

# Google Summer of Code 2013

## WordPress mobile app for BlackBerry 10 devices

### Background

In the past years I've always keep on eye opened on the Google Summer of Code program, I loved it since the first time, but I had no good project to submit.

This year, I've seen that the WordPress community has proposed an idea for the GSoC that was to create a native WordPress app for the BlackBerry 10 platform.

Since I've already working with new BlackBerry 10 platform and developed some apps for it, I had the feeling that it was the right time to proposed this idea for the GSoC. So, I get in touch with some internal at WordPress and with my future mentor I've organized the idea, created a schedule and submit it as a project for the GsoC. That was the real right time, cause I've been accepted.

### Knowing parts

WordPress doesn't need any presentation, as it is the most used open source tool for blog management. It has a gorgeous community always on the cutting edge about new technologies.

The new BlackBerry 10 platform get his power from QNX (an already famous operating system) and Qt (that I already known, since I've used it for other opensource mobile platforms like Nokia maemo/meego).

So for this project I've choose to use Qt/Cascades: Qt is a cross-platform application and UI framework for developers using C++, but for the UI creation I choose to use Cascades (derived from QML) that is a CSS & JavaScript like language, with Cascades is it possible to create beautiful and clean UIs using the out of the box controls and animations and take the advantage of using the BlackBerry 10 interactions.

To build the project I've used the QNX Momentics IDE with the BlackBerry 10 Native SDK (SDK ver. 10.1.\*) for Linux.

### Initial state

For a first phase, I get in touch my mentor that help me for the kick off of the project. He suggested to me to use some software for doing some initial mock-up(s) of how the UI could come. This was an interesting part for me since I've never done something similar in a so professional way. So I've started use balsamiq (my mentor's suggestion), that is a tool to create mock-up. I've played around with it for the first two weeks, and after that I come up with a first mock-up.

## Initial code

Once the mock-up has been done, I've start playing with WordPress XML-RPC API. Firstly I've tried to understand how they work, what call I need to implement, and how they response. After that I've created a standalone code to see how to get this stuff with Qt.

For any call you need to give an well-formed XML to the sever, and than you get a response that his again an XML.

I've used Qt for this: create the XML for the request, send the request (e.g. make the api call) , receive the response, and parse the received XML in order to extrapolate the informations I need and that I want to show to the user.

For the XML creation I've used *QXmlStreamWriter* – a class that provides an XML writer with a simple streaming API – , to make the call I've used *QNetworkAccessManager* – a class that allows the application to send network requests and receive replies – along with *QNetworkRequest* – part of the Network Access API and is the class holding the information necessary to send a request over the network – . For receive the response I've used *QNetworkReply* – a class that contains the data and headers for a request sent with *QNetworkAccessManager* – in some (rare) situations (e.g. the endpoint discovery) I need to wait until the response arrives in order to let the user go on to use the app, so I've used a *QEventLoop* – a class that provides a means of entering and leaving an event loop –

Once I got the response I use *QXmlStreamReader* – a class that provides a fast parser for reading well-formed XML via a simple streaming API – to parse the XML.

When I parse the XML, I put the informations into a *QVariantMap* – that is a synonym for *QMap<QString, QVariant>*.

This QMap contains a pair of key/value, I choose to use this solution so I can put the XML tag as a key of the map, and the content of the tag as the value. Furthermore I can easily insert this data into a *GroupDataModel* – represents an ordered map of QVariantMap objects and/or QObject\* pointers, to be used as data for ListView – that I use it as a model for the *ListView* – a scrollable container used to display a list of items – in order to present to the user a list with all the necessary information, sorted by date.

But, sometimes the reply is an XML that I don't need to use as a model for some ListView, this is the case, for example, when a post has been deleted, the returned XML doesn't contain data that need to be shown to the user, it only contains a boolean that is setted true if the operation has been executed successfully, false if it fails somewhere. In this case, there is a JavaScript function that will intercept this situation and handling this case in a different way, without showing this crude result to the user.

After some test, the first hiccup comes up. The login part wasn't as easy as I was thinking, there my mentor came in my help. For a successfull login process I've to discover where the endpoint is. The endpoint is a file (xmlrpc.php) that take care about the request and response to/from server, that is where the blog has been placed. The most common fault, is that the user, in the 'Blog Address' entry, will put something wrong, so I need to create a function to correct what the user insert, if it is somehow wrong. Due to this, I've started to create a C++ function that will take care of this, this step takes me a long time, because the things the function has to do were clear on my mind, but translate it into some piece of code was kinda hard.

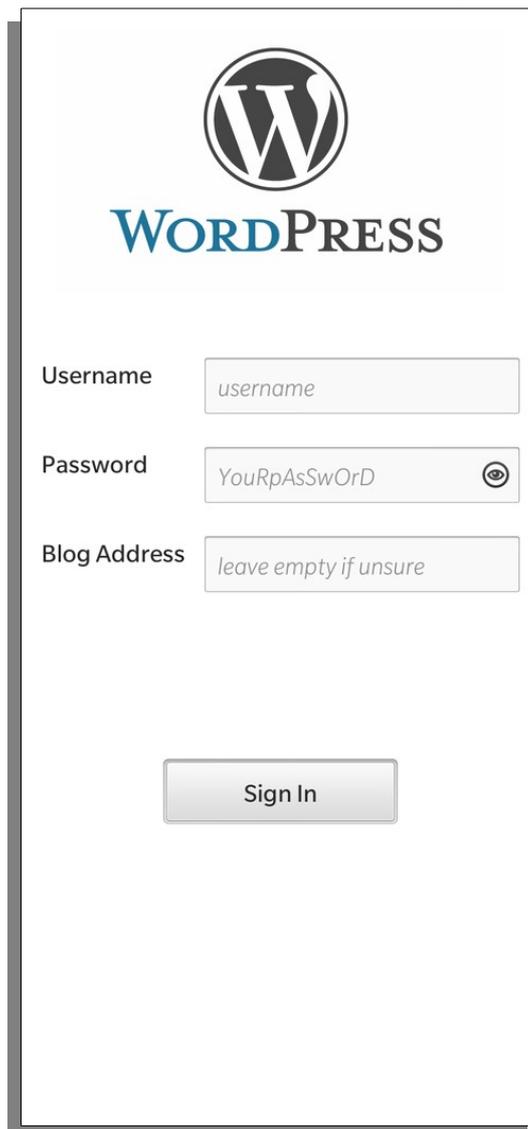
At least I came up with a line guide that is:

- the user insert nothing for 'Blog Address', than use the common <https://wordpress.com/xmlrpc.php> as the endpoint.
- the user insert an url that end with 'xmlrpc.php' , that is the endpoint (after validation)
- the previous fails, try to guess the XML-RPC URL by adding “/xmlrpc.php” and removing if present the string (“/wp-admin/”)
- the previous fails, try to get the value from the pingback from the content of the page at the URL inserted by the user
- the previous fails, download the content of the page at the URL inserted by the user and find the following link element with rel='EditURI'
  - get the href attribute and download the RSD Document
  - parse the RSD document and get the apiLink for WordPress
  - the apiLink value is the XML-RPC URL endpoint of the blog

For the validation I do a simple API call on the method *getUsersBlogs*, if it returns a valid response, then this is the endpoint.

