AGL/Phase2 - Devkit

Build your 1st AGL application

Version 2.0

July 2016

Abstract

In the previous AGL DevKit document, entitled "**AGL Devkit - Image and SDK for porter**", we rebuilt a complete bootable AGL image from source code. We also generated and installed related SDK files in our development container. This document describes the next step: using the SDK to build applications and deploy them on a target board running an AGL image.

AGL DevKit as a whole contains:

- "**AGL Devkit - Image and SDK for porter**" guide for preparing a Docker container with AGL SDK, ready to build AGL applications
- Application templates and demos, allowing developers to start developing various types of applications:
  - Services
  - Native applications
  - HTML5 applications
  - ...
- This guide, focused on how to use these applications templates

This document focuses on: initializing the SDK environment, building the development templates, modifying them, and installing them on the target.

Document revisions

Date	Version	Designation	Author
21 Jun 2016	0.1	Initial release	M.Bachmann [Iot.bzh]
12 Jul 2016	0.2	More templates	S. Desneux [Iot.bzh] M.Bachmann [Iot.bzh]
15 Jul 2016	1.0	Templates finalized	S. Desneux [Iot.bzh] M.Bachmann [Iot.bzh]
17 Jul 2016	1.1	Demos added	M.Bachmann [Iot.bzh]
18 Jul 2016	2.0	Final Review – Version 2.0 aligned wtth other documents	S. Desneux [IoT.bzh]

Table of contents

1. Initializing SDK environment and templates

1.1. Initializing the SDK environment

*(This document assumes that you are logged inside the **bsp-devkit** Docker container, used to produce Porter BSP image and AGL SDK in the previous **"Image and SDK for porter"** document)*

To be able to use the toolchain and utilities offered by AGL SDK, it is necessary to source the proper setup script. This script is in the SDK root folder and is named **/xdt/sdk/environment-setup-*** (full name depends on the target machine)

For Porter board, we source the required SDK environment variables like this:

```
devel@bsp-devkit:~$ source /xdt/sdk/environment-setup-cortexa15hf-vfp-neon-poky-linux-gnueabi
```

To verify that it succeeded, we should obtain a non-empty result for this command:

```
devel@bsp-devkit:~$ echo $CONFIG_SITE | grep sdk
/xdt/sdk/site-config-cortexa15hf-vfp-neon-poky-linux-gnueabi
```

1.2. Application categories

We provide multiple development templates for the AGL SDK:

- **Service**

  A Service is a headless background process, allowing Bindings to expose various APIs accessible through the transports handled by the application framework, which are currently:

  - HTTP REST (HTTP GET, POST…)[1]
  - WebSocket[2]
  - D-Bus[3]

- **Native application**

  A Native application is a compiled application, generally written in C/C++, accessing one or more services, either by its own means or using a helper library with HTTP REST/WebSocket capabilities.

  (our template is written in C and uses the "libafbwsc" helper library available in the app-framework-binder source tree on AGL Gerrit)

- **HTML5 application**

  An HTML5 application is a web application, generally written with a framework (AngularJS, Zurb Foundation…), accessing services with its built-in HTTP REST/WebSocket capabilities.

- **QML application**

  An QML application is a Qt application written in QML/QtQuick descriptive language, accessing a service with its built-in HTTP REST/WebSocket capabilities.

- **Hybrid application (composed of a backend and a frontend)**

  A Hybrid application contains at the same time (an) **Application-specific Binding(s)** as backend(s) and a **User Interface** (Native, HTML5, QML …) as a frontend.

  This is probably the most pertinent real-world case, since it allows developers to provide capabilities through Bindings, and an end-user experience through the UI. For instance, a GPS Binding giving device localization status, and a HTML5 GPS frontend displaying it on the screen.

1 REST: https://en.wikipedia.org/wiki/Representational_state_transfer
2 WebSocket: https://en.wikipedia.org/wiki/WebSocket
3 D-Bus: https://www.freedesktop.org/wiki/Software/dbus/

1.3. Getting application templates

Application Framework Templates live in a dedicated Git Repository, currently hosted on GitHub at the following address:

https://github.com/iotbzh/app-framework-templates

To get the templates in our development container, let us simply clone the source repository:

```
devel@bsp-devkit:~$ cd ~
devel@bsp-devkit:~$ git clone https://github.com/iotbzh/app-framework-templates
Cloning into 'app-framework-templates'...
remote: Counting objects: 315, done.
remote: Compressing objects: 100% (62/62), done.
remote: Total 315 (delta 25), reused 0 (delta 0), pack-reused 250
Receiving objects: 100% (315/315), 512.81 KiB | 284.00 KiB/s, done.
Resolving deltas: 100% (125/125), done.
Checking connectivity... done.
```

1.4. Organization of templates

Templates are provided with the following layout:

```
devel@bsp-devkit:/$ tree -L 2 app-framework-templates
app-framework-templates
|-- build_all
|-- demos
|   |-- cpu-hybrid-html5
|   `-- cpu-hybrid-qml
`-- templates
    |-- html5
    |-- hybrid-html5
    |-- hybrid-qml
    |-- native
    |-- qml
    `-- service
```

There are 2 main categories in the repository:

- **demos/:** projects demonstrating fully-working applications, that are still simple enough to be used as starting points.
- **templates/:** one subdirectory per template type, see folder name.

Here are some details on the files encountered in the projects:

- **CMakeLists.txt**: our templates use CMake for automatic configuration and building. In your projects, you can of course adapt templates to use your preferred solution (Autoconf, Scons…).
- **gulpfile.js**: these are some kind of "Makefiles" used by the Gulp tool. g*ulp* is often used in HTML5 projects as it is able to execute all needed tasks to process web source files (JavaScript, CSS, HTML templates, images…) and create a directory suitable for deployment on a website.
- **package.json**: this is a Node.js file used to specify prjoject dependencies. Basically, *gulp* and *gulpfile.js w*ill download and install all packages mentioned here to assemble the HTML5 project during the "npm install" step.
- **config.xml(.in)**: XML configuration file required by the application framework. This file is mandatory for an AGL Application to be installed and launched by the framework.
- **export.map**: for Bindings (a.k.a. shared libraries) only, this file must contain a list of exported API verbs. Only the symbols specified in this export file will be accessible at runtime. So *export.map* should contain all verbs you intend to provide in your Binding.

2. Trying out the templates

In this section, we describe how to build the various templates, how to package them and finally how to deploy and run them on the target board.

Every template is stored in a single directory and is independent of the rest of the tree: each template can be used as a foundation for a fresh new application.

