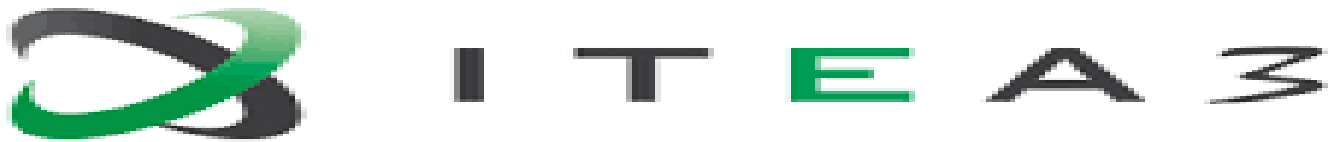


Kuksa User Manual.



SI No.	Author	Version	Date	verified by	Remark
1.	Kirubel	0.2	28.07.2019	Nill	No remarks

Contents

1. Introduction to kuksa

- 1.1. Need of kuksa
- 1.2. Advantage of kuksa
- 1.3. How do we use kuksa

2. Kuksa App IDE

2.1 Getting Started with the App IDE

2.1.1 Required System Configuration

2.1.2 How to set up the IDE

2.1.3 Writing your first Kuksa In-Vehicle App

2.1.4 Writing your first Kuksa Cloud App

2.1.5 Testing your Apps

2.1.6 Deploying your apps

2.1.7 Install your In-Vehicle App in your car (see In-Vehicle platform Getting Started)

2.1.8 Install your Cloud App in your Cloud (see Cloud platform Getting Started)

2.2 Some Kuksa App examples

3. Kuksa In-Vehicle platform

3.1 Getting started with the In-Vehicle platform

3.1.1 Required System Configuration (HW/SW)

3.1.2 Set up the platform

3.1.2.1 using a RPi

3.1.2.2 using XXX?

3.1.3 Connect the platform to a Kuksa portal

3.2.4 Search for an In-Vehicle App

3.1.5 Install an In-Vehicle App

3.1.6 Test the In-Vehicle App

3.2 Configure the In-Vehicle platform

3.3 Overview of the In-Vehicle platform and its architecture

3.4 Overview of the Kuksa In-Vehicle API

4. Kuksa Cloud Platform

- 4.1 Getting started with the Cloud platform
 - 4.1.1 Required System Configuration
 - 4.1.2 Installing and testing the Cloud platform
 - 4.1.3 Installing a Cloud App and its In-Vehicle App
 - 4.1.4 Testing the Cloud App
- 4.2 Configuring the Cloud platform
- 4.3 Overview of the Cloud platform and its architecture
- 4.4 Overview of the Kuksa Cloud API
- 4.5 Marketplace presentation and features

1. Introduction To Kuksa

1.1 Need of Kuksa:

Because today's software-intensive automotive systems are still developed in silos by each car manufacturer or OEM in-house, long-term challenges in the industry are yet unresolved. Establishing a standard for car-to-cloud scenarios significantly improves comprehensive domain-related development activities and opens the market to external applications, service provider, and the use of open source software wherever possible without compromising security. Connectivity, OTA maintenance, automated driving, electric mobility, and related approaches increasingly demand technical innovations applicable across automotive players.

1.2 Advantage of Kuksa:

The open and secure Eclipse Kuksa project will contain a cloud platform that interconnects a wide range of vehicles to the cloud via in-car and internet connections. This platform will be supported by an integrated open source software development environment including technologies to cope especially with software challenges for vehicles designed in the IoT, Cloud, and digital era.

1.3 How do we use Kuksa

The kuksa ecosystem will provide a comprehensive environment across various frameworks and technologies for *the in-vehicle platform, the cloud platform, and an app development IDE* - that is, the complete tooling stack for the connected vehicle domain [see Figure 1 below](#). Essential to this environment will be the capabilities for collecting, storing, and analysing vehicle data in the cloud as well as the transmission of diverse information such as cloud calculation results (e.g. improved routing), software maintenance updates or even complete new applications. While many IoT solutions exist in the Eclipse IoT ecosystem, Eclipse Kuksa combines the necessary existing technologies and fills the gaps for the specific requirements of the connected embedded real-time nature of the automotive domain.[see here for more details](#).

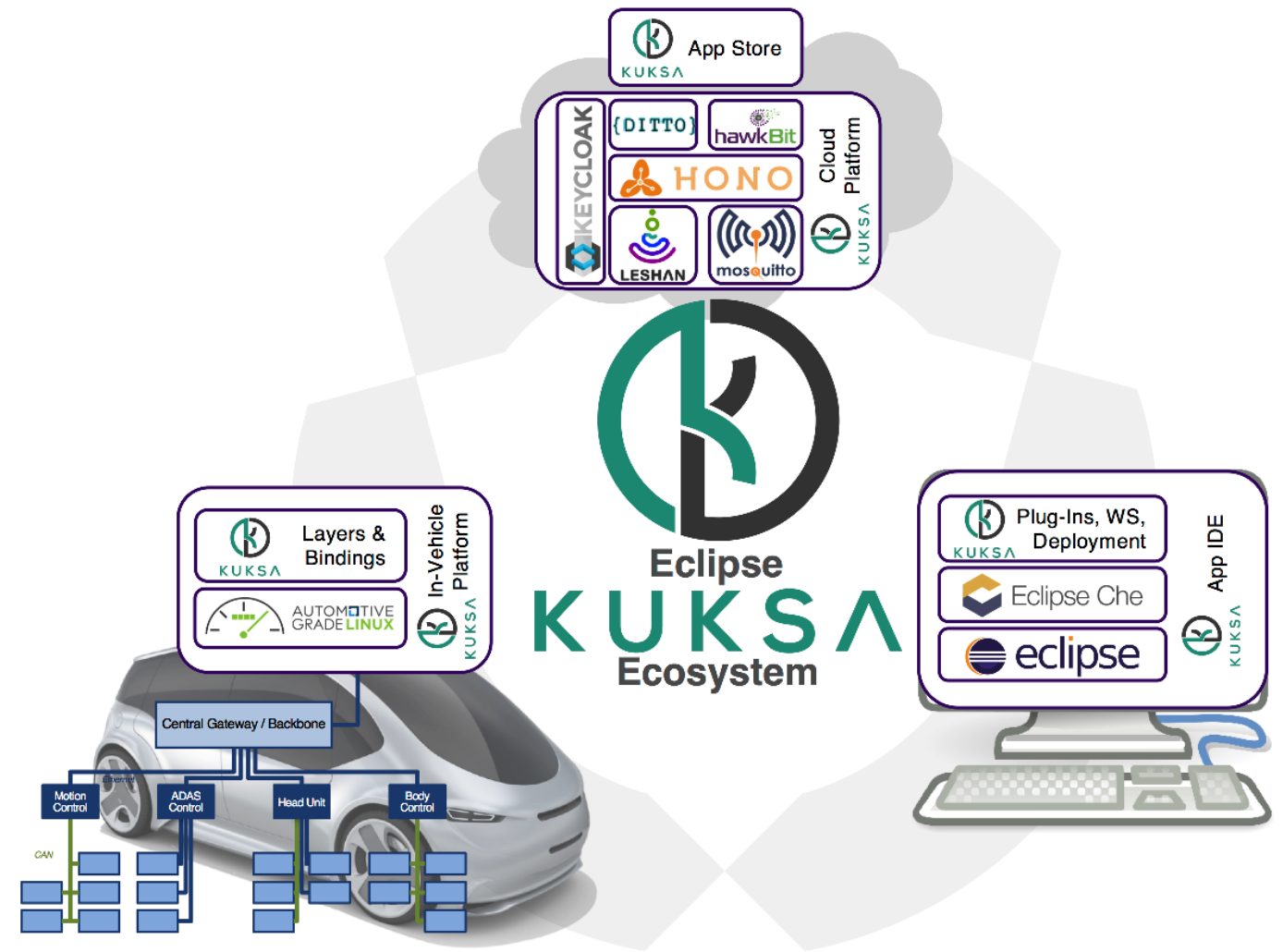


Figure 1. Eclipse kuksa ecosystem

(Source: <https://www.eclipse.org/kuksa/about/EKuksa.png>)

2. Kuksa App IDE

Table 1. Eclipse Che Kuksa instance version

Available	Version number
Current	6.10
Plan	7.0

2.1 Getting Started with the App IDE

Kuksa IDE is built as a full custom Eclipse Che Assembly. Therefore, it includes all assembly components specified and described in Eclipse Che Assembly are included into Kuksa IDE’s build system. A quick start steps can be found from the link [here](#).

