Felinetek Public API Specification

Version **1.0.1**

Prepared by **Panos Hadjikomninos, FelineTek**

Last Revision: 25 March 2010

1. Introduction

1.1. Purpose

Documentation of the Public Domain Felinetek (FT) classes distributed under LGPL license. For more information see:

http://www.felinetek.com/fttoolbox

1.2. General Usage Instructions

To use any of the 3 classes included in this document, you **must** initialize **FTToolBox** and **FTCustomization**, even though you may not be subsequently using one or both of them (you may only be using **FTAsyncRequest**).

You also **must** add the following parameter in *info.plist* in order to use **FTCustomization**: *FTCustomizationString mycomp*

This implies you should have a *mycomp.lproj* localization directory with its own *Localizable.strings* file.

We suggest placing the initialization code in your ***AppDelegate*** code, inside the method: `didFinishLaunchingWithOptions`

Example

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
 ...
     !  [FTCustomization FTinitializeCustomization]; !

     !  [FTToolBox FTinitializeLog:YES atLevel:FT_DEBUGGING];

...
```

2. Class Description

Class FTToolBox

This is effectively a "*static*" class. By "*static*" we mean that no class variables or methods are used. The (few) variables needed are defined as **static**, and all the methods defined for this class are **class (+) methods**.

Therefore it is meaningless to create objects to this class, and any initialization necessary before use, is provided by the class method ***FTinitializeLog***. This method **must** be used prior to using this class, preferably from the *application delegate*.

The *FTToolBox* class is used for saving logging information for debugging purposes, during an application run. The logging process is enhanced using a few usefull concepts:

permanent logging statements
enhanced logging output
different severity logging levels
threshold-level may be set both statically & dynamically both crash-log and live-log

Lets examine these concepts in more detail :

1. Permanent Logging Statements

This class has been written to solve the problem of temporary logging statements for debugging purposes.

These statements have to be removed or commented out after debugging is complete, but may be needed later as more or unexpectedissues are found: Now the programmer may need to re-enter or uncomment the same statements.

The ***FT_LOGG*** function solves these problems, as it may be left permanently inside the code. Logging or not its contents are depended on 2 factors:

• If the app is ready for the app store, the **FTInitializeLog** parameter *debug_flag* may be set to **NO**, effectively removing all logging (until its changed to **YES** again)

• The *threshold_level* may be set statically (via **FTInitializeLog*)*** to a different value, thus effectively logging only ***FT_LOGG*** calls of severity level higher than the threshold (i.e. lower numeric value) , and omitting all other ***FT_LOGG*** calls. Thus we may only elect to log calls with *FT_ERROR* or higher level of severity, which are very few

• The *threshold_level* may be changed dynamically to a different (typically lower level) to effectively log more ***FT_LOGG*** calls.

2. Enhanced Logging Output

Logging automatically include the names of the **class** and the **method** were the ***FT_LOGG*** call occured.

3. Different Severity Logging Levels

Lower levels have higher severity, and vice versa. If the logging level is set to *FT_WARNING*, only ***FT_LOGG*** calls with levels 0..3 will be logged.

```
#define FT_UNRECOVERABLE 0
#define FT_CRITICAL 1
#define FT_ERROR 2
#define FT_WARNING 3
#define FT_INFORMATION 4
#define FT_LOGGING 5
#define FT_DEBUGGING 6
#define FT_ANALYTIC 7
```

```
#define FT_VERBOSE 8
```

FT_VERBOSE should be used sparingly for heavy logging (such as inside loops etc.). You should very rarely -- and for very short periods -- have to set your threshold level to *FT_VERBOSE* as this may slow the application to a crawl.

4. Threshold Levels may be set both Statically & Dynamically

The logging threshold level may be set statically and changed dynamically:

•The *threshold_level* may be set statically (via **FTInitializeLog*)*** to a different value, thus effectively logging only ***FT_LOGG*** calls of severity level higher than the threshold (i.e. lower numeric value) , and omitting all other ***FT_LOGG*** calls. Thus we may only elect to log calls with *FT_ERROR* or higher level of severity, which are very few

• The *threshold_level* may be changed dynamically to a different (typically lower level) to effectively log more ***FT_LOGG*** calls.

Avoid setting your threshold level to *FT_VERBOSE* as this may slow the application to a crawl.

5. Both Crash Log and Live Log

FTgetLogAsString returns the live log from the current application run.

FTgetCrashLogAsString returns the live log from the prior application run, wether the application has crashed or not. Therefore this is **not** necessarily a crash log. But if used judiciously by the user (is recovered immediately after he restarts the app , after a crash) you have effectively a “crash- log”, i.e. a log of the “crashed” prior run of the app.

Static Methods

+ *(void)* **FTinitializeLog:** *(BOOL) debug_flag* **atLevel***: (int) threshold_level*
Method called at the first start of the application in the appDelegate.m with the default method: *applicationDidFinishLaunchingWithOptions*.

(BOOL) debugFlag YES when you want debug to appear, be careful just before setting the application on the appStore you have to set this variable to NO.

(int) threshold_level Statically set threshold level.

+ *(NSString *)* **FTsetLogSeverity:** *(int) threshold*
Dynamically set the logging threshold
(int) threshold_level Dynamically set threshold level (resets static or prior value).

+ *(NSString *)* **FTgetLogAsString**

Returns the live log from this application run.

Returns:
*(NSString *)* The live-log (application current run).

+ *(NSString *)* **FTgetCrashLogAsString**

Returns the live log from the prior application run, wether the application has crashed or not. Therefore this is **not** necessarily a crash log. But if used judiciously by the user (is recovered immediately after he restarts the app , after a crash) you have effectively a “crash-log”, i.e. a log of the “crashed” prior run of the app.

Returns:
*(NSString *)* The crash-log (application prior run).

Main Logging Function

