Model Driven Development for Kuksa Applications Documentation

Release 0.0.1

Pedro Cuadra

Mar 20, 2018

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction and Goals

AGL (Automotive Grade Linux) provides many development interfaces. For instance, HTML5, JavaScript, and C/C++ applications can be developed to run on top AGL. However, development methodologies aren't explicitly mentioned from AGL's development team.

1.1 Requirements Overview

This documentation presents an MDD (Model Drive Development) methodology to simplify and abstract the development process.

1.2 Quality Requirements

Below, the quality requirements are presented.

Requirement: Transparency REQ_001

links incoming: None
links outgoing: None

The MDD methodology shall show a clear mapping between the components from involved layers.

Requirement: Abstraction REQ_002

links incoming: None
links outgoing: None

The MDD methodology shall provide a simplified abstract of the concepts in the underlying layers; e.g. Application Framework.

Requirement: Standardization REQ_003

links incoming: None
links outgoing: None

The developed solutions for the MDD methodology, shall use standard and predefined processes, methodologies, tools, and interfaces to facilitate their adoption.

Requirement: Flexibility REQ_004

links incoming: None
links outgoing: None

The MDD methodology should provide customization mechanisms.

Requirement: Testability and Debugability REQ_005

links incoming: None
links outgoing: None

The MDD methodology should provide mechanisms for testing and debug all main components.

CHAPTER 2

Constraints

2.1 Technical Constraints

The technical constraints are shown in Table 2.1.

Table 2.1: Technical Constraints Table

ID	Constraint	Description
Software and programming constraints		
TC1	Programming Language	There's no explicit constraint regarding the programming language to be used.
Operating system constraints		
TC2	AGL distribution	The developed MDD methodology shall apply for developing for AGL Linux Distribution
Hardware Constraints		
TC3	Memory friendly	The applications developed with the MDD approach shall consider good memory management practices.

2.2 Conventions

Finally, conventions used by this project are shown in Table 2.2.

Table 2.2: Organizational Constraints Table

ID	Constraint	Description
C1	Documentation	The documentation is written using the arc42 document structure and using Sphinx.
C2	Coding conventions	For C/C++ and Python (used for the MDD) development the coding styles used were the Linux Kernel coding style *[8]* and PEP8 *[11]*, respectively.

CHAPTER 3

System Scope and Context

The work presented in this document proposed a Proof-of-Concept of an MDD Approach. The approach is focused on showing a possible workflow to develop AGL applications and services. Fig. 3.1 shows the context diagram of such an approach. Note that the proposed solution should consider *AGL* components in order to provide a smooth integration with the AGL Linux Distribution.

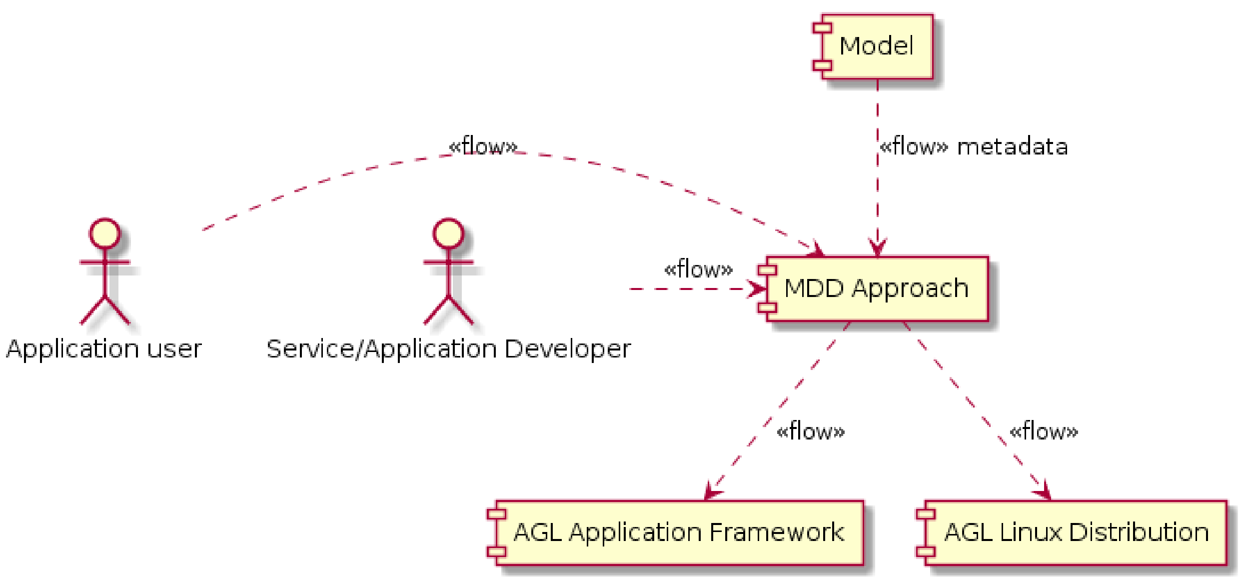

Fig. 3.1: MDD Approach Context

CHAPTER 4

Solution Strategy

The MDD approach developed is focused on developing applications to run on top of AGL. The AGL architecture specifies different layers of abstraction and the MDD workflows shall be compliant with this architecture. Therefore, the MDD process presented in this work focuses on the development of AGL Services that use AGL's Applications Framework APIs.