2.1.1 Requirement system configuration

The necessary prerequisite to build and deploy Eclipse Che Kuksa are:

- A running docker instance on a Linux host system (tested with Ubuntu 18.04).
- Maven

Additionally, steps for building and running a Eclipse Che Kuksa instance in version 6.x are available [here](#).

2.1.2 How to setup the IDE

To set up Eclipse Che Kuksa, the following steps are necessary. If one wants to build and deploy AGL applications & services within Che:

- Create a new workspace with Automotive Grade Linux (AGL) as selected stack
- Go to "Profile" → "Preferences" → "Remote Targets" and add a new Remote Target with the device IP and the according User, e.g. "root". Then select the Target.
- Go to Profile → Preferences → Yocto Settings and add a new SDK with a Name, e.g. "agl-rover", a Version, e.g. "1.0.0" and a link to an appropriate AGL SDK, e.g. the AGL [Rover](#) SDK from "https://owncloud.idial.institute/s/kpntqpTz8cgx7X6/download", as Download Link. Then select the added Yocto SDK. To avoid connection trouble, open the Terminal and ssh into the appropriate Device: ssh < User >@< IP >

2.1.2.1 Model-driven Development of AGL Applications and Services

The Eclipse Che Kuksa instance simplifies the development of AGL applications and services. AGL features the usage of automotive applications based on HTML5, JavaScript, and C/C++, which run on top of AGL. While applications realize a distinct use case, services offers functionality to all applications. For more information please refer to the [AGL documentation](#).

The following sections demonstrate the development of AGL applications/services in a model-driven way based on the tool RAML2AGL (cf. Section [Raml2AGL](#)) as well as the building and deployment of AGL applications/services to a remote device running AGL (cf. Section [Building & Deploying](#)).

2.1.2.2 IDE

The IDE provided under Eclipse Kuksa not only support the development of applications for the vehicle component, but also the creation of applications for the cloud. Users should be able to choose between two different workspaces and technology stacks that contain the preconfigured and embedded APIs as well as software libraries of the respective applications to be developed. This allows the car to be equipped with new functions and new services to be deployed in the cloud.

Kuksa offers various APIs for implementing vehicle applications, a project template for cloud services, and wizards for easily providing vehicle applications in the App Store via the IDE. The extensive provision of the various APIs and libraries in the IDE enables accessing existing communication interfaces for the secure data transmission, storage, management, and authentication without having to take separate measurements for processing or interpreting the data.

Kuksa also supports the simplified deployment of new applications for both the cloud and vehicle components. This is provided by a pre-configured Eclipse Che stack, to which only the address of a target platform must be specified. Configuration, building and deployment can be done at the push of a button without further configuration or processing. Depending on the application, different development tools (e.g. Logging, Debugging, Tracing,...) can be included. Of course, syntax highlighting, code completion, and other necessary IDE functions are supported. For instance, the in-vehicle Eclipse Kuksa Che stack for AGL development activities features including Yocto based SDKs in order to support target specific programming shown in the screenshot below. After compiling and building software, specifying a target IP allows also the deployment process.

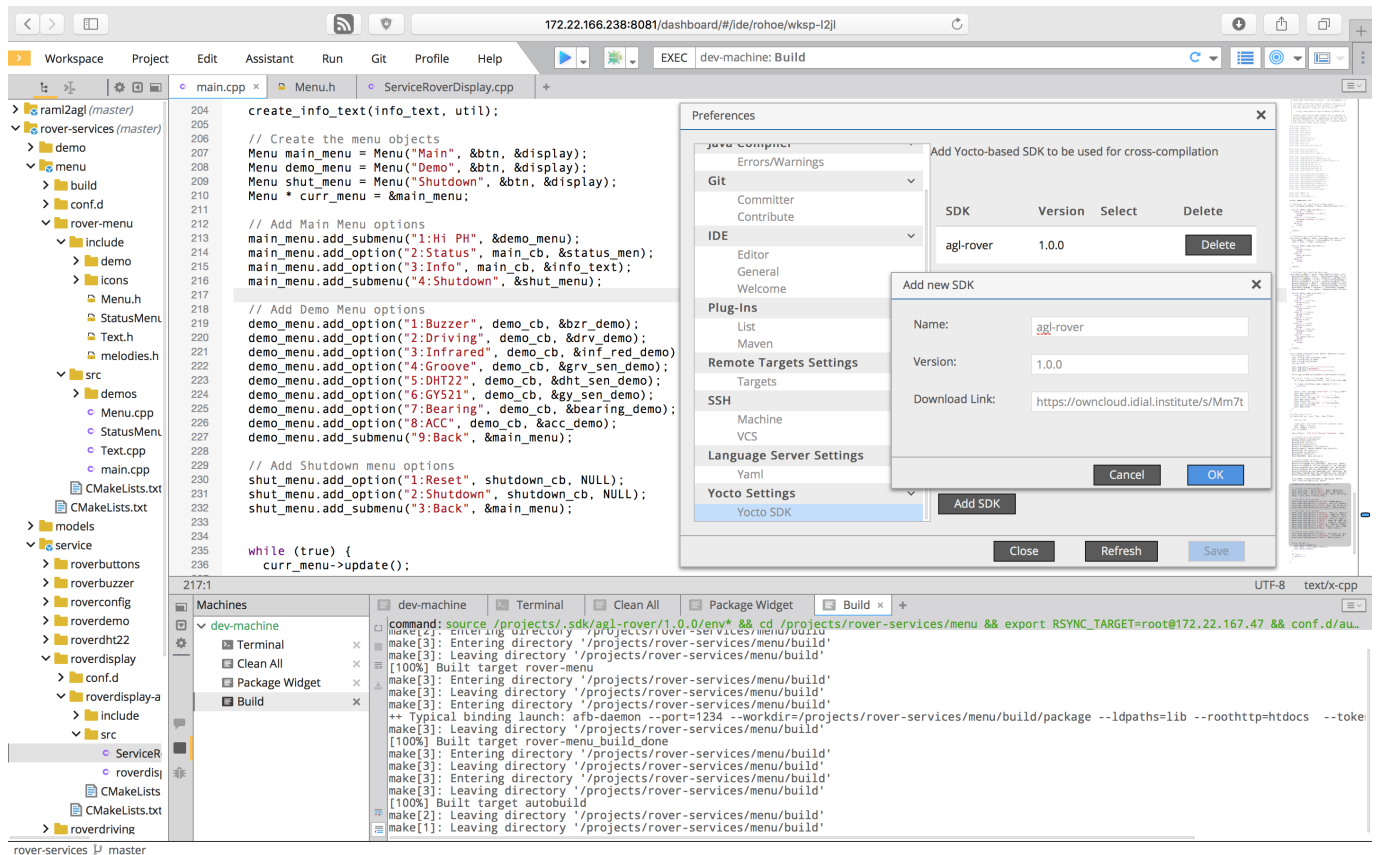


Figure 2. Kuksa IDE screenshot.

