing	GET	XML, JSON, HTML (Play queue)
/radios/my-radio-name/history	GET	XML, JSON, HTML (Play historic)
/radios/my-radio-name/stats	GET	XML, JSON, HTML (Statistics of the radio), PNG (Charts)
/radios/my-radio-name/comments	GET POST	XML, JSON, HTML (See/add a review)

/radios/my-radio-name/tracks/3/rate	GET POST	XML, JSON, HTML (See/add a mark), PNG (mark chart)

5 – Mobile application

5.1 – Architecture

The mobile application gives his user the ability to do some actions on the client software remotely:

- Play
- Pause
- Stop
- Stop en fin
- PushToTalk
- Crossfade
- Playing queue re-ordering

5.2 – Technologies

The development will first be done for Android devices. We use the « Eclipse » IDE, with Google's Android SDK. We use the Java language, allowing to get the most out of Android. Some tests was realized with the Titanium 1.4 framework, but it does not actually fit our network requirements.

5.3 – Interactions

The mobile application is – most of the time – embedded on small devices like tablets. Of course the hardware must include a network connection allowing the application to communicate with the client software. The interaction is done through a TCP socket. Data are caught on the client side by the network core, then transmitted to the mobile API plug-in.

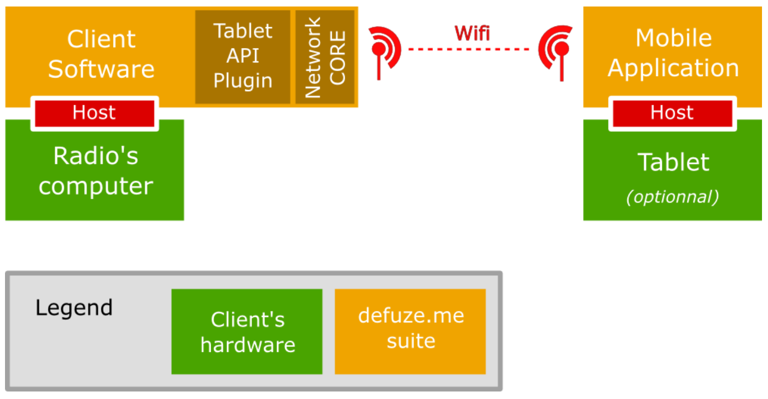

Illustration 8 : Mobile application - Interactions

5.4 – API

See Appendix B - Mobile API

6 – Known bugs

6.1 – Website

1. Missing English translation on invitation page
2. “join existing radio” button is disabled yet

Appendix

A - Table of figures

B – Mobile API

```
Defuze.me                       Mobile API                          2010-2012


Revision     Date           Author              Description
-------------------------------------------------------
v0.1         2011-01-22     Adrien Jarthon      Architecture & Events
v0.2         2011-03-31     Adrien Jarthon      Authentication + TCP format update


The mobile API allow the Defuze.me mobile application to communicate with the
client software. The commincation is etablished through the TCP protocol, and
the data formatting using JSON encoding.

1. Message architecure
----------------------

The communication between the mobile application and the client software will be
2-way and event-oriented. Each pair can send messages to the other, multiple
messages can be sent at once. One message describe one event. One or more
messages CAN reply to another one if an answer is needed, but not necessary
right after the request, tons of messages can be send between a request and its
answers.

1.1 Message format

Each message will be represented by a JSON hash containing:
* the type of event
* the UID of the message (choosed by sender)
* the UID of the message we reply to (optional)
* any additional event-dependent data
```

Example of message:
```
{
      event:      'newQueueTrack',
      uid:  345,
      data: {
            track:      {
                  title:      'Dezzered',
                  artist:     'Daft punk',
                  album:      'Tron legacy OST',
                  year: '2010',
                  id:         4987
            },
            position:   12
      }
}
```

Every key and event name will be camelCased.

This way of asynchronous and non-blocking communication handling (using reply uid) allows us to keep the app very reactive. For example if the mobile app trigger a huge library search, nothing will block and the user is still able to use the live controls and the playing queue while the search is being processed by the client.

1.2 TCP format

The JSON flow will be formatted this way:

```
{message1}\0{message2}\0{message3}\0
```

It's a list of messages (json hash) separated by a null char ('\0'). When receiving data, the client will have to put the different TCP packets together until they form a valid json hash. The trailing '\0' at the end of each message will help the receiver split the data flow. Every binary data will be encoded using JSON string format or base64 to avoid the null char inside of our content.