AGL services expose functionality to all the applications that might run on top *[32]*. To be more specific AGL services are implemented as `systemd` user-defined services in AGL. The way they expose the functionality is exposing a RESTfull API through a Web Sockets (or dbus). Meaning that in order to access functionality exposed by an AGL service, the application has to open a Web Socket use the RESTfull API.

The MDD approach presented in this document focuses in defining a model of the RESTfull API. The model is then used as an input for automatically generate the communication components of both the AGL service and the AGL application.

For modeling the RESTfull API, RAML (RESTfull API) was used. RAML is a recently developed community standard that has already been widely adopted in other projects like; *API Workbench* and *API Designer [17]*. It's a markup language based in YAML, which makes it both; machine readable and human readable.

`raml2agl` is written in Python (Python 3), which makes it really fast to develop and portable. Although Python has already two reference implementations of a RAML parser called `pyraml-parser` *[13]* and `ramlifications` (developed by Spotify) *[36]*, they were not used for developing `raml2agl` since they only support RAML 0.8 and `raml2agl` plans to support RAML 1.0. Therefore, a custom RAML 1.0 parser was designed and implemented. `ramlifications` plans to support RAML 1.0 in the future. *[36]* Therefore, `raml2agl` could adopt it in the future.

Another reason to use Python to write raml2agl is the variety of already implemented components. Especially the support for Jinja2 templating language was of high importance here. Jinja2 is a very powerful and complete templating language with bindings for Python. *[35]* The code generation was implemented using Jinja2 templates, which makes the code generation highly flexible and fast to develop.

The final outcome of the automatic code generation is a set of C++ classes that implement the entire RESTfull API communication. Moreover, simple C++ classes methods abstract the complex Web Socket handling and RESTfull API message marshaling and unmarshaling. This approach can be compared with other projects like Google's protobuffer *[25]* that aims to automatically generate the communication components.

CHAPTER 5

Building Block View

To understand where the proposed MDD approach has its importance, the components involved in the Unix Web Socket communication have to be presented. Fig. 5.1 presents these components.

Since the *AGL Application Framework* and its API are already given in the AGL architecture, the rationale behind the design was to wrap the *AGL Application Framework API* and the Web Socket communication in an RPC-like approach. Moreover, the components were encapsulated applying functional decomposition. Table 5.1 shows the responsibilities for each of the components in Fig. 5.1.

Table 5.1: Top Block Components Responsibilities

Name	Responsibility
AGL Application Framework	Manage all AGL Services and their life cycle, Create Unix Web Socket for the RESTfull API to be served by the AGL Services, Forward RESTfull API verb calls to AGL Services verbs callbacks, Verbs authentication process handling.
AGL Service	Initialize service resources, serve the RESTfull API, Forward the RESTfull API verbs to the corresponding Service Class method, Unmarshal JSON messages as to parse corresponding Service Class method parameters, Marshal output parameters of Service Class as JSON to reply through Unix Web Socket.
Service Class	Implements the intended functionality at service side for each RESTfull API verb.
Application	Use functionality exposed by the AGL Services to achieve a user-visible purpose.
APP Class	Exposes all RESTfull API verbs as methods with input and output parameters, Marshal parameters as JSON to send requests to the Unix Web Socket, Unmarshal JSON replies to update output parameters.
WebSocketApi	Handle Unix Web Socket connection, Form RESTfull API request, Wait for RESTfull API replies.

`raml2agl` features an automatic code generation tool developed. Fig. 5.2 shows the building blocks of the tool and its relations with the possible outputs.

As shown in Fig. 5.2, `raml2agl` generates code for the *Service Class*, *App Class*, and the *AGL Service*; the last two are fully generated. Note that the automatically generated components are the ones with more responsibilities, as shown in Table 5.1. This fact was also the rationale behind the definition of the components, to automate most