For deployment, we assume that the target board is booted with an AGL image and available on the network. Let us use the "BOARDIP" variable for the board IP address (replace by appropriate address or hostname) and test the SSH connection:

```
$ export BOARDIP=1.2.3.4
$ ssh root@$BOARDIP cat /etc/os-release
ID="poky-agl"
NAME="Automotive Grade Linux"
VERSION="1.0+snapshot-20160717 (master)"
VERSION_ID="1.0+snapshot-20160717"
PRETTY_NAME="Automotive Grade Linux 1.0+snapshot-20160717 (master)"
```

2.1. Building

All operations are executed inside the Docker "Devkit" container. For convenience, a build script named "build_all" can build all templates at once. The following instructions allow to build each template separately.

2.1.1. Service

Compile the Service:

```
$ cd ~/app-framework-templates/templates/service
$ mkdir build; cd build
$ cmake ..
-- The C compiler identification is GNU 5.2.0
[snip]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/devel/app-framework-templates/tem-
plates/service/build
$ make
Scanning dependencies of target xxxxxx-service
[snip]
[100%] Generating xxxxxx-service.wgt
NOTICE: -- PACKING widget xxxxxx-service.wgt from directory package
[100%] Built target widget
```

This produced a "xxxxxx-service.wgt" package. Let us copy it to the target:

```
$ scp xxxxxx-service.wgt root@$BOARDIP:~/
xxxxxx-service.wgt                                100%   24KB  24.3KB/s   00:00
```

2.1.2. Native application

Compile the Native application:

```
$ cd ~/app-framework-templates/templates/native
$ mkdir build; cd build
$ cmake ..
-- The C compiler identification is GNU 5.2.0
[snip]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/devel/app-framework-templates/tem-
plates/native/build
$ make
Scanning dependencies of target xxxxxx-native
[ 33%] Building C object CMakeFiles/xxxxxx-native.dir/xxxxxx-native-client.c.o
[ 66%] Linking C executable xxxxxx-native
[ 66%] Built target xxxxxx-native
Scanning dependencies of target widget
[100%] Generating xxxxxx-native.wgt
NOTICE: -- PACKING widget xxxxxx-native.wgt from directory package
[100%] Built target widget
```

This produced a "xxxxxx-native.wgt" package. Let us copy it to the target:

```
$ scp xxxxxx-native.wgt root@$BOARDIP:~/
xxxxxx-native.wgt                                     100%   15KB  15.2KB/s   00:00
```

2.1.3. QML application

Package the QML application:

```
$ cd ~/app-framework-templates/templates/qml
$ mkdir build; cd build
$ cmake ..
-- Creation of xxxxxx-qml for AFB-DAEMON
-- Configuring done
-- Generating done
-- Build files have been written to: /home/devel/app-framework-templates/tem-
plates/qml/build
$ make
Scanning dependencies of target widget
[100%] Generating xxxxxx-qml.wgt
NOTICE: -- PACKING widget xxxxxx-qml.wgt from directory package
[100%] Built target widget
```

This produced a "xxxxxx-qml.wgt" package. Again, copy it onto the target:

```
$ scp xxxxxx-qml.wgt root@$BOARDIP:~/
xxxxxx-qml.wgt                                        100% 5852     5.7KB/s   00:00
```

2.1.4. HTML5 Application

For HTML5 applications, we first need to install Node.js and Gulp (*remember: password for user 'devel' is 'devel'*):

```
$ curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
[snip]
$ sudo apt-get install nodejs
[snip]
$ sudo npm install --global gulp
[snip]
```

Then we need to install required Gulp dependencies to package this particular HTML5 application, written in AngularJS and using Zurb Foundation 6 for styling:

```
$ cd ~/app-framework-templates/templates/html5
$ npm install
[snip]
```

This can take some time as many Node.js modules are required.

We are then able to create the widget using the usual commands:

```
$ mkdir build; cd build
$ cmake ..
-- The C compiler identification is GNU 5.2.0
[snip]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/devel/app-framework-templates/tem-
plates/html5/build
$ make
Scanning dependencies of target widget
[100%] Generating xxxxxx-html5.wgt
[16:46:23] Using gulpfile ~/app-framework-templates/templates/html5/gulpfile.js
[16:46:23] Starting 'build-app-prod'...
[16:46:28] gulp-imagemin: Minified 11 images (saved 35.13 kB - 8.8%)
[16:46:28] gulp-inject 1 files into index.html.
[16:46:28] gulp-inject 2 files into index.html.
[16:46:28] gulp-inject 1 files into index.html.
[16:46:28] gulp-inject 1 files into index.html.
[16:46:28] Finished 'build-app-prod' after 4.8 s
[16:46:28] Starting 'widget-config-prod'...
[16:46:28] Finished 'widget-config-prod' after 3.54 ms
NOTICE: -- PACKING widget xxxxxx-html5.wgt from directory package
[100%] Built target widget
```

This produced a "xxxxxx-html5.wgt" package. Let us copy it to the target:

```
$ scp xxxxxx-html5.wgt root@$BOARDIP:~/
xxxxxx-html5.wgt                                  100%  657KB 656.6KB/s   00:00
```

2.1.5. Hybrid QML Application

This application is composed of a UI written in QML and a specific binding available though the binder.

Build and package the application:

```
$ cd ~/app-framework-templates/templates/hybrid-qml
$ mkdir build; cd build
$ cmake ..
-- The C compiler identification is GNU 5.2.0
[snip]
-- Creation of xxxxxx-hybrid-qml for AFB-DAEMON
-- Configuring done
-- Generating done
-- Build files have been written to: /home/devel/app-framework-templates/tem-
plates/hybrid-qml/build
$ make
Scanning dependencies of target xxxxxx-hybrid-qml
[ 33%] Building C object CMakeFiles/xxxxxx-hybrid-qml.dir/xxxxxx-hybrid-qml-
binding.c.o
[ 66%] Linking C shared module xxxxxx-hybrid-qml.so
[ 66%] Built target xxxxxx-hybrid-qml
Scanning dependencies of target widget
[100%] Generating xxxxxx-hybrid-qml.wgt
NOTICE: -- PACKING widget xxxxxx-hybrid-qml.wgt from directory package
[100%] Built target widget
```

This produced a "xxxxxx-hybrid-qml.wgt" package. Let us copy it to the target:

```
$ scp xxxxxx-hybrid-qml.wgt root@$BOARDIP:~/
xxxxxx-hybrid-qml.wgt                                 100%   13KB  12.7KB/s   00:00
```

2.1.6. Hybrid HTML5 Application

The same dependencies as for a "pure HTML5" application apply: we need to install Gulp dependencies first:

```
$ cd ~/app-framework-templates/templates/hybrid-html5
$ npm install
[snip]
```