```
FT_LOGG(FT_LEVEL, format, …);
```

`int level` The level to be associated with this logging entry

`NSString * format` The format of the string to display

`... format arguments` The format arguments

Example

```
+(void) get {
NSString *requeststr = [request URL] absoluteString];
FT_LOGG(FT_DEBUGGING,@"URL = %@",requeststr);
...
```

Output:

```
FT_DEBUGGING(6):: FTAsyncRequest, get:: !URL = http://www.4gsecure.fr/test.php!
```

Class FTCustomization

This is effectively a "*static*" class. By "*static*" we mean that no class variables or methods are used. The (few) variables needed are defined as **static**, and all the methods defined for this class are **class (+) methods**.

Therefore it is meaningless to create objects to this class, and any initialization necessary before use, is provided by the class method ***FTinitializeCustomization***. This method **must** be used prior to using this class, preferably from the *application delegate*.

The **FTCustomization** class is based on apple's bundle customization primitives using *localizable.strings* files. But the process is enhanced and made easier with a few more advanced concepts:

localization sequence
cross data organization
incomplete Localizable.strings files
int, bool, float, UIColor & UIImage entries, in addition to Strings
compatibility with Android

Lets examine these concepts in more detail :

1. Localization Sequence

A sequence of "reads" determines the customization of each string, for example :

en.lproj --> fr.lproj --> ubiqus.lproj

The string localization outcome can change many times as the result may be rewritten by each file. The last outcome remains, so in the sequence above a later file has precedence over a previous one.

The first localization directory in the sequence is almost always "hardcoded" as **en.lproj**.

The second localization directory is the one that corresponds to the telephone language, so if you your iPhone is in French the 2nd directory in the sequence if **fr.lproj**. You may want the user of the application to be able to change the language of the application without changing the

language of the telephone (say, by clicking to an English flag inside the application.

In this case you can use the class function ***FToverrideCustomization*** *: (NSString *) lang* to override the second language in the sequence dynamically. If you call

[FTcustomization FToverrideCustomization:@"en"]

the sequence above changes to : **en.lproj --> en.lproj --> ubiqus.lproj**

Finally the 3rd file in the sequence is determined by setting the ***FTCustomizationString*** property in the *info.plist* file. In ther above case we would have the following entry in *info.plist:*

FTCustomizationString ubiqus

*This **FTCustomizationString** entry in **info.plist**, as well as the **FTinitializeCustomization** call are mandatory in order to use this class.*

2. Cross Data Organization

We may be organized on 2 vertical types of customization for each project:

1. Customization according to language

! Here we should have allmost all strings of the project in different languages under the language directories en.lproj, fr.lproj, de.lproj etc.

2. *Customization according to customer.*
! Here we should have colors, images and other project parameters (of type int, float etc)

under the customer directory (**ubiqus.lproj**).

In the same project we may have different customers (**ubiqus.lproj, customer2.lproj, generic.lproj** etc.), in case the same project may be re-marketed to different customers. We may equally have different themes (such as **gray.lproj, red.lproj**). To change the project to a different customer or theme, all we have to do is change ***FTCustomizationString*** and rebuild the project.

Thus we can have the same project customized with different customer logos, colors etc, and make different builds as necessary.

3. Incomplete Localizable.strings files

Under Apple's localization paradigm the *Localizable.strings* files must be complete, ie the **fr.lproj** file must contain all the strings necessary to the French localization.

With ***FTCustomization***, you should always put all your strings in English (which is kind of a universal language) in the **en.lproj** *Localizable.strings* file.

• If your phone is in French but an **fr.lproj** directory does not exist, the **en.lproj** entries will be used instead (as they come first in the localization sequence).
• If **fr.lproj** exists, but a specific string is not contained therein, then the **en.lproj** string will be used instead (again, as it comes first in the localization sequence).

Example

en.lproj:

```
"CancelString" = "Cancel";
"OKString" = “OK";
```

fr.lproj:

```
"CancelString" = "Annuler";
// “OKstring” is absent //
```

Customization result:

(Device language is French)

```
!  ! FTCustomString(@"CancelString") returns @"Annuler"

!  ! FTCustomString(@"OKString") returns @"OK"
```

(Device language is English)

```
!  ! FTCustomString(@"CancelString") returns @"Cancel"
!  ! FTCustomString(@"OKString") returns @"OK"
```

4. A variety of localizations

A variety of localizations is available:

- string
- int
- float
- BOOL
- UIColor
- UIImage

Example:

```
"CancelString" = "Cancel";
"PINminimumLength" = "4";
"PINmaximumLength" = "8";
"isPINrequired" = "YES";
"TimeIntervalInSeconds" = "0.8";
"TabBarColor"  = "92,66,134,54";
"SearchImage" = "search.png";
```

5. Compatibility with Android

A set of Java classes are provided that make it possible to use the same

x.lproj/Localizable.strings

directory structure on Android or any other Java based environment. This means you can just copy your files from one environment to another and reuse them.

Static Methods

+ *(void)* **FTinitializeCustomization**
Initializes the "static" class based on the value *FTCustomizationString* in the *info.plist* file.

- You **must** have an *FTCustomizationString* entry in the *info.plist* file.

- You **must** call this function, before using this class, preferably in the *didFinishLaunchingWithOptions function of your app delegate*

The localization sequence established by this call is :
en.lproj --> fr.lproj --> custom.lproj
where custom is the value of your *FTCustomizationString* property.

+ *(void)* **FTinitializeCustomization :** *(NSString *) lang*
Same as the previous call, only we use lang (say "fr") instead of "en" as the default 1st sequence entry. **Avoid using this call, as we always want "en" to be the base.**

*(NSString *) lang* The "base" language to be used first in the localization sequence.

The localization sequence established by this call is :
lang.lproj --> fr.lproj --> custom.lproj

+ *(void)* **FToverrideCustomization :** *(NSString *) lang*
Dynamically override the device language as the 2nd language in the localization sequence,**.**

*(NSString *) lang* The "overriding " language to be used as the 2nd language in the localization sequence.
The localization sequence established by this call is :

en.lproj --> lang.lproj --> custom.lproj

+ *(NSString *)* **FTdeviceLanguage**

Returns the language the device is set to.

Returns:
*(NSString *)* The device language such as “fr”.

+ *(NSString *)* **FTdefaultLanguage**

Returns the language the device is set to. Normally this is “en”.

Returns:
*(NSString *)* The default language normally “en”.

+ *(NSString *)* **FToverrideLanguage**

Returns the language that overrides the device language for customization.

Returns:
*(NSString *) nil* if not overriden.

Functions

```
UIColor * FTCustomColor(NSString * str);

UIImage * FTCustomImage(NSString * img);

NSString * FTCustomString(NSString * str);

float FTCustomFloat(NSString * str);

int FTCustomInt(NSString * str);

BOOL FTCustomBool(NSString*str);
```

Class FTAsyncRequest

This is a simple class for asynchronous ***GET*** and ***POST*** connections.

Properties

```
int timeout;                // If not set the default value is 24 seconds.
NSStringEncoding encoding;  // If not set the default value is NSUTF8StringEncoding.
NSString*contentType;       // If not set the default value is "text/plain; charset=utf-8".
```

Methods

- *(id)* **initWithUrl:** *(NSString *) url* **delegate***: (id) delegate* **onSuccess***: (SEL) success* **onFailure***: (SEL) failure*

Initialization method

*(NSString *) url* (id) *delegate*

(SEL) success

(SEL) failure

Returns: (id)

- *(void)* **getAsynchronously** Start an asynchronous ***GET***.

The HTTP URL to connect to.

The delegate, the object on which methods **success** and **failure** will be called.

The selector that will be called after the asynchronous connection has successfully completed. The selector must have one argument of type *(NSData *)* that contains the data retrieved via the asynchronous call.

The selector will be called if the asynchronous connection fails. The selector must have one argument of type *(NSNumber *)* that contains the error code for the failure.

A pointer to *self,* as this method is an ***init*** replacement.

- *(void)* **postAsynchronously:** *(NSData *) data* Start an asynchronous ***POST***.
*(NSData *) data* The data to post.

Example

```
rec = [[FTAsyncRequest alloc] FTinitWithUrl:@"http://
services.project.com/GetProfil/"
delegate:self onSuccess:@selector(successAction:)
onFailure:@selector(failAction:)];
rec.contentType=@"application/json; charset=utf-8";
[rec setTimeout:15];
NSString * str = [NSString stringWithString:@"{\"userFirstName\":
\"Boris\",
\"userLastName\":\"OUDET\"}"];
[rec FTpostAsynchronously:[str
dataUsingEncoding:NSUTF8StringEncoding]];
```