Abstract

The Application Framework is one of the key components of the AGL platform. It allows
to install/uninstall applications securely on a target device and handles applications' life cycle. The Application Framework is also responsible for setting up the secure contexts used to run applications as well as enabling privilege checks at various levels. It finally defines a necessary isolation between non-trusted high level applications run by end users and low level BSP trusted services running at lower level.

This document is the developer documentation.

**Document revisions**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Designation</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 May 2016</td>
<td>0.9</td>
<td>Initial release</td>
<td>J. Bollo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>M. Bachmann</td>
</tr>
<tr>
<td>30 May 2016</td>
<td>1.0</td>
<td>Master document edition, final review</td>
<td>S. Desneux</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F. Ar Foll</td>
</tr>
<tr>
<td>21 Sept 2016</td>
<td>2.0</td>
<td>Updated with new sections (events, widgets)</td>
<td>J. Bollo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S. Desneux</td>
</tr>
</tbody>
</table>
Table of contents

1. AGL framework, overview of the proposal of IoT.bzh ........................................... 7
   1.1. Foreword ............................................................................................................ 7
   1.2. Introduction ......................................................................................................... 7
   1.3. Overview ............................................................................................................ 9
   1.4. Links between the "Security framework" and the "Application framework" ........................................... 11
   1.5. The security framework .................................................................................. 11
   1.6. The application framework ............................................................................. 11
2. The afm-system-daemon ....................................................................................... 13
   2.1. Foreword ........................................................................................................... 13
   2.2. Introduction ....................................................................................................... 13
   2.3. Starting afm-system-daemon ........................................................................... 14
   2.4. The D-Bus interface .......................................................................................... 14
      2.4.1. Overview of the dbus interface .................................................................. 14
      2.4.2. The protocol over D-Bus ......................................................................... 15
3. The afm-user-daemon ............................................................................................. 17
   3.1. Foreword ........................................................................................................... 17
   3.2. Introduction ....................................................................................................... 17
   3.3. Tasks of afm-user-daemon ............................................................................... 18
      3.3.1. Maintaining list of applications .................................................................. 18
      3.3.2. Launching application .............................................................................. 18
      3.3.3. Managing instances of running applications .......................................... 19
      3.3.4. Installing and un installing applications ................................................... 19
   3.4. Starting afm-user-daemon ............................................................................... 19
   3.5. Launcher Configuration ..................................................................................... 20
      3.5.1. mode local .................................................................................................... 21
      3.5.2. mode remote ............................................................................................... 21
      3.5.3. %substitutions ............................................................................................. 22
   3.6. The D-Bus interface ......................................................................................... 23
      3.6.1. Overview of the dbus interface .................................................................. 23
      3.6.2. Using afm-util ............................................................................................. 24
      3.6.3. The protocol over D-Bus ......................................................................... 25
4. The widgets ............................................................................................................... 29
   4.1. The widgets ....................................................................................................... 29
      4.1.1. The configuration file config.xml ............................................................... 29
      4.1.2. Tools for managing widgets ....................................................................... 29
      4.1.3. Signing a widget ........................................................................................ 30
      4.1.4. Packing a widget ....................................................................................... 30
      4.1.5. Getting data about a widget file ................................................................ 30
   4.2. Writing a widget ............................................................................................... 31
      4.2.1. What kind of application? .......................................................................... 31
      4.2.2. The steps for writing a widget .................................................................. 31
   4.3. Organization of directory of applications ...................................................... 31
      4.3.1. directory where are stored applications ..................................................... 31

Version 2.0  September 2016

- 3 / 91 -
8. HOWTO WRITE an APPLICATION above AGL FRAMEWORK

8.1. Programming Languages for Applications

8.1.1. Writing an HTML5 Application

8.1.2. Writing a Qt Application

8.1.3. Writing "C" Application

8.2. Handling sessions within applications
10. HOW TO WRITE a BINDING for AFB-DAEMON

10.1. Summary

10.1.1. Nature of a binding

10.1.2. Class of bindings

10.1.3. Live cycle of bindings within afb-daemon

10.1.4. Binding Contend

10.2. The TicTacToe example

10.3. Dependencies when compiling

10.4. Header files to include

10.5. Choosing names

10.5.1. Names for API (binding)

10.5.2. Names for methods

10.5.3. Names for arguments

10.5.4. Forging names widely available

10.6. Writing a synchronous method implementation

10.6.1. The incoming request

10.6.2. Associating a client context to a session

10.6.3. Sending reply to a request

10.7. Getting argument of invocation

10.7.1. Basic functions for querying arguments

10.7.2. Arguments for received files

10.7.3. Arguments as a JSON object

10.8. Initialisation of the binding and declaration of methods

10.9. Sending messages to the log system

10.9.1. Verbs for logging messages

10.9.2. Managing verbosity

10.9.3. Output format and destination
10.10. Sending events ................................................................. 75
10.11. Writing an asynchronous method implementation .......... 76
10.12. How to build a binding .................................................. 77
  10.12.1. Example for cmake meta build system ....................... 78
  10.12.2. Exporting the function afbBindingV1Register .............. 79
  10.12.3. Building within yocto ............................................ 79
11. Guide for developing with events ......................................... 80
  11.1. Overview of events ....................................................... 80
    11.1.1. Subscribing and unsubscribing .................................. 80
    11.1.2. Generating and pushing signals and data .................... 82
    11.1.3. Receiving the signals ............................................ 83
    11.1.4. The exceptional case of wide broadcast ...................... 83
  11.2. Reference of functions ................................................ 84
    11.2.1. Function afb_event afb_daemon_make_event .................. 84
    11.2.2. Function afb_event_push ........................................ 84
    11.2.3. Function afb_event_drop ........................................ 84
    11.2.4. Function afb_req_subscribe ..................................... 85
    11.2.5. Function afb_req_unsubscribe .................................. 85
    11.2.6. Function afb_event_broadcast .................................. 85
    11.2.7. Function afb_daemon_broadcast_event ....................... 85
  11.3. Architectural digressions ............................................. 86
    11.3.1. Strict separation .................................................. 86
    11.3.2. Soft composition .................................................. 87
12. Overview of tests shipped with AFB-Daemon ......................... 88
  12.1. List of tests ............................................................. 88
    12.2.1. afb-client-demo (command-line WebSockets) ................ 88
    12.2.2. token-websock.qml (Qt/QML WebSockets) ................. 88
    12.2.3. *.html (HTML5/JS HTTP-REST & WebSockets) .............. 89
13. Vocabulary for AFB-DAEMON ............................................. 90
  13.1. Binding ................................................................. 90
  13.2. Event ................................................................. 90
  13.3. Level of assurance (LOA) ........................................... 90
  13.4. Plugin ............................................................... 90
  13.5. Request ............................................................. 90
  13.6. Reply/Response ...................................................... 90
  13.7. Service ............................................................. 90
  13.8. Session .............................................................. 91
  13.9. Token ............................................................... 91
  13.10. UUID ............................................................... 91
  13.11. x-afb-reqid ....................................................... 91
  13.12. x-afb-token ...................................................... 91
  13.13. x-afb-uuid ...................................................... 91
1. AGL framework, overview of the proposal of IoT.bzh

1.1. Foreword

This document describes what we intend to do. It may happen that our current implementation and the content of this document differ.

In case of differences, it is assumed that this document is right and the implementation is wrong.

1.2. Introduction

During the first works in having the security model of Tizen integrated in AGL (Automotive Grade Linux) distribution, it became quickly obvious that the count of components specific to Tizen to integrate was huge.

Here is a minimal list of what was needed:

- platform/appfw/app-installers
- platform/core/security/cert-svc
- platform/core/appfw/ail
- platform/core/appfw/aul-1
- platform/core/appfw/libslp-db-util
- platform/core/appfw/pkgmgr-info
- platform/core/appfw/slp-pkgmgr

But this list is complete because many dependencies are hidden. Those hidden dependencies are including some common libraries but also many tizen specific sub-components (iniparser, bundle, dlog, libtzplatform-config, db-util, vconf-buxton, ...).

This is an issue because AGL is not expected to be Tizen. Taking it would either need to patch it for removing unwanted components or to take all of them.

However, a careful study of the core components of the security framework of Tizen showed that their dependencies to Tizen are light (and since some of our work, there is no more dependency to tizen). Those components are cynara, security-manager, D-Bus aware of cynara.

Luckily, these core security components of Tizen are provided by meta-intel-iot-security, a set of yocto layers. These layers were created by Intel to isolate Tizen specific security components from the initial port of Tizen to Yocto. The 3 layers are providing components for:

- Implementing Smack LSM
- Implementing Integrity Measurement Architecture

Version 2.0 September 2016
• Implementing Tizen Security Framework

The figure below shows the history of these layers.

```
2014     2015
Tizen OBS ------------------------->  \
  \                                    
  Tizen Yocto --------------------->  \
  \                              
meta-intel-iot-security +---------->
```

We took the decision to use these security layers that provides the basis of the Tizen security, the security framework.

For the components of the application framework, built top of the security framework, instead of pulling the huge set of packages from Tizen, we decided to refit it by developing a tiny set of components that would implement the same behaviour but without all the dependencies and with minor architectural improvements for AGL.

These components are **afm-system-daemon** and **afm-user-daemon**. They provides infrastructure for installing, uninstalling, launching, terminating, stopping and resuming applications in a multi user secure environment.

A third component exists in the framework, the binder **afb-daemon**. The binder provides the easiest way to provide secured API for any tier. Currently, the use of the binder is not absolutely mandatory.

This documentation explains the framework created by IoT.bzh by rewriting the Tizen Application Framework. Be aware of the previous foreword.
1.3. Overview

The figure below shows the major components of the framework and their interactions going through the following scenario: APPLICATION installs an other application and then launch it.

Let follow the sequence of calls:

1. APPLICATION calls its **binder** to install the OTHER application.
2. The binding **afm-main-binding** of the **binder** calls, through **D-Bus** system, the system daemon to install the OTHER application.

3. The system **D-Bus** checks whether **APPLICATION** has the permission or not to install applications by calling **CYNARA**.

4. The system **D-Bus** transmits the request to **afm-system-daemon**. **afm-system-daemon** checks the application to install, its signatures and rights and installs it.

5. **afm-system-daemon** calls **SECURITY-MANAGER** for fulfilling security context of the installed application.

6. **SECURITY-MANAGER** calls **CYNARA** to install initial permissions for the application.

7. **APPLICATION** calls its binder to start the nearly installed OTHER application.

8. The binding **afm-main-binding** of the **binder** calls, through **D-Bus** session, the user daemon to launch the OTHER application.

9. The session **D-Bus** checks whether **APPLICATION** has the permission or not to start an application by calling **CYNARA**.

10. The session **D-Bus** transmits the request to **afm-user-daemon**.

11. **afm-user-daemon** checks whether **APPLICATION** has the permission or not to start the OTHER application **CYNARA**.

12. **afm-user-daemon** uses **SECURITY-MANAGER** features to set the security context for the OTHER application.

13. **afm-user-daemon** launches the OTHER application.

This scenario does not cover all the features of the frameworks. Shortly because details will be revealed in the next chapters, the components are:

- **SECURITY-MANAGER**: in charge of setting Smack contexts and rules, of setting groups, and, of creating initial content of **CYNARA** rules for applications.

- **CYNARA**: in charge of handling API access permissions by users and by applications.

- **D-Bus**: in charge of checking security of messaging. The usual D-Bus security rules are enhanced by **CYNARA** checking rules.

- **afm-system-daemon**: in charge of installing and uninstalling applications.

- **afm-user-daemon**: in charge of listing applications, querying application details, starting, terminating, stopping, resuming applications and their instances for a given user context.

- **afb-binder**: in charge of serving resources and features through an HTTP
interface.

*afm-main-binding*: This binding allows applications to use the API of the AGL framework.

### 1.4. Links between the "Security framework" and the "Application framework"

The security framework refers to the security model used to ensure security and to the tools that are provided for implementing that model.

The security model refers to how DAC (Discretionary Access Control), MAC (Mandatory Access Control) and Capabilities are used by the system to ensure security and privacy. It also includes features of reporting using audit features and by managing logs and alerts.

The application framework manages the applications: installing, uninstalling, starting, stopping, listing ...

The application framework uses the security model/framework to ensure the security and the privacy of the applications that it manages.

The application framework must be compliant with the underlying security model/framework. But it should hide it to the applications.

### 1.5. The security framework

The implemented security model is the security model of Tizen 3. This model is described [here](#).

The security framework then comes from Tizen 3 but through the meta-intel. It includes: Security-Manager, Cynara and D-Bus compliant to Cynara.

Two patches are applied to the security-manager. These patches are removing dependencies to packages specific of Tizen but that are not needed by AGL. None of these patches adds or removes any behaviour.

**Theoretically, the security framework/model is an implementation details that should not impact the layers above the application framework.**

The security framework of Tizen provides "nice lad" a valuable component to scan log files and analyse auditing. This component is still in development.

### 1.6. The application framework

The application framework on top of the security framework provides the components to install and uninstall applications and to run it in a secured environment.

The goal is to manage applications and to hide the details of the security framework to
the applications.

For the reasons explained in introduction, we did not used the application framework of Tizen as is but used an adaptation of it.

The basis is kept identical: the applications are distributed in a digitally signed container that must match the specifications of widgets (web applications). This is described by the technical recommendations widgets and widgets-digsig of the W3 consortium.

This model allows the distribution of HTML, QML and binary applications.

The management of signatures of the widget packages This basis is not meant as being rigid and it can be extended in the futur to include for example incremental delivery.
2. The afm-system-daemon

2.1. Foreword

This document describes application framework system daemon fundamentals. FCF (Fully Conform to Specification) implementation is still under development. It may happen that current implementation somehow diverges with specifications.

2.2. Introduction

The daemon **afm-system-daemon** is in charge of installing applications on AGL system. Its main tasks are:

- installs applications and setup security framework for newly installed application.
- uninstall applications

The **afm-system-daemon** takes its orders from system instance of D-Bus.

The figure below summarizes the situation of **afm-system-daemon** in the system.
2.3. Starting afm-system-daemon

`afm-system-daemon` is launched as a `systemd` service attached to system. Normally, the service file is located at `/lib/systemd/system/afm-system-daemon.service`.

The options for launching `afm-system-daemon` are:

```
-r
--root directory
    Set the root application directory.
    Note that the default root directory is defined to be /usr/share/afm/applications (may change).

-d
--daemon
    Daemonizes the process. It is not needed by systemd.

-q
--quiet
    Reduces the verbosity (can be repeated).

-v
--verbose
    Increases the verbosity (can be repeated).

-h
--help
    Prints a short help.
```

2.4. The D-Bus interface

2.4.1. Overview of the dbus interface

`afm-system-daemon` takes its orders from the session instance of D-Bus. The use of D-Bus is great because it allows to implement discovery and signaling.