Then we can create the application:

```
$ mkdir build; cd build
$ cmake ..
[snip]
-- Build files have been written to: /home/devel/app-framework-templates/tem-
plates/hybrid-html5/build
$ make
Scanning dependencies of target xxxxxx-hybrid-html5
[snip]
Scanning dependencies of target widget
```

```
[snip]
NOTICE: -- PACKING widget xxxxxx-hybrid-html5.wgt from directory package
[100%] Built target widget
```

This produced a "xxxxxx-hybrid-html5.wgt" package. Let us copy it to the target:

```
$ scp xxxxxx-hybrid-html5.wgt root@$BOARDIP:~/
xxxxxx-hybrid-html5.wgt                          100%  660KB 660.2KB/s   00:00
```

2.2. Installing on the target

And then, in a separate window, log in to the target board:

```
$ ssh root@$BOARDIP
```

and install our packages:

```
# afm-util install xxxxxx-service.wgt
{ "added": "xxxxxx-service@0.1" }
# afm-util install xxxxxx-native.wgt
{ "added": "xxxxxx-native@0.1" }
# afm-util install xxxxxx-qml.wgt
{ "added": "xxxxxx-qml@0.1" }
# afm-util install xxxxxx-html5.wgt
{ "added": "xxxxxx-html5@0.1" }
# afm-util install xxxxxx-hybrid-qml.wgt
{ "added": "xxxxxx-hybrid-qml@0.1" }
# afm-util install xxxxxx-hybrid-html5.wgt
{ "added": "xxxxxx-hybrid-html5@0.1" }
```

or alternatively, to install all packages at once:

```
# for x in *.wgt; do afm-util install $x; done
```

We can verify that they were correctly installed by listing all available applications:

```
# afm-util list
[ { "id": "xxxxxx-service@0.1", "version": "0.1", "width": 0, "height": 0,
"name": "App Framework - xxxxxx-service", "description": "This service is used
for ... (TODO: add description here)", "shortname": "", "author": "John Doe
<john.doe@example.com>" },
 { "id": "xxxxxx-hybrid-html5@0.1", "version": "0.1", "width": 0, "height": 0,
"name": "App Framework - xxxxxx-hybrid-html5", "description": "This application
is used for ... (TODO: add description here)", "shortname": "", "author": "John
Doe <john.doe@example.com>" },
 { "id": "xxxxxx-html5@0.1", "version": "0.1", "width": 0, "height": 0, "name":
"App Framework - xxxxxx-html5", "description": "This application is used
for ... (TODO: add description here)", "shortname": "", "author": "John Doe
<john.doe@example.com>" },
```

```
 { "id": "xxxxxx-native@0.1", "version": "0.1", "width": 0, "height": 0,
"name": "App Framework - xxxxxx-native", "description": "This application is
used for ... (TODO: add description here)", "shortname": "", "author": "John
Doe <john.doe@example.com>" },
 { "id": "xxxxxx-hybrid-qml@0.1", "version": "0.1", "width": 0, "height": 0,
"name": "App Framework - xxxxxx-hybrid-qml", "description": "This application
is used for ... (TODO: add description here)", "shortname": "", "author": "John
Doe <john.doe@example.com>" },
 { "id": "xxxxxx-qml@0.1", "version": "0.1", "width": 0, "height": 0, "name":
"App Framework - xxxxxx-qml", "description": "This application is used for ...
(TODO: add description here)", "shortname": "", "author": "John Doe <john.-
doe@example.com>" } ]
```

2.3. Running the templates

Even if the templates are mostly skeleton applications, we can still start them and see them in action.

All templates have been installed through the application framework, so we will run them with the afm-util tool (only exception: Native application – see explanations in chapter 2.3.2).

When the afm-user-daemon process receives a request to start an application, it reads the application's "**config.xml**" configuration file located in its specific directory (*/usr/share/afm/applications/<appname>/<version>/config.xml*). Depending on the specified MIME type (or the default Linux MIME type in none was given), it then starts (an) appropriate process(es).

Final behavior is controlled by the global "**/etc/afm/afm-launch.conf**" config file:

```
# cat /etc/afm/afm-launch.conf
[snip]
application/vnd.agl.service
     /usr/bin/afb-daemon --ldpaths=/usr/lib/afb:%r/%c --mode=local --readyfd=
%R --alias=/icons:%I --port=%P --rootdir=%r/htdocs --token=%S -sessiondir=
%D/.afb-daemon

application/vnd.agl.native
     /usr/bin/afb-daemon --ldpaths=/usr/lib/afb:%r/lib --mode=local --readyfd=
%R --alias=/icons:%I --port=%P --rootdir=%r/htdocs --token=%S --sessiondir=
%D/.afb-daemon
     %r/%c %P %S

application/vnd.agl.qml
     /usr/bin/qt5/qmlscene -fullscreen -I %r -I %r/imports %r/%c

application/vnd.agl.qml.hybrid
     /usr/bin/afb-daemon --ldpaths=/usr/lib/afb:%r/lib --mode=local --readyfd=
%R --alias=/icons:%I --port=%P --rootdir=%r/htdocs --token=%S --sessiondir=
%D/.afb-daemon
     /usr/bin/qt5/qmlscene %P %S -fullscreen -I %r -I %r/imports %r/%c

application/vnd.agl.html.hybrid
     /usr/bin/afb-daemon --ldpaths=/usr/lib/afb:%r/lib --mode=local --readyfd=
%R --alias=/icons:%I --port=%P --rootdir=%r/htdocs --token=%S --sessiondir=
%D/.afb-daemon
     /usr/bin/web-runtime http://localhost:%P/%c?token=%S

[snip]
```

This file defines how various applications types are handled by the application framework daemons.

2.3.1. Run a Service

Let us first run the Service:

```
# afm-util run xxxxxx-service@0.1
1
```

Then confirm it is running:

```
# afm-util ps
[ { "runid": 1, "state": "running", "id": "xxxxxx-service@0.1" } ]
# ps -ef | grep afb
ps -ef|grep afb
root       848   650  0 17:34 ?        00:00:00 /usr/bin/afb-daemon
--ldpaths=/usr/lib/afb:/usr/share/afm/applications/xxxxxx-service/0.1/lib
--mode=local --readyfd=8 --alias=/icons /usr/share/afm/icons --port=12345
--rootdir=/usr/share/afm/applications/xxxxxx-service/0.1/htdocs
--token=2F4DCA47 --sessiondir=/home/root/app-data/xxxxxx-service/.afb-daemon
```

We can see that an afb-daemon (Binder) process for this Service started on port 12345 with security token 2F4DCA47, and that the directory containing the shared library (*"/usr/share/afm/applications/xxxxxx-service/0.1/lib"*) has been added to the --ldpaths option. Please take note of your specific port and security token, as we will re-use them in the next chapter.