In order to make new applications applicable to a greater amount of vehicles, applications need to be centrally checked, managed, and organized with regard to various in-vehicle derivatives and variants in such a way that only vehicle-appropriate applications are accessible. Similar to a Smartphone App Store, it has to be possible to add new functions and applications to their vehicle or perform updates or upgrades. Therefore, standardized interfaces of the in-vehicle and cloud platforms are required and they must offer the most diverse and yet simple infrastructure for vehicle owners. Authentication methods, security concepts, variant management, and suitable data transmission technologies in combination with the publicly accessible ecosystem form mandatory components as well as the difference to existing solutions. [Click here to read more.](#)

2.1.3 Writing your first Kuksa In-Vehicle App

Kuksa Demo-Apps for the In-Vehicle platform can be found from the repository link [here](#).

2.1.4 Writing your first Kuksa Cloud App

Demo Kuksa Cloud App can be found from the repository link [here](#). However, by the time of preparing this document, the repository content was void.

2.1.5 Testing your Apps

2.1.6 Deploying your Apps

2.1.7 Install your In-Vehicle App in your car (see In-Vehicle platform Getting Started)

2.1.8 Install your Cloud App in your Cloud (see Cloud platform Getting Started)

2.2 Some Kuksa App examples

The eclipse Che doesn't provide a standard mechanism to add custom sample projects during build time. Therefore, Kuksa IDE provides an easy and straight forward mechanism to append them to ones provided by Eclipse Che during build time. [Sample Projects](#) can be found from this link.

3 Kuksa In-Vehicle platform

3.1 Getting started with the In-vehicle platform

[Eclipse Kuksa](#) includes an open and secure cloud platform that interconnects a wide range of vehicles to the cloud via open in-car and Internet connection and is supported by an integrated open source software development ecosystem. The Eclipse Kuksa project contains a set of repositories and this repo is one among those that contains in-vehicle platform code and also contains required layers and bindings to build a Kuksa adapted AGL (Automotive Grade Linux) distribution. The in-vehicle platform is primarily designed to work with AGL. However the individual components found in [this repo](#) could be used on other platforms as well.

Kuksa is a wrapper project around Automotive Grade Linux (AGL). From its side, AGL uses Yocto/Bitbake building system to build an automotive domain specific Linux distribution. Therefore, this project provides a building system that adds Kuksa's specific Bitbake layers on top of the original AGL. The scripts in this project help ease the process of building an AGL image by simply using a few commands. This project includes the yocto recipes found in meta-kuksa project.

3.1.1 Required System Configuration (HW/SW)

In order to Build the Image/SDK with cmake scripts, the required system configuration both (hardware and software) are:

Need Ubuntu 16

Fast Internet connection.

Minimum of 100 GB memory.

Some patience as it takes about 8 hours the first time.

To build the Image/SDK, run;

```
cd mkdir build cd build cmake .. make
```

Where can be;

```
agl-kuksa-sdk: AGL kuksa image and SDK
agl-kuksa: AGL kuksa Image only
Other Targets to follow.
```

The output images can be seen at /build/images and the SDKs at /build/sdk.

To set up and build the Image using yocto/bitbake, the necessary prerequisites are:

```
Need Ubuntu 16

Fast Internet connection.

Minimum of 100 GB memory.

Some patience as it takes about 8 hours the first time.
```

Steps

Setup the machine

Execute

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat libstdc++2.12-dev xterm cpio curl
```

This will install the necessary packages.

Execute

```
export AGL_TOP=$HOME/workspace_agl; mkdir -p $AGL_TOP
```

Execute

```
mkdir -p ~/bin ; export PATH=~/bin:$PATH ;curl https://storage.googleapis.com/git-repo-downloads/repo >
~/bin/repo; chmod a+x ~/bin/repo
```

This will set up the repo tool. Repo tool is used to download the recipes for AGL image.

Execute

```
cd $AGL_TOP ;repo init -b flounder -m flounder_6.0.1.xml -u https://gerrit.automotivelinux.org/gerrit/AGL/AGL-
repo ;repo sync
```

This will download the Funky Flounder version of AGL. This version has been tested and is recommended.

Start Building

Execute

```
source meta-agl/scripts/aglsetup.sh -m raspberrypi3 agl-demo agl-netboot agl-appfw-smack ; bitbake agl-demo-platform
```

This will start the build system and would take about 7 hours to complete if you are running for the first time, so you could take a nap 😴. The Yocto/bitbake build system has a caching mechanism and hence from the next time on, this would only take a few minutes.

Adding Kuksa layers

Go to \$HOME/workspace_agl/build/conf folder and open bblayers.conf file.

Append the following lines to the end of the file.

```
BBLAYERS += "  
${METADIR}/meta-kuksa  
${METADIR}/meta-kuksa/meta-kuksa-bsp  
${METADIR}/meta-virtualization  
"
```

Now copy the meta-kuksa folder (Link : <https://github.com/eclipse/kuksa.invehicle/agl-kuksa>) into the \$HOME/workspace_agl directory.

3.1.2 Set up the platform

To get started with In-Vehicle platform: AGL KUKSA Build and Run on Raspberry Pi 3 / Compute Module 3 (Lite) can be found from the link [here](#).

Building for the Raspberry Pi Compute Module 3 (Lite)

To build for the Raspberry Pi CM3 (Lite) platform, go to \$HOME/workspace_agl/build/conf folder and open local.conf file.

Append the following lines to the end of the file.

```
KERNEL_IMAGETYPE = "zImage"
```

Configure meta-kuksa layer

The kuksa layer contains recipes for the APIs and Apps contained in Eclipse kuksa Invehicle repo.

The AGL image with meta-kuksa layer adds w3c-visserver-api and elm327-visdatafeeder as systemd services. It will install the datalogger apps in the respective locations /usr/bin/datalogger-

Set up wifi

--- Ignore this step if wifi is not required ---

With meta-kuksa layer the wifi connection could be set up while building an Image so that the target device connects to the specified wifi, which make it easier to ssh into the device. The wifi settings could be configured

by modifying the meta-kuksa/recipes-devtools/wifi-conf/files/wifi_default.config file. Update the "Name" and the "Passphrase" of the wifi you want the device to connect to. More more secured wifi connection please refer to the link

configure Bluetooth connection with ELM 327 bluetooth adapter

The elm327-datafeeder service connects to an ELM327 Bluetooth adapter to retrieve data from the vehicle. Hence the bluetooth connection with the ELM327 adapter needs to be established before the service starts. The BT connection can be configured by Updating the MAC-Address of the adapter along with its pairing PIN. The MAC-Addr and PIN can be updated in file meta-kuksa/recipes-elm327-visdatafeeder/elm327-visdatafeeder/files/bt_setup.sh

Update the fields

```
#!/bin/bash

# Enter the MAC-ADDR of your bluetooth elm327 adapter here.
MACID='00:11:03:01:04:35'
# Enter the pair key of your elm327 adapter here.
PAIRID='6789'
```

Figure 3. schreenshot of bluetooth connection setup.

Alt text

Now Execute the below line to build image with Kuksa layers

```
source meta-agl/scripts/aglsetup.sh -m raspberrypi3 agl-demo agl-netboot agl-appfw-smack ; bitbake agl-demo-platform
```

This would take a few minutes to execute and at the end of the process the bootable image for RaspberryPi 3 will be found in the below location