Once I got this piece of code working, I've started to insert it into the project sources in order to have a working login UI and a good working login process.

For the login UI I've used Cascades and created it in a bit, this is how it look at an early stage:



The image shows a screenshot of a WordPress login form. At the top center is the WordPress logo, a circular emblem with a stylized 'W' and a quill pen. Below the logo, the word "WORDPRESS" is written in a blue, serif font. The form consists of three input fields stacked vertically. The first field is labeled "Username" and contains the placeholder text "username". The second field is labeled "Password" and contains the placeholder text "YouRpAsSwOrD", with a small eye icon to its right for toggling visibility. The third field is labeled "Blog Address" and contains the placeholder text "leave empty if unsure". Below these fields is a single "Sign In" button with a slight gradient and rounded corners.

Once I've accomplished the login part, I've started keeping in time with my schedule and proceed to the next step that is to add a blog(s).

For this part the most interesting thing has been the UI creation, I've created a ListView that contains the blog(s) name and url returned from the API call *getUserBlogs*, I've implemented this list in order to support multi-selection, that wasn't too hard to do since Cascades support it, in a particular but working way.

After the user successfully login and select the blog(s) he want to use, I save this data in a *SQLite* – SQLite is the in-process database system with the best test coverage and support on all platforms - by using *QSqlDatabase* – a class that represents a connection to a database – and *QSqlQuery* – a class provides a means of executing and manipulating SQL statements – and store it in the app data directory.

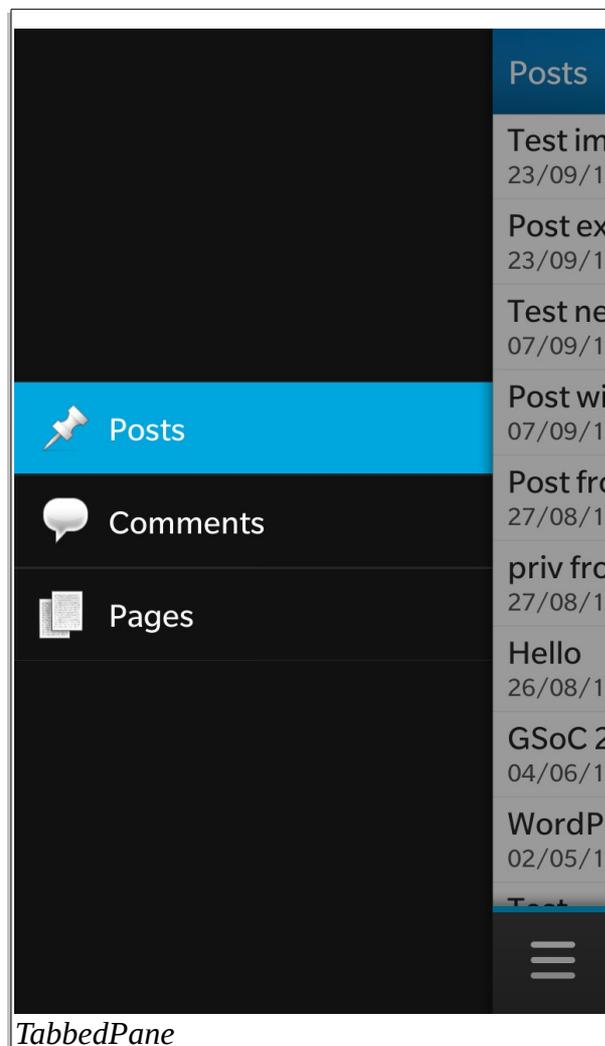
## Keep ready for the midterm

To be in time with the midterm evaluation, I've created all the necessary core part for reading posts/comments and pages. I've also wrote the code to make a new post/comment/page. Here I had to focus my attention on how to present this things to the user, how to put everything togheter. So I've created a first mock-up of how it would be, but it doesn't look so good. So I changed idea and started creating the best solution that comes to my mind.

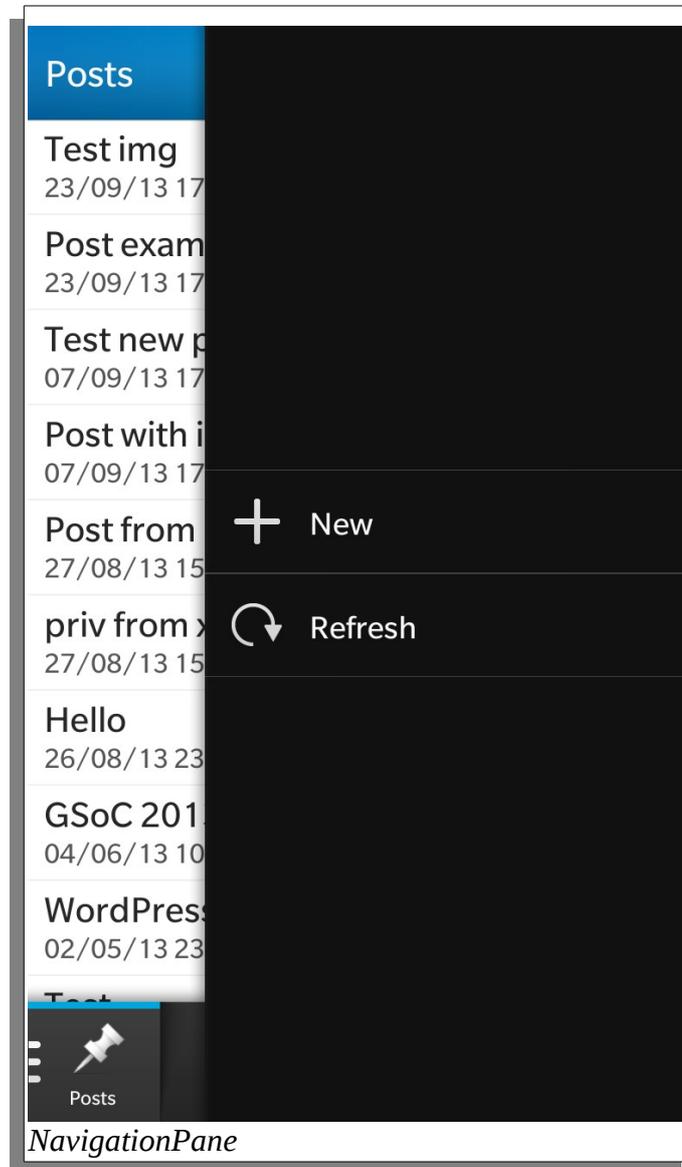
In Cascades you have two methods to show a page to a user : a *NavigationPane* – a class that is used for stack-like navigation between Page objects. The NavigationPane keeps track of a stack of Page objects that can be pushed and popped on the stack – or a *TabbedPane* – a navigation control that allows the user to switch between tabs. The tabs can be used to either completely replace displayed content by setting new panes or to filter existing content in a single pane based on which tab is currently selected.