A Service has no user interface, but we can still try the binding API. Looking at the Service source code, we see that the Service implements an API named '**xxxxxx**' providing a '**ping**' verb. Let us try this using a simple REST request that will prove that the service is functional:

```
# web-runtime http://localhost:12345/api/xxxxxx/ping
```

2.3.2. Run a Native application

Now, let us see if our Native application is able to connect to the previous Service using WebSockets capabilities.

Please note that the Native application we built is a console application. For this reason, we won't start it through the application framework because we need a text console to see its output. If the Native application had been a visual one (say a Qt, GTK+ or EFL application), we would have launched this app using the "afm-util start" command.

First, let us make the application executable:

```
# chmod +x /usr/share/afm/applications/xxxxxx-native/0.1/xxxxxx-native
```

This Native application takes the following arguments:

- Service WebSocket URL (*ws://localhost:<PORT>/api?token=<TOKEN>*)
- API name ('*xxxxxx*' in our case)
- verb name ('*ping*' for instance)

Replace PORT and TOKEN with the appropriate values you noted during chapter 2.3.1, then run the application:

```
# PORT=12345
# TOKEN=2F4DCA47
# /usr/share/afm/applications/xxxxxx-native/0.1/xxxxxx-native ws://localhost:
$PORT/api?token=$TOKEN xxxxxx ping
ON-REPLY 1:xxxxxx/ping: {"response":"Some String","jtype":"afb-reply","re-
quest":{"status":"success","info":"Ping Binder Daemon tag=pingSample count=4
query=\"null\"","uuid":"72fdce2a-c029-4d3e-b03a-f564393cb65c"}}
```

2.3.3. Run a QML Application

For the QML Application, we can start it using afm-util:

```
# afm-util start xxxxxx-qml@0.1
4
```

A new window should appear on the display, similar to this capture:

We can see that the application is running:

```
# afm-util ps
[ { "runid": 1, "state": "running", "id": "xxxxxx-service@0.1" },
 { "runid": 4, "state": "running", "id": "xxxxxx-qml@0.1" } ]
```

Note that in this case, no Binder is started for the application. This may change in the future depending on the security model chosen by AGL, and how we want to handle Qt/QML apps.

When clicking on the Quit button, the application terminates and the application framework daemon detects the end of the process:

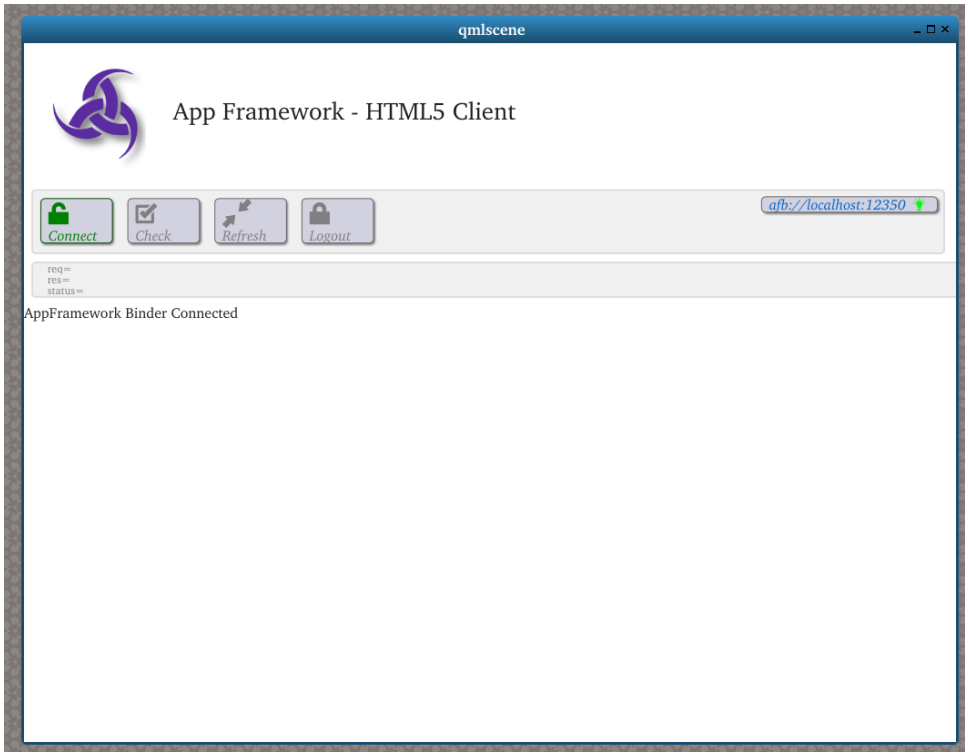
```
# afm-util ps
[ { "runid": 1, "state": "running", "id": "xxxxxx-service@0.1" } ]
```

2.3.4. Run a HTML5 Application

We can then start the HTML5 application:

```
# afm-util start xxxxxx-html5@0.1
6
```

Its user interface looks like this:



Buttons allow to demonstrate default authentication API, provided by “auth” binding.

Note that a Binder is started to support the HTML5 application (at least to allow the webengine to download the static data: HTML page, javascript code …) even if the application doesn't provide any specific binding.

2.3.5. Run a Hybrid QML Application

Let us start the hybrid QML application:

```
# afm-util start xxxxxx-hybrid-qml@0.1
7
```

The following UI appears:

In the hybrid-qml source code, the binding provides an API named '**xxxxxx**' with a '**ping**' verb. When clicking on the "Ping!" button, the QML client sends the "**xxxxxx/ping**" API call to the Binder, which routes it to xxxxxx-hybrid-qml-binding and executes the 'ping' verb.