The `afm-system-daemon` is listening with the destination name `org.AGL.afm.system` at the object of path `/org/AGL/afm/system` on the interface `org.AGL.afm.system` for the below detailed members `install` and `uninstall`.

D-Bus is mainly used for signaling and discovery. Its optimized typed protocol is not used except for transmitting only one string in both directions.

The client and the service are using JSON serialization to exchange data.

The D-Bus interface is defined by:
• DESTINATION: `org.AGL.afm.system`
• PATH: `/org/AGL/afm/system`
• INTERFACE: `org.AGL.afm.system`

The signature of any member of the interface is `string -> string` for `JSON -> JSON`.

This is the normal case. In case of error, the current implementation returns a dbus error that is a string.

Here is an example using `dbus-send` to query data on installed applications.

```
dbus-send --session --print-reply \\  
 --dest=org.AGL.afm.system \\  
 /org/AGL/afm/system \\  
 org.AGL.afm.system.install 'string:"/tmp/appli.wgt"'
```

### 2.4.2. The protocol over D-Bus

**Method `org.AGL.afm.system.install`**

**Description**: Install an application from a widget file.

When an application with the same `id` and `version` already exists. Outside of using `force=true` the application is not reinstalled.

Applications are installed the subdirectories of applications common directory. If `root` is specified, the application is installed under the sub-directories of the `root` defined.

Note that this methods is a simple accessor method of `org.AGL.afm.system.install` from `afm-system-daemon`.

After the installation and before returning to the sender, `afm-system-daemon` sends a signal `org.AGL.afm.system.changed`.

**Input**: The `path` of the widget file to install and, optionally, a flag to `force` reinstallation, and, optionally, a `root` directory.

Either just a string being the absolute path of the widget file:

```
"/a/path/driving/to/the/widget"
```

Or an object:

```json
{
  "wgt": "/a/path/to/the/widget",
  "force": false,
  "root": "/a/path/to/the/root"
}
```

"wgt" and "root" must be absolute paths.

**output**: An object with the field "added" being the string for the id of the added application.
Method org.AGL.afm.system.uninstall

Description: Uninstall an application from its id.

Note that this method is a simple method accessor of org.AGL.afm.system.uninstall from afm-system-daemon.

After the uninstallation and before returning to the sender, afm-system-daemon sends a signal org.AGL.afm.system.changed.

Input: the id of the application and optionally the application root path.

Either a string:

"appli@x.y"

Or an object:

{ 
  "id": "appli@x.y",
  "root": "/a/path/to/the/root"
}

Output: the value 'true'.
3. The afm-user-daemon

3.1. Foreword

This document describes application framework user daemon fundamentals. FCF (Fully Conform to Specification) implementation is still under development. It may happen that current implementation somehow diverges with specifications.

3.2. Introduction

The daemon **afm-user-daemon** is in charge of handling applications on behalf of a user. Its main tasks are:

- enumerate applications that end user can run and keep this list available on demand
- start applications on behalf of end user, set user running environment, set user security context
- list current runnable or running applications
- stop (aka pause), continue (aka resume), terminate a running instance of a given application
- transfer requests for installation/uninstallation of applications to the corresponding system daemon **afm-system-daemon**

The **afm-user-daemon** takes its orders from the session instance of D-Bus.
The figure below summarizes the situation of afm-user-daemon in the system.

### 3.3. Tasks of afm-user-daemon

#### 3.3.1. Maintaining list of applications

At start afm-user-daemon scans the directories containing applications and load in memory a list of available applications accessible by current user.

When afm-system-daemon installs or removes an application. On success it sends the signal org.AGL.afm.system.changed. When receiving such a signal, afm-user-daemon rebuilds its applications list.

afm-user-daemon provides the data it collects about applications to its clients. Clients may either request the full list of available applications or a more specific information about a given application.

#### 3.3.2. Launching application

afm-user-daemon launches application. Its builds a secure environment for the application before starting it within a secured environment.

Different kind of applications can be launched.

This is set using a configuration file that describes how to launch an application of a given kind within a given mode.
There are two launching modes: local or remote.

Launching an application locally means that the application and its binder are launched together.

Launching application remotely translates in only launching the application binder. The UI by itself has to be activated remotely by the requested (ie: HTML5 homescreen in a browser).

Once launched, running instances of application receive a runid that identify them.

### 3.3.3. Managing instances of running applications

**afm-user-daemon** manages the list of applications that it launched.

When owning the right permissions, a client can get the list of running instances and details about a specific running instance. It can also terminates, stops or continues a given application.

### 3.3.4. Installing and uninstalling applications

If the client own the right permissions, **afm-user-daemon** delegates that task to **afm-system-daemon**.

### 3.4. Starting afm-user-daemon

**afm-user-daemon** is launched as a **systemd** service attached to user sessions. Normally, the service file is located at /usr/lib/systemd/user/afm-user-daemon.service.

The options for launching **afm-user-daemon** are:

```
-a
---application directory
    Includes the given application directory to the database base of applications.
    Can be repeated.

-r
---root directory
    Includes root application directory or directories when passing multiple rootdir to applications database.
    Note that default root directory for applications is always added. In current version /usr/share/afm/applications is used as default.

-m
---mode (local|remote)
    Set the default launch mode.
    The default value is 'local'
```
3.5. Launcher Configuration

It contains rules for launching applications. When `afm-user-daemon` has to launch an application, it looks for launch mode (local or remote), as well as for the type of application describe in `config.xml` widget configuration file.

This tuple mode+type allows to select the adequate rule.

Configuration file is `/etc/afm/afm-launch.conf`.

It contains sections and rules. It can also contain comments and empty lines to improve readability.

The separators are space and tabulation, any other character should have a meaning.

The format is line oriented. The new line character separate the lines.

Lines having only separators are blank lines and ignored. Line having character `#` (sharp) at first position are comment lines and ignored.

Lines not starting with a separator are different from lines starting with a separator character.

The grammar of the configuration file is defined below:

```plaintext
CONF: *COMMENT *SECTION
SECTION: MODE *RULE
RULE: +TYPE VECTOR ?VECTOR
MODE: 'mode' +SEP ('local' | 'remote') *SEP EOL
TYPE: DATA *SEP EOL
```
Here is a sample of configuration file for defining how to launch an application of types `application/x-executable`, `text/x-shellscript` and `text/html` in local mode:

```
mode local
  application/x-executable
  text/x-shellscript
      %r/%c
  text/html
      /usr/bin/afb-daemon --mode=local --readyfd=%R --alias=/icons:%I --port=%P --rootdir=%r --token=%S --sessiondir=%D/.afb-daemon
      /usr/bin/web-runtime http://localhost:%P/%c?token=%S
```

This shows that:

- within a section, several rules can be defined
- within a rule, several types can be defined
- within a rule, one or two vectors can be defined
- vectors are using %substitution
- launched binaries must be defined with their full path

### 3.5.1. mode local

Within this mode, the launchers have either one or two description vectors. All of those vectors are treated as programs and are executed with `execve` system call.

The first vector is the leader vector and it defines the process group. The second vector (if any) is attached to the group defined by this first vector.

### 3.5.2. mode remote

Within this mode, the launchers have either one or two vectors describing them.

The first vector is process as a program and is executed with system call `execve`.

The second vector (if any) defines a text that is returned to the caller. This mechanism can be used to return a uri for remote UI to connect on the newly launched application.

The daemon `afm-user-daemon` allocates a port for each new remote application. The current implementation port allocation is incremental. A smarter (cacheable and
discoverable) allocation should be defined.

### 3.5.3. %substitutions

Vectors can include sequences of 2 characters that have a special meaning. These sequences are named %substitution because their first character is the percent sign (%) and because each occurrence of the sequence is replaced, at launch time, by the value associated to sequences.

Here is the list of %substitutions:

- **%%**: %.
  - This simply emits the percent sign %
- **%a**: appid
  - Holds application Id of launched application.
  - Defined by the attribute `id` of the element of `config.xml`.
- **%b**: bindings
  - In the future should represent the list of bindings and bindings directory separated by ','. Warning: not supported in current version.
  - **%c**: content
  - The file within the widget directory that is the entry point.
  - For HTML applications, it represents the relative path to main page (aka index.html).
  - Defined by attribute `src` of the element within `config.xml`.
- **%D**: datadir
  - Path of the directory where the application runs (cwd) and stores its data.
  - It is equal to %h/%a.
- **%H**: height
  - Requested height for the widget.
  - Defined by the attribute `height` of the element of `config.xml`.
- **%h**: homedir
  - Path of the home directory for all applications.
  - It is generally equal to $HOME/app-data
- **%i**: icondir
  - Path of the directory were the icons of the applications can be found.
- **%m**: mime-type
  - Mime type of the launched application.
  - Defined by the attribute `type` of the element of `config.xml`.
- **%n**: name
Name of the application as defined by the content of the element of config.xml.

- **%P**: port
  A port to use. It is currently a kind of random port. The precise model is to be defined later.

- **%R**: readyfd
  Number of file descriptor to use for signaling readiness of launched process.

- **%r**: rootdir
  Path of directory containing the widget and its data.

- **%S**: secret
  An hexadecimal number that can be used to initialize pairing of client and application binder.

- **%W**: width
  Requested width for the widget.

Defined by the attribute width of the element of config.xml.

### 3.6. The D-Bus interface

#### 3.6.1. Overview of the dbus interface

**afm-user-daemon** takes its orders from the session instance of D-Bus. D-Bus is nice to use in this context because it allows discovery and signaling.

The dbus session is by default addressed by environment variable **DBUS_SESSION_BUS_ADDRESS**. Using **systemd** variable **DBUS_SESSION_BUS_ADDRESS** is automatically set for user sessions.

The **afm-user-daemon** is listening on destination name **org.AGL.afm.user** at object path **/org/AGL/afm/user** on interface **org.AGL.afm.user** for following members: **runnables, detail, start, terminate, stop, continue, runners, state, install** and **uninstall**.

D-Bus is mainly used for signaling and discovery. Its optimized typed protocol is not used except for transmission of standalone strings.

Clients and Services are using JSON serialisation to exchange data.

The D-Bus interface is defined by:

- **DESTINATION**: **org.AGL.afm.user**
- **PATH**: **/org/AGL/afm/user**
- **INTERFACE**: **org.AGL.afm.user**

The signature of any member of the interface is **string -> string** for **JSON -> JSON**.

This is the normal case. In case of error, the current implementation returns a dbus
error as a string.

Here an example using `dbus-send` to query data on installed applications.

```bash
dbus-send --session --print-reply \
    --dest=org.AGL.afm.user \
    /org/AGL/afm/user \
    org.AGL.afm.user.runnables string:true
```

### 3.6.2. Using `afm-util`

The command line tool `afm-util` uses `dbus-send` to send orders to `afm-user-daemon`. This small scripts allows to send command to `afm-user-daemon` either interactively at shell prompt or scriptically.

The syntax is simple: it accept a command and when requires attached arguments.

Here is the summary of `afm-util`:

- **afm-util runnables**: list the runnable widgets installed
- **afm-util install wgt**: install the wgt file
- **afm-util uninstall id**: remove the installed widget of id
- **afm-util detail id**: print detail about the installed widget of id
- **afm-util runners**: list the running instance
- **afm-util start id**: start an instance of the widget of id
- **afm-util terminate rid**: terminate the running instance rid
- **afm-util stop rid**: stop the running instance rid
- **afm-util continue rid**: continue the previously rid
- **afm-util state rid**: get status of the running instance rid

Here is how to list applications using `afm-util`:

```bash
afm-util runnables
```
3.6.3. The protocol over D-Bus

Recall:

- **DESTINATION**: org.AGL.afm.user
- **PATH**: /org/AGL/afm/user
- **INTERFACE**: org.AGL.afm.user

**Method org.AGL.afm.user.detail**

**Description**: Get details about an application from its id.

**Input**: the id of the application as below.

Either just a string:

"appli@x.y"

Or an object having the field "id" of type string:

{"id":"appli@x.y"}

**Output**: A JSON object describing the application containing the fields described below.

```
{
  "id": string, the application id (id@version)
  "version": string, the version of the application
  "width": integer, requested width of the application
  "height": integer, requested height of the application
  "name": string, the name of the application
  "description": string, the description of the application
  "shortname": string, the short name of the application
  "author": string, the author of the application
}
```

**Method org.AGL.afm.user.runnables**

**Description**: Get the list of applications that can be run.

**Input**: any valid json entry, can be anything except null.

**output**: An array of description of the runnable applications. Each item of the array contains an object containing the detail of an application as described above for the method org.AGL.afm.user.detail.
Method `org.AGL.afm.user.install`

**Description:** Install an application from its widget file.

If an application of the same *id* and *version* exists, it is not reinstalled except when `force=true`.

Applications are installed in the subdirectories of the common directory reserved for applications. If `root` is specified, the application is installed under sub-directories of defined `root`.

Note that this methods is a simple accessor to the method `org.AGL.afm.system.install` of `afm-system-daemon`.

After the installation and before returning to the sender, `afm-user-daemon` sends the signal `org.AGL.afm.user.changed`.

**Input:** The *path* of widget file to be installed. Optionally, a flag to `force` reinstallation and/or a `root` directory.

Simple form a simple string containing the absolute widget path:

```
"/a/path/driving/to/the/widget"
```

Or an object:

```json
{
    "wgt": "/a/path/to/the/widget",
    "force": false,
    "root": "/a/path/to/the/root"
}
```

"wgt" and "root" MUST be absolute paths.

**output:** An object containing field "added" to use as application ID.

```json
{"added":"appli@x.y"}
```

Method `org.AGL.afm.user.uninstall`

**Description:** Uninstall an application from its id.

Note that this methods is a simple accessor to `org.AGL.afm.system.uninstall` method from `afm-system-daemon`.

After the uninstallation and before returning to the sender, `afm-user-daemon` sends the signal `org.AGL.afm.user.changed`.

**Input:** the *id* of the application and, optionally, the path to application `root`.

Either a string:

"appli@x.y"
Or an object:

```json
{
    "id": "appli@x.y",
    "root": "/a/path/to/the/root"
}
```

**output**: the value 'true'.

**Method org.AGL.afm.user.start**

**Description**:  
**Input**: the *id* of the application and, optionally, the start *mode* as below.  
Either just a string:

```
"appli@x.y"
```

Or an object containing field "id" of type string and optionally a field mode:

```json
{"id":"appli@x.y","mode":"local"}
```

The field "mode" is a string equal to either "local" or "remote".