Well, I had to come across a new method, using both classes, that (maybe) is not the best way, but it works fine without so much trouble.

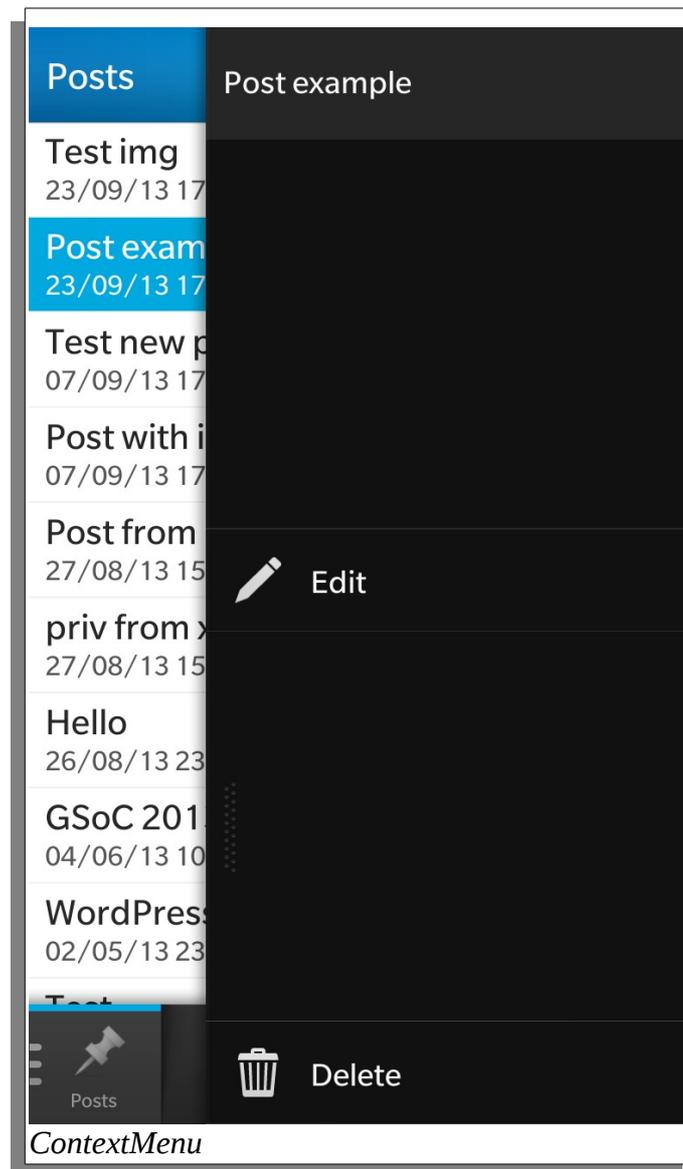
So, I've used a NaviagationPane for the login process, after the user logged in and register the blog(s) a TabbedPane show up, to let the user choice where he want to go, view posts/comments/pages, by default the Post View will be shown.



There a *NavigationPane* will take care of showing the other pages for the post/comment/page management, like creating a new one or refresh the page by using the respective API call method : *wp.getPosts*, *wp.getComments*, for the pages I've used the same API call that I use for the posts by simply changing the *post\_type* to 'page'. This saves me a lot of times, since I can use the same UI that I made for posts also for the pages management. There is a boolean variable that will be setted accordingly.



There is also a *ContextMenu* – a context menu (sometimes known as the cross-cut menu) is displayed when a user touches and holds a UI control – to let the user editing or delete an item.



Making this two things dealing together has been a little bit hard, but at the end they decided to work together without too much problems.

After I managed how to show the lists containing the post/comment/page, I start to implement a code to make a new blog post with an image. I read the WordPress XML-RPC API documentation several times to accomplish this work, I've created a first solution but it doesn't seems to work good, then I've ask for help to my mentor, and I found out that the solution was just a little forgotten tag in the XML creation. Once I fixed it, the app has been capable of make new blog post with image successfully, using the *uploadFile* API call method.

The app is also capable of showing the image, once you choose to view a post a JavaScript function will try to see if there is any image, if so, a *WebView* – A control that is used to display dynamic web content – is opened, else a simple *Label* – A non interactive label with one (or more) line(s) of text – will be shown.