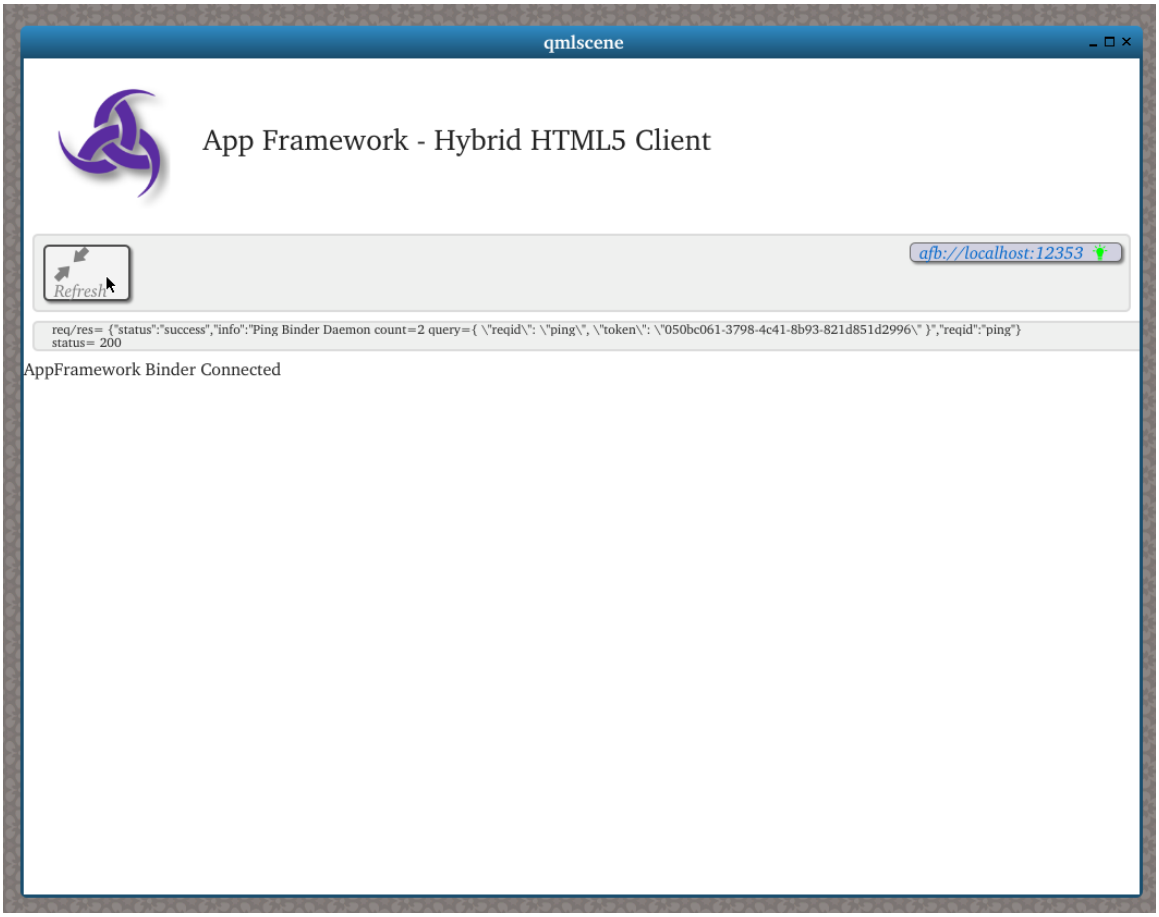
The status is updated by the response to the API call.

2.3.6. Run a Hybrid HTML5 Application

Finally, let us start the Hybrid HTML5 application:

```
# afm-util start xxxxxx-hybrid-html5@0.1
7
```

The following UI appears:



This application uses the same binding as the Hybrid QML one. It also demonstrates the same functionnality: when clicking on the 'Refresh" button, it calls **xxxxxx/ping** on the binder and gets the result then displays it.

3. Developing smoothly with the container

The previous chapter pretty much illustrated the virtues of the Docker container by letting you compile AGL applications, independently of your host machine.

There is one catch, though, in that the container does not feature a full graphical environment similar to those which developers are used to. But this can be circumvented.

For this reason, and before we start developing our own apps, we will explain how to get the best from the provided development container.

3.1. Remote display

Apart from popular console tools such as *vi, git, patch, diff*... our container also features graphical applications: *gvim* text editor, *gitg* frontend for Git, *gvimdiff…*

You can display them on your host machine by taking advantage of X11 protocol remoting capabilities. The procedure differs depending on your host machine.

3.1.1. Linux

You have to connect to your container by specifying the *-X* option:

```
$ ssh -X -p 2222 devel@localhost
```

and then any graphical window, such as *gvim*'s, should display on your host screen.

3.1.2. Mac OS X

You have to connect to your container by specifying the *-X* option:

```
$ ssh -X -p 2222 devel@localhost
```

together with a running X11 server such as XQuartz.

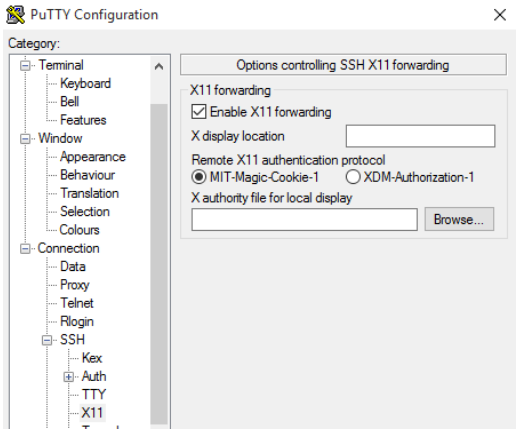
XQuartz was included in old versions such as 10.5 Leopard; you can find it under "Applications" → "Utilities" → "X11". For more recent versions starting from 10.6.3, you may have to download and install it from the following URL: https://dl.bintray.com/xquartz/downloads/XQuartz-2.7.9.dmg (it will end up in the same location).

And then after having activated the "X11" icon, any graphical window, such as *gvim*'s, should display on your host screen.

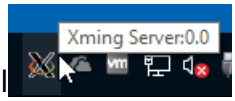
3.1.3. Windows

You have to use PuTTY, as suggested in the previous ***"Image and SDK for porter"*** *document*, together with a running X server such as Xming (https://sourceforge.net/projects/xming/files/latest/download).

Before connecting with PuTTY as usual, you have to go to "Connection" → "SSH" → "X11" and check the "Enable X11 forwarding" checkbox.



Then, if Xming is installed and running, as displayed in the bottom right of the screen:



any graphical window, such as *gvim's*, should display on your screen.

3.2. Installing new applications (IDE...)

The container has access to the whole Linux Debian distribution library, and can benefit of only package available for it.

For instance, to install the popular Eclipse IDE, please type:

```
$ sudo apt-get install eclipse
```

And then, using the method described in section 3.1, you can run it on your host screen by just typing:

```
$ eclipse
```

4. Creating your own hybrid application

In this section, we describe how to extend the provided templates to create a real application that provides information about CPUs:

- obtain target CPU count;
- obtain target CPU load, by processor;
- display results with a graphical interface.

This will consist in 4 main steps:

- get a minimal template, with a default Service;
- extend the Service to add API verbs (“cpucount”, “cpuload”);
- add a Qt/QML Application to use and display these verbs results;
- package the new Service/Application couple as a **Hybrid Application**.

In this section, we get the detailed steps to modify the templates but the result is also available directly in app-framework-templates/demos.