```
$HOME/workspace_agl/build/tmp/deploy/images/raspberrypi3
```

3.1.3 Connect the platform to Kuksa portal

Once the image is ready, burn it onto a SD-card and boot up the image on raspi 3. The w3c-visserver-api requires the vss_rel_1.0.json file to set up the vss tree structure. This file can be copied to the /usr/bin/w3c-visserver folder by using scp command (sample file is available under https://github.com/GENIVI/vehicle_signal_specification or could also be generated using the tools in the repo). Once the file has been copied reboot the raspi 3.

Launch Datalogger apps

The Datalogger apps connect to a remote HONO-Instance and hence the IP-address of the Hono-Instance needs to be updated. The datalogger apps are already installed if you have followed the above steps. Now update the Hono configuration in the file /usr/bin/datalogger-*/start.sh with valid IP-address for the respective adapters (eg: HTTP and MQTT), Port, Device and password for Hono.

And you could start the datalogger apps by executing ./usr/bin/datalogger-*/start.sh

The apps connect to the w3c-visserver service using a websocket connection and retrieves Signal.OBD.RPM and Signal.OBD.Speed values to send to hono by packing the retrieved data into a json which looks like this {SPEED:xxx} & {RPM:yyy}

The Datalogger example for kuksa-app which connects to the w3c-visserver service via Websocket and talks to the Eclipse Hono MQTT adapter are :- [DataLogger-HTTP app](#) and [DataLogger-Mqtt](#).

3.1.4 Search for an In-Vehicle App

3.1.5 Install an In-Vehicle App

In-vehicle application software installation/update

– Admin user shall be able to perform a command to install or update In-vehicle application software from management web page of OTA server. For this purpose, OTA server shall send a message to the OTA client running on In-vehicle platform.'

This message shall include the name of last version of the software and binary file repository link of the software.

– OTA client shall handle the message coming from OTA server and install or update In-vehicle application software from the binary file repository link. OTA client shall reply a message to the OTA server about the status of installation (succeeded or failed).

– OTA client shall display a status message on HMI after software installation process is completed. – OTA server shall keep and maintain the data as a time series data in OTA server database.

Installing RoverApp

This tutorial [link](#) contains how to set up a Rover-specific Raspbian image from scratch and the basic workflow to run the Roverapp applications.

3.1.6 Test the In-Vehicle App

3.2 Configure the In-Vehicle platform

3.3 Overview of the In-Vehicle platform and its architecture.

In-Vehicle Platform

The APPSTACLE environment is created to provide addon services to the connected vehicles. This provides a complete functionality of the vehicles through deploying the Apps on the in-vehicle platform. Therefore, three layers of components are required to enable this purpose:

Core Layer: Contains the in-vehicle platform components, such as operating system and application runtime. It, furthermore, allows the vehicle owner to interact with the vehicle, e.g., via smartphone access. In addition, it provides an interface to the 5G infrastructure similar to the core layer of the cloud back-end.

API / Binding Layer: consists of relevant APIs and components for internal and external communication.

Application Layer: It represents the arrangement of all Apps that are running within the in-vehicle platform. Some of the Eclipse base Open source solutions to inreach kuksa components are:

- Automotive Grade Linux (AGL)

- Eclipse hawkBit
- Eclipse Hono
- Eclipse Ditto
- Eclipse Che
- Keycloak
- ...



Figure 4. Eclipse base Open source solutions to inreach kuksa components

(source:https://www.researchgate.net/profile/Marco_Wagner2/publication/330281127_Innovation_through_Openness_-_The_Open_Source_Connected_Vehicle_Framework_Eclipse_Kuksa/links/5c371b5892851c22a3691df8/Innovation-through-Openness-The-Open-Source-Connected-Vehicle-Framework-Eclipse-Kuksa.pdf?origin=publication_detail)

The in-vehicle platform additionally provides means for retrieving telemetry data collected by the vehicle itself as well as a human machine interface (HMI) for user interaction.

In-vehicle connectivity :

This section provides an overview of the communication protocols that are currently used in the existing automotive architectures as well as their interconnections in the Electrical / Electronic (E/E) in-vehicle architecture. The scope of these protocols defines the in-vehicle communication interfaces for the APPSTACLE platform.

Protocols

Automotive protocols are classified by the Society of Automotive Engineers (SAE) into four categories according to the transmission rate and their role in the automotive architecture. Specifically, Class A defines the protocols that are used for convenience systems (e.g. lighting, windows, seatcontrols) and require inexpensive, low-speed communication. Class B defines the protocols supporting instrument cluster or vehicle speed communication and require medium-speed communication. Furthermore, Class C is defined for real-time control ECUs such as the engine, braking and steer-by-wire and require high-speed communication. Finally, telematics systems usually

require higher communication speed for multimedia (audio / video) and navigation, and therefore SAE defined the additional Class D communications. All four protocol Class categories are illustrated in Table 1 along with the protocols that belong to each category and are used for in-vehicle communication in terms of their characteristics.

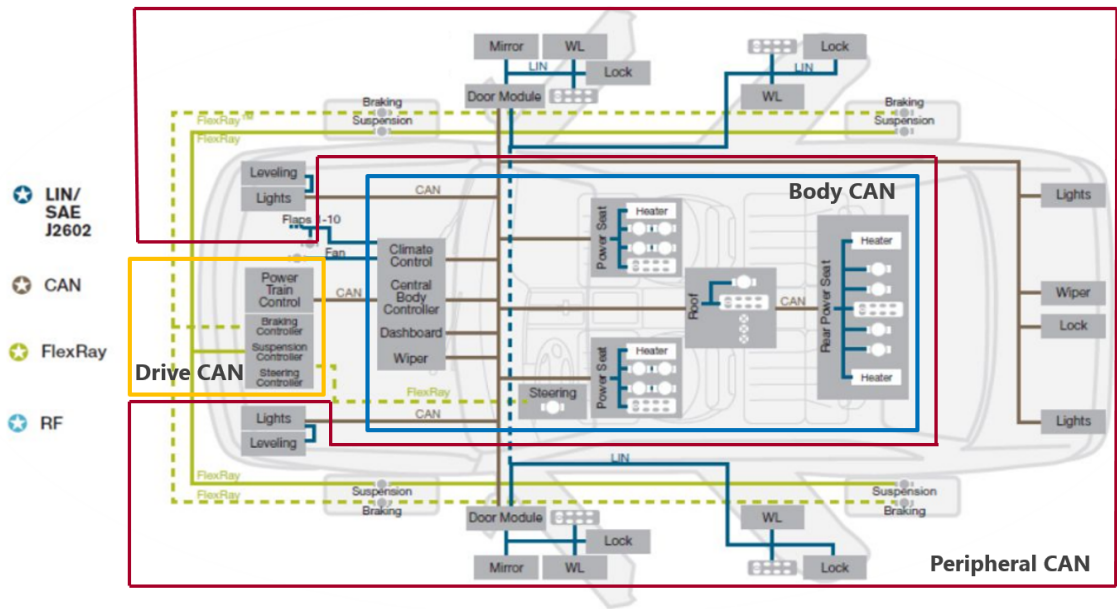


Figure 5. Automotive Network.

Table 12.: Characteristics of the communication protocols

Bus	LIN	CAN	CAN FD	FlexRay	MOST	Automotive Ethernet
Used in Application domains					Multimedia	Multimedia
Message transmission	Subnets	Soft real-time	Soft real-time	Hard real-time	Multimedia	Telematics
Access control	Body Soft	Powertrain, Chassis	a a CSMA/CA	Chassis, Powertrain	and Telematics	and active safety
Maximum Data Rate	Synchronous Polling 20 kbps A	Asynchronous CSMA/CA 1 Mbps BC	10 Mbps D	Synchronous and Asynchronous TDMA 10 Mbps D	Synchronous and Asynchronous CSMA/CA 24Mbps D	Synchronous and Asynchronous CSMA/CD 100Mbps D
Protocol Class						

Architectural Overview of In-vehicle

Modern automotive embedded systems consist of several subsystems, which are comprised of one or several Electronic Control Units (ECUs). In turn, the ECUs are made up of a micro-controller and a set of sensors and actuators. They are able to communicate through the transmission of electronic or optical signals through a