**output**: The *runid* of the application launched. *runid* is an integer.

**Method org.AGL.afm.user.terminate**

**Description**: Terminates the application attached to *runid*.

**Input**: The *runid* (an integer) of running instance to terminate.

**output**: the value 'true'.

**Method org.AGL.afm.user.stop**

**Description**: Stops the application attached to *runid* until terminate or continue.

**Input**: The *runid* (integer) of the running instance to stop.

**output**: the value 'true'.

**Method org.AGL.afm.user.continue**

**Description**: Continues the application attached to *runid* previously stopped.

**Input**: The *runid* (integer) of the running instance to continue.

**output**: the value 'true'.

**Method `org.AGL.afm.user.state`**

**Description:** Get informations about a running instance of `runid`.

**Input:** The `runid` (integer) of the running instance inspected.

**output:** An object describing instance state. It contains: the runid (integer), the id of the running application (string), the state of the application (string either: "starting", "running", "stopped").

Example of returned state:

```json
{
  "runid": 2,
  "state": "running",
  "id": "appli@x.y"
}
```

**Method `org.AGL.afm.user.runners`**

**Description:** Get the list of currently running instances.

**Input:** anything.

**output:** An array of states, one per running instance, as returned by the method `org.AGL.afm.user.state`.
4. The widgets

4.1. The widgets

The widgets are described by the technical recommendations widgets and widgets-digsig.

In summary, widgets are ZIP files that can be signed and whose content is described by the file.

4.1.1. The configuration file config.xml

This is one of the important file of the widget. It fully describes the widget.

Here is the example of the config file for the QML application SmartHome.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://www.w3.org/ns/widgets" id="smarthome" version="0.1">
    <name>SmartHome</name>
    <icon src="smarthome.png"/>
    <content src="qml/smarthome/smarthome.qml" type="text/vnd.qt.qml"/>
    <description>This is the Smarthome QML demo application. It shows some user interfaces for controlling an automated house. The user interface is completely done with QML.</description>
    <author>Qt team</author>
    <license>GPL</license>
</widget>
```

The most important items are:

- `<widget id="......">`: gives the id of the widget. It must be unique.
- `<widget version="......">`: gives the version of the widget
- `<icon src="...">`: gives a path to the icon of the application (can be repeated with different sizes)
- `<content src="..." type="...">`: this indicates the entry point and its type. The types handled are set through the file /etc/afm/afm-launch.conf

Further development will add handling of for requiring and providing permissions and services.

4.1.2. Tools for managing widgets

This project includes tools for managing widgets. These tools are:

- `wgtpkg-info`: command line tool to display informations about a widget file.
- `wgtpkg-installer`: command line tool to install a widget file.
- **wgtpkg-pack**: command line tool to create a widget file from a widget directory.
- **wgtpkg-sign**: command line tool to add a signature to a widget directory.

For all these commands, a tiny help is available with options `-h` or `--help`.

There is no tool for unpacking a widget. For doing such operation, you can use the command `unzip`.

To list the files of a widget:

```
$ unzip -l WIDGET
```

To extract a widget in some directory:

```
$ unzip WIDGET -d DIRECTORY
```

*Note that DIRECTORY will be created if needed.*

### 4.1.3. Signing a widget

To sign a widget, you need a private key and its certificate.

The tool **wgtpkg-sign** creates or replace a signature file in the directory of the widget BEFORE its packaging.

There are two types of signature files: author and distributor.

Example 1: add an author signature

```
$ wgtpkg-sign -a -k me.key.pem -c me.cert.pem DIRECTORY
```

Example 2: add a distributor signature

```
$ wgtpkg-sign -k authority.key.pem -c authority.cert.pem DIRECTORY
```

### 4.1.4. Packing a widget

This operation can be done using the command `zip` but we provide the tool **wgtpkg-pack** that may add checking.

Example:

```
$ wgtpkg-pack DIRECTORY -o file.wgt
```

### 4.1.5. Getting data about a widget file

The command **wgtpkg-info** opens a widget file, reads its `config.xml` file and displays its content in a human readable way.
4.2. Writing a widget

4.2.1. What kind of application?
The file /etc/afm/afm-launch.conf explain how to launch applications. (It is the current state that use afm-user-daemon. In a future, it may be replace by systemd features.)

Currently the applications that can be run are:

- binary applications: their type is application/x-executable
- HTML5 applications: their type is text/html
- QML applications: their type is text/vnd.qt.qml

4.2.2. The steps for writing a widget

1. make your application
2. create its configuration file config.xml
3. sign it
4. pack it

Fairly easy, no?

4.3. Organization of directory of applications

4.3.1. directory where are stored applications

Applications can be installed in different places: the system itself, extension device. On a phone application are typically installed on the sd card.

This translates to:

- /usr/applications: system wide applications
- /opt/applications: removable applications

From here those paths are referenced as: "APPDIR".

The main path for applications is: APPDIR/PKGID/VER.

Where:

- APPDIR is as defined above
- PKGID is a directory whose name is the package identifier
- VER is the version of the package MAJOR.MINOR

This organization has the advantage to allow several versions to leave together. This is needed for some good reasons (rolling back) and also for less good reasons (user habits).
4.3.2. Identity of installed files
All files are installed as user "afm" and group "afm". All files have rw(x) for user and r-(x) for group and others.
This allows every user to read every file.

4.3.3. labeling the directories of applications
The data of a user are in its directory and are labelled by the security-manager using the labels of the application.
5. Writing the configuration file "config.xml"

5.1. About "config.xml"

The file `config.xml` describes important data of the application to the framework:

- the unique identifier of the application
- the name of the application
- the type of the application
- the icon of the application
- the permissions linked to the application
- the services and dependancies of the application

The file MUST be at the root of the widget and MUST be case sensitively named `config.xml`.

The file `config.xml` is a XML file described by the document `widgets`.

5.2. Example of "config.xml"

Here is the example of the config file for the QML application SmartHome.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://www.w3.org/ns/widgets" id="smarthome" version="0.1">
  <name>SmartHome</name>
  <icon src="smarthome.png"/>
  <content src="qml/smarthome/smarthome.qml" type="text/vnd.qt.qml"/>
  <description>
    This is the Smarthome QML demo application.
    It shows some user interfaces for controlling an automated house.
    The user interface is completely done with QML.
  </description>
  <author>Qt team</author>
  <license>GPL</license>
</widget>
```

5.3. Standard elements of "config.xml"

5.3.1. The element `widget`

**the attribute `id` of `widget`**

The attribute `id` is mandatory (for version 2.x, blowfish) and must be unique.

Values for `id` are any non empty string containing only latin letters, arabic digits, and the three characters `.` (dot), `-` (dash) and `_` (underscore).

Authors can use a mnemonic id or can pick a unique id using command `uuid` or
uuidgen.

**The attribute version of widget**
The attribute *version* is mandatory (for version 2.x, blowfish).

Values for *id* are any non empty string containing only latin letters, arabic digits, and the three characters ' ' (dot), '-' (dash) and '_' (underscore).

Version values are dot separated fields MAJOR.MINOR.REVISION.

**5.3.2. The element content**
The element *content* is mandatory (for version 2.x, blowfish) and must designate a file (subject to localisation) with its attribute *src*.

The content designed depends on its type. See below for the known types.

**5.3.3. The element icon**
The element *icon* is mandatory (for version 2.x, blowfish) and must be unique. It must designate an image file with its attribute *src*.

**5.4. Known widget types and content**
The configuration file `/etc/afm/afm-launch.conf` defines the types of widget known and how to launch it.

Known types for the type of content are (for version 2.x, blowfish):

- **text/html**: HTML application, *content.src* designates the home page of the application
- **application/x-executable**: Native application, *content.src* designates the relative path of the binary
- **application/vnd.agl.url**: Internet url, *content.src* designates the url to be used
- **application/vnd.agl.service**: AGL service defined as a binder, *content.src* designates the directory of provided binders, http content, if any, must be put in the subdirectory `htdocs` of the widget
- **application/vnd.agl.native**: Native application with AGL service defined as a binder, *content.src* designates the relative path of the binary, bindings, if any must be put in the subdirectory `lib` of the widget, http content, if any, must be put in the subdirectory `htdocs` of the widget
- **text/vnd.qt.qml, application/vnd.agl.qml**: QML application, *content.src* designate the relative path of the QML root, imports must be put in the subdirectory `imports` of the widget
- **application/vnd.agl.qml.hybrid**: QML application with bindings, *content.src* designate
designate the relative path of the QML root, bindings, if any must be put in the subdirectory \textit{lib} of the widget, imports must be put in the subdirectory \textit{imports} of the widget

- \textit{application/vnd.agl.html.hybrid}: HTML application, content.src designates the home page of the application, bindings, if any must be put in the subdirectory \textit{lib} of the widget, http content must be put in the subdirectory \textit{htdocs} of the widget

### 5.5. AGL features

The AGL framework uses the feature tag for specifying security and binding requirement of the widget.

The current version of AGL (up to 2.0.1, blowfish) has no fully implemented features.

The features planned to be implemented are described below.

#### 5.5.1. feature name="urn:AGL:required-binding"

List of the bindings required by the widget.

Each required binding must be explicited using a entry.

\texttt{\textit{param name=}[required binding name]}

The value is either:

- \textit{required}: the binding is mandatorily needed except if the feature isn't required (required="false") and in that case it is optional.
- \textit{optional}: the binding is optional

#### 5.5.2. feature name="urn:AGL:required-permission"

List of the permissions required by the widget.

Each required permission must be explicited using a entry.

\texttt{\textit{param name=}[required permission name]}

The value is either:

- \textit{required}: the permission is mandatorily needed except if the feature isn't required (required="false") and in that case it is optional.
- \textit{optional}: the permission is optional

#### 5.5.3. feature name="urn:AGL:provided-binding"

Use this feature for each provided binding of the widget. The parameters are:

\texttt{\textit{param name}="name"}

REQUIRED
The value is the string that must match the binding prefix. It must be unique.

**param name=\"src\"**

REQUIRED

The value is the path of the shared library for the binding.

**param name=\"type\"**

REQUIRED

Currently it must be `application/vnd.agl.binding.v1`.

**param name=\"scope\"**

REQUIRED

The value indicates the availability of the binding:

- private: used only by the widget
- public: available to allowed clients as a remote service (requires permission+)
- inline: available to allowed clients inside their binding (unsafe, requires permission+++)

**param name=\"needed-binding\"**

OPTIONAL

The value is a space-separated list of bindings' names that the binding needs.

5.5.4. **feature name=\"urn:AGL:defined-permission\"**

Each required permission must be explicitly using an entry.

**param name=[defined permission name]**

The value is the level of the defined permission. Standard levels are:

- system
- platform
- partner
- public

This level defines the level of accreditation required to get the given permission. The accreditations are given by signatures of widgets.
6. AGL Application Framework: A Quick Tutorial

6.1. Introduction

This document proposes a quick tutorial to demonstrate the major functionalities of the AGL Application Framework. For more complete information, please refer to the inline documentation available in the main git repository:

- https://gerrit.automotivelinux.org/gerrit/#/admin/projects/src/app-framework-main
- https://gerrit.automotivelinux.org/gerrit/#/admin/projects/src/app-framework-binder

For more information on AGL, please visit: https://www.automotivelinux.org/

6.2. Sample applications

4 sample applications (.wgt files) are prebuilt and available at the following address: https://github.com/iotbzh/afm-widget-examples

You can get them by cloning this git repository on your desktop (will be useful later in this tutorial):

```
$ git clone https://github.com/iotbzh/afm-widget-examples
```

6.3. Using the CLI tool

6.3.1. Setup Environment

Connect your AGL target board to the network and copy some sample widgets on it through SSH (set BOARDIP with your board IP address):

```
$ cd afm-widget-examples
$ BOARDIP=1.2.3.4
$ scp *.wgt root@$BOARDIP:~/
```

Connect through SSH on the target board and check for Application Framework daemons:

```
$ ssh root@$BOARDIP
root@porter:~# ps -ef|grep bin/afm
afm 409 1 0 13:00 ? 00:00:00 /usr/bin/afm-system-daemon
root 505 499 0 13:01 ? 00:00:00 /usr/bin/afm-user-daemon
root 596 550 0 13:22 pts/0 00:00:00 grep afm
```
We can see that there are two daemons running: *afm-system-daemon* runs with a system user 'afm' and is responsible for installing/uninstalling packages *afm-user-daemon* runs as a user daemon (currently as root because it's the only real user on the target board) and is responsible for the whole lifecycle of the applications running inside the user session.

The application framework has a tool running on the Command Line Interface (CLI). Using the *afm-util* command, you can install, uninstall, list, run, stop ... applications.

To begin, run *afm-util help* to get a quick help on commands:

```bash
root@porter:~# afm-util help
usage: afm-util command [arg]

The commands are:

- list runnables (list the runnable widgets installed)
- add wgt (install the wgt file)
- remove id (remove the installed widget of id)
- uninstall id (uninstall the widget of id)
- info id (print detail about the installed widget of id)
- detail id (print detail about the installed widget of id)
- ps runners (list the running instance)
- run id (start an instance of the widget of id)
- start id (start an instance of the widget of id)
- kill rid (terminate the running instance rid)
- terminate rid (terminate the running instance rid)
- stop rid (stop the running instance rid)
- pause rid (pause the running instance rid)
- resume rid (resume the previously rid)
- continue rid (continue the previously rid)
- status rid (get status of the running instance rid)
```

### 6.3.2. Install an application

You can then install your first application:

```bash
root@porter:~# afm-util install /home/root/annex.wgt
{ "added": "webapps-annex@0.0" }
```

Let's install a second application:

```bash
root@porter:~# afm-util install /home/root/memory-match.wgt
{ "added": "webapps-memory-match@1.1" }
```
Note that usually, **afm-util** will return a **JSON result**, which is the common format for messages returned by the Application Framework daemons.

### 6.3.3. List installed applications

You can then list all installed applications:

```
root@porter:~# afm-util list
[ { "id": "webapps-annex@0.0", "version": "0.0.10", "width": 0, "height": 0,
  "name": "Annex", "description": "Reversi/Othello", "shortname": ", "author":
  "Todd Brandt <todd.e.brandt@intel.com>" },
  { "id": "webapps-memory-match@1.1", "version": "1.1.7", "width": 0, "height": 0,
  "name": "MemoryMatch", "description": "Memory match", "shortname": ", "author":
  "Todd Brandt <todd.e.brandt@intel.com>" } ]
```

Here, we can see the two previously installed applications.

### 6.3.4. Get information about an application

Let's get some details about the first application:

```
root@porter:~# afm-util info webapps-annex@0.0
{ "id": "webapps-annex@0.0", "version": "0.0.10", "width": 0, "height": 0,
  "name": "Annex", "description": "Reversi/Othello", "shortname": ", "author":
  "Todd Brandt <todd.e.brandt@intel.com>" }
```

Note that AGL applications are mostly handled by afm-util through their IDs. In our example, the application ID is 'webapps-annex@0.0'.

### 6.3.5. Start application

Let's start the first application Annex:

```
root@porter:~# afm-util start webapps-annex@0.0
```

As the application is a HTML5 game, you should then get a webview running with QML on the board display.