4.1. Get the default Hybrid application template

Log into the DevKit container, clone the app-framework-templates repository (if not already done) and copy the default Hybrid Application template in the home directory:

```
$ cd $HOME
$ git clone https://github.com/iotbzh/app-framework-templates
$ cp -a app-framework-templates/templates/hybrid-qml ~/cpu-hybrid-qml
$ cd ~/cpu-hybrid-qml
```

4.2. Rename application, add API verbs to Service

First, rename “xxxxxx-hybrid-binding.c” to “cpu-hybrid-qml-binding.c”

```
$ mv xxxxxx-hybrid-qml-binding.c cpu-hybrid-qml-binding.c
```

Then in “cpu-hybrid-qml-binding.c”, line 42, replace “xxxxxx” by “cpu”:

```
.v1 = {
      .info = “cpu hybrid service”,
      .prefix = “cpu”,
      .verbs = verbs
}
```

In "CMakeLists.txt", line 20, change project name to "cpu-hybrid-qml".

```
##############################################################################

project(cpu-hybrid-qml)

cmake_minimum_required(VERSION 3.3)
```

Then, in your text editor, open "cpu-hybrid-binding.c" and do the following modifications:

4.2.1. 'cpucount' verb

At line 18, add required header files:

```
#define GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <json-c/json.h>
```

At line 32, add new API verb "CPUCount" function:

```
}

static void CPUCount (struct afb_req request)
{
      long count = sysconf(_SC_NPROCESSORS_ONLN);
      char count_str[8];

      snprintf (count_str, 8, "%ld", count);
      afb_req_success(request, NULL, count_str);
}

// NOTE: this sample does not use session to keep test a basic as possible
```

At line 47, define new API verb "count":

```
  {"ping",     AFB_SESSION_NONE, ping  , "Ping the binder"},
  {"count",  AFB_SESSION_NONE, CPUCount , "returns number of CPUs on target \
board"},
  {NULL}
```

The "CPUCount()" C function uses standard Linux calls to retrieve the CPU count in a numeric (long) variable, then converts it to a string and returns it as an argument of the default Binder success function (afb_req_success(…)).

The function is then associated to the "count" API verb.

4.2.2. 'cpuload' verb

At line 24, add systemd and <time.h> header files, and some definitions:

```
#include <afb/afb-plugin.h>

#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <systemd/sd-event.h>
#define MAXCPUS 16
static long cpucount;
static long double loadpast[4][MAXCPUS], loadnow[4][MAXCPUS], load[MAXCPUS];

const struct AFB_interface *interface;
```

At line 50, add following helper and API functions:

```
}

static char* LookupStringInFile (char *string, char *filename)
{
      FILE *file;
      char *line;

      file = fopen (filename, "r");
      line = malloc (256);
      /* lookup string on file line, stop there if found */
      while (fgets (line, 256, file)) {
            if (strstr (line, string))
                  break;
      }
      fclose(file);

      return line;
}

static int MeasureCPULoad (sd_event_source *src, uint64_t now, void *data)
{
      sd_event *loop = (sd_event *)data;
      char cpuname[6] = "cpu";
      char num_str[2];
      int num, i;
      char *line;

      /* iterate on each CPU */
      for (num = 0; num < cpucount; num++) {
            /* construct lookup string ("cpu1" e.g.) */
            snprintf (num_str, 2, "%d", num);
            strncat (cpuname, num_str, 2);

            /* lookup string in file, get current load values */
            line = LookupStringInFile(cpuname, "/proc/stat");
            sscanf (line, "%*s %Lf %Lf %Lf %Lf", &loadnow[0][num], \
                  &loadnow[1][num], &loadnow[2][num], &loadnow[3][num]);
            free (line);
```

```
            /* calculate average load by comparing with previous load */
            load[num] = ((loadnow[0][num]+loadnow[1][num]+loadnow[2][num])
              - (loadpast[0][num]+loadpast[1][num]+loadpast[2][num])) /
            ((loadnow[0][num]+loadnow[1][num]+loadnow[2][num]+loadnow[3][num])
              - (loadpast[0][num]+loadpast[1][num]+loadpast[2][num]+\
            loadpast[3][num]));

            /* store current load values as previous ones */
            for (i = 0; i < 4; i++)
                  loadpast[i][num] = loadnow[i][num];
      }

      /* re-fire the function in 5 seconds ("now" + 5 microseconds) */
      sd_event_add_time (loop, &src, CLOCK_MONOTONIC, now+5000000, 0,
                                    MeasureCPULoad, loop);

      return 1;
}

static void CPULoad (struct afb_req request)
{
      const char *num_str = afb_req_value (request, "num");
      int num;
      char load_str[4];

      /* no "num" argument was given : fail */
      if (!num_str)
            afb_req_fail (request, "failed", "please provide CPU number as \
                                                argument");

      /* prevent negative number, or superior to CPU count */
      num = atoi (num_str);
      if ((num < 0) || (num >= cpucount)) {
            afb_req_fail (request, "failed", "invalid CPU number argument");
            return;
      }

      /* convert load to readable format and return it */
      snprintf (load_str, 4, "%.0Lf%%", load[num]*100);
      afb_req_success(request, NULL, load_str);
}
```

At line 133, define new API verb:

```
  {"count",  AFB_SESSION_NONE, CPUCount , "returns number of CPUs on target
board"},
  {"load",   AFB_SESSION_NONE, CPULoad  , "returns designated CPU load on
target board"},
  {NULL}
};
```

At line 149, initialize static variables and callbacks in the Service registration function:

```
const struct AFB_plugin *pluginAfbV1Register (const struct AFB_interface *itf)
{
      interface = itf;
      sd_event *loop;
      sd_event_source *src;
      uint64_t now;
      int i;

      /* get CPU count, limiting it to MAXCPUS (default : 16) */
      cpucount = sysconf(_SC_NPROCESSORS_ONLN);
      if (cpucount > MAXCPUS)
            cpucount = MAXCPUS;

      /* initialize past load to 0 for each CPU */
      for (i = 0; i < cpucount; i++)
            loadpast[0][i] = loadpast[1][i] = loadpast[2][i] \
                                              = loadpast[3][i] = 0;

      /* register the CPU load measuring function, fires immediately ("now") */
      loop = afb_daemon_get_event_loop (interface->daemon);
      sd_event_now (loop, CLOCK_MONOTONIC, &now);
      sd_event_add_time (loop, &src, CLOCK_MONOTONIC, now, 0,
                                    MeasureCPULoad, loop);

      return &plugin_desc;
}
```

This example is slightly more complicated, and illustrates more advanced concepts:

- getting argument strings from the request;
- initializing variables, launching custom functions at startup;
- using the Binder event loop with callbacks.

Some remarks on the code:

- Our new "load" verb will require a "num" argument matching the CPU we want to query ("0" for "CPU0", "1" for "CPU1"…).

  For this, we use the "afb_req_value()" helper function to retrieve it as a string. If no argument is given, or the argument is invalid (-1 or 200 for instance), we use the"afb_req_fail()" helper function to give an explicit error message.

- Measuring CPU load effectively consists in comparing 2 load values (past and current) within a certain time interval.