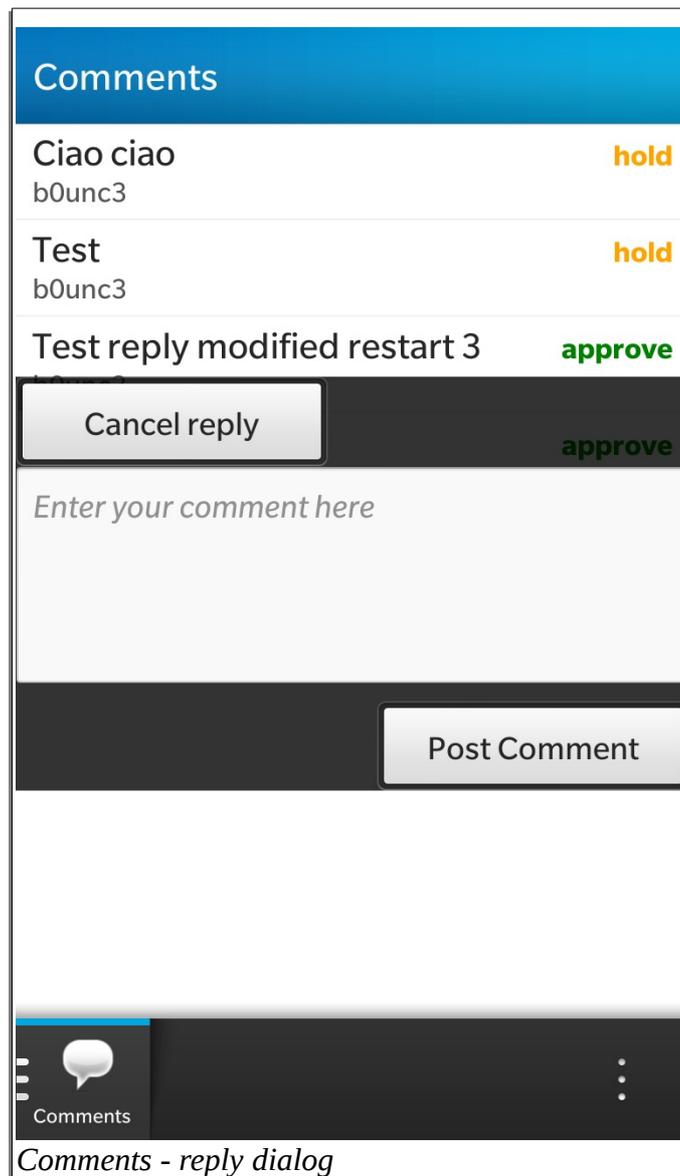
That was the milestone I've been accomplished for the midterm evaluation.

## Finalizing

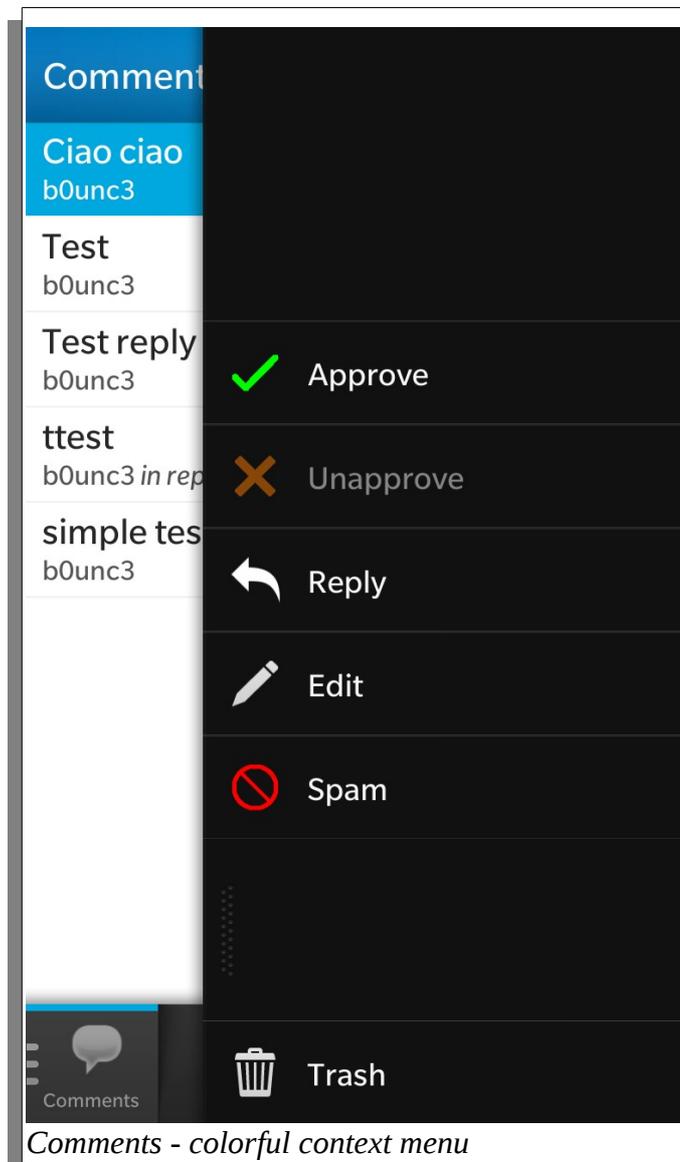
Once the midterm successfully goes over, I continued the work following the schedule and I've implemented an important part, that is the comments management.

For this part the core functions wasn't so hard to create, it was almost similar to all the other API call that I've already implemented.

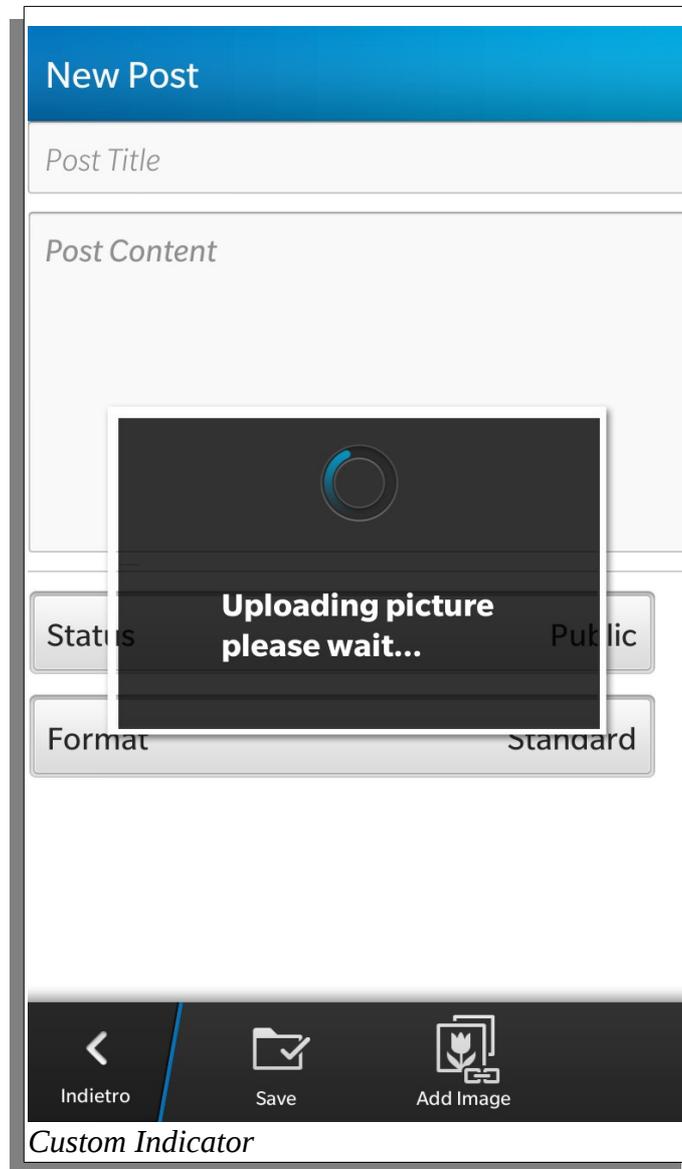
The funny part has been to create the UI. I'm not an active blogger so I don't know how to make it better for who will use this app daily. Here a nice idea (or at least I hope so) come up in my mind, showing the comments with the state bolded and colored: green for the approved status, orange for the comments that are waiting to be approved; in this way I hope that the user will easily found how to manage the comments. Another useful thing (I think), is that I've created a reply dialog, it has two buttons and a text area where the user can put the reply, easy and efficient as it sounds.