### 6.3.6. Security Context

The application has been started in the user session, with a dedicated security context, enforced by SMACK. To illustrate this, we can take a look at the running processes and their respective SMACK labels:

```
root@porter:~# ps -efZ |grep webapps-annex | grep -v grep
User::App::webapps-annex        root       716   491  0 13:19 ?        00:00:00
/usr/bin/afb-daemon --mode=local --readyfd=8 --alias=/icons
/usr/share/afm/icons --port=12348
--rootdir=/usr/share/afm/applications/webapps-annex/0.0 --token=7D6D2F16
--sessiondir=/home/root/app-data/webapps-annex/.afb-daemon
User::App::webapps-annex        root       717   491  0 13:19 ?        00:00:00
/usr/bin/qt5/qmlscene http://localhost:12348/index.html?token=7D6D2F16
/usr/bin/web-runtime-webkit.qml
```
In the previous result, we see that the application is composed of two processes: * the application binder (afb-daemon) * the application UI (qmlscene ...)

While most system processes run with the label 'System', we see that the application runs with a specific SMACK label 'User::App::webapps-annex': this label is used to force the application to follow a Mandatory Access Control (MAC) scheme. This means that those processes run in their own security context, isolated from the rest of the system (and other applications). Global security rules can then be applied to restrict access to all other user or system resources.

### 6.3.7. Check running applications

To check for running applications, just run:

```bash
root@porter:~# afm-util ps
[ { "runid": 1, "state": "running", "id": "webapps-annex@0.0" } ]
```

The 'runid' is the application instance ID and is used as an argument for the subcommands controlling the application runtime state (kill/stop/resume/status).

### 6.3.8. Stop application

To stop the application that was just started (the one with RUNID 1), just run the stop command:

```bash
root@porter:~# afm-util terminate 1
true
```

The application is now stopped, as confirmed by a list of running apps:

```bash
root@porter:~# afm-util ps
[ ]
```

### 6.3.9. Uninstall application

To uninstall an application, simply use its ID:

```bash
root@porter:~# afm-util uninstall webapps-annex@0.0
true
```

Then list the installed apps to confirm the removal:

```bash
root@porter:~# afm-util list
[ { "id": "webapps-memory-match@1.1", "version": "1.1.7", "width": 0, "height": 0, "name": "MemoryMatch", "description": "Memory match", "shortname": ", "author": "Todd Brandt <todd.e.brandt@intel.com>" } ]
```

### 6.4. afm-client: a sample HTML5 'Homescreen'

**afm-client** is a HTML5 UI that allows to install/uninstall applications as well as starting/stopping them as already demonstrated with afm-util.
The HTML5 UI is accessible remotely through this URL: http://[board_ip]:1234/opa?token=132456789

6.4.1. Installing an application
By clicking on the 'Upload' button on the right, you can send an application package (WGT file) and install it. Select for example the file 'rabbit.wgt' that was cloned initially from the git repository afm-widget-examples.

Then a popup requester ask for a confirmation: 'Upload Application rabbit.wgt ?'. Click on the 'Install' button.

You should then see some changes in the toolbar: a new icon appeared, representing the freshly installed application.

6.4.2. Running an application
In the toolbar, click on the button representing the Rabbit application. You'll get a popup asking to: * start the application * or get some info about it * or uninstall it

Click on the 'start' item: the application starts and should be visible as a webview on the target board display. Note that at this point, we could also run the application remotely, that is in the same browser as the Homescreen app. By default, the application framework is configured to run applications 'locally' on the board display.

6.4.3. Stopping an application
In the Homescreen application, click again on the Rabbit application button, then select 'stop': the application then stops.

6.4.4. Uninstalling an application
From the same popup menu, you can select 'uninstall' to remove the application from the system. As a consequence, the application icon should disappear from the toolbar.

6.5. afb-client: a template for Angular Applications
Another package 'afb-client' is also available for testing. This is a sample HTML5 application demonstrating various basic capabilities of the Binder daemon. It can be used by developers as a template to start writing real AGL Applications.

This application is not available as WGT file yet and it should be started manually without any specific security context:

```
root@porter:~# /usr/bin/afb-daemon --mode=remote --port=1235 --token='' --sessiondir=/home/root/.afm-daemon --rootdir=/usr/share/agl/afb-client --alias=/icons:/usr/share/afm/icons
```

Then you can access it from a browser: http://[board_ip]:1235/opa/?token=132456789

afb-client is a simple application to demonstrate the built-in capabilities of the binder
daemon (handling sessions and security tokens, testing POSTs uploads...) and was used during the application framework development to validate the proposed features.
7. Overview of AFB-DAEMON

7.1. Roles of afb-daemon

The name afb-daemon stands for Application Framework Binder Daemon. That is why afb-daemon is also named the binder.

Afb-daemon is in charge to bind one instance of an application to the AGL framework and AGL system.

On the following figure, you can use a typical use of afb-daemon:

Figure: binder afb-daemon, basis

The application and its companion binder run in secured and isolated environment set for them. Applications are intended to access to AGL system through the binder.

The binder afb-daemon serves multiple purposes:

1. It acts as a gateway for the application to access the system;
2. It acts as an HTTP server for serving files to HTML5 applications;
3. It allows HTML5 applications to have native extensions subject to security enforcement for accessing hardware resources or for speeding parts of algorithm.
7.2. Use cases of the binder afb-daemon

This section tries to give a better understanding of the binder usage through several use cases.

7.2.1. Remotely running application

One of the most interesting aspect of using the binder afb-daemon is the ability to run applications remotely. This feature is possible because the binder afb-daemon implements native web protocols.

So the figure binder, basis would become when the application is run remotely:

Figure: binder afb-daemon and remotely running application

7.2.2. Adding native features to HTML5/QML applications

Applications can provide with their packaged delivery a binding. That binding will be instanciated for each application instance. The methods of the binding will be accessible by applications and will be executed within the security context.
7.2.3. Offering services to the system

It is possible to run the binder afb-daemon as a daemon that provides the API of its bindings.

This will be used for:

1. offering common APIs
2. provide application's services (services provided as application)

In that case, the figure showing the whole aspects is

Figure: binder afb-daemon for services

For this case, the binder afb-daemon takes care to attribute one single session context to each client instance. It allows bindings to store and retrieve data associated to each of its client.

7.3. The bindings of the binder afb-daemon

The binder can instanciate bindings. The primary use of bindings is to add native methods that can be accessed by applications written with any language through web technologies ala JSON RPC.
This simple idea is declined to serves multiple purposes:

1. add native feature to applications
2. add common API available by any applications
3. provide customers services

A specific document explains how to write an afb-daemon binder binding: [HOWTO WRITE a BINDING for AFB-DAEMON](#)

### 7.4. Launching the binder afb-daemon

The launch options for binder **afb-daemon** are:

```
--help
  Prints help with available options
--version
  Display version and copyright
--verbose
  Increases the verbosity, can be repeated
--port=xxxx
  HTTP listening TCP port [default 1234]
--rootdir=xxxx
  HTTP Root Directory [default $AFBDIR or else $HOME/.AFB]
--rootbase=xxxx
  Angular Base Root URL [default /opa]
  This is used for any application of kind OPA (one page application).
  When set, any missing document whose url has the form /opa/zzz
  is translated to /opa/#!zzz
--rootapi=xxxx
  HTML Root API URL [default /api]
  The bindings are available within that url.
--alias=xxxx
  Maps a path located anywhere in the file system to the
  a subdirectory. The syntax for mapping a PATH to the
  subdirectory NAME is: --alias=/NAME:PATH.
  Example: --alias=/icons:/usr/share/icons maps the
  content of /usr/share/icons within the subpath /icons.
```
This option can be repeated.

--apitimeout=xxxx

binding API timeout in seconds [default 20]

Defines how many seconds maximum a method is allowed to run. 0 means no limit.

--cntxttimeout=xxxx

Client Session Timeout in seconds [default 3600]

--cache-eol=xxxx

Client cache end of live [default 100000 that is 27,7 hours]

--sessiondir=xxxx

Sessions file path [default rootdir/sessions]

--session-max=xxxx

Maximum count of simultaneous sessions [default 10]

--ldpaths=xxxx

Load bindings from given paths separated by colons as for dir1:dir2:binding1.so:... [default = $libdir/afb]

You can mix path to directories and to bindings. The sub-directories of the given directories are searched recursively.

The bindings are the files terminated by '.so' (the extension so denotes shared object) that contain the public entry symbol.

--binding=xxxx

Load the binding of given path.

--token=xxxx

Initial Secret token to authenticate.

If not set, no client can authenticate.

If set to the empty string, then any initial token is accepted.

--mode=xxxx

Set the mode: either local, remote or global.

The mode indicate if the application is run locally on the host or remotely through network.

--readyfd=xxxx

Set the #fd to signal when ready
If set, the binder afb-daemon will write "READY=1\n" on the file descriptor whose number if given (/proc/self/fd/xxx).

--dbus-client=xxxx

Transparent binding to a binder afb-daemon service through dbus.

It creates an API of name xxxx that is implemented remotely and queried via DBUS.

--dbus-server=xxxx

Provides a binder afb-daemon service through dbus.

The name xxxx must be the name of an API defined by a binding. This API is exported through DBUS.

--foreground

Get all in foreground mode (default)

--daemon

Get all in background mode

7.5. Future development of afb-daemon

- The binder afb-daemon would launch the applications directly.
- The current setting of mode (local/remote/global) might be reworked to a mechanism for querying configuration variables.
- Implements "one-shot" initial token. It means that after its first authenticated use, the initial token is removed and no client can connect anymore.
- Creates some intrinsic APIs.
- Securisation of LOA.
- Integration of the protocol JSON-RPC for the websockets.
8. HOWTO WRITE an APPLICATION above AGL FRAMEWORK

8.1. Programmatic Languages for Applications

8.1.1. Writing an HTML5 application
Developers of HTML5 applications (client side) can easily create applications for AGL framework using their preferred HTML5 framework.

Developers may also take advantage of powerful server side plugins to improve application behavior. Server side plugins return an application/json mine-type and can be accessed through either HTTP or Websockets.

In a near future, JSON-RPC protocol should be added to complete current x-afb-json1 protocol.

Two examples of HTML5 applications are given:

- **afb-client**: a simple "hello world" application template
- **afm-client**: a simple "Home screen" application template

8.1.2. Writing a Qt application
Writing Qt applications is also supported. Qt offers standard API to send request through HTTP or WebSockets.

It is also possible to write QML applications. A sample QML application [token-websock] is available.

- **token-websock**: a simple "hello world" application in QML

8.1.3. Writing "C" application
C applications can use afb-daemon binder through a websocket connection.

The library **libafbwsc** is provided for C clients that need to connect with an afb-daemon binder.

The program **afb-client-demo** is the C example that uses **libafbwsc** library. Source code is available here [src/afb-client-demo.c].

Current implementation relies on libsystemd and file descriptors. This model might be reviewed in the future to support secure sockets and free the dependency with libsystemd.
8.2. Handling sessions within applications

Applications should understand sessions and tokens management when interacting with aflb-daemon binder.

Applications are communicating with their private binder(aflb-daemon) using a network connection or potentially any other connection channel. While current version does not yet implement unix domain this feature might be added in a near future. Developers need to be warn that HTTP protocol is a none connected protocol. This prevents from using HTTP socket connection to authenticate clients.

For this reason, the binder should authenticate the application by using a shared secret. The secret is named "token" and the identification of client is named "session".

The examples token-websock.qml and aflb-client are demonstrating how authentication and sessions are managed.

8.2.1. Handling sessions

Plugins and other binder feature need to keep track of client instances. This is especially important for plugins running as services as they may typically have to keep each client's data separated.

For HTML5 applications, the web runtime handles the cookie of session that the binder aflb-daemon automatically sets.

Session identifier can be set using the parameter uuid or x-aflb-uuid in URI requests. Within current version of the framework session UUID is supported by both HTTP requests and websocket negotiation.

8.2.2. Exchanging tokens

At application start, AGL framework communicates a shared secret to both binder and client application. This initial secret is called the "initial token".

For each of its client application, the binder manages a current active token for session management. This authentication token can be use to restrict access to some plugin's methods.

The token must be included in URI request on HTTP or during websockets connection using parameter token or x-aflb-token.

To ensure security, tokens must be refreshed periodically.

8.2.3. Example of session management

In following examples, we suppose that aflb-daemon is launched with something equivalent to:

```
$ aflb-daemon --port=1234 --token=123456 [...]
```

making the expectation that AuthLogin plugin is requested as default.
**Using curl**

First, connects with the initial token, 123456:

```
$ curl http://localhost:1234/api/auth/connect?token=123456
{
    "jtype": "afb-reply",
    "request": {
        "status": "success",
        "token": "0aef6841-2ddd-436d-b961-ae78da3b5c5f",
        "uuid": "850c4594-1be1-4e9b-9fcc-38cc3e6ff015"
    },
    "response": {"token": "A New Token and Session Context Was Created"}
}
```

It returns an answer containing session UUID, 850c4594-1be1-4e9b-9fcc-38cc3e6ff015, and a refreshed token, 850c4594-1be1-4e9b-9fcc-38cc3e6ff015.