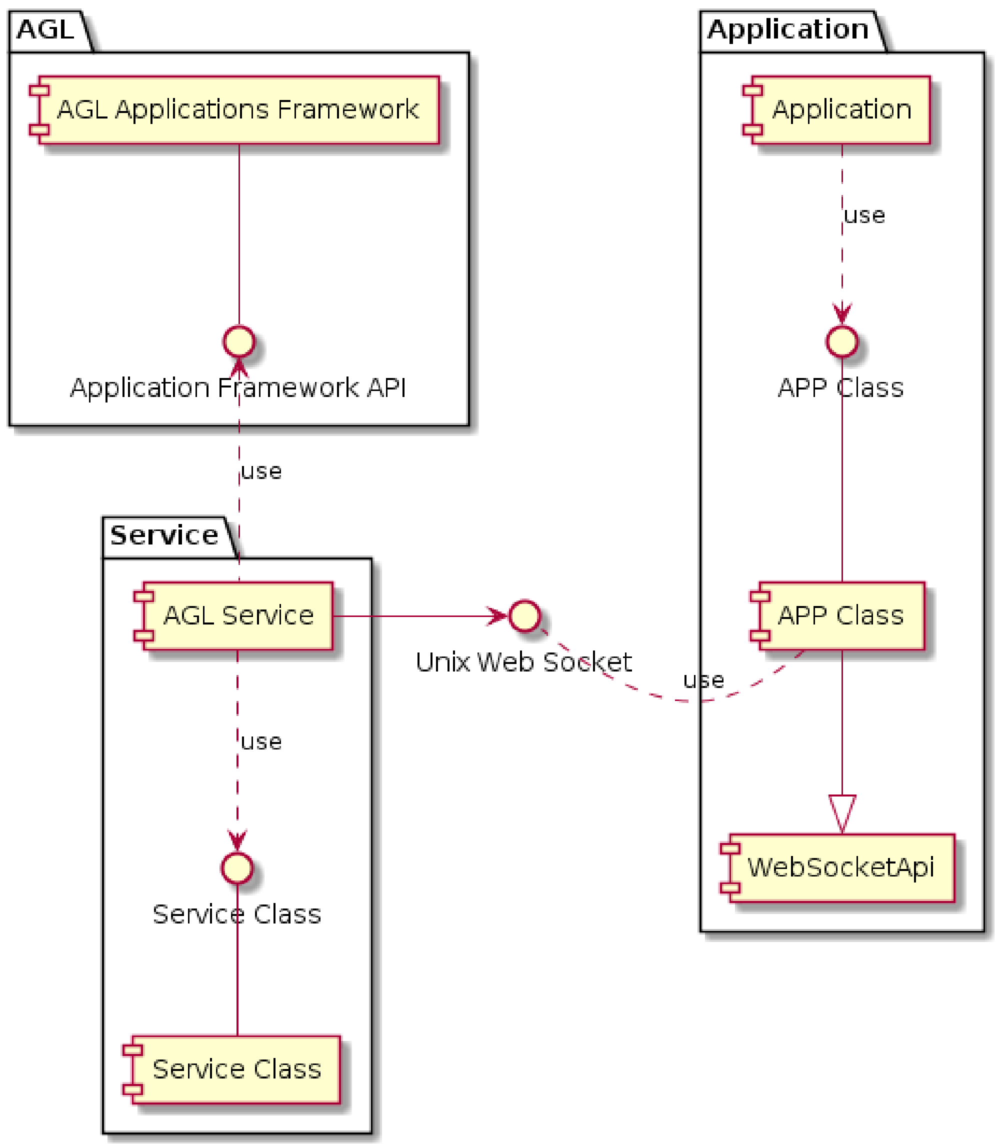

Fig. 5.1: Web Socket Communication Component Diagram

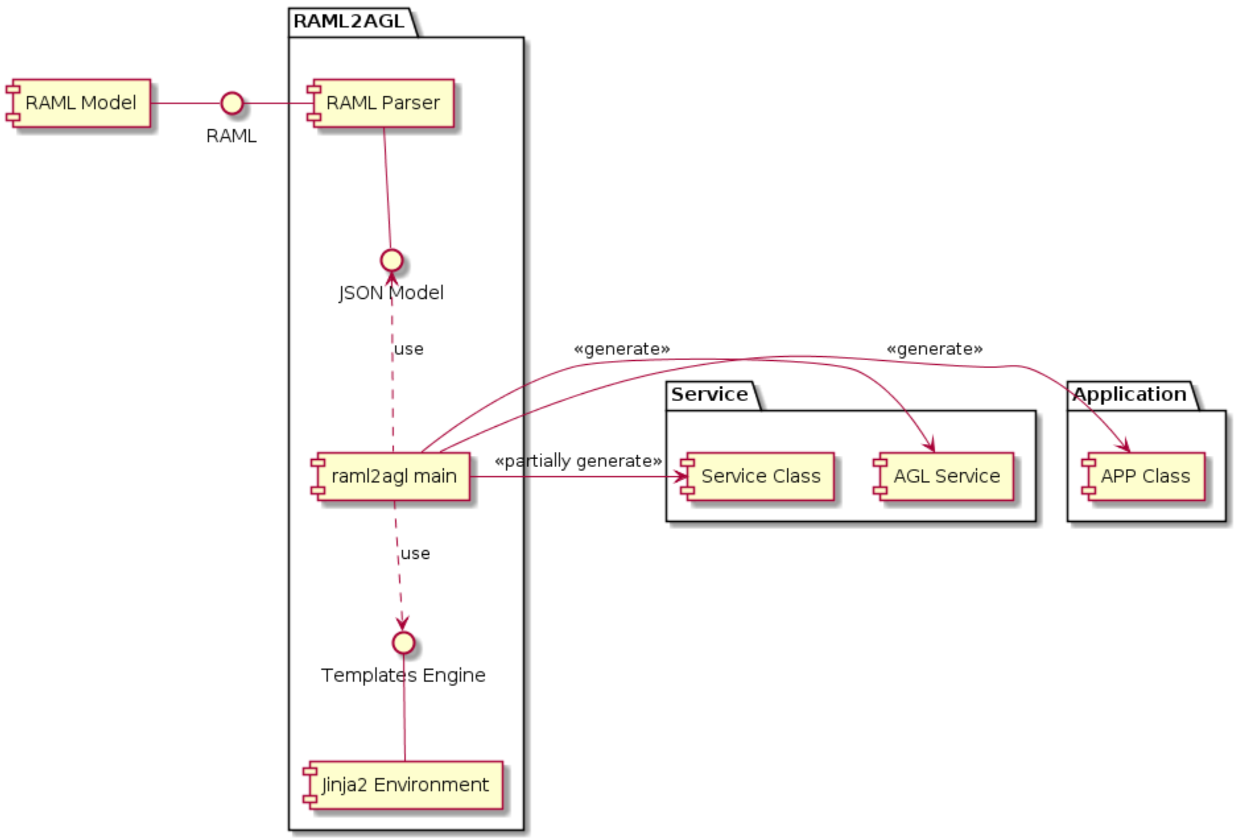

Fig. 5.2: RAML2AGL Block Diagram

of the process and reduce the overhead of creating a new *Service* and/or *Application*. Moreover, Table 5.2 shows the responsibilities of each of the `raml2agl` components.