Here I've changed a bit the context-menu, to let the user quickly take an action about a comment. So the app is capable, through the context-menu, of changing (almost 'on-the-fly') the status of a comment or to mark it as spam or delete it.

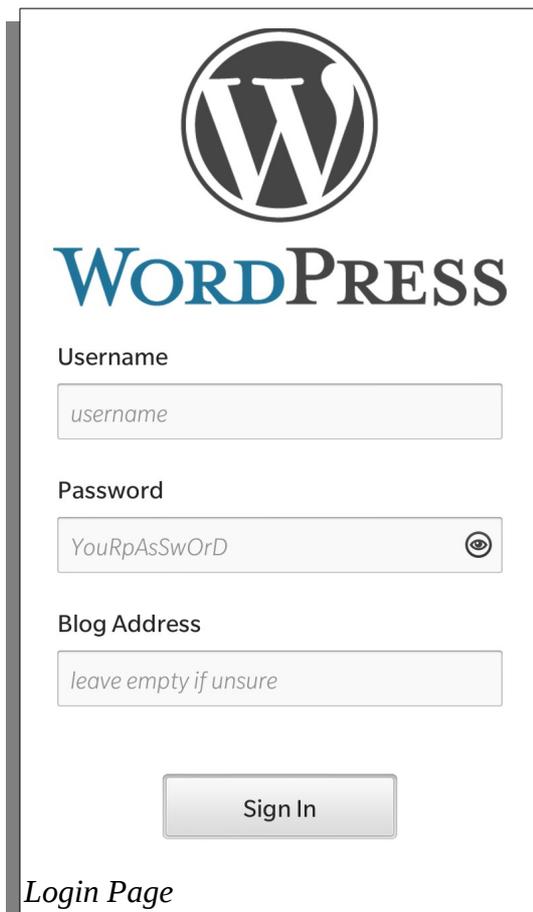


Another small problem I faced out with, was the need of a 'waiter': something that will tell to the user that there is an ongoing operation, like fetching/posting data. I wasn't able to find a suitable solutions from the ones offered by Cascades, so I decide to make my own component. This new component consist in a basic *Dialog* with a body, where you can put a message, and an *ActivityIndicator* – a control that indicates that a process is being completed –



Once this has been done, I've finished up the other pages, take care about the UI but also of the C++ part, by cleaning it up and adjust some functions and fix some (not so small) bugs.

Here is a step-by-step screenshots :



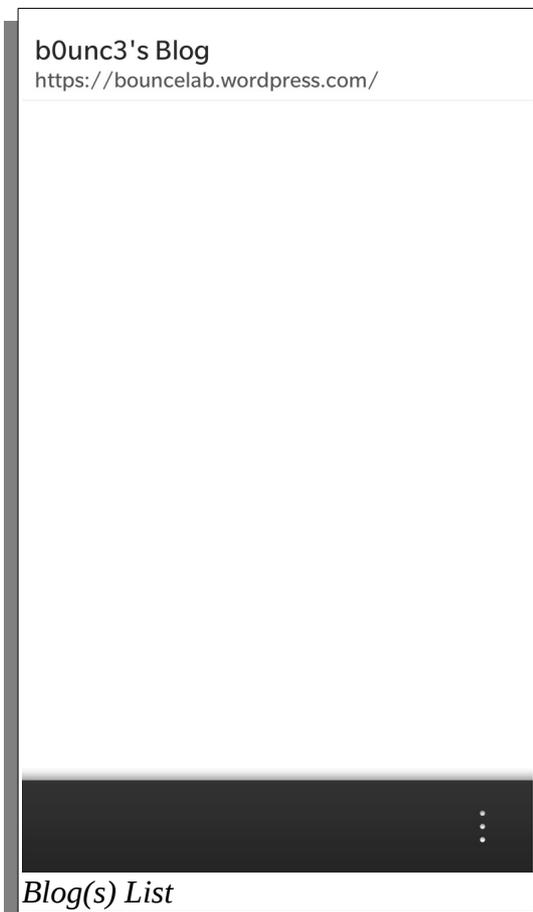
The screenshot shows the WordPress login page. At the top is the WordPress logo and the word "WORDPRESS" in blue. Below are three input fields: "Username" with the placeholder "username", "Password" with the placeholder "YouRpAsSwOrD" and an eye icon, and "Blog Address" with the placeholder "leave empty if unsure". A "Sign In" button is at the bottom.

Login Page



The screenshot shows the WordPress login page with the fields filled. The "Username" field contains "b0unc3", the "Password" field contains a series of dots, and the "Blog Address" field contains "leave empty if unsure". The "Sign In" button is at the bottom.