Check if session and token is valid:

```
$ curl http://localhost:1234/api/auth/check?token=0aef6841-2ddd-436d-b961-ae78da3b5c5f\&uuid=850c4594-1be1-4e9b-9fcc-38cc3e6ff015
{
    "jtype": "afb-reply",
    "request": {
        "status": "success",
    },
    "response": {"isValid":true}
}
```

Refresh the token:

```
$ curl http://localhost:1234/api/auth/refresh?token=0aef6841-2ddd-436d-b961-ae78da3b5c5f\&uuid=850c4594-1be1-4e9b-9fcc-38cc3e6ff015
{
    "jtype": "afb-reply",
    "request": {
        "status": "success",
        "token": "b8ec3ec3-6ffe-448c-9a6c-efda69ad7bd9"
    },
    "response": {"token": "Token was refreshed"}
}
```

Close the session:

```
curl http://localhost:1234/api/auth/logout?token=b8ec3ec3-6ffe-448c-9a6c-efda69ad7bd9\&uuid=850c4594-1be1-4e9b-9fcc-38cc3e6ff015
{
    "jtype": "afb-reply",
    "request": {
        "status": "success",
    },
    "response": {"info": "Token and all resources are released"}
}
```

Checking on closed session for uuid should be refused:

```
curl http://localhost:1234/api/auth/check?token=b8ec3ec3-6ffe-448c-9a6c-efda69ad7bd9\&uuid=850c4594-1be1-4e9b-9fcc-38cc3e6ff015
{
    "jtype": "afb-reply",
    "request": {
```
Using \texttt{afb-client-demo}

The program is packaged within AGL in the rpm \texttt{libafbwsc-dev}

Here is an example of exchange using \texttt{afb-client-demo}:

```
$ afb-client-demo ws://localhost:1234/api?token=123456
auth connect
ON-REPLY 1:auth/connect: \{"jtype":"afb-reply","request":\{"status":"success","token":"63f71a29-8b52-4f9b-829f-b302b9a46b68","uuid":"5fcc3f3d-4b84-4fc7-ba66-2d8bd34ae7d1"\},
\"response\":\{"token\":\"A New Token and Session Context Was Created\"\}\
auth check
ON-REPLY 2:auth/check: \{\"jtype\":\"afb-reply\",\"request\":\{\"status\":\"success\"\},\"response\":\{"isvalid\":true\}\}
auth refresh
ON-REPLY 4:auth/refresh: \{\"jtype\":\"afb-reply\",\"request\":\{\"status\":\"success\",\"token\":\"8b8ba8f4-1b0c-48fa-962d-4a00a8c9157e\"\},\"response\":\{"token\":\"Token was refreshed\"\}\
auth check
ON-REPLY 5:auth/check: \{\"jtype\":\"afb-reply\",\"request\":\{\"status\":\"success\"\},\"response\":\{"isvalid\":true\}\}
auth refresh
ON-REPLY 6:auth/refresh: \{\"jtype\":\"afb-reply\",\"request\":\{\"status\":\"success\",\"token\":\"e83b36f8-d945-463d-b983-5d8ed73ba529\"\},\"response\":\{"token\":\"Token was refreshed\"\}\
```

After closing connection, reconnect as here after:

```
$ afb-client-demo ws://localhost:1234/api?token=e83b36f8-d945-463d-b983-5d8ed73ba529\&uuid=5fcc3f3d-4b84-4fc7-ba66-2d8bd34ae7d1 auth check
ON-REPLY 1:auth/check: \{\"jtype\":\"afb-reply\",\"request\":\{\"status\":\"success\"\},\"response\":\{"isvalid\":true\}\}
```

Same connection check using \texttt{curl}:

```
$ curl http://localhost:1234/api/auth/check?token=e83b36f8-d945-463d-b983-5d8ed73ba529\&uuid=5fcc3f3d-4b84-4fc7-ba66-2d8bd34ae7d1
\{"jtype\":\"afb-reply\",\"request\":\{\"status\":\"success\"\},\"response\":\{"isvalid\":true\}\}
```

\section*{8.3. Format of replies}

Replies use javascript object returned as serialized JSON.

This object contains at least 2 mandatory fields of name \texttt{jtype} and \texttt{request} and one optional field of name \texttt{response}.  


8.3.1. Template

This is a template of replies:

```json
{
    "jtype": "afb-reply",
    "request": {
        "status": "success",
        "info": "informational text",
        "token": "e83b36f8-d945-463d-b983-5d8ed73ba52",
        "uuid": "5fcc3f3d-4b84-4fc7-ba66-2d8bd34ae7d1",
        "reqid": "application-generated-id-23456"
    },
    "response": ....any response object....
}
```

8.3.2. Field jtype

The field `jtype` must have a value of type string equal to "afb-reply".

8.3.3. Field request

The field `request` must have a value of type object. This request object has at least one field named `status` and four optional fields named `info`, `token`, `uuid`, `reqid`.

**Subfield request.status**

`status` must have a value of type string. This string is equal to "success" only in case of success.

**Subfield request.info**

`info` is of type string and represent optional information added to the reply.

**Subfield request.token**

`token` is of type string. It is sent either at session creation or when the token is refreshed.

**Subfield request.uuid**

`uuid` is of type string. It is sent at session creation.

**Subfield request.reqid**

`reqid` is of type string. It is sent in response to HTTP requests that added a parameter of name `reqid` or `x-afb-reqid` at request time. Value returns in the reply has the exact same value as the one received in the request.

8.3.4. Field response

This field response optionally contains an object returned when request succeeded.
8.4. Format of events

Events are javascript object serialized as JSON.

This object contains at least 2 mandatory fields of name \texttt{jtype} and \texttt{event} and one optional field of name \texttt{data}.

8.4.1. Template

Here is a template of event:

```
{
  "jtype": "afb-event",
  "event": "sample_api_name/sample_event_name",
  "data": ...any event data...
}
```

8.4.2. Field \texttt{jtype}

The field \texttt{jtype} must have a value of type string equal to "\texttt{afb-event}".

8.4.3. Field \texttt{event}

The field \texttt{event} carries the event's name.

The name of the event is made of two parts separated by a slash: the name of the name of the API that generated the event and the name of event within the API.

8.4.4. Field \texttt{data}

This field data if present holds the data carried by the event.
9. Overview of bindings shipped with AFB-Daemon

9.1. List of bindings

Here are the bindings shipped in the source tree:

- Hello World
- Authentication
- Tic Tac Toe
- Audio (2 backends: ALSA/PulseAudio)
- Radio (1 backend: RTLSDR RTL2832U)
- Media (1 backend: Rygel UPnP)

All bindings may not be built, depending on the development libraries present on the system at build time.

9.2. Detail of bindings

9.2.1. Hello World

A sample Hello World binding for demonstration and learning purposes.

This binding provides a few unauthenticated requests, all beginning with "ping", to demonstrate basic binder capabilities.

**Verbs:**

- ping: returns a success response
- pingfail: returns a failure response
- pingnull: returns a success response, with an empty JSON response field
- pingbug: does a memory violation (intercepted by the binder)
- pingjson: returns a success response, with a complex JSON response field
- pingevent: broadcasts a global event

9.2.2. Authentication

An sample Authentication binding for demonstration purposes.

This binding provides a few requests to demonstrate the binder's token-based security mechanism.

Calling "connect" with a security token will initiate a session, calling "refresh" will issue a new token and invalidate the previous one, calling "logout" will invalidate all tokens and close the session.

**Verbs:**
• **ping:** returns a success response  
• **connect:** creates a session and returns a new token  
• **refresh:** returns a new token  
• **check:** verifies the passed token is valid  
• **logout:** closes the session

### 9.2.3. Tic Tac Toe

A sample Tic Tac Toe game binding.

This binding provides an interactive Tic Tac Toe game where the binder returns the grid as a JSON response.

**Verbs:**

• **new:** starts a new game  
• **play:** asks the server to play  
• **move:** gives a client move  
• **board:** gets the current board state, as a JSON structure  
• **level:** sets the server level  
• **join:** joins an existing board  
• **undo:** undo the last move  
• **wait:** wait for a move

### 9.2.4. Audio

A sample Audio binding with 2 backends:

• ALSA (mandatory)  
• PulseAudio (optional)

This binding is able to initialize a specific soundcard, define volume levels, channels (mono/stereo...), mute sound, and play a 22,050 Hz PCM stream.

**Verbs:**

• **ping:** returns a success response  
• **init:** initializes backend, on the "default" sound card  
• **volume:** gets or sets volume, in % (0-100)  
• **channels:** gets or sets channels count (1-8)  
• **mute:** gets or sets the mute status (on-off)  
• **play:** gets or sets the playing status (on-off)

*(if PulseAudio development libraries are not found at build time, only ALSA will be available)*

*(if a PulseAudio server is not found at runtime, the binding will dynamically fall back to ALSA)*

*(a specific backend can be forced by using this syntax before running afb-daemon: $ export AFB_AUDIO_OUTPUT=Alsa)*
9.2.5. Radio

A sample AM/FM Radio binding with 1 backend:

- RTLSDR - Realtek RTL2832U dongles (mandatory)

This binding is able to initialize specific RTL2832U dongles, switch between AM/FM modes, define frequency, mute sound, and play sound (if combining with the audio binding).

Verbs:

- ping: returns a success response
- init: initializes backend, looking for plugged-in devices
- power: sets device power status (on-off)
- mode: sets device reception mode (AM-FM)
- freq: sets device frequency (in Hz)
- mute: sets device mute status (on-off)
- play: sets device playing status (on-off)

(if rtlsdr development libraries are not found at build time, this binding will not be built)

9.2.6. Media

A sample Media Server binding with 1 backend:

- Rygel

This binding is able to detect a local Rygel UPnP media server, list audio files, select an audio file for playback, play/pause/seek in this file, upload an audio file to the server.

Verbs:

- ping: returns a success response
- init: initializes backend, looking for an active local UPnP server
- list: returns list of audio files, as a JSON structure
- select: select an audio files, by index number (001-...)
- play: plays the currently selected audio file
- stop: stops the currently selected audio file
- pause: pauses the currently selected audio file
- seek: seeks in the currently selected audio file, in seconds
- upload: uploads an audio file, with a POST request

(if GUPnP/GSSDP development libraries are not fund at build time, this binding will not be built)

Sample command-line applications: afb-client-demo (built by default)

Sample HTML5 applications: **test/*.html, afb-client, afb-radio**

Sample Qt/QML applications: test/token-websock.qml
10. HOWTO WRITE a BINDING for AFB-DAEMON

10.1. Summary

Afb-daemon binders serve files through HTTP protocol and offers to developers the capability to expose application API methods through HTTP or WebSocket protocol.

Binder bindings are used to add API to afb-daemon. This part describes how to write a binding for afb-daemon.

Excepting this summary, this document target developers.

Before moving further through an example, here after a short overview of binder bindings fundamentals.

10.1.1. Nature of a binding

A binding is an independent piece of software. A binding is self contain and exposes application logic as sharable library. A binding is intended to be dynamically loaded by afb-daemon to expose application API.

Technically, a binder binding does not reference and is not linked with any afb-daemon library.

10.1.2. Class of bindings

Application binder supports two kinds of bindings: application bindings and service bindings. Technically both class of binding are equivalent are use the same coding convention. Only sharing mode and security context diverge.

Application-bindings

Application-bindings implements the glue in between application's UI and services. Every AGL application has a corresponding binder that typically activates one or many bindings to interface the application logic with lower platform services. When an application is started by the AGL application framework, a dedicate binder is started that loads/activates application binding(s). API expose by application-binding are executed within corresponding application security context.

Application bindings generally handle a unique context for a unique client. As the application framework start a dedicated instance of afb_daemon for each AGL application, if a given binding is used within multiple application each of those application get a new and private instance of eventually "shared" binding.
**Service-bindings**

Service-bindings enable API activation within corresponding service security context and not within calling application context. Service-bindings are intended to run as a unique instance. Service-bindings can be shared in between multiple clients.

Service-bindings can either be stateless or manage client context. When managing context each client get a private context.

Sharing may either be global to the platform (ie: GPS service) or dedicated to a given user (ie: user preferences)

10.1.3. Live cycle of bindings within afb-daemon

Application and service bindings are loaded and activated each time a new afb-daemon is started.

At launch time, every loaded binding initialise itself. If a single binding initialisation fail corresponding instance of afb-daemon self aborts.

Conversely, when a binding initialisation succeeds, it should register its unique name as well as the list of verbs attached to the methods it exposes.

When initialised, on request from application clients to the right API/verb, binding methods are activated by the afb-daemon attached to the application or service.

At exit time, no special action is enforced by afb-daemon. When a specific actions is required at afb-daemon stop, developers should use 'atexit/on_exit' during binding initialisation sequence to register a custom exit function.

10.1.4. Binding Contend

Afb-daemon's binding register two classes of objects: names and functions.

Bindings declare categories of names: - A unique binding name to access all API expose by this binding, - One name for each methods/verbs provided by this binding.

Bindings declare two categories of functions: - function use for the initialisation - functions implementing exposed API methods

Afb-daemon parses URI requests to extract the API(binding name) and the VERB(method to activate). As an example, URI `foo/bar` translates to binding named `foo` and method named `bar`. To serve such a request, afb-daemon looks for an active binding named `foo` and then within this binding for a method named `bar`. When find afb-daemon calls corresponding method with attached parameter if any.

Afb-daemon ignores letter case when parsing URI. Thus `TicTacToe/Board` and `tictactoe/board` are equivalent.

**The name of the binding**

The name of a given binding is also known as the name of the API prefix that defines
the binding.

The name of a binding SHOULD be unique within a given afb-daemon instance.

For example, when a client of afb-daemon calls a URI named `foo/bar`. Afb-daemon extracts the prefix `foo` and the suffix `bar`. `foo` must match a binding name and `bar` a VERB attached to some method.

**Names of methods**

Each binding exposes a set of methods that can be called by the clients of a given afb-daemon.

VERB's name attached to a given binding (API) MUST be unique within a binding.

Bindings static declaration link VERBS to corresponding methods. When clients emit requests on a given API/VERB corresponding method is called by afb-daemon.

**Initialisation function**

Binding's initialisation function serves several purposes.

1. It allows afb-daemon to control binding version depending on initialisation function name. As today, the only supported initialisation function is `afbBindingV1Register`. This identifies version "one" of bindings.
2. It allows bindings to initialise itself.
3. It enables names declarations: descriptions, requirements and implementations of exposed API/VERB.

**Functions instantiation of API/VERBs**

When an API/VERB is called, afb-daemon constructs a request object. Then it passes this request object to the implementation function corresponding to requested method, this within attached API binding.

An implementation function receives a request object that is used to: get arguments of the request, send answer, store session data.

A binding MUST set an answer to every received requests.

Nevertheless it is not mandatory to set the answer before returning from API/VERB implementing function. This behaviour is important for asynchronous actions.

API/VERB implementation that set an answer before returning are called *synchronous implementations*. Those that do not systematically set an answer before returning are called *asynchronous implementations*.

Asynchronous implementations typically launch asynchronous actions. They record some context at request time and provide answer to the request only at completion of asynchronous actions.
10.2. The Tic-Tac-Toe example

This part explains how to write an afb-binding. For the sake of being practical it uses many examples based on tic-tac-toe. This binding example is in bindings/samples/tic-tac-toe.c.

This binding is named tictactoe.

10.3. Dependencies when compiling

Afb-daemon provides a configuration file for pkg-config. Typing the command

```bash
pkg-config --cflags afb-daemon
```

Print flags use for compilation:

```
$ pkg-config --cflags afb-daemon
-I/opt/local/include -I/usr/include/json-c
```

For linking, you should use

```
$ pkg-config --libs afb-daemon
-ljson-c
```

Afb-daemon automatically includes dependency to json-c. This is activated through Requires keyword in pkg-config. While almost every binding replies on json-c this is not a must have dependency.

Internally, afb-daemon relies on libsystemd for its event loop, as well as for its binding to D-Bus. Bindings developers are encouraged to leverage libsystemd when possible. Nevertheless there is no hard dependency to libsystemd if ever you rather not use it, feel free to do so.

Afb-daemon binding are fully self contain. They do not enforce dependency on any libraries from the application framework. Afb-daemon dependencies requerier to run AGL bindings are given at runtime through pointers leveraging read-only memory feature.