Table 5.2: RAML2AGL Components Responsibilities

Name	Responsibility
RAML Parser	Read the RAML model and create a JSON model to be pass to the Jinja2 templates.
Jinja2 Environment	Manage the templates, render the templates using the JSON model.
raml2agl main	Read the RAML model from a file, Control the entire generation flow, reads input command line parameters, Calls the RAML Parser to generate JSON model, Calls the Jinja2 Environment to render the corresponding templates.

5.1 Service Class

Fig. 5.3 shows an example of the output of `raml2agl` using the following model;

```
#%RAML 1.0
title: Example
mediaType: application/json
version: v1
baseUri: localhost:8000/api?token=x
/method_1:
  post:
    body:
      properties:
        param_in_1:
          type: integer
  get:
    responses:
      200:
        body:
          properties:
            param_out_1:
              type: integer
/method_2:
  post:
    body:
      properties:
        param_in_1:
          type: string
  get:
    responses:
      200:
        body:
          properties:
            param_out_1:
              type: string
```

Note that *Service Class* isn't fully automatic generated. Nevertheless, a skeleton of the entire class with all the methods definition is generated. Is the task of the *Service* developer to finish the implementation of the functionality. Moreover, each method represents a verb of the RESTfull API. Hence, `/example/method_1` will shall be implemented in `ServiceExample.method_1(...)`. Furthermore, the model title is the parsed to name the RESTfull API and both classes.

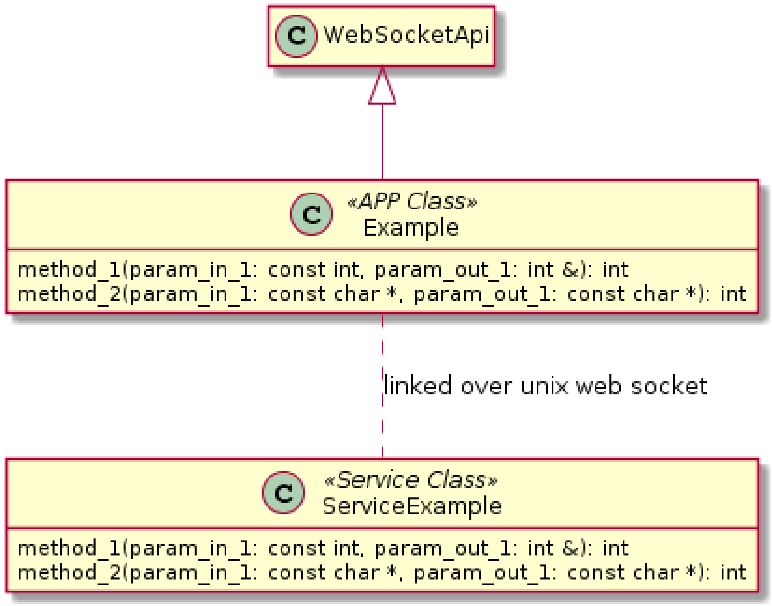

Fig. 5.3: Generated Example

5.2 WebSocketApi

Fig. 5.4 class diagram shows the definition of the `WebSocketApi` class.

Moreover, below the description of each of the classes members.

class WebSocketApi
Handle Unix Web Socket connection and transmission

Public Functions

WebSocketApi (**const** char **uri*, **const** char **api_name*)
Constructor

Creates Unix Web Socket connection and initialize the wait loop

Parameters

- `uri`: Base uri to the web socket
- `api_name`: API name

~WebSocketApi ()
Destructor

Releases the resources and disconnect from the Unix Web Socket

Protected Functions

json_object ***emit** (**const** char **verb*, **const** char **object*)
Send string to the specified API's verb