Login Page - filled



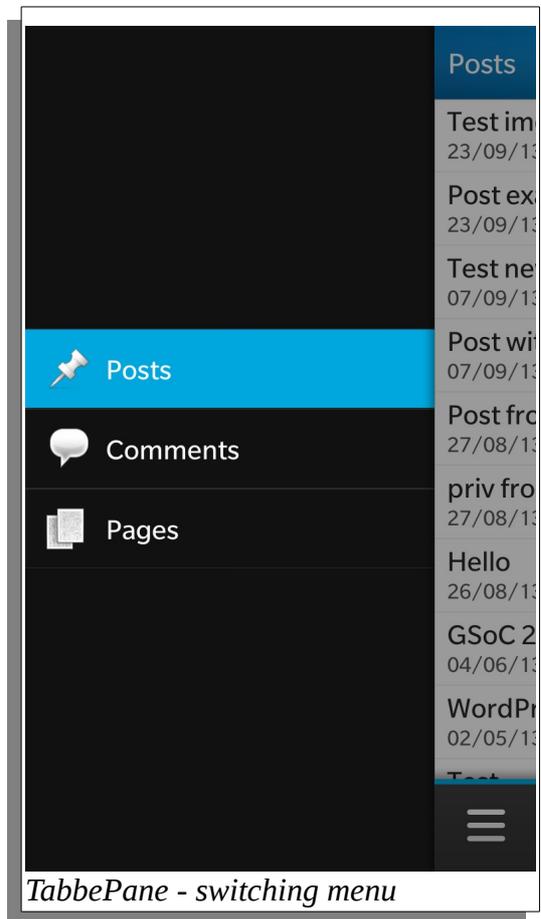
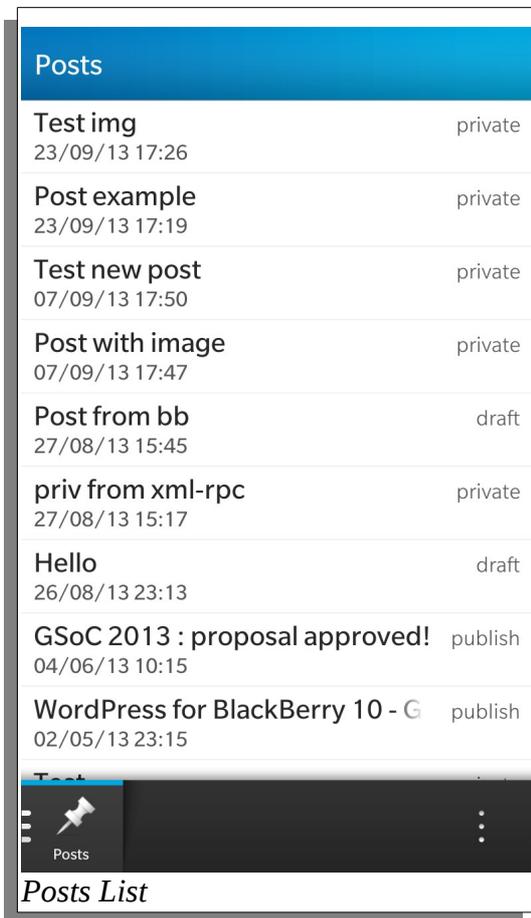
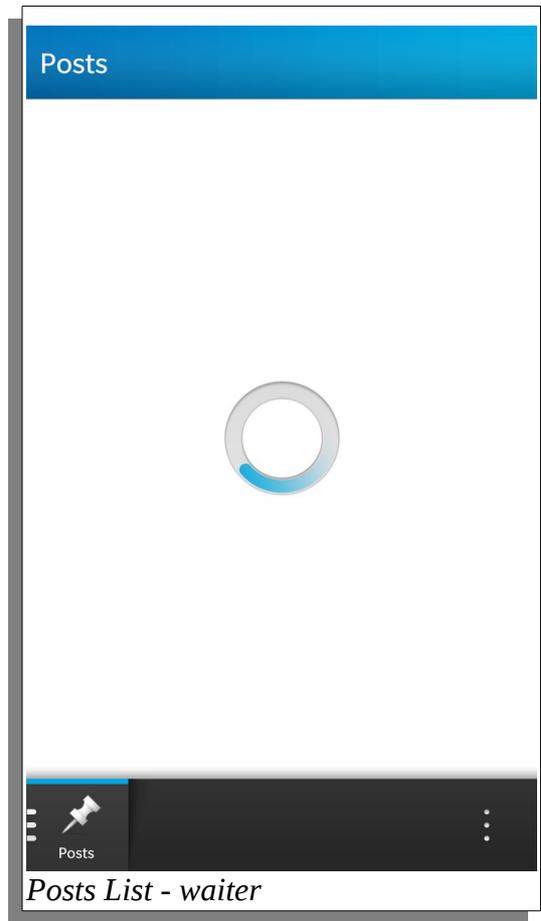
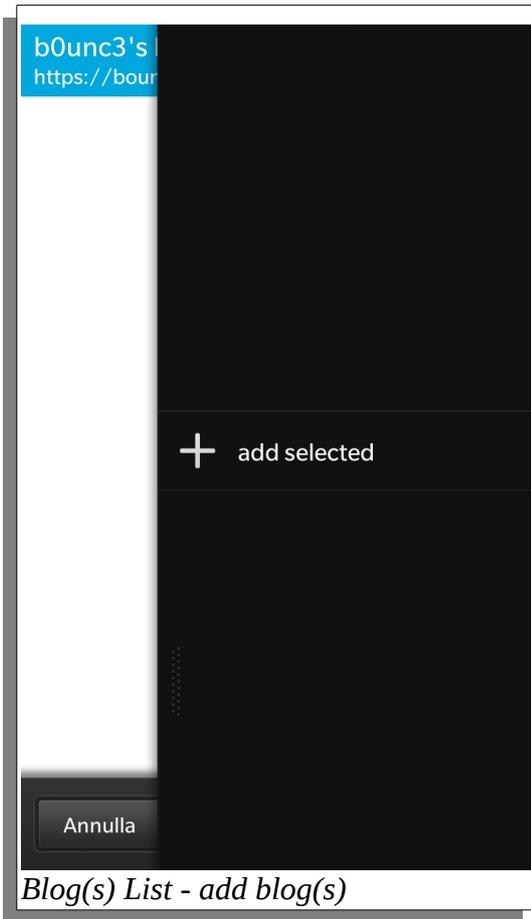
The screenshot shows the WordPress blog list page. At the top, it says "b0unc3's Blog" and "https://bouncelab.wordpress.com/". The main content area is empty. At the bottom, there is a dark bar with three vertical dots.