10.4. Header files to include

Binding tictactoe has following includes:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <json-c/json.h>
#include <afb/afb-binding.h>
```

Header afb/afb-binding.h is the only hard dependency, it includes all features that a binding MUST HAVE. Outside of includes used to support application logic, common external headers used within bindings are:
• `json-c/json.h`: should be include to handle json objects;
• `systemd/sd-event.h`: should be include to access event main loop;
• `systemd/sd-bus.h`: should be include for dbus connections.

The `tictactoe` binding does not leverage systemd features, also only `json.h` is used on top of mandatory `afb/afb-binding.h`.

When including `afb/afb-binding.h`, the macro `**_GNU_SOURCE**` MUST be defined.

### 10.5. Choosing names

Designers of bindings should define a unique name for every API binding as well as for methods VERBs. They should also define names for request arguments passed as name/value pair in URI.

While forging names, designers should respect few rules to ensure that created names are valid and easy to use across platforms.

All names and strings are UTF-8 encoded.

#### 10.5.1. Names for API (binding)

Binding API name are checked. All characters are authorised except:

- the control characters (000 .. 01f)
- the characters of the set `{ ' ' , '"', '#', '%', '&', ''', '/', '?', '`', '7f' }`

In other words the set of forbidden characters is `{ 000..020, 022, 023, 025..027, 02f, 03f, 060, 07f }`.

Afb-daemon makes no distinction between lower case and upper case when searching for API/VERB.

#### 10.5.2. Names for methods

The names of methods VERBs are totally free and not checked.

However, the validity rules for method's VERB name are the same as for Binding API name except that the dot(.) character is forbidden.

Afb-daemon makes no case distinction when searching for an API by name.

#### 10.5.3. Names for arguments

Argument's name are not restricted and can be everything you wish.

Warning arguments search is case sensitive and "index" and "Index" are not two different arguments.

#### 10.5.4. Forging names widely available

The key names of javascript object can be almost anything using the arrayed notation:
Nevertheless this is not the case with javascript dot notation:

```javascript
object.key = value
```

Using the dot notation, the key must be a valid javascript identifier and dash(-) as well as few other reserved characters cannot be used.

For this reason, we advise developer to chose name compatible with both javascript and HTML notation.

It is a good practice, even for arguments not to rely on case sensitivity. This may reduce headache strength at debug time, especially with interpreted language like javascript that may not warn you that a variable was not defined.

### 10.6. Writing a synchronous method implementation

The method `tictactoe/board` is a synchronous implementation. Here is its listing:

```c
static void board(struct afb_req req)
{
    struct board *board;
    struct json_object *description;

    /* retrieves the context for the session */
    board = board_of_req(req);
    INFO(afbitf, "method 'board' called for boardid %d", board->id);

    /* describe the board */
    description = describe(board);

    /* send the board's description */
    afb_req_success(req, description, NULL);
}
```

This example shows many aspects of a synchronous method implementation. Let summarise it:

1. The function `board_of_req` retrieves the context stored for the binding: the board.
2. The macro `INFO` sends a message of kind `INFO` to the logging system. The global variable named `afbitf` used represents the interface to afb-daemon.
3. The function `describe` creates a `json_object` representing the board.
4. The function `afb_req_success` sends the reply, attaching to it the object `description`. 
10.6.1. The incoming request

For any implementation, the request is received by a structure of type `struct afb_req`.

Note that this is a PLAIN structure, not a pointer to a structure.

The definition of `struct afb_req` is:

```c
/*
 * Describes the request by bindings from afb-daemon
 */
struct afb_req {
    const struct afb_req_itf *itf; /* the interfacing functions */
    void *closure; /* the closure for functions */
};
```

It contains two pointers: first one `itf`, points to functions used to handle internal request. Second one `closure` point onto function closure.

The structure must never be used directly. Instead developer should use the intended functions provided by afb-daemon as described here after.

`req` is used to get arguments of the request, to send answer, to store session data.

This object and its interface is defined and documented in the file names `afb/afb-req-itf.h`

The above example uses twice `req` object request.

The first time, to retrieve the board attached to the session of the request.

The second time, to send the reply: an object that describes the current board.

10.6.2. Associating a client context to a session

When `tic-tac-toe` binding receives a request, it musts get the board describing the game associated to the session.

For a binding, having data associated to a session is common. This data is called "binding context" for the session. Within `tic-tac-toe` binding the context is the board.

Requests `afb_req` offer four functions for storing and retrieving session associated context.

These functions are:

- **afb_req_context_get**: retrieves context data stored for current binding.
- **afb_req_context_set**: store context data of current binding.
- **afb_req_context**: if exist retrieves context data of current binding. if context does not yet exist, creates a new context and store it.
- **afb_req_context_clear**: reset the stored context data.

The binding `tictactoe` use a convenient function to retrieve its context: the board. This
function is `board_of_req`:

```c
/*
 * retrieves the board of the request
 */
static inline struct board *board_of_req(struct afb_req req)
{
    return afb_req_context(req, (void*)get_new_board, (void*)release_board);
}
```

The function `afb_req_context` ensures an existing context for the session of the request. Its two last arguments are functions to allocate and free context. Note function type casts to avoid compilation warnings.

Here is the definition of the function `afb_req_context`:

```c
/*
 * Gets the pointer stored by the binding for the session of 'req'.
 * If the stored pointer is NULL, indicating that no pointer was
 * already stored, afb_req_context creates a new context by calling
 * the function 'create_context' and stores it with the freeing function
 * 'free_context'.
 */
static inline void *afb_req_context(struct afb_req req, void (*create_context) (), void (*free_context)(void*))
{
    void *result = afb_req_context_get(req);
    if (result == NULL) {
        result = create_context();
        afb_req_context_set(req, result, free_context);
    }
    return result;
}
```

The second argument if the function that creates the context. For binding `tic-tac-toe` (function `get_new_board`). The function `get_new_board` creates a new board and set usage its count to 1. The boards are checking usage count to free resources when not used.

The third argument is a function that frees context resources. For binding `tic-tac-toe` (function `release_board`). The function `release_board` decrease usage count of the board passed in argument. When usage count falls to zero, data board are freed.

Definition of other functions dealing with contexts:

```c
/*
 * Gets the pointer stored by the binding for the session of 'req'.
 * When the binding has not yet recorded a pointer, NULL is returned.
 */
void *afb_req_context_get(struct afb_req req);

/*
 * Stores for the binding the pointer 'context' to the session of 'req'.
 * The function 'free_context' will be called when the session is closed
 * or if binding stores an other pointer.
 */
```
### 10.6.3. Sending reply to a request

Two kinds of replies: successful or failure.

Sending a reply to a request MUST be done once and only once.

It exists two functions for "success" replies: `afb_req_success` and `afb_req_success_f`.

```c
/* Sends a reply of kind success to the request 'req'. */
void afb_req_success(struct afb_req req, struct json_object *obj, const char *info);

/* Same as 'afb_req_success' but the 'info' is a formatting string followed by arguments. */
void afb_req_success_f(struct afb_req req, struct json_object *obj, const char *info, ...);
```

It exists two functions for "failure" replies: `afb_req_fail` and `afb_req_fail_f`.

```c
/* Sends a reply of kind failure to the request 'req'. */
void afb_req_fail(struct afb_req req, const char *status, const char *info);

/* Same as 'afb_req_fail' but the 'info' is a formatting string followed by arguments. */
void afb_req_fail_f(struct afb_req req, const char *status, const char *info, ...);
```

Note that calling `afb_req_fail("success", info)` is equivalent to call `afb_req_success(NULL, info)`. Thus even if possible it is strongly recommended to NEVER use "success" for status.
For convenience, these functions automatically call `json_object_put` to release `obj`. Because `obj` usage count is null after being passed to a reply function, it SHOULD not be used anymore. If exceptionally `obj` needs to remain usable after reply function then using `json_object_get` on `obj` to increase usage count and cancels the effect the `json_object_put` is possible.

### 10.7. Getting argument of invocation

Many methods expect arguments. Afb-daemon's bindings retrieve arguments by name and not by position.

Arguments are passed by requests through either HTTP or WebSockets.

For example, the method `join` of binding `tic-tac-toe` expects one argument: the `boardid` to join. Here is an extract:

```c
/*
 * Join a board
 */
static void join(struct afb_req req)
{
    struct board *board, *new_board;
    const char *id;

    /* retrieves the context for the session */
    board = board_of_req(req);
    INFO(afbitf, "method 'join' called for boardid %d", board->id);

    /* retrieves the argument */
    id = afb_req_value(req, "boardid");
    if (id == NULL)
        goto bad_request;

    ...
}
```

The function `afb_req_value` searches in the request `req` for argument name passed in the second argument. When argument name is not passed, `afb_req_value` returns
The search is case sensitive and *boardid* is not equivalent to *BoardId*. Nevertheless having argument names that only differ by name case is not a good idea.

### 10.7.1. Basic functions for querying arguments

The function **afb_req_value** is defined here after:

```c
static inline const char *afb_req_value(struct afb_req req, const char *name) {
    return afb_req_get(req, name).value;
}
```

It is defined as a shortcut to call the function **afb_req_get**. That function is defined here after:

```c
struct afb_arg afb_req_get(struct afb_req req, const char *name);
```

That function takes 2 parameters: the request and the name of the argument to retrieve. It returns a PLAIN structure of type **struct afb_arg**.

There is a special name that is defined when the request is of type HTTP/POST with a Content-Type being application/json. This name is "" (the empty string). In that case, the argument of name "" receives the value of the body of the HTTP request.

The definition of **struct afb_arg** is:

```c
struct afb_arg {
    const char *name;    /* name of the argument or NULL if invalid */
};
```
The structure returns the data arguments that are known for the request. This data include a field named `path`. This `path` can be accessed using the function `afb_req_path` defined here after:

```c
/*
 * Gets from the request 'req' the path for file attached to the argument of
 * 'name'.
 * Returns NULL if when there is no argument of 'name' or when there is no
 * file,
 * Returns the path of the argument of 'name' otherwise.
 * 
 * Shortcut for: afb_req_get(req, name).path
 */
static inline const char *afb_req_path(struct afb_req req, const char *name)
{
    return afb_req_get(req, name).path;
}
```

The path is only defined for HTTP/POST requests that send file.

### 10.7.2. Arguments for received files

As it is explained above, clients can send files using HTTP/POST requests.

Received files are attached to "file" argument name. For example, the following HTTP fragment (from test/sample-post.html) will send an HTTP/POST request to the method `post/upload-image` with 2 arguments named `file` and `hidden`.

```html
<h2>Sample Post File</h2>
<form enctype="multipart/form-data">
    <input type="file" name="file" />
    <input type="hidden" name="hidden" value="bollobollo" />
    <br>
    <button formmethod="POST" formaction="api/post/upload-image">Post File</button>
</form>
```

Argument named `file` should have both its value and path defined.

The value is the name of the file as it was set by the HTTP client. Generally it is the filename on client side.

The path is the effective path of saved file on the temporary local storage area of the application. This is a randomly generated and unique filename. It is not linked with the original filename as used on client side.

After success the binding can use the uploaded file directly from local storage path.
with no restriction: read, write, remove, copy, rename... Nevertheless when request reply is set and query terminated, the uploaded temporary file at path is destroyed.

10.7.3. Arguments as a JSON object

Bindings may also request every arguments of a given call as one single object. This feature is provided by the function `afb_req_json` defined here after:

```c
/*
 * Gets from the request 'req' the json object hashing the arguments.
 * The returned object must not be released using 'json_object_put'.
 */
struct json_object *afb_req_json(struct afb_req req);
```

It returns a json object. This object depends on how the request was built:

- For HTTP requests, this json object uses key names mapped on argument name. Values are either string for common arguments or object ie: `{ "file": "...", "path": "..." }`
- For WebSockets requests, returned directly the object as provided by the client.

In fact, for Websockets requests, the function `afb_req_value` can be seen as a shortcut to:

```c
json_object_get_string(json_object_object_get(afb_req_json(req), name))
```

10.8. Initialisation of the binding and declaration of methods

To be active, binding's methods should be declared to afb-daemon. Furthermore, the binding itself must be recorded.

The registration mechanism is very basic: when afb-need starts, it loads all bindings listed in: command line or configuration file.

Loading a binding follows the following steps:

1. Afb-daemon loads the binding with `dlopen`.
2. Afb-daemon searches for a symbol named `afbBindingV1Register` using `dlsym`. This symbol is assumed to be the exported initialisation function of the binding.
3. Afb-daemon builds an interface object for the binding.
4. Afb-daemon calls the found function `afbBindingV1Register` with interface pointer as parameter.
5. Function `afbBindingV1Register` setups the binding and initialises it.
6. Function `afbBindingV1Register` returns the pointer to a structure describing the binding: version, name (prefix or API name), and list of methods.
7. Afb-daemon checks that the returned version and name can be managed. If so, binding and its methods are register to become usable as soon as afb-daemon initialisation is finished.

Here after the code used for `afbBindingV1Register` from binding *tic-tac-toe*:

```c
/*
* activation function for registering the binding called by afb-daemon
*/
const struct afb_binding *afbBindingV1Register(const struct afb_binding_interface *itf)
{
    afbitf = itf;         // records the interface for accessing afb-daemon
    return &binding_description;  // returns the description of the binding
}
```

It is a very minimal initialisation function because *tic-tac-toe* binding doesn't have any application related initialisation step. It merely record daemon's interface and returns its description.

The variable `afbitf` is a binding global variable. It keeps the interface to afb-daemon that should be used for logging and pushing events. Here is its declaration:

```c
/*
* the interface to afb-daemon
*/
const struct afb_binding_interface *afbitf;
```

The description of the binding is defined here after.