  For this, we create a recurring "MeasureCPULoad()" function which will re-fire itself every 5 seconds to measure loads and compare it with the former ones. We use the "sd_event_add_time()" function for this purpose, passing "5000000" for 5 seconds.

- A recurring function needs to be run at least once at startup.

  For this, we use the mandatory "pluginAfbV1register()" function, run at Binder startup. We initialize CPU count and load variables ("cpucount", "loadpast[4][...]") and most importantly retrieve the Binder event loop with "afb_daemon_get_event_loop()" to run our first occurence of "MeasureCPULoad()". It will then run indefinitely.

4.3. Add a Qt/QML frontend

We will now create a QML frontend for our service.

The full source code can be found in app-framework-templates/demos/cpu-hybrid-qml/cpu-hybrid-qml-app.qml. Let's copy it into our project and remove old qml file:

```
$ cp ~/app-framework-templates/demos/cpu-hybrid-qml/cpu-hybrid-qml-app.qml .
$ rm xxxxxx-hybrid-qml-app.qml
```

The most important bits are the following, line 9:

```
property string port_str: Qt.application.arguments[1]
property string token_str: Qt.application.arguments[2]
property string address_str: "ws://localhost:"+port_str+"/api?token="+token_str
```

then later:

```
WebSocket {
      url: address_str
[...]
      active: true
}
```

This forges a WebSocket URL string (2 components, port and token, are predefined and passed by the Application Framework manager) and passes it as the "url" property of our WebSocket widget. The "active" property makes sures the connection gets initialized when the application is launched.

The following code:

```
Button {
text: "Re-count CPUs"
onClicked: {
      verb_str = "count"
      request_str = '[' + msgid_enu.call + ',"99999","' + api_str+'/'+verb_str
+ '", null ]'
      websocket.sendTextMessage (request_str)
}
```

defines a Button widget which, when clicked on, creates a request with the "cpu" API and "load" verb, and sends it.

Finally, the following code

```
WebSocket {
[...]
      onTextMessageReceived: {
            var message_json = JSON.parse (message)
            var request_json = message_json[2].request
            if (verb_str == "count")
                  count = request_json.info
}
```

parses the final response as JSON and, if the verb was "count", retrieves the "info" field which contains the answer (1, 2, 3…).

4.4. Package our Hybrid application

Run the following commands:

```
$ mkdir build; cd build
$ cmake ..
|snip]
$ make
Scanning dependencies of target cpu-hybrid-qml
[ 33%] Building C object CMakeFiles/cpu-hybrid-qml.dir/cpu-hybrid-qml-bind-
ing.c.o
[ 66%] Linking C shared module cpu-hybrid-qml.so
[ 66%] Built target cpu-hybrid-qml
Scanning dependencies of target widget
[100%] Generating cpu-hybrid-qml.wgt
NOTICE: -- PACKING widget cpu-hybrid-qml.wgt from directory package
[100%] Built target widget
```

The package "cpu-hybrid-qml.wgt" is then produced in the build directory and ready to be installed and run on the target.

4.5. Install our Hybrid application on the target

Copy the package to the target board:

```
$ scp cpu-hybrid-qml.wgt root@$BOARDIP:~/
```

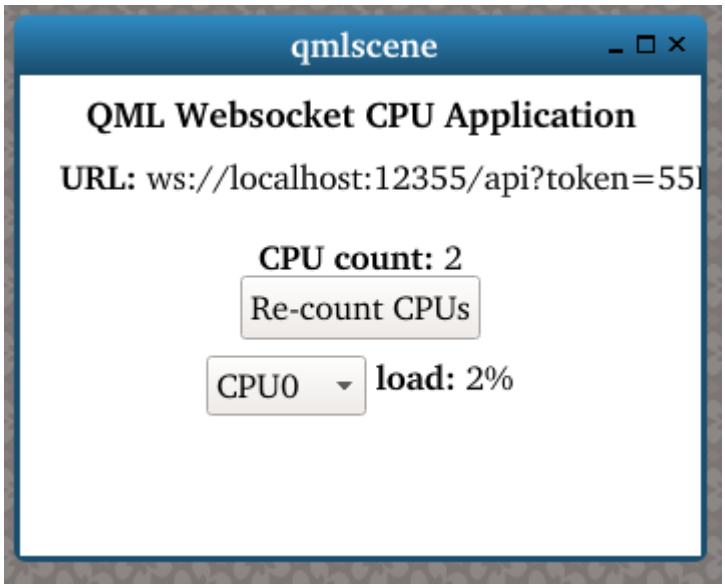
Then open a new session on the target and install the package:

```
$ ssh root@$BOARDIP
# afm-util install cpu-hybrid-qml.wgt
{ "added": "cpu-hybrid-qml@0.1" }
```

Next, run the application:

```
# afm-util start cpu-hybrid-qml@0.1
11
```

The following UI appears and displays the number of CPUs and the load for the selected CPU, updated every 5 seconds:



4.6. Add a HTML5 frontend

We will now create a HTML5 frontend for our service.

For this, we will re-use the layout of the HTML5 template found in app-framework-templates/templates/hybrid-html5.

The most noticeable steps are the following:

- duplicate the template app-framework/templates/hybrid-html5 to a new directory ~/cpu-hybrid-html5
- copy cpu-hybrid-qml-binding.c to ~/cpu-hybrid-html5/binding/cpu-hybrid-html5-binding.c and remove the previous binding code
- in CMakeLists.txt, line 20, rename "xxxxxx-hybrid-html5" to "cpu-hybrid-html5", and line 64, rename "xxxxxx-hybrid-binding.c" to "cpu-hybrid-html5-binding.c"
- in app/etc/AppDefaults.js, line 24, rename "xxxxxx-hybrid-html5" to "cpu-hybrid-html5"
- in package.json, line 2, rename "xxxxxx-hybrid-html5" to "cpu-hybrid-html5"

4.6.1. 'cpucount' verb

Open "app/Frontend/pages/SampleHome/SampleHome.html" and, at line 19, replace:

```
 <submit-button class="session-button {{ctrl.class.refresh}}" icon="fi-arrows-
compress" label="Refresh" clicked="ctrl.RefreshSession" ></submit-button>
```

with:

```
 <submit-button class="session-button {{ctrl.class.refresh}}" label="CPU
status" clicked="ctrl.CpuCount" ></submit-button>
```

Open "app/Frontend/pages/SampleHome/SampleHome.js" and, at line 51, replace:

```
scope.RefreshSession = function() {
    console.log ("RefreshSession");
    AppCall.get ("xxxxxx", "ping", {/*query*/}, scope.OnResponse, scope.Inval-
idApiCall);
}
```

with:

```
scope.CpuCount = function() {
    AppCall.get ("cpu", "count", {/*query*/}, scope.OnResponse, scope.Invalid-
ApiCall);
}
```

then at line 29, replace:

```
case 'PING':
    break;
```

with:

```
case 'COUNT':
    // Get CPU count from response
    var cpucount = jresp.request.info;
    Notification.success ({message: "CPU count: " + cpucount , delay: 3000});
    break;
```

We first associate a HTML button with the «CPUCount()» function found in the JavaScript code. Then, in the JavaScript code, we make this function use the «AppCall.get» helper function to fire the «cpu/count» request.