dedicated communication unit. The subsystems that rely on network communication in automotive systems are divided into five main categories: power train, chassis, body, HMI, and telematics (illustrated in [Figure 3](#)). Each subsystem uses a different protocol to communicate, which is selected based on the architectural requirements and the subsystem functionality. Specifically, the powertrain domain is related to the systems that participate in the longitudinal propulsion of the vehicle, including engine, transmission and all subsidiary components. This domain is supported by a dedicated subsystem called Drive CAN using the Controller Area Network (CAN) for data exchange. The chassis domain refers to the four wheels and their relative position and movement; in this domain the systems are mainly steering and braking. In this subsystem category we find two protocols that are used for high-critical communication, namely CAN and FlexRay, as well as the Local Interconnect Network (LIN) for the lower critical functionalities (e.g. door locking, window raising / lowering). According to the EAST-EEA 1 project definition the body domain includes the entities that do not belong to the vehicle dynamics (i.e., being those that support the car's user) such as airbags, wipers, lighting, etc. Today's cars sometimes use two CAN buses (peripheral CAN and body CAN) which interconnect the ECUs of the comfort domain. The telematics domain includes the equipment allowing information exchange between electronic systems and the driver (displays and switches). Such interactions are possible through the infotainment subsystem that is supported by the MOST protocol. Finally additional peripheral systems (e.g., cameras) allow the in-vehicle system to monitor and extract information from its physical environment through the use of Automotive Ethernet technologies. All the aforementioned systems are able to exchange data through a central gateway ([Figure 3](#)) that is able to map (through packet encapsulation) or forward messages from one subsystem to another.

ID	Property
1.	It enables communication with the infotainment unit or the ECUs internal to the vehicle
2.	<ul style="list-style-type: none"> • Telemetry data or data specified in the Cloud platform related to internal vehicle functionalities. • System metadata (which components are present, heartbeats etc)
3.	<ul style="list-style-type: none"> • Interacts with the in-vehicle HW for data gathering and with the ex-vehicle connectivity for data forwarding
4.	<ul style="list-style-type: none"> • One component per vehicle with different interfaces according to the employed protocols.
5.	<ul style="list-style-type: none"> • Information about the protocols used in the internal vehicle architecture • Depending on the chosen protocols: what components are communicating what information • Development of software modules for handling data for each protocol

Ex-vehicle connectivity concept:

The technology evolution in the automotive vehicles contributed to the demands for smarter mobility solutions. These solutions are focused on several types of V2X communication:

- Vehicle-2-Vehicle (V2V)
- Vehicle-2-Infrastructure/Infrastructure-2-Vehicle (V2I,I2V)
- Vehicle-2-Pedestrian (V2P) / Pedestrian-2-Vehicle (P2V)
- Vehicle-2-Network (V2N) / Network-2-Vehicle (N2V), 5) Infrastructure-2-Network (I2N) / Network-2-Infrastructure (N2I).

These types along with their interactions are demonstrated in [Figure 6](#).

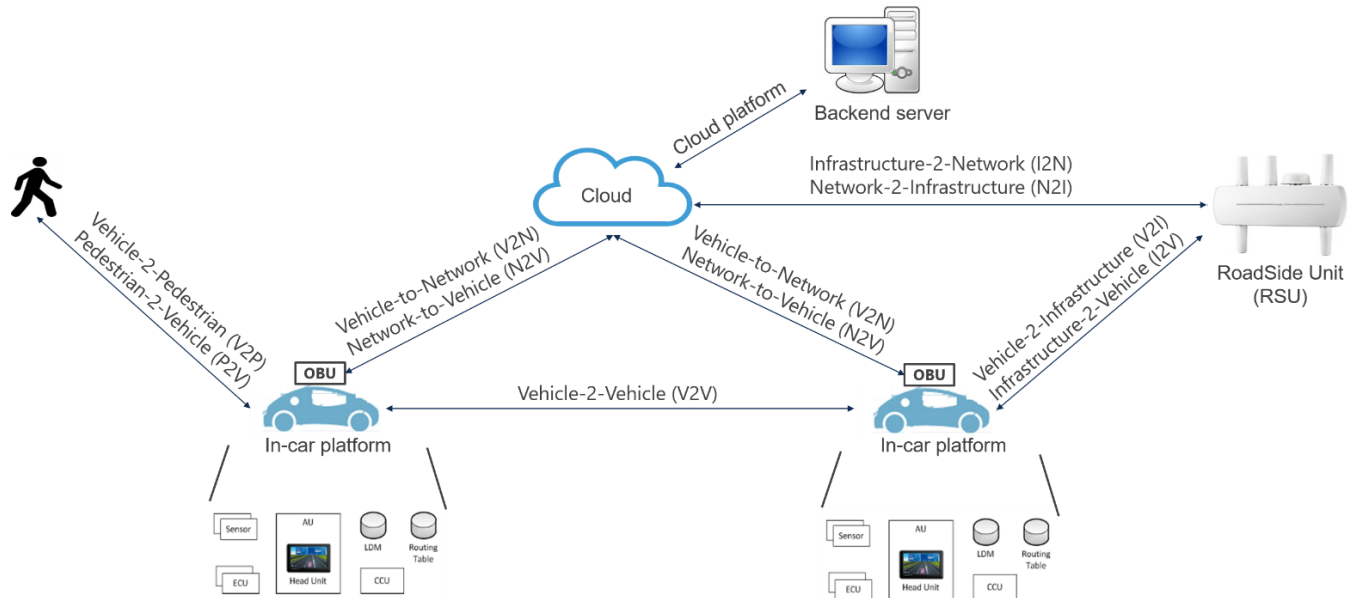


Figure 6. V2X communication types

The units supporting V2X communication are:

- **RoadSide unit (RSU):** It is connected to road sensors (e.g. induction loops, cameras) and a local control center, such that it performs actions or exchanges critical information other vehicles or servers about road or traffic management.
- **OnBoard unit (OBU):** The on-board unit (OBU) is a radio built-in vehicle device mounted on each vehicle that transmits vehicle data (i.e. identification and location) to a transponder. The OBU itself is a transponder, that is, a data exchange takes place automatically and only on request of one of the participating devices. It allows Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I, I2V) communications with other OBUs or RSUs.
- **Backend server :** It is composed by a PKI, traffic management and roadside unit management servers, all accessible via the RSU's or cellular base stations. In order to facilitate this evolution a couple of solutions were defined that are split into 3 main categories:
 - **5G radio access technologies:** This technology provides wide area, broadband access. The 5G technology is currently in the process of conceptual development and standardization by the World Radiocommunication Conference (WRC). The 5G technology is expected to have a specific V2X aspect of the 5G technology in a practical scale after 2020. However, in this document we are leveraging the limited standardization to illustrate conceptually its main scope and architectural view.
 - **Pre-5G radio access technologies:** Multiple cellular technologies were identified by the ETSI 3rd Generation Partnership Project (3GPP), LoRa Alliance and other organizations, such as Narrowband IoT , Long Term Evolution for Machines (LTE-M), LoRA are considered. Even though these technologies are already used in V2P/P2V, the main challenge when adopting them in other V2X communication types are reliability and safety, which are currently not addressed in the scope of Low-Power Wide Area Networks (LPWAN).
 - **Non-cellular technologies providing wireless access:** IEEE has defined different standards for wireless communication, such as 802.11ac and 802.11p, however only 802.11p is flexible in terms of throughput and offers higher reliability, even though its maximal throughput is more limited than 802.11ac (from 3 to 27 Mbps raw data rate). The reason behind this is that 802.11p was designed particularly for for safety-related Vehicular Ad-hoc NETworks (VANET), including the V2V and V2I/I2V concepts. IEEE 802.11p technology is currently fully specified and already deployed in different locations. The following paragraphs start with a description of the

scenarios supported by 802.11p communication and cellular communication. This is followed by a description on both the 802.11p and 5G technologies. In the scope of this section we focus on these two technologies, because, to the best of our knowledge, they are considered as the leading candidates for V2X communication.

This browser does not support PDFs. Please download the PDF to view it: [Download D4.1. APPSTACLE - Concept of Vehicle to Cloud to Ecosystem to Cloud to Vehicle HMI Use Case.pdf](#).

ID	Property
1.	It enables outward and inward communication between the vehicle and the external entity
2.	<ul style="list-style-type: none"> • No data processing as such. • Re-Packs the data received from bus to appropriate format for the external entity and vice versa.