```c
/*
* array of the methods exported to afb-daemon
*/
static const struct afb_verb_desc_v1 binding_methods[] = {
    /* VERB'S NAME     SESSION MANAGEMENT          FUNCTION TO CALL  SHORT DESCRIPTION */
    { .name= "new",   .session= AFB_SESSION_NONE, .callback= new,   .info= "Starts a new game" },
    { .name= "play",  .session= AFB_SESSION_NONE, .callback= play,  .info= "Asks the server to play" },
    { .name= "move",  .session= AFB_SESSION_NONE, .callback= move,  .info= "Tells the client move" },
    { .name= "board", .session= AFB_SESSION_NONE, .callback= board, .info= "Get the current board" },
    { .name= "level", .session= AFB_SESSION_NONE, .callback= level, .info= "Set the server level" },
    { .name= "join",  .session= AFB_SESSION_CHECK,.callback= join,  .info= "Join a board" },
    { .name= "undo",  .session= AFB_SESSION_NONE, .callback= undo,  .info= "Undo the last move" },
    { .name= "wait",  .session= AFB_SESSION_NONE, .callback= wait,  .info= "Wait for a change" },
    { .name= NULL }, /* marker for end of the array */
};
```

/*
* description of the binding for afb-daemon
*/
The structure `binding_description` describes the binding. It declares the type and version of the binding, its name, a short description and its methods list.

The list of methods is an array of structures describing the methods and terminated by a NULL marker.

In version one of afb-damon binding, a method description contains 4 fields:

- the name of the method,
- the session management flags,
- the implementation function to be call for the method,
- a short description.

The structure describing methods is defined as follows:

```c
/*
 * Description of one method of the API provided by the binding
 * This enumeration is valid for bindings of type 1
 */
struct afb_verb_desc_v1
{
    const char *name;            /* name of the method */
    enum AFB_session_v1 session; /* authorisation and session requirements of the method */
    void (*callback)(struct afb_req req); /* callback function implementing the method */
    const char *info;            /* textual description of the method */
};
```

For technical reasons, the enumeration `enum AFB_session_v1` is not exactly an enumeration but the wrapper of constant definitions that can be mixed using bitwise or (the C operator `|`).

The constants that can bit mixed are:

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFB_SESSION_CREATE</td>
<td>Equals to AFB_SESSION_LOA_EQ_0</td>
</tr>
<tr>
<td>AFB_SESSION_CLOSE</td>
<td>Closes the session after the reply and set</td>
</tr>
</tbody>
</table>
## 10.9. Sending messages to the log system

Afb-daemon provides 4 levels of verbosity and 5 methods for logging messages.

The verbosity is managed. Options allow the change the verbosity of afb-daemon and the verbosity of the bindings can be set binding by binding.

The methods for logging messages are defined as macros that test the verbosity level and that call the real logging function only if the message must be output. This avoid evaluation of arguments of the formatting messages if the message must not be output.

### 10.9.1. Verbs for logging messages

The 5 logging methods are:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Verbosity</th>
<th>Meaning</th>
<th>syslog level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>0</td>
<td>Error conditions</td>
<td>3</td>
</tr>
<tr>
<td>WARNING</td>
<td>1</td>
<td>Warning conditions</td>
<td>4</td>
</tr>
<tr>
<td>Macro</td>
<td>Verbosity</td>
<td>Meaning</td>
<td>syslog level</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>--------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>NOTICE</td>
<td>1</td>
<td>Normal but significant condition</td>
<td>5</td>
</tr>
<tr>
<td>INFO</td>
<td>2</td>
<td>Informational</td>
<td>6</td>
</tr>
<tr>
<td>DEBUG</td>
<td>3</td>
<td>Debug-level messages</td>
<td>7</td>
</tr>
</tbody>
</table>

You can note that the 2 methods WARNING and INFO have the same level of verbosity. But they don’t have the same syslog level. It means that they are output with a different level on the logging system.

All of these methods have the same signature:

```c
void ERROR(const struct afb_binding_interface *afbitf, const char *message, ...);
```

The first argument afbitf is the interface to afb daemon that the binding received at initialisation time when afbBindingV1Register is called.

The second argument message is a formatting string compatible with printf/sprintf.

The remaining arguments are arguments of the formatting message like with printf.

### 10.9.2. Managing verbosity

Depending on the level of verbosity, the messages are output or not. The following table explains what messages will be output depending on the verbosity level.

<table>
<thead>
<tr>
<th>Level of verbosity</th>
<th>Outputed macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ERROR</td>
</tr>
<tr>
<td>1</td>
<td>ERROR + WARNING + NOTICE</td>
</tr>
<tr>
<td>2</td>
<td>ERROR + WARNING + NOTICE + INFO</td>
</tr>
<tr>
<td>3</td>
<td>ERROR + WARNING + NOTICE + INFO + DEBUG</td>
</tr>
</tbody>
</table>

### 10.9.3. Output format and destination

The syslog level is used for forging a prefix to the message. The prefixes are:

<table>
<thead>
<tr>
<th>syslog level</th>
<th>prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;0&gt; EMERGENCY</td>
</tr>
<tr>
<td>1</td>
<td>&lt;1&gt; ALERT</td>
</tr>
<tr>
<td>2</td>
<td>&lt;2&gt; CRITICAL</td>
</tr>
<tr>
<td>3</td>
<td>&lt;3&gt; ERROR</td>
</tr>
<tr>
<td>4</td>
<td>&lt;4&gt; WARNING</td>
</tr>
<tr>
<td>5</td>
<td>&lt;5&gt; NOTICE</td>
</tr>
<tr>
<td>6</td>
<td>&lt;6&gt; INFO</td>
</tr>
<tr>
<td>7</td>
<td>&lt;7&gt; DEBUG</td>
</tr>
</tbody>
</table>

The message is pushed to standard error. The final destination of the message depends on how systemd service was configured through its variable `StandardError`. It can be journal, syslog or kmsg. (See man sd-daemon).
10.10. Sending events

Since version 0.5, bindings can broadcast events to any potential listener. As today only unattended even are supported. Targeted events are expected for next coming version.

The binding tic-tac-toe broadcasts events when the board changes. This is done in the function changed:

```c
/*
 * signals a change of the board
 */
static void changed(struct board *board, const char *reason)
{
    struct json_object *description;
    /* get the description */
    description = describe(board);
    ...
    afb_daemon_broadcast_event(afbitf->daemon, reason, description);
}
```

The description of the changed board is pushed via the daemon interface.

Within binding tic-tac-toe, reason indicates the origin of the change. In function afb_daemon_broadcast_event the second parameter is the name of broadcasted event. The third argument is the object that is transmitted with the event.

Function afb_daemon_broadcast_event is defined here after:

```c
/*
 * Broadcasts widely the event of 'name' with the data 'object'.
 * 'object' can be NULL.
 * 'daemon' MUST be the daemon given in interface when activating the binding.
 * For convenience, the function calls 'json_object_put' for 'object'.
 * Thus, in the case where 'object' should remain available after
 * the function returns, the function 'json_object_get' shall be used.
 */
void afb_daemon_broadcast_event(struct afb_daemon daemon, const char *name,
                                 struct json_object *object);
```

Be aware, as with reply functions object is automatically released using json_object_put when using this function. Call json_object_get before calling afb_daemon_broadcast_event to keep object available after function returns.

Event name received by listeners is prefixed with binding name. So when a change occurs after a move, the reason is move and every clients receive an event tictactoe/move.

Note that nothing is said about case sensitivity of event names. However,
the event is always prefixed with the name that the binding declared, with the same case, followed with a slash /. Thus it is safe to compare event using a case sensitive comparison.

10.11. Writing an asynchronous method implementation

The tic-tac-toe example allows two clients or more to share the same board. This is implemented by the method **join** that illustrated partly how to retrieve arguments.

When two or more clients are sharing a same board, one of them can wait until the state of the board changes, but this could also be implemented using events because an even is generated each time the board changes.

In this case, the reply to the wait is sent only when the board changes. See the diagram below:

<table>
<thead>
<tr>
<th>CLIENT A</th>
<th>CLIENT B</th>
<th>TIC-TAC-TOE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>wait . . . . . . . .</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>+-----------------&gt;</td>
<td>move . . . . . . .</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;-----------------+ success of move</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=-----------------</td>
<td>success of wait &lt;=</td>
<td></td>
</tr>
</tbody>
</table>

Here, this is an invocation of the binding by an other client that unblock the suspended **wait** call. Nevertheless in most case this should be a timer, a hardware event, a sync with a concurrent process or thread, ...

Common case of an asynchronous implementation.

Here is the listing of the function **wait**:

```c
static void wait(struct afb_req req) {
    struct board *board;
    struct waiter *waiter;

    /* retrieves the context for the session */
    board = board_of_req(req);
    INFO(afbitf, "method 'wait' called for boardid %d", board->id);

    /* creates the waiter and enqueues it */
    waiter = calloc(1, sizeof *waiter);
    waiter->req = req;
    waiter->next = board->waiters;
    afb_req_addref(req);
    board->waiters = waiter;
}
```
After retrieving the board, the function adds a new waiter to waiters list and returns without setting a reply.

Before returning, it increases req request's reference count using afb_req_addref function.

When a method returns without setting a reply, it MUST increment request's reference count using afb_req_addref. If unpredictable behaviour may pop up.

Later, when a board changes, it calls tic-tac-toe changed function with reason of change in parameter.

Here is the full listing of the function changed:

```c
/*
 * signals a change of the board
 */
static void changed(struct board *board, const char *reason)
{
    struct waiter *waiter, *next;
    struct json_object *description;

    /* get the description */
    description = describe(board);

    waiter = board->waiters;
    board->waiters = NULL;
    while (waiter != NULL) {
        next = waiter->next;
        afb_req_success(waiter->req, json_object_get(description), reason);
        afb_req_unref(waiter->req);
        free(waiter);
        waiter = next;
    }

    afb_event_sender_push(afb_daemon_get_event_sender(afbitf->daemon), reason, description);
}
```

The list of waiters is walked and a reply is sent to each waiter. After sending the reply, the reference count of the request is decremented using afb_req_unref to allow resources to be freed.

The reference count MUST be decremented using afb_req_unref to free resources and avoid memory leaks. This usage count decrement should happen AFTER setting reply or bad things may happen.

### 10.12. How to build a binding

Afb-daemon provides a pkg-config configuration file that can be queried by providing afb-daemon in command line arguments. This configuration file provides data that should be used for bindings compilation. Examples:
10.12.1. Example for cmake meta build system

This example is the extract for building the binding *afm-main* using *CMAKE*.

```cmake
pkg_check_modules(afb afb-daemon)
if(afb_FOUND)
    message(STATUS "Creation afm-main-binding for AFB-DAEMON")
    add_library(afm-main-binding MODULE afm-main-binding.c)
    target_compile_options(afm-main-binding PRIVATE ${afb_CFLAGS})
    target_include_directories(afm-main-binding PRIVATE ${afb_INCLUDE_DIRS})
    target_link_libraries(afm-main-binding utils ${afb_LIBRARIES})
    set_target_properties(afm-main-binding PROPERTIES
        LINK_FLAGS "-Wl,--version-script=${CMAKE_CURRENT_SOURCE_DIR}/afm-main-binding.export-map"
    )
    install(TARGETS afm-main-binding LIBRARY DESTINATION ${binding_dir})
else()
    message(STATUS "Not creating the binding for AFB-DAEMON")
endif()
```

Let now describe some of these lines.

```cmake
pkg_check_modules(afb afb-daemon)
```

This first lines searches to the *pkg-config* configuration file for *afb-daemon*. Resulting data are stored in the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>afb_FOUND</td>
<td>Set to 1 if afb-daemon binding development files exist</td>
</tr>
<tr>
<td>afb_LIBRARIES</td>
<td>Only the libraries (w/o the <code>-l</code>) for compiling afb-daemon bindings</td>
</tr>
<tr>
<td>afb_LIBRARY_DIRS</td>
<td>The paths of the libraries (w/o the <code>-L</code>) for compiling afb-daemon bindings</td>
</tr>
<tr>
<td>afb_LDFLAGS</td>
<td>All required linker flags for compiling afb-daemon bindings</td>
</tr>
<tr>
<td>afb_INCLUDE_DIRS</td>
<td>The <code>-I</code> preprocessor flags (w/o the <code>-I</code>) for compiling afb-daemon bindings</td>
</tr>
<tr>
<td>afb_CFLAGS</td>
<td>All required cflags for compiling afb-daemon bindings</td>
</tr>
</tbody>
</table>

If development files are found, the binding can be added to the set of target to build.

```cmake
add_library(afm-main-binding MODULE afm-main-binding.c)
```

This line asks to create a shared library having a single source file named *afm-main-binding.c* to be compiled. The default name of the created shared object is *libafm-main-binding.so*.

```cmake
set_target_properties(afm-main-binding PROPERTIES
    PREFIX ""
)
This lines are doing two things:

1. It renames the built library from `libafm-main-binding.so` to `afm-main-binding.so` by removing the implicitly added prefix `lib`. This step is not mandatory because afb-daemon doesn't check names of files at load time. The only filename convention used by afb-daemon relates to `.so` termination. `*.so` pattern is used when afb-daemon automatically discovers binding from a directory hierarchy.

2. It applies a version script at link time to only export the reserved name `afbBindingV1Register` for registration entry point. By default, when building a shared library linker exports all the public symbols (C functions that are not static).

Next line are:

```cpp
target_include_directories(afm-main-binding PRIVATE ${afb_INCLUDE_DIRS})
target_link_libraries(afm-main-binding utils ${afb_LIBRARIES})
```

As you can see it uses the variables computed by `pkg_check_modules(afb afb-daemon)` to configure the compiler and the linker.

**10.12.2. Exporting the function afbBindingV1Register**

The function `afbBindingV1Register` MUST be exported. This can be achieved using a version script at link time. Here after is a version script used for `tic-tac-toe` (bindings/samples/export.map).

```plaintext
{ global: afbBindingV1Register; local: *; };
```

This sample `version_script` exports as global the symbol `afbBindingV1Register` and hides any other symbols.

This version script is added to the link options using the option `--version-script=export.map` is given directly to the linker or using the option `-WI,-version-script=export.map` when the option is given to the C compiler.

**10.12.3. Building within yocto**

Adding a dependency to afb-daemon is enough. See below:

```plaintext
DEPENDS += " afb-daemon "
```
# 11. Guide for developing with events

Signaling agents are services that send events to any clients that subscribed for receiving it. The sent events carry any data.

To have a good understanding of how to write a signaling agent, the actions of subscribing, unsubscribing, producing, sending, receiving events must be described and explained.

## 11.1. Overview of events

The basis of a signaling agent is shown on the following figure:

*scenario of using events*

This figure shows the main role of the signaling framework for the propagation of events.

For people not familiar with the framework, a signaling agent and a “binding” are similar.

### 11.1.1. Subscribing and unsubscribing

Subscribing and subscription is the action that makes a client able to receive data from a signaling agent. Subscription must create resources for generating the data and for delivering the data to the client. These two aspects are not handled by the same piece of software: generating the data is the responsibility of the developer of the signaling agent while delivering the data is handled by the framework.

When a client subscribes for data, the agent must:

1. check that the subscription request is correct;
2. establish the computation chain of the required data, if not already done;
3. create a named event for the computed data, if not already done;
4. ask the framework to establish the subscription to the event for the request;
5. optionally give indications about the event in the reply to the client.

The first two steps are not involving the framework. They are linked to the business logic of the binding. The request can be any description of the requested data and the computing stream can be of any nature, this is specific to the binding.

As said before, the framework uses and integrates “libsystemd” and its event loop. Within the framework, "libsystemd" is the standard API/library for bindings expecting to setup and handle I/O, timer or signal events.

Steps 3 and 4 are bound to the framework.

The agent must create an object for handling the propagation of produced data to its clients. That object is called “event” in the framework. An event has a name that
allows clients to distinguish it from other events.

Events are created using the `afb_daemon_make_event` function that takes the name of the event. Example:

```c
event = afb_daemon_make_event(afb_daemon, name);
```

Once created, the event can be used either to push data to its subscribers or to broadcast data to any listener.

