Creating HTML5 apps for AGL

Lorenzo Tilve / Roger Zanoni

Igalia - AGL F2F @ Panasonic Center Tokyo - 20.10.2022

igalia

Agenda

- About us
- Goals of AGL Web Runtime
- Chromium and Web Application Manager (WAM)
- How to create and debug web applications in AGL
- Status and future plans

About Igalia

- Open Source Consultancy with HQ in Galicia, Spain
- Over 120 employees around the world
- Web rendering and browsers experience in Chromium, WebKit, WPE and Firefox, Compilers, JavaScript engines (V8, JSC), Graphics, Multimedia, Kernel, Accessibility

Goals of AGL Web Runtime

Provide full Web Platform support into AGL platform

- Not framework specific. Any front-end framework allowed.
- Out-of-the-box compatibility with standard web APIs

Goals of AGL Web Runtime

- Native-like experience for web-applications
- Smooth integration with V2C services

Goals of AGL Web Runtime

Potential to reach a big community of developers

- Development tools already available and well known
- Interoperability with other frontend and backend services

Chromium and WAM

WAM = Web Application Manager

Lorenzo Tilve / Roger Zanoni, Igalia

Chromium and WAM

WAM is the web application runtime for LG Electronics webOS

Open-sourced as part of webOS Open Source Edition

Lorenzo Tilve / Roger Zanoni, Igalia

Chromium and WAM

Built on top of Google Chromium

- Using Google/Igalia upstream Ozone Wayland backend
- State of the art GPU acceleration
- Solution tested in multiple embedded devices
- Support for cloud-native technologies

Chromium and WAM

WAM provides:

- Browser-like architecture for web applications
- Broad optimizations of memory usage, application launch time
- Life-cycle control of web applications
- Extensible both in web platform and system integration

Reference Hardware

- Detailed startup documentation:
  - https://wiki.automotivelinux.org/start/getting-started
- Different hardware target architectures can be used to build and test AGL:
  - Renesas R-Car starter kit: h3ulcb / m3ulcb
  - Intel 64-Bit Hardware Platforms: intel-corei7-64
  - RaspberryPi: raspberrypi3 / raspberrypi4
  - Emulation with QEMU / Virtualbox: qemux86-64

Building the HTML5 image

- Getting the AGL code (needs depot_tools and Yocto):

```
repo init -b master \
  -u https://gerrit.automotivelinux.org/gerrit/AGL/AGL-repo
repo sync
```

- Configuring the build and compiling all the stack with Yocto:

```
source meta-agl/scripts/aglsetup.sh -f -m <target_architecture> \
  -b build agl-devel agl-demo
bitbake agl-ivi-demo-platform-html5
```

- If you want to work on the current release, check the AGL wiki

Chromium/WAM Yocto layer

- This fetches and builds the recipes of the meta-agl-demo Yocto layer:
  - WAM https://gerrit.automotivelinux.org/gerrit/gitweb?p=AGL/meta-agl-demo.git;a=blob;f=recipes-wam/wam/wam_git.bb
  - Chromium https://gerrit.automotivelinux.org/gerrit/gitweb?p=AGL/meta-agl-demo.git;a=blob;f=recipes-wam/chromium/chromium_git.bb

Flashing the built images

- The generated image will be located at:

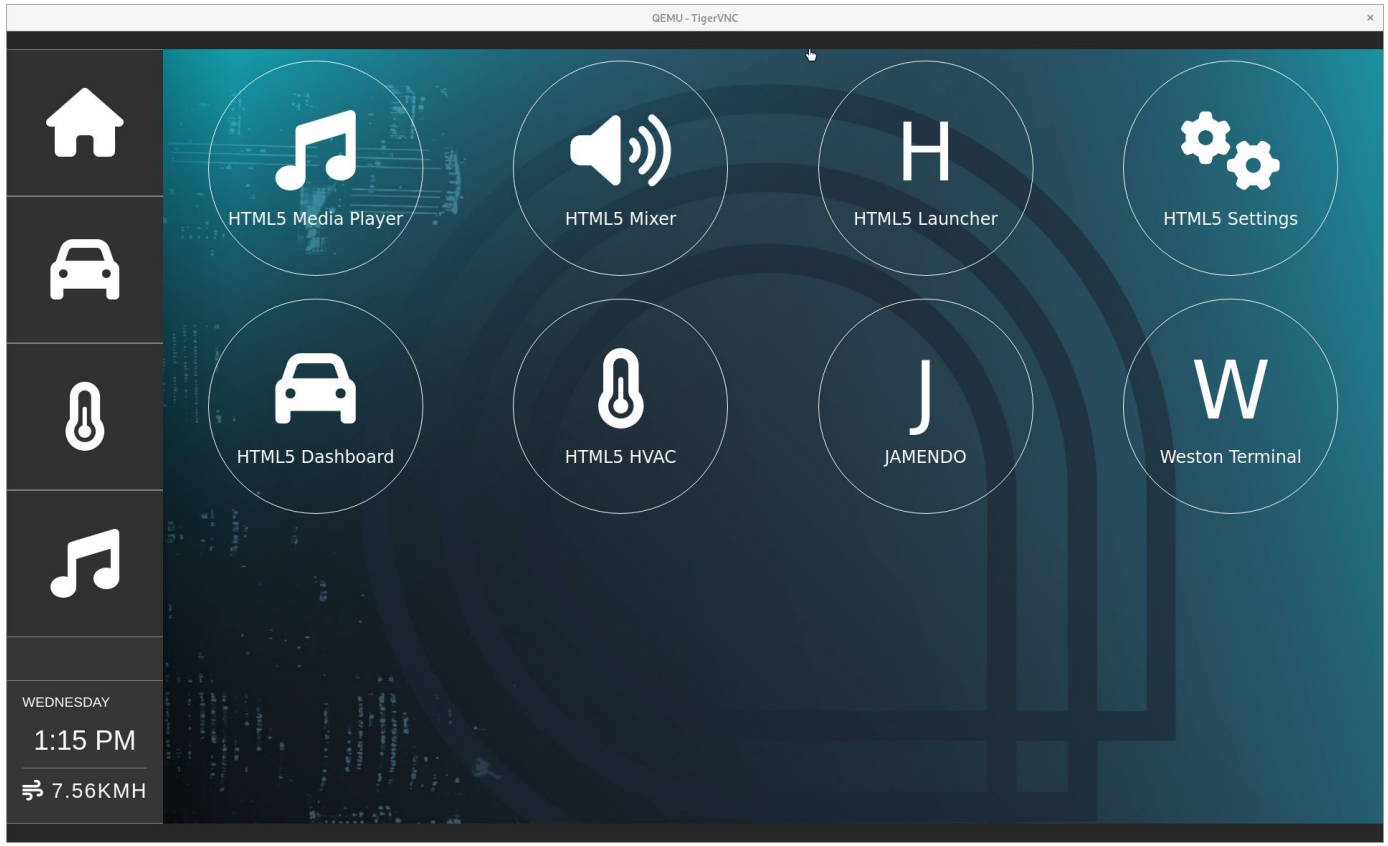
```
build/tmp/deploy/images/<arch>/agl-ivi-demo-platform-html5-<arch>.wic.xz
```

- It can be flashed with the following command:

```
xzcat path/to/arch/agl-ivi-demo-platform-html5-<arch>.wic.xz \
  | sudo dd of=/dev/mmcblk0 bs=4M && sync
```

- Then the SD card can be inserted on the device and booted for the first time.

The html5 image



Structure of a webapp

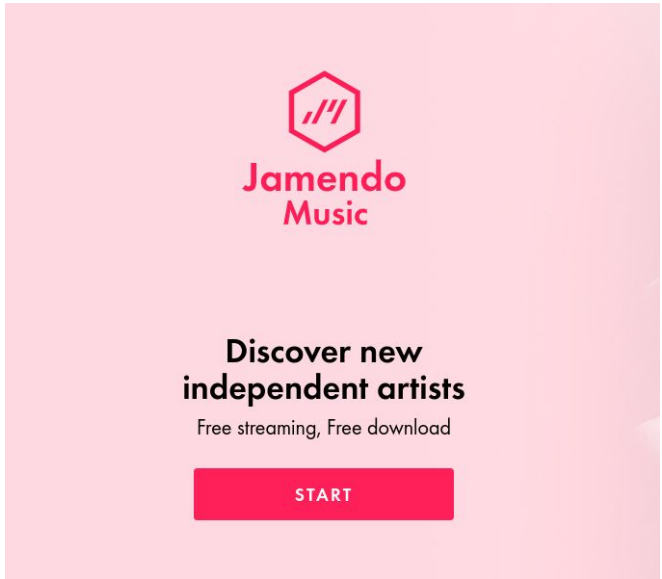
- They can be as simple as:
  - An appinfo.json file that contains metadata such as the application id, title, description, application type, the main file and icon.
  - The source with any of the HTML application resources and a LICENSE file
- There is no dependency of any specific web technology:
  - Pure HTML+JavaScript, WASM
  - Any frameworks or libraries as Enact, AngularJS, React...