3.	<ul style="list-style-type: none"> • It has the direct connection to the BUS and no direct user interaction.
4.	<ul style="list-style-type: none"> • There will be multiple instances in the minimum two cases.
5.	<ul style="list-style-type: none"> • Driver for hardware component • (Since Development Stage) Need manual configuration at the moment for 5G mm Radio.

App Runtime concept:

ID	Property
1.	The APP Runtime provides the environment for executing APPs and starts / stops APPs. It has to provide and control resources for the APP, enforce access control (permissions), and isolate APPs from each other.
2.	APPs, configuration data (permissions, options, ...), APP data
3.	The APP Runtime permits or denies communication between APPs, or APP and backend (depending on "the policy"). The APP Runtime obtains APPs from the marketplace. It can be configured by the OEM and/or the vehicle owner (via backend and/or an in-vehicle user interface; probably also by devices [with authorization]).
4.	There is one APP Runtime per in-vehicle platform. It could be part of the operating system.
5.	Initialisation / start up: The APP Runtime is started during the (secure) boot process. It can be configured by the OEM and the vehicle owner (details are left open in this document), e.g., to configure permissions ("the policy").

Automotive API concept:

The Application Programming Interface (API) for vehicles are introduced and discussed in here. The automotive API's try to achieve (a) merging the potentially very complex device and network structure of a car into a single virtual device and (b) hiding the differences between manufacturers, models and makes behind a common interface. On the other hand these interfaces strongly differ in their scope (data-subset or use-case), technological approach and creators.

AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) is a cooperation between car manufacturers, OEMs and tool manufacturers and defines a software development paradigm for Electronic Control Units (ECUs) in the automotive domain. In order to separate the development process of application software from the chosen ECU hardware platform, AUTOSAR is introducing a layer model with the three layers Application Software, Runtime Environment and Basic Software (illustrated IN [Figure 5](#)).

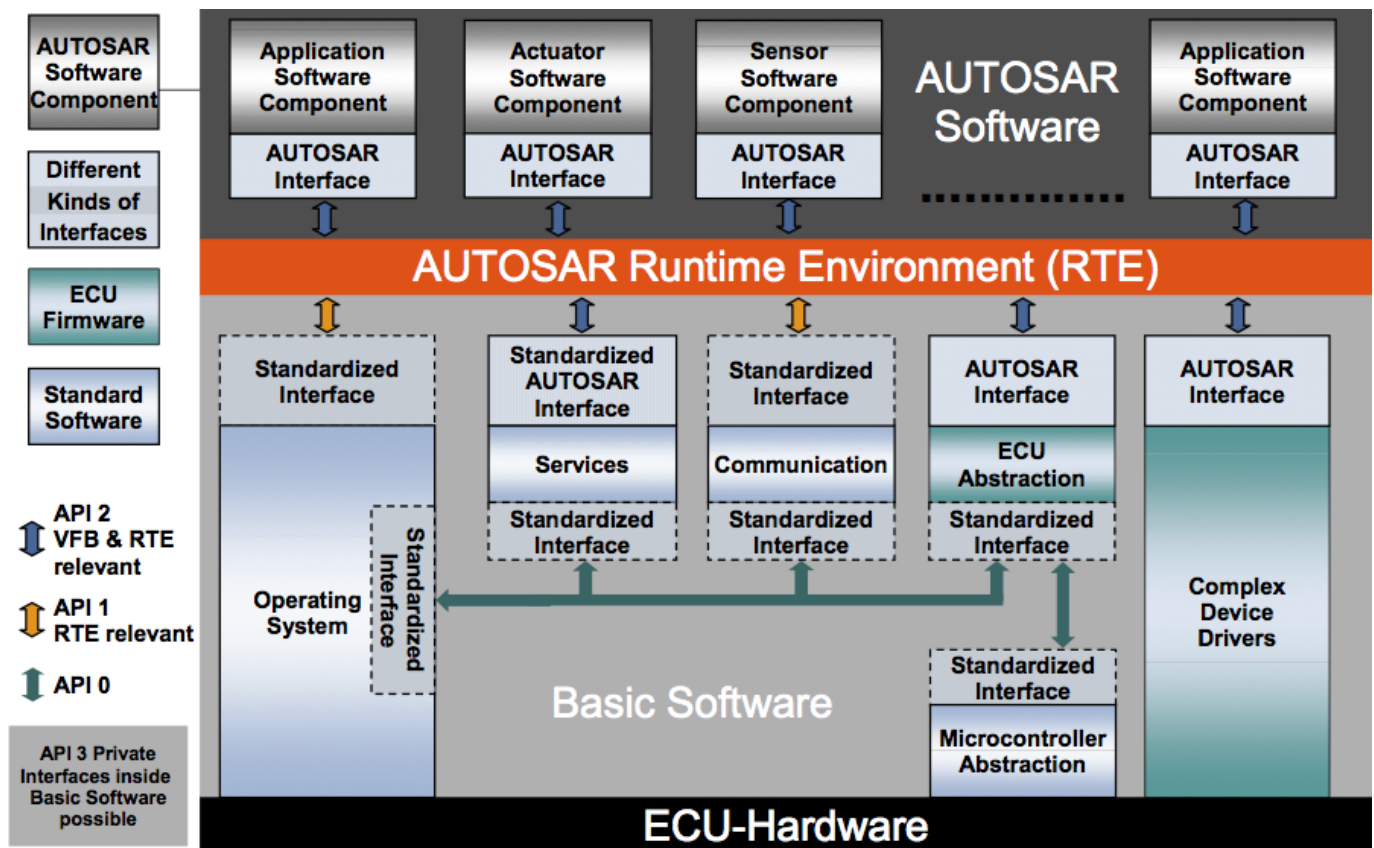



Figure 7. Autosar Layers

The top layer is formed by the application software. It is divided into software components, each of which realizes a part of the application and can consume and provide data via so-called ports. Any communication that does not take place via port connections is forbidden. A port is classified via a port interface (here referred to as interface). Two ports can only be connected to each other if both ports use compatible interfaces. Two important communication paradigms, that are selected by interfaces, are client-server and sender-receiver communication. For client-server communication, a server component provides functions (C, C++) which can be called by clients. A 1:n communication is also possible (i. e. a server can provide its functionality to several clients). In sender-receiver communication, a sender provides data that can be consumed by receiver components. Both 1:n and m:1 communication is possible here (i.e. a date can be consumed by several components or several senders provide a date for one receiver concurrently). Many-to-many communication is not provided.

The lower layer consists of the basic software and contains the hardware drivers, the operating system and the communication stack. The communication stack handles communication from and to other ECUs that are connected via network interfaces like CAN, LIN, Flexray, automotive Ethernet, etc.

All communication, whether between software components on the upper level or between software components and basic software on the lower level, is realized via the runtime environment (RTE), which forms the middle layer. The RTE specification document defines a schema for API functions (C, C++), which are usually generated by code generators of the AUTOSAR modeling tools according to the modeled communication between software components and basic software. All communication must take place via the (generated) API functions. Other communication is not permitted. Likewise, all communication interfaces must be defined at the time of development, which makes it impossible to dynamically extend the software architecture at runtime.

 For detail information

ID	Property
1.	The Automotive API provides an interface to in-vehicle data for APPs (and cloud ser-vices? other entities??). An (operating system) service ("Automotive API server") implements the Automotive API, for instance similar to the "Vehicle Information Service" specified by the W3C.
2.	Vehicle data (sensor data, diagnosis information, configuration of vehicle compo-nents, vehicle status information, ...)
3.	It can be used by APPs (and cloud services, etc.) to retrieve information from the vehicle, to send data to in-vehicle components, and to write (vehicle) configuration data.
4.	There is one Automotive API per in-vehicle platform.
5.	Initialisation / start up: The "Automotive API" service is started during the (secure) boot process.

Apps concept:

ID	Property
1.	(In-vehicle) APPs are programs that provide new features to the vehicle.
2.	App data (depends on APP / use case)
3.	APPs are executed and controlled by the APP Runtime and can access in-vehicle data via the Automotive API. They can communicate with other entities via con-nectivity APIs (cf. architecture picture). If permitted (by "the policy"), they might communicate with cloud services (backend providers) and other APPs, and they could interact with the driver via a GUI (if available). APPs can access resources via the APP Runtime (subject to "the policy"). A user can install APPs from the Marketplace in the vehicle.
4.	There can be multiple APPs per in-vehicle platform.
5.	Initialisation / start up: APPs are started by the APP Runtime. Configuration data depends on the APP / use case.

Device Management Client concept:

ID	Property
----	----------

ID Property

The DM is responsible for keeping the devices compliant to the whole system land-scape. This starts with building up a base in communication and process protocols. It is also providing features for securely

1. enrolling new devices, governing and con-figuring them while being out in the field, monitoring and debugging their behavior remotely and maintaining the devices with software updates. Four subjects can be distinguished:

- Enrollment includes provisioning and authentication for bringing new devices into an IoT landscape. The authentication assures that only trust-worthy devices are added to the network and connected to cloud services. Also only authorized users should be able to bring in new devices and gain access according to their roles granted.

- Governing contains features for controlling and configuring devices. As IT systems are often under a constant development, some parts change and so do some of their constants, for example network addresses or ports.

- Monitoring keeps an eye on all components of the system and their status. Reports and alerts for incidents are raised and logged. An on-time awareness of system issues is enabled. For diagnosis and solving software bugs it is also imperative to load log dumps remotely from devices.

- Maintenance with the ability to distribute and apply software updates is the fourth subject. The device management assures that the update is delivered and applied to the device according to the present constraints. Feedback mechanisms answer back to the cloud for a detailed status of the update.

2. Process

- System states

- Application states

- Update states

- Management Calls

- Push Calls

- Alarms

- Enrollment policies

- Updates, Update scheduling

- Inventory Hardware/Software listings

- Communication requests

Provide

- Access to resources through OMA-DM Management Objects
(<http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html>)

- Management access through LwM2M)

- Push Service (for notifying in-vehicle Applications form the cloud)

- Monitoring Service

ID	Property
	• Update Service (Maintenance)
	• Keep-Alive Signal Service
	• Inventory Hardware/Software
	• Enrollment Service
	• Control Service (e.g. shutdown certain components)
3.	• Apps, OS: (Re-)Configure, Update
	• ECU: Flashing/Updating ECUs
	• Apps: Notify/Wake Up through push
	• HW/SW: Read Inventory
	• HW/SW: access on resource (if Management Object is defined)
	• User: Update scheduling
4.	The DM can only exist once per vehicle. But it is split into its functional groups (subjects). A hierarchical design of distributed DMs in a vehicle might be the subject of another design.
5.	• Authentication Set (MAC, UUID, Public Key)
	• Cloud contact
	– Address of DM at the desired cloud
	– Certificate

Operating System (OS) concept:

ID	Property
1.	The Operating System is the backbone any platform where it operates including In-vehicle platform. We also consider drivers as being part of the Operating system.
	<i>Operating System</i>
	• consists of kernel and user space components
	• provides security
	• I/O and networking services to applications running in user space
	• runs on bare metal or hypervisor
	• OS kernel provides:
	– processes/thread management including scheduling
	– inter process communication
	– memory management
	– access to underlying HW

ID	Property
	– networking stacks
	– I/O stacks
	– security subsystem include:
	* access control (e.g. MAC)
	* crypto and key management
	* integrity measurement
	* entropy pools and gathering entropy from various sources It is also important to highlight that the security hardening of an OS is crucial to the platform and application security
2.	Process
	• data to/from underlying HW
	• data to/from networking
	• data to/from file system
	• user input
	• data exchanged between kernel and user space
	• data exchanged between processes
	• configuration data such as security policies, system settings,...
	Provide
	• System state information (running processes, CPU utilization, etc.)
	• log data
	• debug or diagnostics data if certain debug features are enabled
3.	• <i>Access to underlying HW</i>
	– peripherals
	– networking interfaces
	– any physical interface
	– HW based crypto and key management functionality
	– HW based TRNG (True Random Number Generator)
	• <i>App-Runtime (transport layer)</i>
	– Provide transport layer interface for Apps (inter-app communication, app to cloud backend communication)
	– Provide system level services (file access, etc.)
	• <i>Device Management Client (transport layer)</i>

ID	Property
	– Provide transport layer interface for communication with device management (cloud backend)
	• <i>5g Infrastructure (data link layer)</i>
	– Connection Management
	• <i>Smartphone (data link layer)</i>
	– Pairing
	– Connection Management
	– Data Transfer
	• <i>Net-IDS</i>
	– Provides Interface to allow the Net-IDS to monitor network traffic.
4.	Different subsystems on a vehicle may be running their own OSes
	• whether on bare metal or virtualized
	• whether micro kernel based, unikernel based or rich OS such as Linux
5.	• build and runtime OS configuration
	– for process, resource management, memory management, functionality, security,...
	– policies (e.g. security)
	– configuration of different processes
	– networking configuration
	• User credentials

3.4 Overview of the Kuksa In-Vehicle API

The Kuksa In-Vehicle API implementation is based on [W3C Vehicle Information Service Specification](#).

The implementation provides all the major functionality defined in the above specification and also uses JWT Token for permissions handling with decent amount of uni-tests covering all the basic functions. This project uses components from other open source projects namely

1. [Simple-WebSocket-Server](#) which is under MIT license.
2. [jsoncons](#) which is under Boost Software license.
3. [jwt-cpp](#) which is under MIT license.

The overall details on how to build and run W3C as well as details of W3C VIS Server Implementation can be found from the repo [link here](#).

4 Kuksa Cloud Platform

4.1 Getting started with the cloud platform

This repo [link](#) contains script directory and it's subdirectories help to setup and deployment of the Kuksa cloud. These scripts assume a running Kubernetes cluster which can be configured using kubectl. More information regarding the parameters of the scripts can be found within the respective script file of the [link](#).

Structure

The deployment scripts are divided into the following parts:

1. Azure for Azure-specific configuration that provides the basis of Kubernetes.
2. Eclipse hawkBit enables the deployment of the corresponding software update components, in particular the update server. Note that this step requires the installation of the command line tool kompose. Installation instructions can be found at <http://kompose.io/>
3. Eclipse Hono enables the deployment of a messaging infrastructure.
4. Kubernetes provides functions for the Kubernetes deployment of the Kuksa cloud.
5. Utils scripts that are included by other parts of the deployment infrastructure (e.g. handling static IP-addresses for the services). It is possible to set static IP-addresses and DNS entries for deployed services. For more details on that configuration see the Readme.md file in the utils directory.

4.1.1 Required System Configuration

To get started, the required infrastructures are:

```
Java 8
Maven
Spring-Boot and other dependencies(data-jpa, feign client,pagination)
Vaadin
Swagger (for Rest API documentation)
```

Prerequisites

Just run AppStoreApplication.java class.Spring boot has an embedded Tomcat instance. Spring boot uses Tomcat7 by default, if you change Tomcat version, you have to define these configuration in *pom.xml*. But you have a few options to have embedded web server deployment instead of Tomcat like Jetty(HTTP (Web) server and Java Servlet container) or Java EE Application Server. You have to configure these replacements from default to new ones in *pom.xml*. For detail information, follow the [link](#).

4.1.2 Installing and testing the Cloud platform

4.1.3 Installing a Cloud App and its In-Vehicle App

4.1.4 Testing the Cloud App

9.2 Configuring the Cloud platform

- The Microsoft Azure Deployment contains the IP addresses and DNS names. This [link](#) explains the deployment of Azure.
- Similarly, the [Eclipse hawkBit Deployment](#) repo link consists of necessary deployment steps.
- Deployment of Eclipse hawkBit can be found from the repo link [here](#).
- Eclipse Hono Deployment is available from this [link](#).
- Kubernetes-specific functionality details can be accessed from the repo link [here](#).

4.3 Overview of the Cloud platform and its architecture

4.4 Overview of the Kuksa Cloud API

4.5 Marketplace presentation and features

Marketplace Backend concept:

ID	Property
1.	<ul style="list-style-type: none"> • Stores and manages the software and data artifacts related to the apps that are installed on the in-vehicle platform
	<ul style="list-style-type: none"> • Initiates and controls the communication with the app provider backends
	<ul style="list-style-type: none"> • Interacts with potential payment providers, depending on the payment methods applied (Credit/Debit Card, SMS)
	<ul style="list-style-type: none"> • Stores the data related to registered users (vehicle owners and app developers)
	<ul style="list-style-type: none"> • Central hub for the interaction with app developers
	<ul style="list-style-type: none"> • Provides app data to the in-vehicle platforms via the device management back-end
	<ul style="list-style-type: none"> • Scan app software and data artifacts regarding vulnerabilities and malware
	<ul style="list-style-type: none"> • Question: Should in-app transactions be possible?
2.	Process
	<ul style="list-style-type: none"> • User input via the Marketplace Frontend
	<ul style="list-style-type: none"> – Vehicle owner core data (Name, Address, Supported payment methods, User vehicle mappings (1 User, N Vehicles))
	<ul style="list-style-type: none"> – App developer core data (Name, Company, Address, Supported payment methods, Apps provided)
	<ul style="list-style-type: none"> – Transactional data regarding buying and returning apps (Transaction ID, Transaction status, Payment method applied, Vehicle Identification Number, Version of the app transferred)
	Provide
	<ul style="list-style-type: none"> • Software and data artifacts that are provided by the app developers
	<ul style="list-style-type: none"> – Software (Compiled source code)
	<ul style="list-style-type: none"> – Data (Licenses, Documentation, Version history, Supported in-vehicle software platforms, Supported in-vehicle hardware platforms)

ID	Property
	– API/Frontend for app developer interaction (Transferring software and data artifacts to the backend, Receiving account and app related information, e.g. number of downloads or total revenue per app)
3.	Input
	• Marketplace frontend (see informations listed above)
	• Payment provider backends (Handling of payment transactions)
	• App developer backends (Receiving data and software artifacts) Output
	• Marketplace frontend (Deliver data requested by the frontend, e.g., results for search queries)
	• In-vehicle platforms, via Device Management component (Software and data artifacts regarding specific apps)
	• App developer backends (Transaction and evaluation data)
	• Identity Management (Requests for user authentication and authorisation)
4.	• There should be only a single instance within the cloud backend
5.	• Deployment configuration regarding micro services
	– External IP addresses
	– Ports
	– Connection to other services, e.g. Identity Management

Marketplace Frontend concept:

ID	Property
1.	• Provides visual interface that enable the interaction with the Marketplace Backend
	• Vehicle owners can search and order apps as well as initiate their transfer to the in-vehicle platform
	• Allows app developers to access app storage and monitoring data
	• Allows vehicle owners and app developers to edit their core data / profiles
2.	Process
	• Core data (vehicle owner and app developer) entered via the visual interface
	• Query database regarding apps available for the specific in-vehicle platform
	• Software and data artifacts send by the app developers
	• Data referring to payment transaction, e.g., data exchange with payment providers
	Provide
	• Visual interface depicts information regarding
	– Vehicle owner and app developer core data
	– App transactions

ID	Property
	– App search query results
3.	Input
	• Vehicle owner
	– Enter and update core data
	– Fill out order forms
	– Search apps via query masks
	• App developer
	– Enter and update core data
	– Upload apps
	– Retrieve monitoring data
	Output
	• Vehicle owner
	– Display core data
	– Display app search queries
	• App developer
	– Display core data
	– Display monitoring data
4.	• There exists only a single services within the backend
	• There might be several container instances behind this service
5.	• Deployment configuration regarding micro services
	– External IP addresses
	– Ports
	– Connection to other services, e.g, Marketplace Backend

In addition:

The Eclipse Kuksa Appstore contains:

Build Eclipse Kuksa Appstore

Script:

build_kuksa_appstore.sh

Purpose:

Build a Docker image for the Eclipse Kuksa Appstore and push it to a Docker registry

Options:

DOCKER_REGISTRY_SERVER: Address of the Docker registry server, e.g. running on Microsoft Azure.
DOCKER_REGISTRY_USERNAME: Username to sign in to the Docker registry.
DOCKER_REGISTRY_PASSWORD: Password to sign in to the Docker registry.
DOCKER_REGISTRY_EMAIL: Email to sign in to the Docker registry.

Stages:

- Clone the Kuksa Git repository
- Build with Maven
- Build Docker image
- Push Docker image to Docker registry
- Final cleanup
 - Remove the clones Git repository

Deploy Eclipse Kuksa Appstore

Script:

deploy_kuksa_appstore.sh

Purpose:

Deploys a Docker image for the Eclipse Kuksa Appstore on a Kubernetes cluster

Options:

APPSTORE_USERNAME: The username to set for the admin user of the appstore
APPSTORE_PASSWORD: The password to set for the admin user of the appstore
HAWKBIT_HOST: The (cluster-internal) hostname of the hawkbit server to use in the appstore
HAWKBIT_PORT: The (cluster-internal) port number of the hawkbit server to use in the appstore
HAWKBIT_USERNAME: The username to be used by the appstore to authenticate with hawkbit.
HAWKBIT_PASSWORD: The password to be used by the appstore to authenticate with hawkbit.
DOCKER_REGISTRY_SERVER: Address of the Docker registry server, e.g. running on Microsoft Azure.
DOCKER_REGISTRY_USERNAME: Username to sign in to the Docker registry.
DOCKER_REGISTRY_PASSWORD: Password to sign in to the Docker registry.
DOCKER_REGISTRY_EMAIL: Email to sign in to the Docker registry.
DOCKER_REGISTRY_SECRET: Name of the secret to access to custom Docker registry. Note that the secret is created during the deploy process within the namespace extension.

Notes:

- Production mode is enabled for Vaadin.
- H2 console is disabled.
- The H2 database is persisted via a persistent volume claim.
- The service uses a ClusterIP so it is only available behind the Ambassador gateway.