The event must be used to establish the subscription for the requesting client. This is done using the `afb_req_subscribe` function that takes the current request object and event and associates them together. Example:

```c
rc = afb_req_subscribe(afb_req, event);
```

When successful, this function make the connection between the event and the client that emitted the request. The client becomes a subscriber of the event until it unsubscribes or disconnects. The `afb_req_subscribe` function will fail if the client connection is weak: if the request comes from a HTTP link. To receive signals, the client must be connected. The AGL framework allows connections using WebSocket.

The name of the event is either a well known name or an ad hoc name forged for the usecase.

Let's see a basic example: client A expects to receive the speed in km/h every second while client B expects the speed in mph twice a second. In that case, there are two different events because it is not the same unit and it is not the same frequency. Having two different events allows to associate clients to the correct event. But this doesn't tell any word about the name of these events. The designer of the signaling agent has two options for naming:

1. names can be the same (“speed” for example) with sent data self-describing itself or having a specific tag (requiring from clients awareness about requesting both kinds of speed isn’t safe).
2. names of the event include the variations (by example: “speed-km/h-1Hz” and “speed-mph-2Hz”) and, in that case, sent data can self-describe itself or not.

In both cases, the signaling agent might have to send the name of the event and/or an associated tag to its client in the reply of the subscription. This is part of the step 5 above.

The framework only uses the event (not its name) for subscription, unsubscription and pushing.

When the requested data is already generated and the event used for pushing it already exists, the signaling agent must not instantiate a new processing chain and must not create a new event object for pushing data. The signaling agent must reuse the existing chain and event.

Unsubscribing is made by the signaling agent on a request of its client. The
**afb_req_unsubscribe** function tells the framework to remove the requesting client from the event's list of subscribers. Example:

```
afb_req_unsubscribe(afb_req, event);
```

Subscription count does not matter to the framework: subscribing the same client several times has the same effect that subscribing only one time. Thus, when unsubscribing is invoked, it becomes immediately effective.

**More on naming events**

Within the AGL framework, a signaling agent is a binding that has an API prefix. This prefix is meant to be unique and to identify the binding API. The names of the events that this signaling agent creates are automatically prefixed by the framework, using the API prefix of the binding.

Thus, if a signaling agent of API prefix *api* creates an event of name *event* and pushes data to that event, the subscribers will receive an event of name *api/event*.

**11.1.2. Generating and pushing signals and data**

This is the responsibility of the designer of the signaling agent to establish the processing chain for generating events. In many cases, this can be achieved using I/O or timer or signal events inserted in the main loop. For this case, the AGL framework uses “libsystemd” and provide a way to integrates to the main loop of this library using `afb_daemon_get_event_loop`. Example:

```
sdev = afb_daemon_get_event_loop(af_daemon);
rc = sd_event_add_io(sdev, &source, fd, EPOLLIN, myfunction, NULL);
```

In some other cases, the events are coming from D-Bus. In that case, the framework also uses “libsystemd” internally to access D-Bus. It provides two methods to get the available D-Bus objects, already existing and bound to the main libsystemd event loop. Use either `afb_daemon_get_system_bus` or `afb_daemon_get_user_bus` to get the required instance. Then use functions of “libsystemd” to handle D-Bus.

In some rare cases, the generation of the data requires to start a new thread.

When a data is generated and ready to be pushed, the signaling agent should call the function `afb_event_push`. Example:

```
rc = afb_event_push(event, json);
if (rc == 0) {
    stop_generating(event);
    afb_event_drop(event);
}
```

The function `afb_event_push` pushes json data to all the subscribers. It then returns
the count of subscribers. When the count is zero, there is no subscriber listening for
the event. The example above shows that in that case, the signaling agent stops to
generate data for the event and delete the event using afb_event_drop. This is one
possible option. Other valuable options are: do nothing and continue to generate and
push the event or just stop to generate and push the data but keep the event existing.

11.1.3. Receiving the signals
Understanding what a client expects when it receives signals, events or data shall be
the most important topic of the designer of a signaling agent. The good point here is
that because JSON¹ is the exchange format, structured data can be sent in a flexible
way.

The good design is to allow as much as possible the client to describe what is needed
with the goal to optimize the processing to the requirements only.

11.1.4. The exceptional case of wide broadcast
Some data or events have so much importance that they can be widely broadcasted to
alert any listening client. Examples of such an alert are:

• system is entering/leaving “power safe” mode
• system is shutting down
• the car starts/stops moving
• ...

An event can be broadcasted using one of the two following methods: afb_daemon_broadcast_event or afb_event_broadcast.

Example 1:

```
afb_daemon_broadcast_event(afb_daemon, name, json);
```

Example 2:

```
  event = afb_daemon_make_event(afb_daemon, name);
  ... 
  afb_event_broadcast(event, json);
```

As for other events, the name of events broadcasted using afb_daemon_broadcast_event are automatically prefixed by the framework with API
prefix of the binding (signaling agent).

¹ There are two aspect in using JSON: the first is the flexible data structure that mixes
common types (booleans, numbers, strings, arrays, dictionaries, nulls), the second, is the
streaming specification. Streaming is often seen as the bottleneck of using JSON (see
http://bjson.org). When the agent share the same process, there is no streaming at all.
11.2. Reference of functions

11.2.1. Function afb_event afb_daemon_make_event
The function `afb_daemon_make_event` that is defined as below:

```c
/*
 * Creates an event of 'name' and returns it.
 * 'daemon' MUST be the daemon given in interface when activating the binding.
 */
struct afb_event afb_daemon_make_event(struct afb_daemon daemon, const char *name);
```

The daemon is the handler to the application framework binder daemon received during initialisation steps of the binding.

Calling the function `afb_daemon_make_event` within the initialisation function `afbBindingV1Register` will fail because the plugin name is not known at this time.

The correct way to create the event at initialisation is to call the function `afb_daemon_make_event` within the initialisation function `afbBindingV1ServiceInit`.

11.2.2. Function afb_event_push
The function `afb_event_push` is defined as below:

```c
/*
 * Pushes the 'event' with the data 'object' to its observers.
 * 'object' can be NULL.
 * For convenience, the function calls 'json_object_put' for object'.
 * Thus, in the case where 'object' should remain available after
 * the function returns, the function 'json_object_get' shall be used.
 * Returns the count of clients that received the event.
 */
int afb_event_push(struct afb_event event, struct json_object *object);
```

As the function `afb_event_push` returns 0 when there is no more subscriber, a binding can remove such unexpected event using the function `afb_event_drop`.

11.2.3. Function afb_event_drop
The function `afb_event_drop` is defined as below:

```c
/*
 * Drops the data associated to the event
 * After calling this function, the event
 * MUST NOT BE USED ANYMORE.
 */
void afb_event_drop(struct afb_event event);
```
11.2.4. Function afb_req_subscribe

The function `afb_req_subscribe` is defined as below:

```c
/*
 * Establishes for the client link identified by 'req' a subscription
 * to the 'event'.
 * Returns 0 in case of successful subscription or -1 in case of error.
 */
int afb_req_subscribe(struct afb_req req, struct afb_event event);
```

The subscription adds the client of the request to the list of subscribers to the event.

11.2.5. Function afb_req_unsubscribe

The function `afb_req_unsubscribe` is defined as below:

```c
/*
 * Revokes the subscription established to the 'event' for the client
 * link identified by 'req'.
 * Returns 0 in case of successful unsubscription or -1 in case of error.
 */
int afb_req_unsubscribe(struct afb_req req, struct afb_event event);
```

The unsubscription removes the client of the request from the list of subscribers to the event. When the list of subscribers to the event becomes empty, the function `afb_event_push` will return zero.

11.2.6. Function afb_event_broadcast

The function `afb_event_broadcast` is defined as below:

```c
/*
 * Broadcasts widely the 'event' with the data 'object'.
 * 'object' can be NULL.
 *
 * For convenience, the function calls 'json_object_put' for 'object'.
 * Thus, in the case where 'object' should remain available after
 * the function returns, the function 'json_object_get' shall be used.
 *
 * Returns the count of clients that received the event.
 */
int afb_event_broadcast(struct afb_event event, struct json_object *object);
```

This uses an existing event (created with `afb_daemon_make_event`) for broadcasting an event having its name.

11.2.7. Function afb_daemon_broadcast_event

The function `afb_daemon_broadcast_event` is defined as below:

```c
/*
 * Broadcasts widely the event of 'name' with the data 'object'.
 */
```
* 'object' can be NULL.
* 'daemon' MUST be the daemon given in interface when activating the binding.
* For convenience, the function calls 'json_object_put' for 'object'.
* Thus, in the case where 'object' should remain available after
  the function returns, the function 'json_object_get' shall be used.
* Returns the count of clients that received the event.
*/
int afb_daemon_broadcast_event(struct afb_daemon daemon, const char *name, 
struct json_object *object);

The name is given here explicitly. The name is automatically prefixed with the name of the binding. For example, a binding of prefix "xxx" would broadcast the event "xxx/name".

11.3. Architectural digressions

Based on their dependencies to hardware, signaling agents can be split into 2 categories: low-level signaling agents and high-level signaling agents.

Low-level signaling agents are bound to the hardware and focused on interfacing and driving.

High-level signaling agent are independent of the hardware and ocused on providing service.

This separation (that may in the corner look artificial) aim to help in the systems design. The main idea here is that high-level signaling agents are providing “business logic”, also known as “application logic”, that is proper to the car industry and that can be reused and that can evolve as a foundation for the future of the industry.

The implementation of this decomposition may follow 2 paths: strict separation or soft composition.

11.3.1. Strict separation

The strict separation implements the modularity composition of signaling agent through the framework. The high-level signaling agent subscribes to the low level signaling agent using the standard client API.

Advantages:

- Modularity
- Separation of responsibilities
- Possible aggregation of multiple sources
- Soft binding of agent good for maintenance

Drawbacks:
• Cost of propagation of data (might serialize)
• Difficulties to abstract low-level signaling agent or to find a trade-off between abstracting and specializing

The key is modularity versus cost of propagation. It can be partly solved when logical group of signaling agent are launched together in the same binder process. In that particular case, the cost of propagation of data between agents is reduced\(^2\) because there is no serialization.

This reduction of the propagation cost (and of the resources used) precludes implementation of strong security between the agents because they share the same memory.

### 11.3.2. Soft composition

The soft composition implements the business logic of high-level signaling agents as libraries that can then be used directly by the low level signaling agents.

**Advantages:**
- No propagation: same memory, sharing of native structures

**Drawbacks:**
- Cannot be used for aggregation of several sources
- Difficulties to abstract low-level signaling agent or to find a trade-off between abstracting and specializing
- Source code binding not good for maintenance

\(^2\) Within the same process, there is not serialization, the propagation has the cost of wrapping a json data and calling callbacks with the benefit of having a powerful callback manager: the event mechanism of the framework.
12. Overview of tests shipped with AFB-Daemon

12.1. List of tests

Here are the tests shipped in the source tree:

- **afb-client-demo** (command-line WebSockets)
- **token-websock.qml** (Qt/QML WebSockets)
- ***.html** (HTML5/JS HTTP-REST & WebSockets)

12.2. Detail of tests

12.2.1. **afb-client-demo (command-line WebSockets)**

This clients interactively calls bindings APIs from the command line, using the binder WebSockets facility.

If **afb-daemon** has been launched with the following parameters:

```
$ afb-daemon --port=1234 --token=123456 [...]
```

Then run the client with:

```
```

For instance, to initialize the Audio binding from the command line:

```
afb-client-demo ws://localhost:1234/api?token=123456
```

The command doesn't return. You should type requests of type `[]`. So, try:

```
auth connect
hello pingjson true
```

12.2.2. **token-websock.qml (Qt/QML WebSockets)**

If **afb-daemon** has been launched with the following parameters:

```
$ afb-daemon --port=1234 --token=123456 [...]
```

and Qt5 is installed.

For installing Qt5 on **Ubuntu 16.04**:  

```
$ apt-get install qmlscene qml-module-qtwebsockets qml-module-qtvquick-controls
```
For installing Qt5 on **Fedora 23**:

```
$ dnf install qt5-qtdeclarative-devel qt5-qtwebsockets-devel qt5-qtquickcontrols
```

Then run the client with:

```
qmlscene test/token-websock.qml
```

and interactively press the buttons, "Connect", "Refresh", "Logout".

### 12.2.3. *html (HTML5/JS HTTP-REST & WebSockets)

If *afb-daemon* has been launched with the following parameters:

```
$ afb-daemon --port=1234 --rootdir=$PWD/test [...]
```

("$PWD/test" being the "test" subdirectory of the source tree)

Then open your preferred Web browser, connect to the following URL:

```
http://localhost:1234
```

and interactively run the various tests.
13. Vocabulary for AFB-DAEMON

13.1. Binding

A shared library object intended to be add a functionnality to an afb-daemon instance. It implements an API. It may provide a service.

Binding made for services can have specific entry point called after initialisation and before serving.

13.2. Event

Message with data propagated from the services to the client and not expecting any reply.

The current implementation allows to widely broadcast events to all clients.

13.3. Level of assurance (LOA)

This level that can be from 0 to 3 represent the level of assurance that the services can expect from the session.

The exact definition of the meaning of this levels and of how to use it remains to be achieved.

13.4. Plugin

Old name for binding, see binding.

13.5. Request

A request is an invocation by a client to a method of a binding using a message transfered through some protocol: HTTP, WebSocket, DBUS... served by afb-daemon

13.6. Reply/Response

This is a message sent to client as the result of the request.

13.7. Service

Service are made of binding runnning by their side on their binder. It can serve many client. Each one being attached to one session.

The framework establishes the connection between the services and the clients. Using
DBus currently but other protocols are considered.

### 13.8. Session

A session is meant to be the unique context of an instance of client, identifying that instance across requests.

Each session has an identifier. Session identifier generated by afb-daemon are UUIDs. Internally, afb-daemon offers a mechanism to attach data to sessions. When the session is closed or disappears, the data attached to that session are freed.

### 13.9. Token

The token is an identifier that the client must give to be authenticated.

At start, afb-daemon get an initial token. This initial token must be presented incoming client to be authenticated.

A token is valid only for a period.

The token must be renewed periodically. When the token is renewed, afb-daemon sends the new token to the client.

Tokens generated by afb-daemon are UUIDs.

### 13.10. UUID

It stand for Universal Unic IDentifier.

Its is designed to create identifier in a way that avoid has much as possible conflicts. It means that if two different instance create a UUID, the probability that they create the same UUID is very low, near to zero.

### 13.11. x-afb-reqid

Argument name that can be used with HTTP request. When this argument is given, it is automatically added to the "request" object of the answer.

### 13.12. x-afb-token

Argument name for giving the token without ambiguity. You can also use the name token but it may conflicts with other arguments.

### 13.13. x-afb-uuid

Argument name for giving explicitely the session identifier without ambiguity. You can also use the name uuid but it may conflicts with other arguments.