Steps for creating a webapp

- Create the new git repository (or local directory) for the new application
  - Add a main html file, all needed resources, etc
- Choose and add a license file
  - It will be needed later when we create the recipe
- Add an appconfig.json file
- Create a Yocto recipe and add it to AGL tree

Example application: Jamendo

As an example, let's create an application that simply redirects the user to jamendo.com



appinfo.json

```
{
  "id": "webapps-jamendo",
  "title": "JAMENDO",
  "description": "Free independent music streaming",
  "version": "0.0.0",
  "vendor": "Igalia, S.L.",
  "type": "web",
  "main": "index.html",
  "uiRevision": "2",
  "icon": "icon.svg"
}
```


index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Jamendo AGL</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
maximum-scale=1.0, user-scalable=no"/>
  </head>
  <body>
    <script>
      window.location = "https://www.jamendo.com"
    </script>
  </body>
</html>
```


Creating the Yocto recipe

- Create a recipe folder in /meta-agl-demo/recipes-demo/
- Create the recipe file: html5-jamendo_git.bb
- Add the basic information (You can check the glossary for the meaning of each variable, and check Yocto guides):

```
SUMMARY     = "Free independent music streaming"
HOMEPAGE    = "<homepage>"
SECTION     = "apps"
LICENSE     = "Apache-2.0"
LIC_FILES_CHKSUM = "file://LICENSE;md5=b1e01b26bacfc2232046c90a330332b3"

PV      = "1.0+git${SRCPV}"
S       = "${WORKDIR}/git"
B       = "${WORKDIR}/build"
```


Creating the Yocto recipe

- Setup the fetcher of the app source:

```
SRC_URI = "git://github.com/rogerzanoni/html5-jamendo;protocol=https;branch=main"
SRCREV = "51624ff085bd5d57a7dc4b196bfd567f91766318"
```

- It's possible to set different kinds of fetchers, for example, setting it up to fetch from a local repository:

```
SRC_URI = "git:///home/<path-to-source-repository>/html5-jamendo;protocol=file;branch=main"
SRCREV = "51624ff085bd5d57a7dc4b196bfd567f91766318"
```


Creating the Yocto recipe

- During development it may be useful to make the recipe skip fetching the source and using a local source directory
- To do so, create a `local.dev.inc` file in your `<AGL_ROOT>/build/conf` directory:

```
INHERIT += "externalsrc"
EXTERNALSRC_pn-<recipe-name> = "/path-to-your-source-dir/"
# For instance:
# EXTERNALSRC_pn-chromium = "/path-to-the-source-dir/chromium91/"
# EXTERNALSRC_pn-wam = "/path-to-the-source-dir/wam/"
# EXTERNALSRC_pn-<webapp> = "/path-to-the-source-dir/<webapp>/"
```

- Then the option "agl-localdev" need to be passed to aglsetup.sh:

```
source meta-agl/scripts/aglsetup.sh -f -m <target_architecture> \
  -b build agl-devel agl-localdev agl-demo
bitbake agl-ivi-demo-platform-html5
```


Creating the Yocto recipe

- Finally, setup the build and install instructions:

```
inherit pythonnative agl-app

AGL_APP_TEMPLATE = "agl-app-web"
AGL_APP_ID = "webapps-jamendo"
AGL_APP_NAME = "JAMENDO"