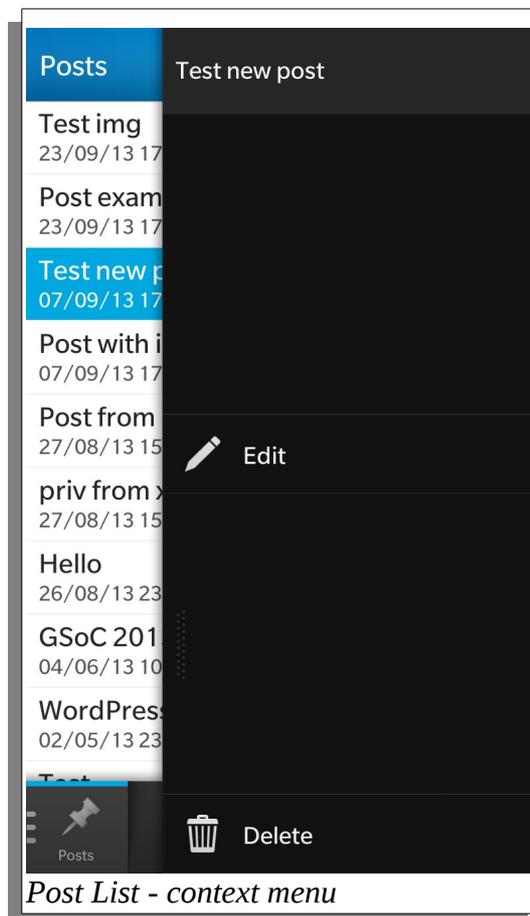
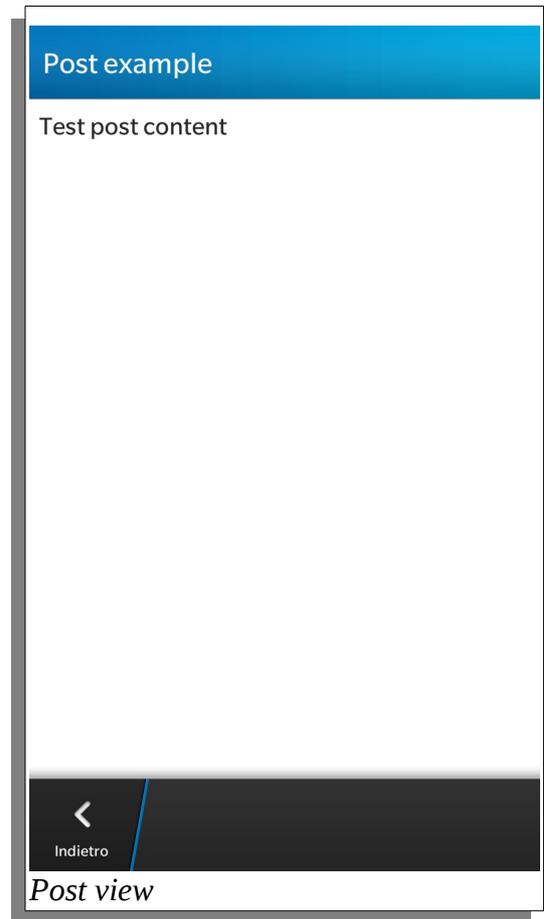
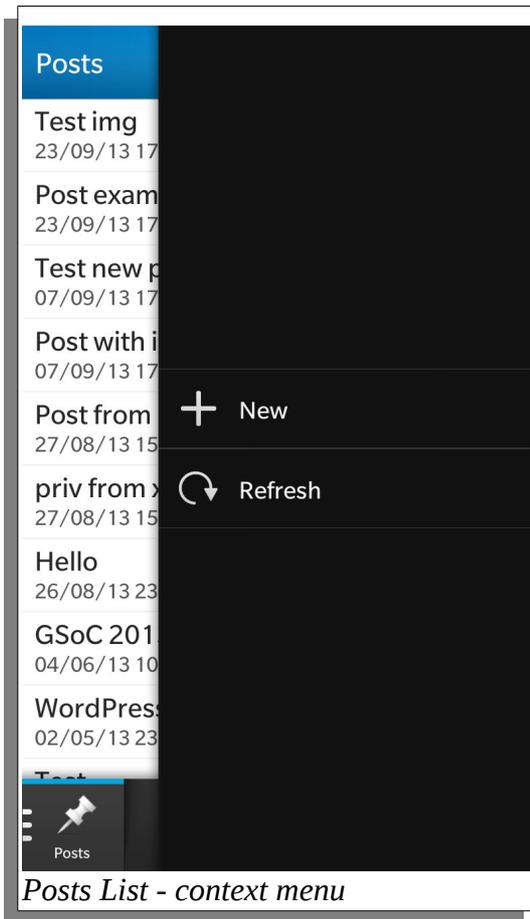
Blog(s) List