The response will then be parsed in the generic «OnResponse()» function, where the verb is converted uppercase («count» → «COUNT»). And then, the response «info» field containing CPU count will be displayed on the page.

4.6.2. 'cpuload' verb

Open "app/Frontend/pages/SampleHome/SampleHome.js" and, at line 33, add:

```
    Notification.success ({message: "CPU count: " + cpucount , delay: 3000});
    // Iterate and fire CPU load request for each CPU
    for (var i = 0; i < cpucount; i++) {
        AppCall.get ("cpu", "load", {num: i}, scope.OnResponse,
scope.InvalidApiCall);
    }
    break;
```

Then at line 39, add:

```
case 'LOAD':
    // Get CPU load from response
    var cpuload = jresp.request.info;
    Notification.success ({message: "CPU load: " + cpuload , delay: 3000});
    break;
```

Now that we have the CPU count number, we can iterate on it to fire one «load » request per CPU («cpu/load {num:0}», «cpu/load{num:1}»….).

Then, as we did before, we parse the response and display its result on the page.

4.7. Package our hybrid HTML5 application

In "config.xml.in", line 2, replace "xxxxxx-hybrid-html5" with "cpu-hybrid-html5".

Then, prepare the directory for gulp processing:

```
$ npm install
[snip]
```

Then, run the following commands:

```
$ mkdir build; cd build
$ cmake ..
|snip]
$ make
Scanning dependencies of target cpu-hybrid-html5
[ 33%] Building C object CMakeFiles/cpu-hybrid-html5.dir/binding/cpu-hybrid-
html5-binding.c.o
[ 66%] Linking C shared module cpu-hybrid-html5.so
[ 66%] Built target cpu-hybrid-html5
Scanning dependencies of target widget
[100%] Generating cpu-hybrid-html5.wgt
[snip]
[100%] Built target widget
```

A "cpu-hybrid-html5.wgt" installable package will be produced in the current directory.

4.8. Install our Hybrid application on the target

Copy the package to the target board:

```
$ scp cpu-hybrid-html5.wgt root@$BOARDIP:~/
```

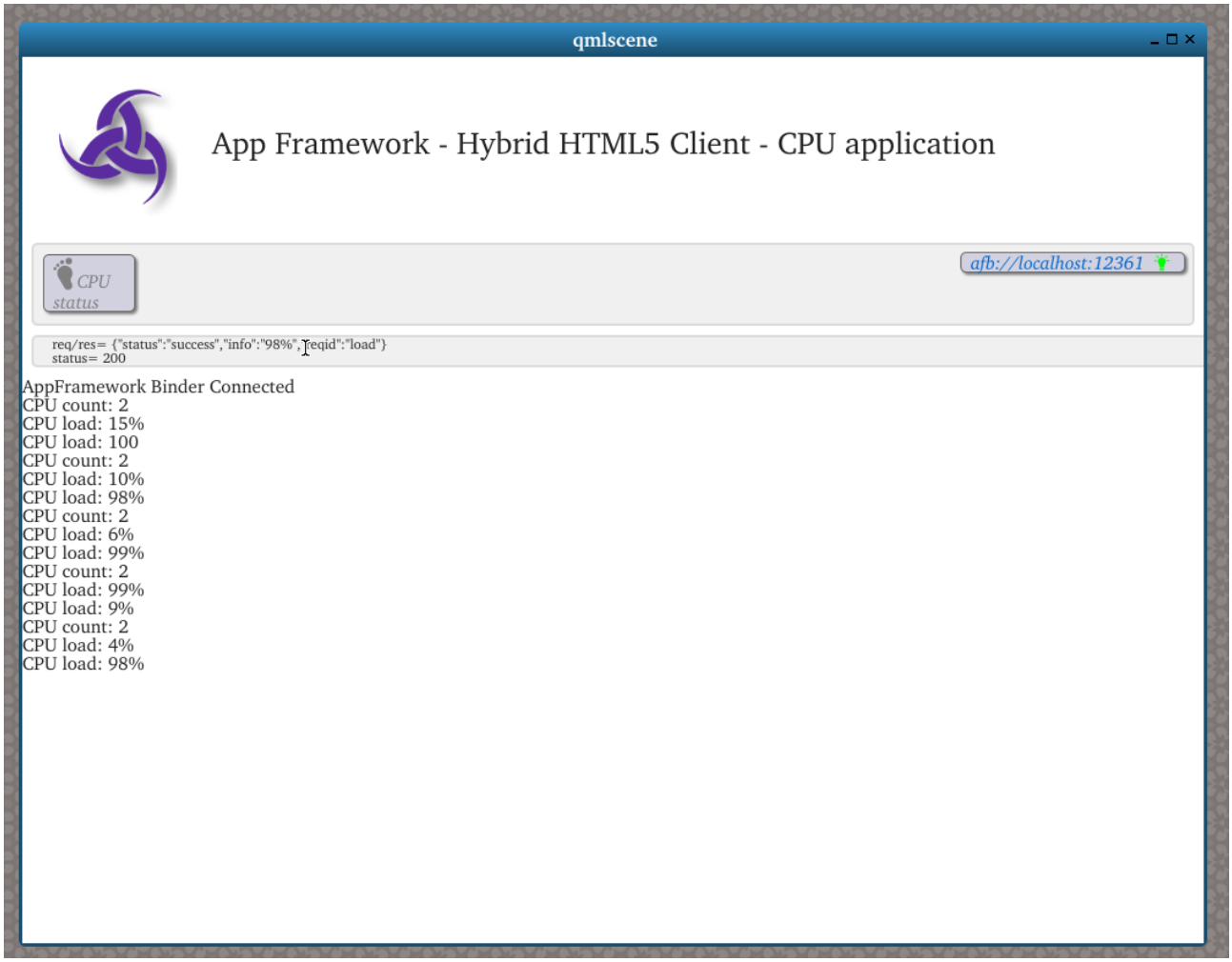
Then open a new session on the target and install the package:

```
$ ssh root@$BOARDIP
# afm-util install cpu-hybrid-html5.wgt
{ "added": "cpu-hybrid-html5@0.1" }
```

Next, run the application:

```
# afm-util start cpu-hybrid-html5@0.1
12
```

The following UI appears:



Each time the user hits the “CPU” button, CPU count and loads are displayed.