WAM_APPLICATIONS_DIR = "${libdir}/wam_apps"

do_install() {
  install -d ${D}${WAM_APPLICATIONS_DIR}/${PN}
  cp -R --no-dereference --preserve=mode,links ${S}/* \
${D}${WAM_APPLICATIONS_DIR}/${PN}
}
```


Changing the html5 packagegroup

- The AGL build system needs to know where the new application is to bake it into new images. For that, an entry needs to be added to

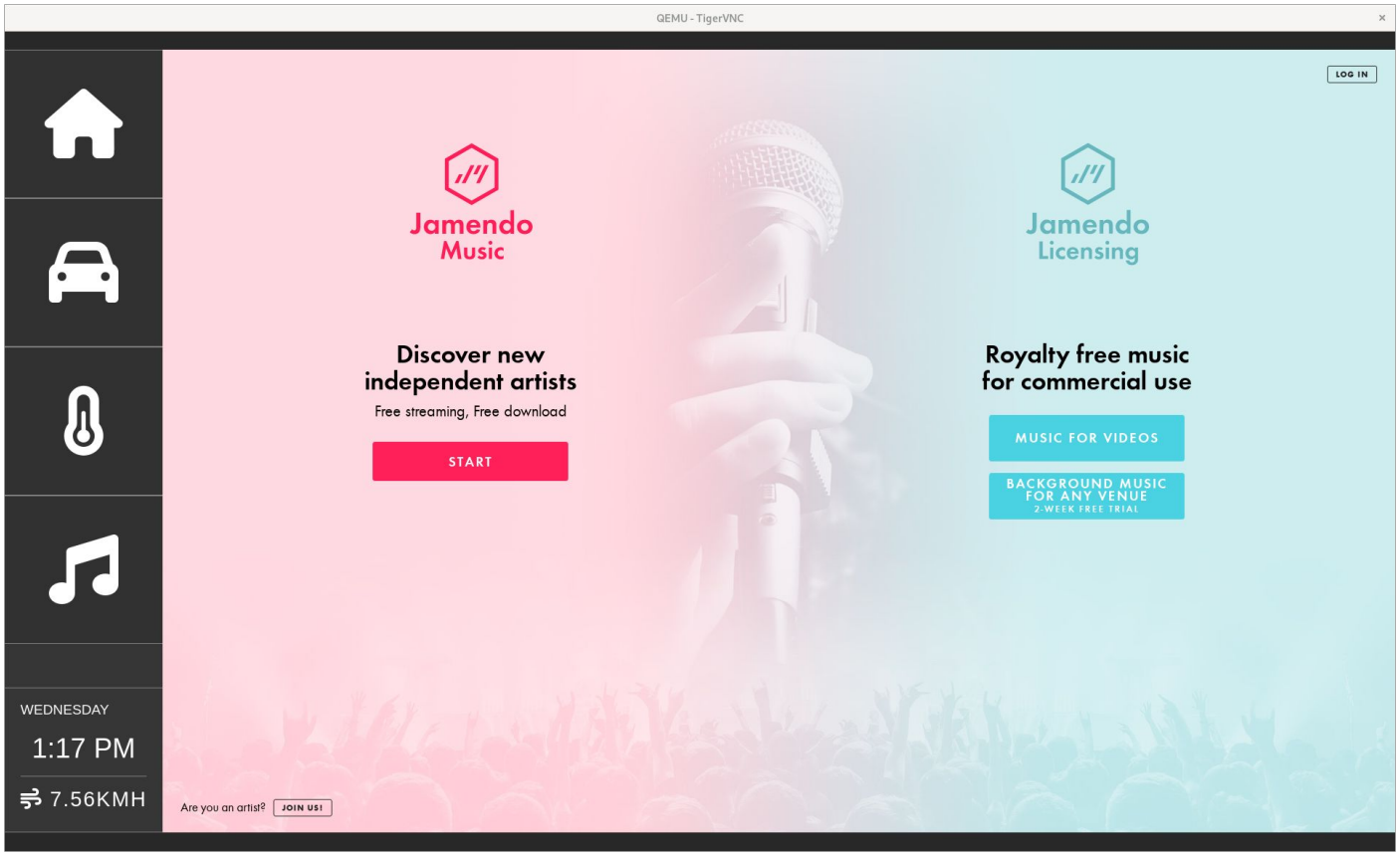
```
<AGL root>/recipes-platform/packagegroups/packagegroup-agl-demo-platform-html5.bb
```