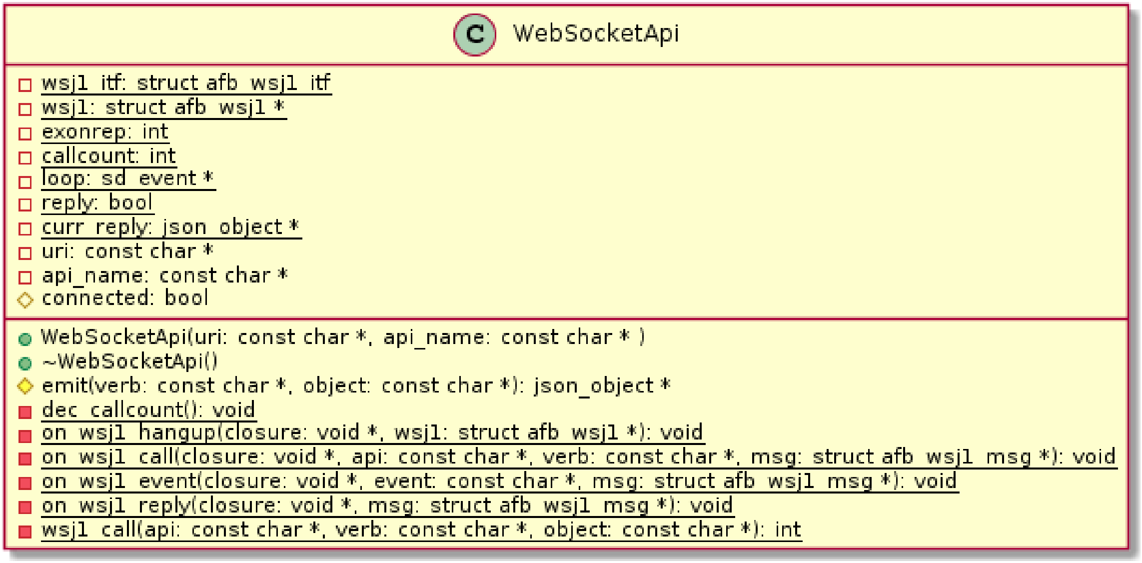

Fig. 5.4: Web Socket API Class Diagram

Return Reply JSON object

Parameters

- `verb`: API's verb
- `object`: Marshaled JSON object

Protected Attributes

bool **connected**
Flags connection status

Private Members

const char ***uri**
Base URI of the API

const char ***api_name**
API name

Private Static Functions

static void **dec_callcount** ()
Decrement the reference count of calls

static void **on_wsj1_hangup** (void **closure*, **struct** afb_wsj1 **wsj1*)
Hang up callback

Parameters

- `closure`: Hangup's closure

- `wsj1`: Connection object

static void **on_wsj1_call**(void **closure*, **const** char **api*, **const** char **verb*, **struct** afb_wsj1_msg **msg*)
Receives a method invocation callback

Parameters

- `closure`: Call's closure
- `api`: API Name
- `verb`: API's verb
- `msg`: Message to be sent

static void **on_wsj1_event**(void **closure*, **const** char **event*, **struct** afb_wsj1_msg **msg*)
Receive an event callback

Parameters

- `closure`: Event's closure
- `event`: Issued event
- `msg`: Received message

static void **on_wsj1_reply**(void **closure*, **struct** afb_wsj1_msg **msg*)
Receive a reply callback

Parameters

- `closure`: Reply's closure
- `msg`: Replied message

static int **wsj1_call**(**const** char **api*, **const** char **verb*, **const** char **object*)
Send a marshaled object to the specified API and API's verb

Return Return POSIX error codes

Parameters

- `api`: API name
- `verb`: API's verb
- `object`: Marshalled JSON object

Private Static Attributes

struct afb_wsj1_itf **wsj1_itf**
The Web Socket callback interface for wsj1

struct afb_wsj1 ***wsj1**
The Web Socket connection object

int **exonrep**
The Web Socket connection object

int **callcount**
Calls Reference counter

sd_event ***loop**
Wait loop event

bool **reply**
    Flags the presens of a reply

json_object ***curr_reply**
    Last received JSON object

5.3 APP Class

As shown in Fig. 5.3 the `Example` *APP Class* has symmetric methods with `ServiceExample`. Therefore, a call to `Example.method_1` will call `/example/method_1` RESTfull API through the Unix Web Socket. Note that every *APP Class* is completely automatically generated. Moreover, *APP Class* inherits `WebSocketApi` and implements the entire Unix Web Socket communication its methods.

5.4 AGL Service

An AGL service is basically the implementation of the *Application Framework API* shown in Fig. 5.5.