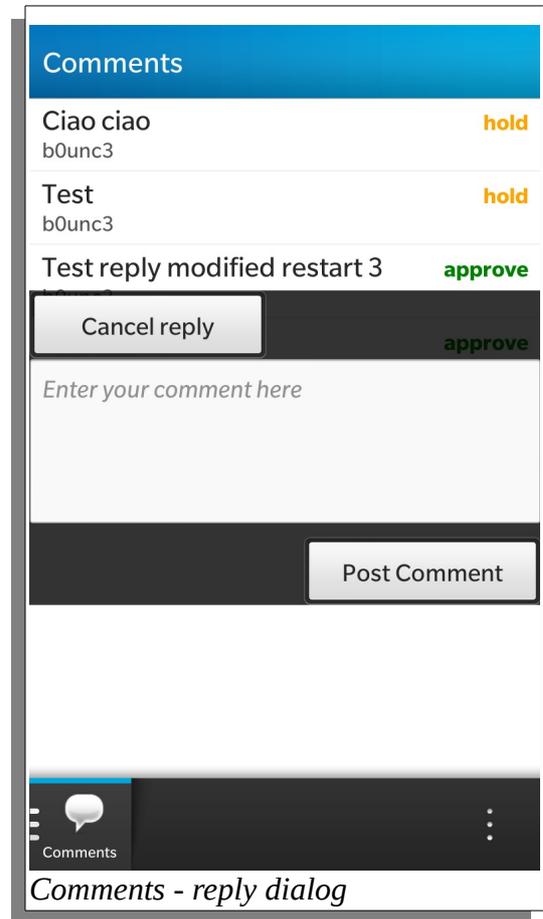
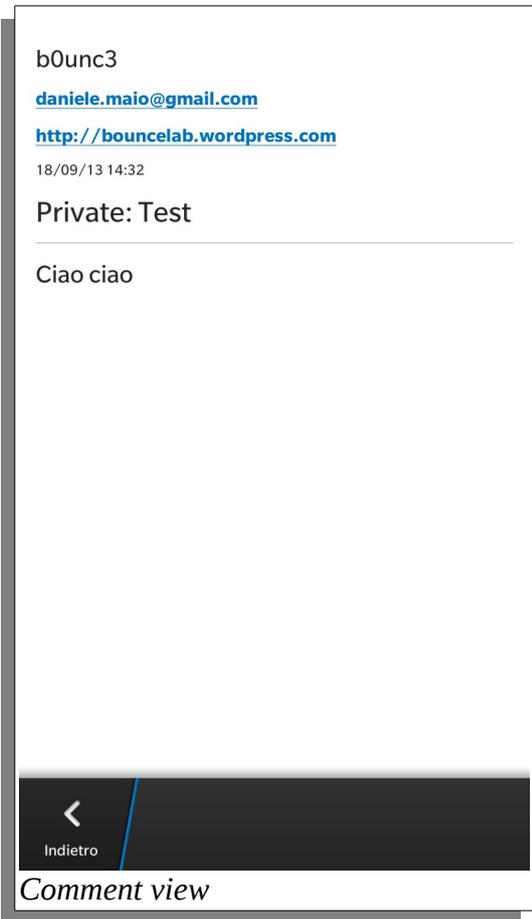
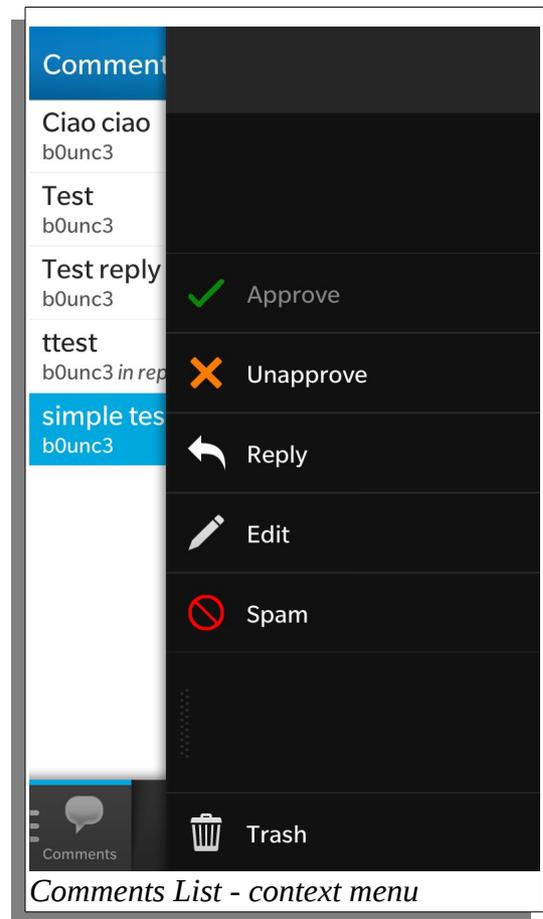
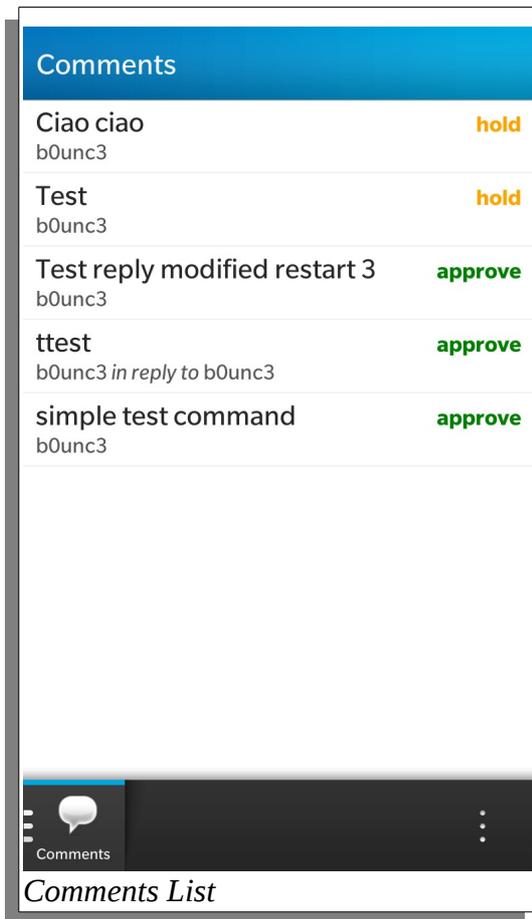


The screenshot shows the WordPress blog list page with a dark overlay menu. The menu has a checkmark icon and the text "select blog(s)".

Blog(s) List - multiselection







### Edit Comment

**Author**

**Author e-mail**

**Author url**

**Content**

**State**      Waiting for approve

Indietro      Edit

*Edit Comment*

### Pages

<b>test</b> 29/08/13 15:33	private
<b>About</b> 29/04/13 15:12	publish

Pages

*Pages List*

### Pages

<b>test</b> 29/08/13 15:33	test
<b>About</b> 29/04/13 15:12	

Edit

Delete

Pages

*Pages list - context menu*

## Final considerations

The project was successfully completed as I planned on my schedule. The app is working and do all the basic stuff for a complete blog management. On the other hand, the code needs a re-check, I haven't checked all the functions, some may need a better error handling, some may fall in some memory leak, there could be some unused function too.

For an example, for the creation of the XML in Qt initially I've started to use a QDom\* , after I switch, partially, to use a QDomWriter, but in the code both method still exists, this is because I would like to dig more on it, to see which is the best to use in this case, also in terms of performances. Speaking about performances, the app doesn't look to eat too much CPU, neither RAM, of course there is a significant increment of used RAM once you start the app (as for all the other apps), but it never gets too high; of course a better profiling must be executed.

There is also an important unchecked part that is the image upload. I didn't check for the initial size of the image, so if you upload a 10MB image (apart from the long time it will take) it will eat more resources, and, since there are some methods to avoid this behavior it would be good to apply it.

The multi-blogs part hasn't been tested, that's mostly because I run out of time. I will check it later on.

Some minor, but maybe useful, features are missing mostly because they weren't on my schedule, but it would be good to implement it one day in order to have a rich-features app.