```
...
AGL_APPS = " \
    ...
    html5-settings \
    html5-aquarium \
    html5-youtube \
    html5-jitsi \
    html5-examples \
==> html5-jamendo \
    "
...
```


Testing the new app on a QEMU image

- After the packagegroup setup, the application should be ready to be build and included in the image.
  - Use "qemux86-64" as architecture and build as described in previous slides
  - Run it with `runqemu qemux86-64 kvm publicvnc slirp`
  - Connect to the running instance using a vnc client

Testing the new app on a QEMU image

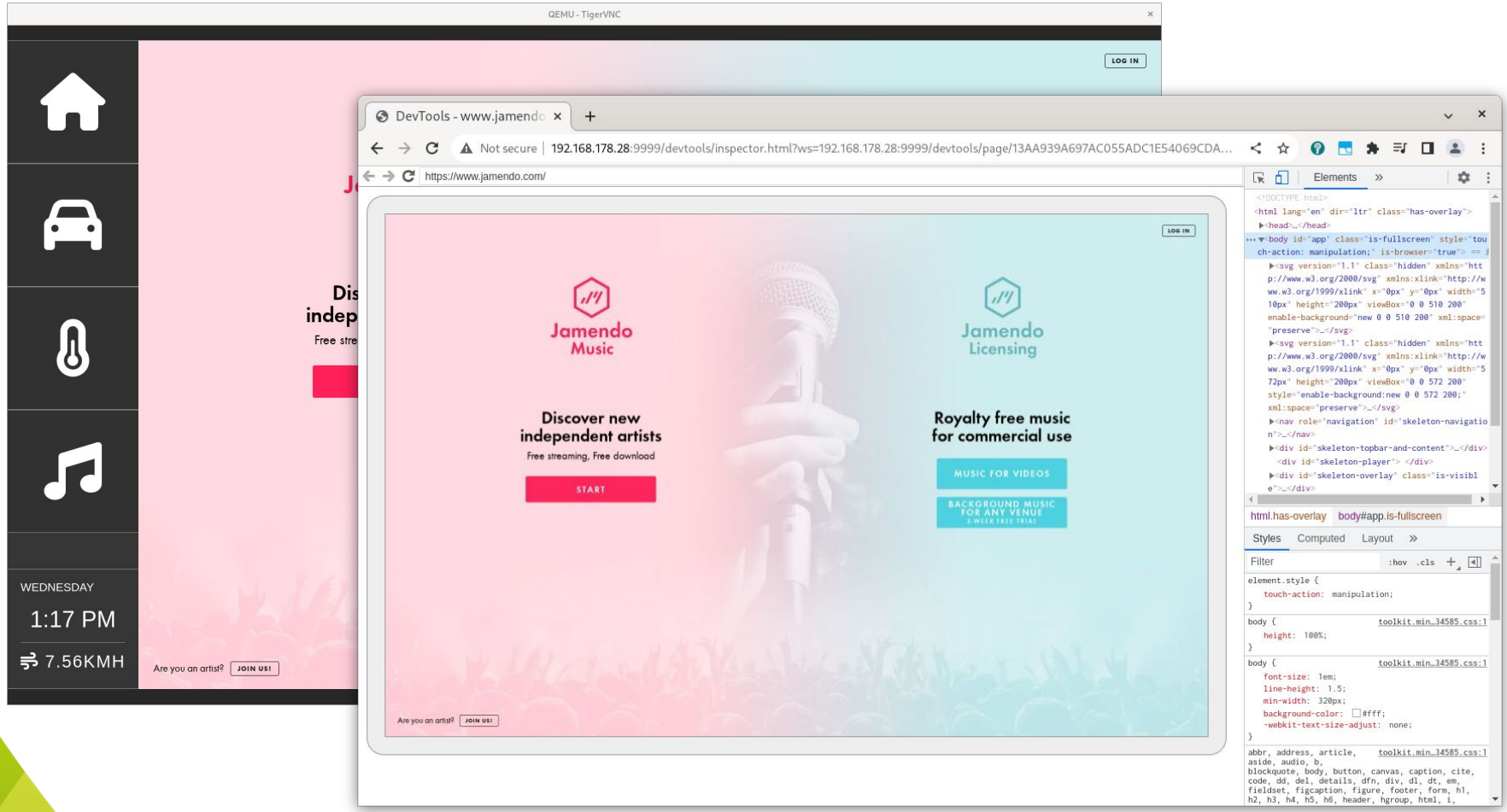


Remote DevTools

- Auto enabled when the image was built with agl-devel, at port 9998
- To test while running a QEMU image, the network options can be overridden to enable port forwarding by using the following environment variable before running runqemu:

```
export QB_SLIRP_OPT="-netdev user,id=net0,hostfwd=tcp::2222-:22,hostfwd=tcp::9999-:9998"
```


Remote DevTools



Interacting with services

- The current HVAC AGL demos uses the kuksa.val server
  - KUKSA.val provides in-vehicle software components for working with in-vehicle signals modelled using the COVESA VSS data model
    - VSS can be used by application to communicate information around the vehicle
- Currently clients use websockets connect to a service listening on port 8090

Communicating with the server

- Connect to the server using a websocket:

```
var socket = new WebSocket('wss://localhost:8090');
```

- After connecting, to be able to use the signals, the client must be authorized:

```
socket.onopen = function(event) {
    init();
}

function init() {
    authorize();
    ...
}
```

```
function authorize() {
    var data = {
        action: 'authorize',
        tokens: <authToken>,
        requestId: <requestId>,
    };
    socket.send(JSON.stringify(data));
}
```


Communicating with the server

- <authToken> is the JSON Web Token of your client
  - On real applications each application must have their own tokens, but for development the kuksa.val keys can be used
  - kuksa.val repository provides a doc with more information
- <requestId> is a unique id set by the client and returned by the server in the response
- More info about the protocol can be found in kuksa.val documentation and VISS specs

Subscribing

- After the authentication, the client can subscribe to vehicle signals to receive notifications of value changes:

```
function subscribe(path) {
    var data = {
        action: 'subscribe',
        tokens: <authToken>,
        requestId: requestId,
        path: path,
    };
    socket.send(JSON.stringify(data));
}
```

```
socket.onmessage = function(event) {
    var jsonData = JSON.parse(event.data);

    if (jsonData.action == 'get' ||
        jsonData.action =='subscription') {
        ...
    }
}
```

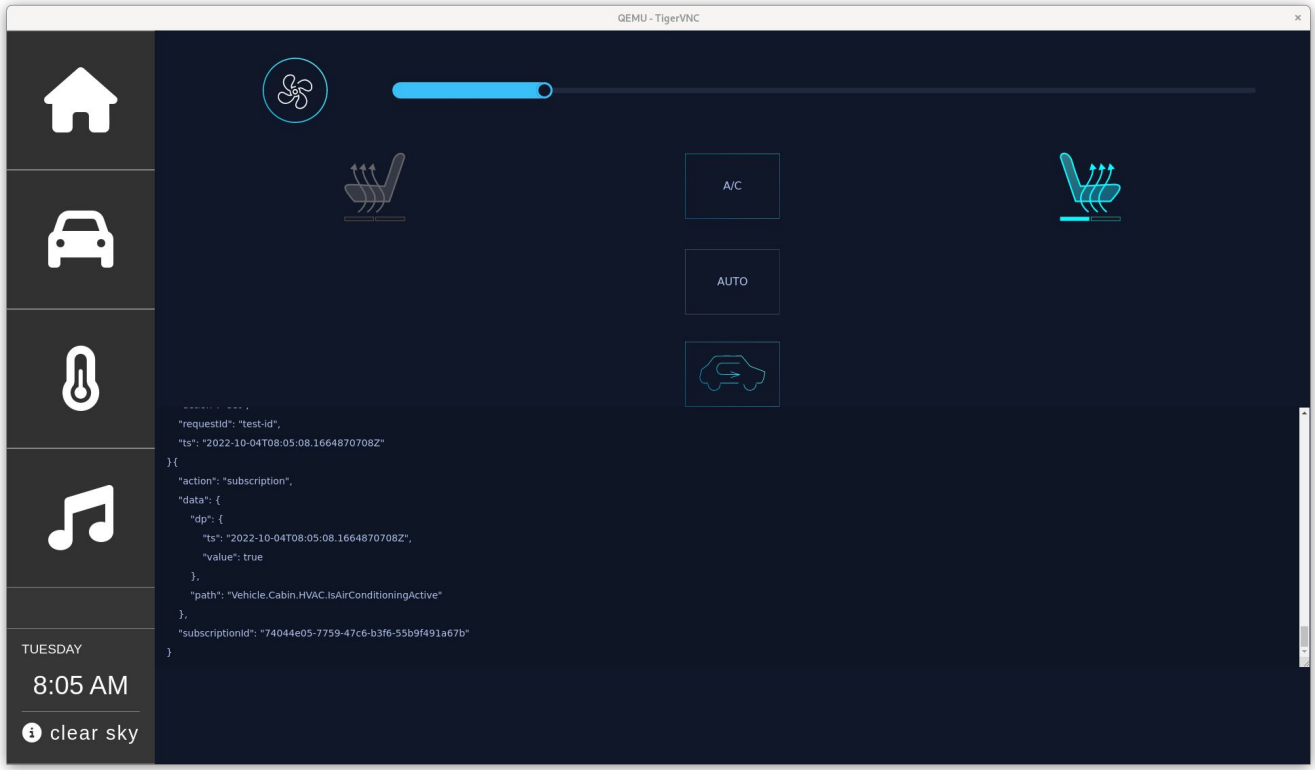

Setting/Getting data

- Similar signals can be used to set/retrieve data:

```
function get(path) {
    var data = {
        action: 'get',
        tokens: <authToken>,
        requestId: <requestId>,
        path: path,
    };
    socket.send(JSON.stringify(data));
}
```

```
function set(path, value) {
    var data = {
        action: 'set',
        tokens: <authToken>,
        requestId: <requestId>,
        path: path,
        value: value,
    };
    socket.send(JSON.stringify(data));
}
```


HVAC demo



Lorenzo Tilve / Roger Zanoni, Igalia

Ongoing work and future plans

- Update current hvac demo to use kuksa.val server API
- Continue integrating more webapps
- Update chromium to milestone 94
- Lower prio:
  - Experiment with kuksa.val gRPC API with webapps
  - Propose APIs for other services
  - Bring back chromium as an app

Ongoing work and future plans

gRPC api sample: kuksa.val client

- Not currently used by the demos
- kuksa.val project provides a protobuf interface that can be used to generate code that interacts with the service

```
// The connecting service definition.
service kuksa_grpc_if {
  rpc get (GetRequest) returns (GetResponse) {}
  rpc set (SetRequest) returns (SetResponse) {}
  rpc subscribe (stream SubscribeRequest) returns (stream
SubscribeResponse) {}
  rpc authorize (AuthRequest) returns (AuthResponse) {}
}
```

```
message AuthRequest {
  string token = 1;
}
```


Ongoing work and future plans

- Generating the code:

```
protoc -I=. proto/kuksa.proto --js_out=import_style=commonjs:<out_dir>
--grpc-web_out=import_style=commonjs,mode=grpcwebtext:<out_dir>
```

- Authenticating

```
var messages = require('./gen/proto/kuksa_pb.js');
var services = require('./gen/proto/kuksa_grpc_web_pb.js');

var target = "localhost:8090";
var client = new services.kuksa_grpc_ifClient(target);

function init() {
    var request = new messages.AuthRequest();
    request.setToken(authToken);
    client.authorize(request);
}
```


Ongoing work and future plans

- Standard way of interacting with the services
  - protobuf interfaces can be used to generate code for multiple languages
- Other services can define similar interfaces
- Needs a proxy service (ex: Envoy for gRPC-web)
  - More on The state of gRPC in the browser
- Maintenance cost

Sample code

- Jamendo
  - App https://github.com/rogerzanoni/html5-jamendo
  - Recipe https://github.com/rogerzanoni/html5-jamendo-recipe
- kuksa.val HVAC
  - App https://github.com/rogerzanoni/html5-tailwind-hvac
  - Recipe https://github.com/rogerzanoni/html5-tailwind-hvac-recipe

Thanks

Contact:

- Jose Dapena - jdapena@igalia.com
- Lorenzo Tilve - ltilve@igalia.com
- Roger Zanoni - rzanoni@igalia.com


igalia