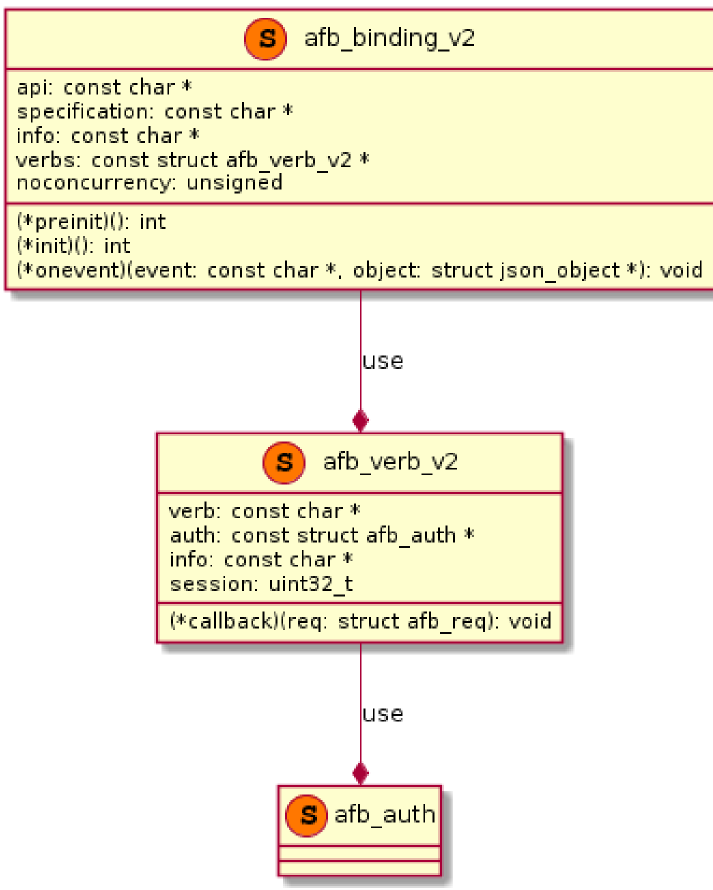

Fig. 5.5: AGL Application Framework API *[29]*

Furthermore, to implement Fig. 5.3, for instance, a null-terminated list of verbs has to be defined as follows;

```
static const struct afb_verb_v2 verbs[] = {
      /*Without security*/
      {.verb = "method_1", .callback = method_1, .auth = NULL, .info = "method_1", .
↪session = 0},
      {.verb = "method_2", .callback = method_2, .auth = NULL, .info = "method_2", .
↪session = 0},
      {.verb = NULL, .callback = NULL, .auth = NULL, .info = NULL, .session = 0 }
};
```

Note that for an initial implementation the authentication mechanisms weren't implemented. Nevertheless, it has been included in the `raml2agl`'s road map, see *[22]*.

And finally, to register the entire API to the *AGL Application Framework* the `afb_binding_v2` structure is automatically generated as follows.

```
const struct afb_binding_v2 afbBindingV2 = {
  .api = "example",
  .specification = "",
  .info = "Auto generated - Example",
      .verbs = verbs,
      .preinit = NULL,
  .init = init,
      .onevent = NULL,
  .noconcurrency = 1
};
```

5.5 RAML Parser

Fig. 5.6 presents the internals of the RAML Parser component. Furthermore, the responsibilities of each of the sub-components are stated in Table 5.3

Table 5.3: RAML2 Parser Sub-components Responsibilities

Name	Responsibility
Root Attributes Parser	Parse the RAML root attributes like; title and base URI.
Methods Parser	Parse the RAML verbs as methods
Input Parameters Parser	Parse the RAML verbs' input parameters
Output Parameters Parser	Parse the RAML verbs' output parameters
Types Parser	Parse the RAML verbs' parameters' types

5.6 raml2agl main

Fig. 5.7 presents the internals of the *RAML2AGL main* component. Furthermore, the responsibilities of each of the sub-components are stated in Table 5.4

Table 5.4: RAML2AGL main Sub-components Responsibilities

Name	Responsibility
Command Line Arguments Parser	Parses the command line arguments to configure the *File Generator*.
Templates Filters	Defines Jinja2 Template filters to convert data types from RAML format to C++.
Files Generator	Passes the JSON model to render the templates to be built and write files to the selected output location.