2. Events

2.1 Live control

Live control events can be sent by the client to update the mobile app display of by the mobile app to control the client software.

Available events:
* play
* pause
* stop
* next
* talk
* endTalk

All this events have no additionnal parameters. If sent by the mobile app, the client MUST answer to the mobile app request with one of the following event:
* ok
* noChanges (if the control was already in the desired state)
* error (SHOULD contain additional error informations)

For the mobile app to know what are these answers for, the client MUST specify the request event id in the 'replyTo' field of the message hash.

Example:

```
(mobile -> client)
{
      event:            'play',
      uid:        42
}

(client -> mobile)
{
      event:            'ok',
      uid:        375,
      replyTo:    42
}
```

2.2 Authentication

Once connected to the client, the mobile app will have to authenticate itself, and wait for acceptance from client side. When launched for the first time, the mobile application must generate a random token and store it for further use.

Then will be a base64 string of length 16.
ex: d2KI1ONxc8123bJF

On connection, the client will send an initial event requesting the application's identification token:

```
{
      event:      'authenticationRequest',
      uid:  32,
}
```

The mobile application must then answer like this:

```
{
  event: 'authentication',
  uid:  765,
  replyTo: 32,
  data: {
    token: 'd2KI1ONxc8123bJF',
    appVersion: '0.1',
    deviceName: 'Samsung S100 Galaxy Tab'
  }
}
```

The client will then check the informations and ask the user if necessary.
If the app is accepted, the mobile app will receive:

```
{
  event: 'authenticated',
  uid: 92,
  replyTo: 765,
}
```

Now every regular event described in this document can be received and sent.
If the authentication fail, the client will send an event like this:

```
{
  event: 'authenticationFailed',
  uid: 93,
  replyTo: 765,
  data: {
    message: "Sorry, your version is too old, please upgrade"
  }
}
```

And the connection will instantly be closed by the client.

2.3 Playing queue

The client software will continously stream to the mobile app every modification to the playing queue using one of the following event:
* popQueueTrack
* newQueueTrack
* removeQueueTrack
* moveQueueTrack

Each "track" in the queue will have a position relative to the currently playing track. The playing track will have the position 0, the next will have the position 1 and the last -1. When the current track fade out to the next track, at the end of the fade (when the first track is unloaded) the second track become the first and optain the position 0, every track position in the queue is shifted by -1.

When this does arrive (at each track change) the client will send a "popQueueTrack" event to the mobile app. The message will look like this:

```
{
      event:      'popQueueTrack',
      uid:  42,
}
```

When a new track is present in the queue (added on client side), the client will send a "newQueueTrack" message looking like this:

```
{
      event:      'newQueueTrack',
      uid:  345,
      data: {
            track:      {
                  title:      'Dezzered',
                  artist:     'Daft punk',
                  album:      'Tron legacy OST',
                  year: '2010',
                  id:         4987
            },
            position:   12
      }
}
```

In that case, the mobile app will have to insert the track at the given position and shift any later track (position >= 12) by +1. If the given position is negative (history), the older tracks are shifted by -1.

When a track is remove from client side, the client will send the following message:

```
{
      event:      'removeQueueTrack',
      uid:  345,
      data: {
            track:      {
                  title:      'Dezzered',
                  artist:     'Daft punk',
                  album:      'Tron legacy OST',
                  year: '2010',
                  id:         4987
            },
            position:   12
      }
}
```

In this case, the mobile app will have to shift the later tracks (> 12) positions by -1. The "moveQueueTrack" event is a combination of the 2 previous events. It is intended to move a track inside the queue and will look like this:

```
{
      event:      'moveQueueTrack',
      uid:  345,
      data: {
            track:      {
                  title:      'Dezzered',
                  artist:     'Daft punk',
                  album:      'Tron legacy OST',
                  year: '2010',
                  id:         4987
            },
            position:         12
            oldPosition:      42
      }
}
```

Of course, the mobile app will have to shift the ids accordingly.

When a queue change occur on the mobile app side, the mobile application must send one of the previous events (except popQueueTrack) and the client will reply with on of the available answer event (ok, noChanges or error).

2.4 Music library