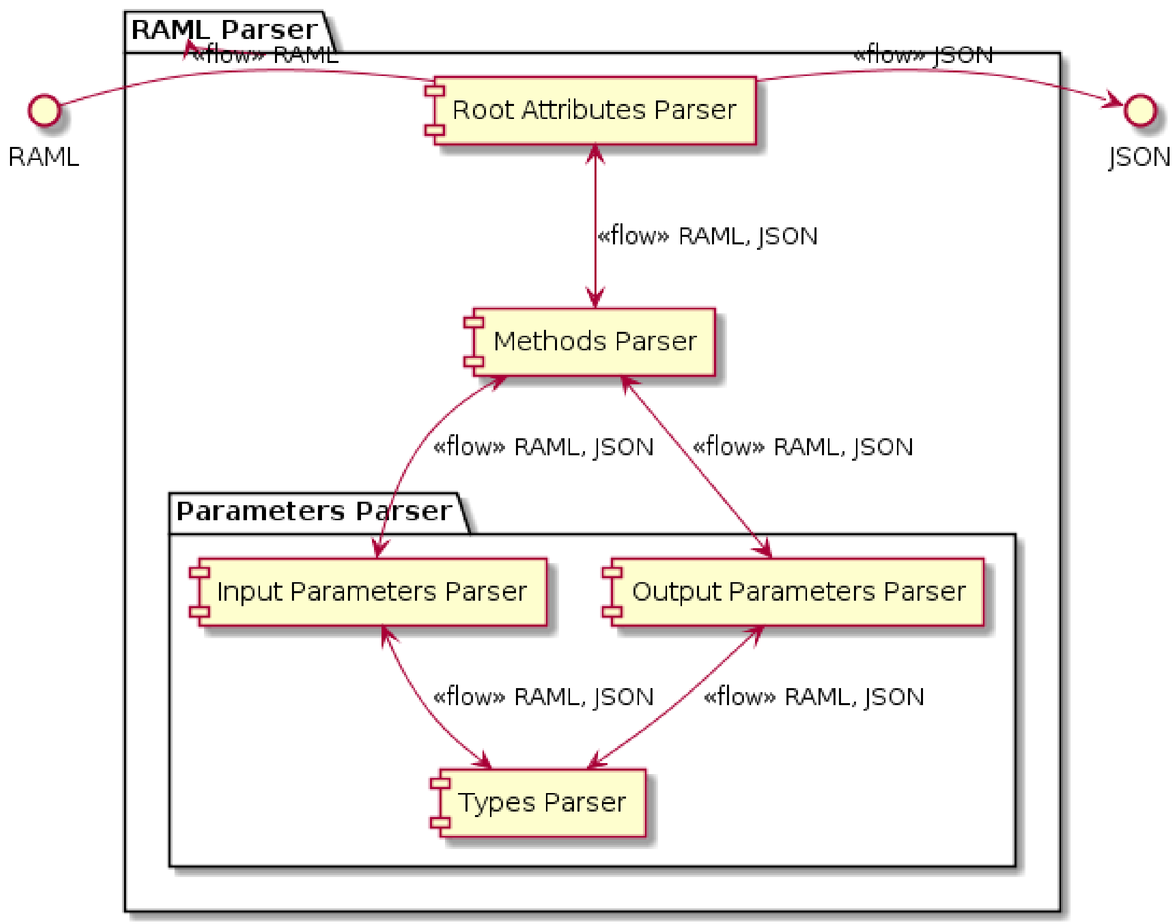

Fig. 5.6: RAML Parser Block Diagram

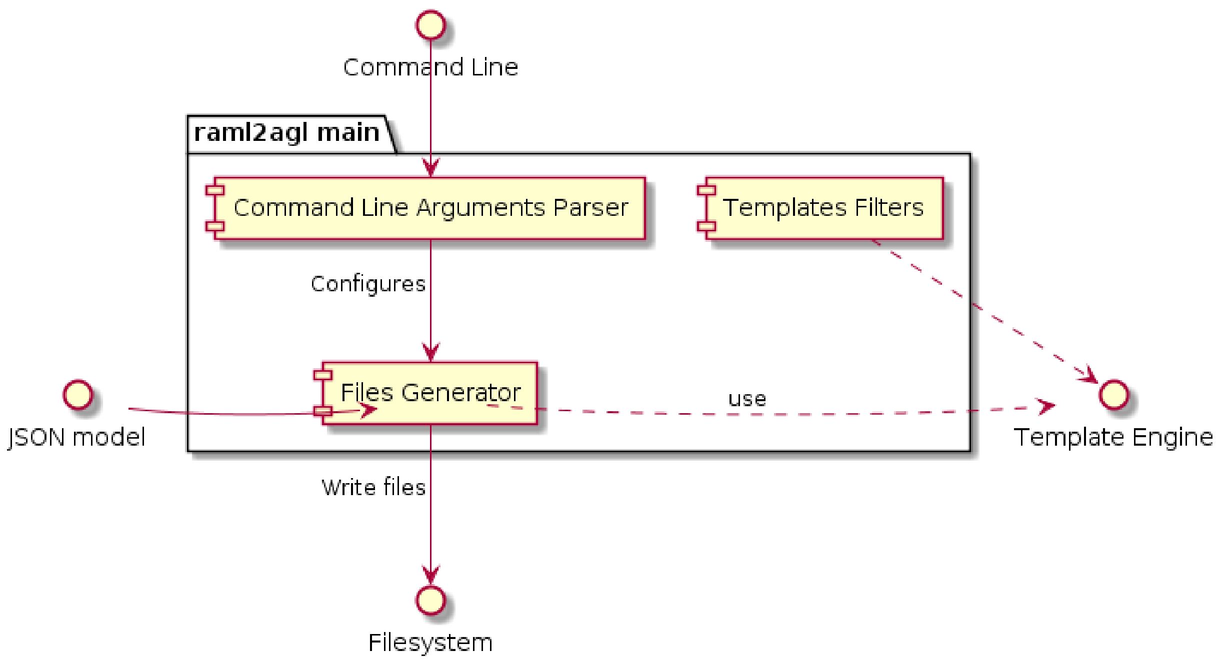

Fig. 5.7: RAML2AGL main Block Diagram

CHAPTER 6

Runtime View

6.1 RAML2AGL Generation

Fig. 6.1 presents the sequence of the `raml2agl` run for automatically generate *APP Class*, *WebSocketApi*, *AGL Service* and *Service Class*.

6.2 AGL Service Start

It's important to have some insight on how *AGL Services* are initialized and how the *Unix Web Socket* gets created. Therefore, Fig. 6.2 shows this process.

6.3 Web Socket Communication

The *Web Socket Communication* can only happen after the AGL Service is already running, thus the *Unix Web Socket* was already created and the RESTfull API is being served. Fig. 6.3 shows the sequence how the entire communication takes place.

Note that the *Application* using the *APP Class* will have the entire Web Socket communication abstracted as simple method calls. Hence, an RPC model is implemented on top of the RESTful API. Fig. 6.4 shows this abstracted communication sequence.

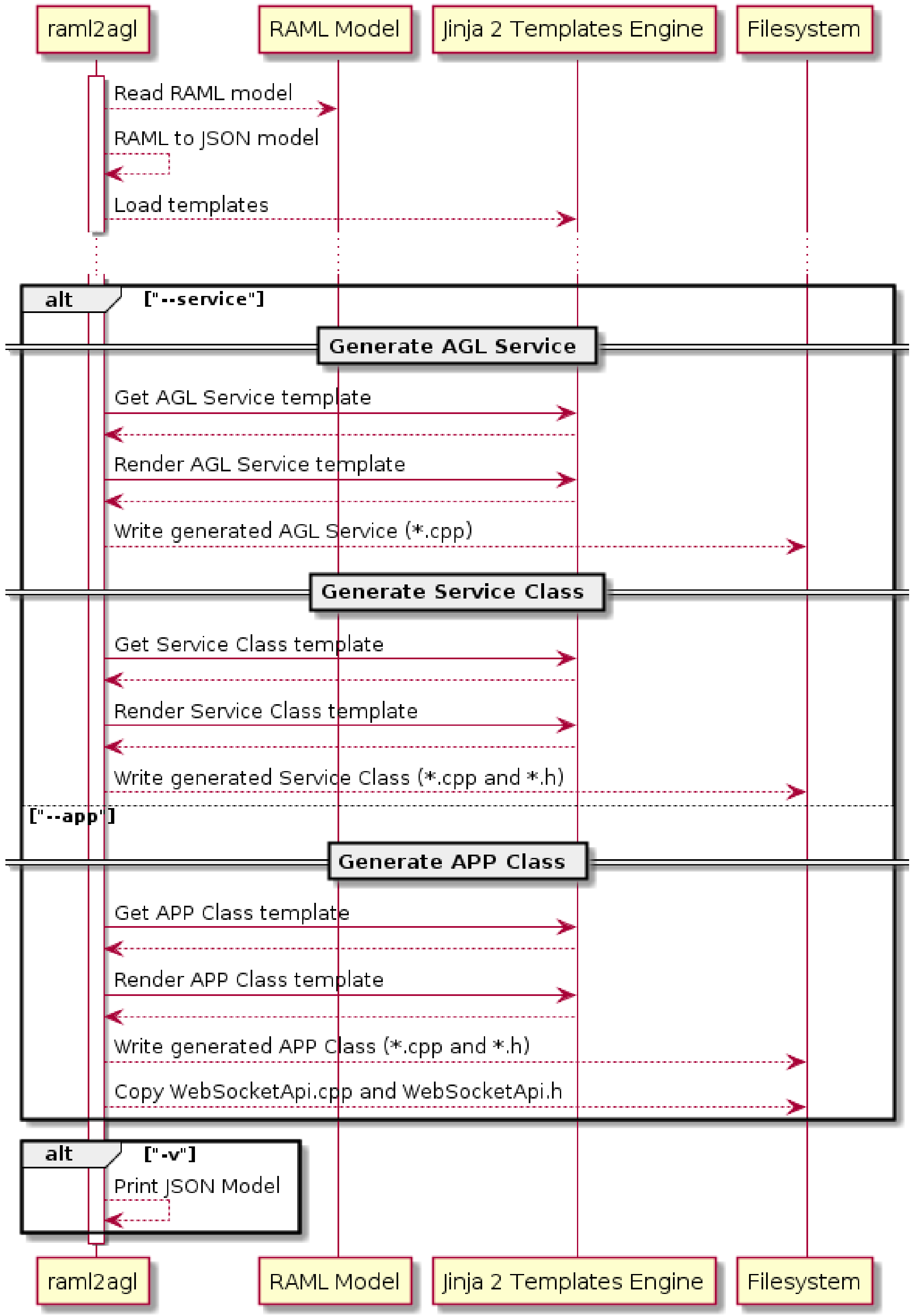

Fig. 6.1: RAML2AGL Generation

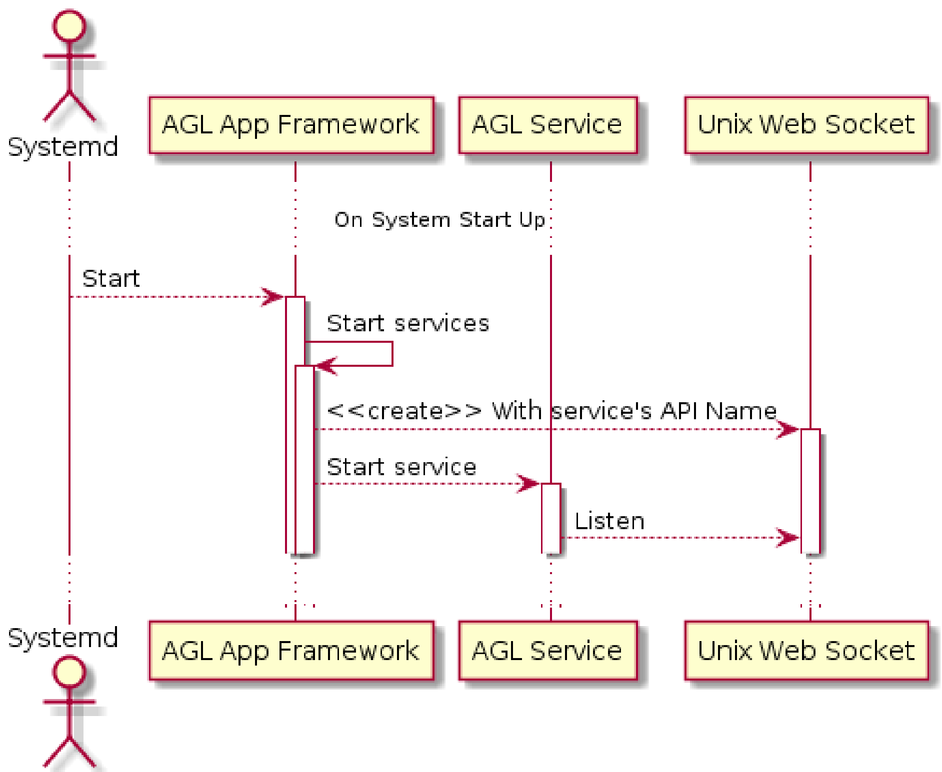

Fig. 6.2: AGL Service Start

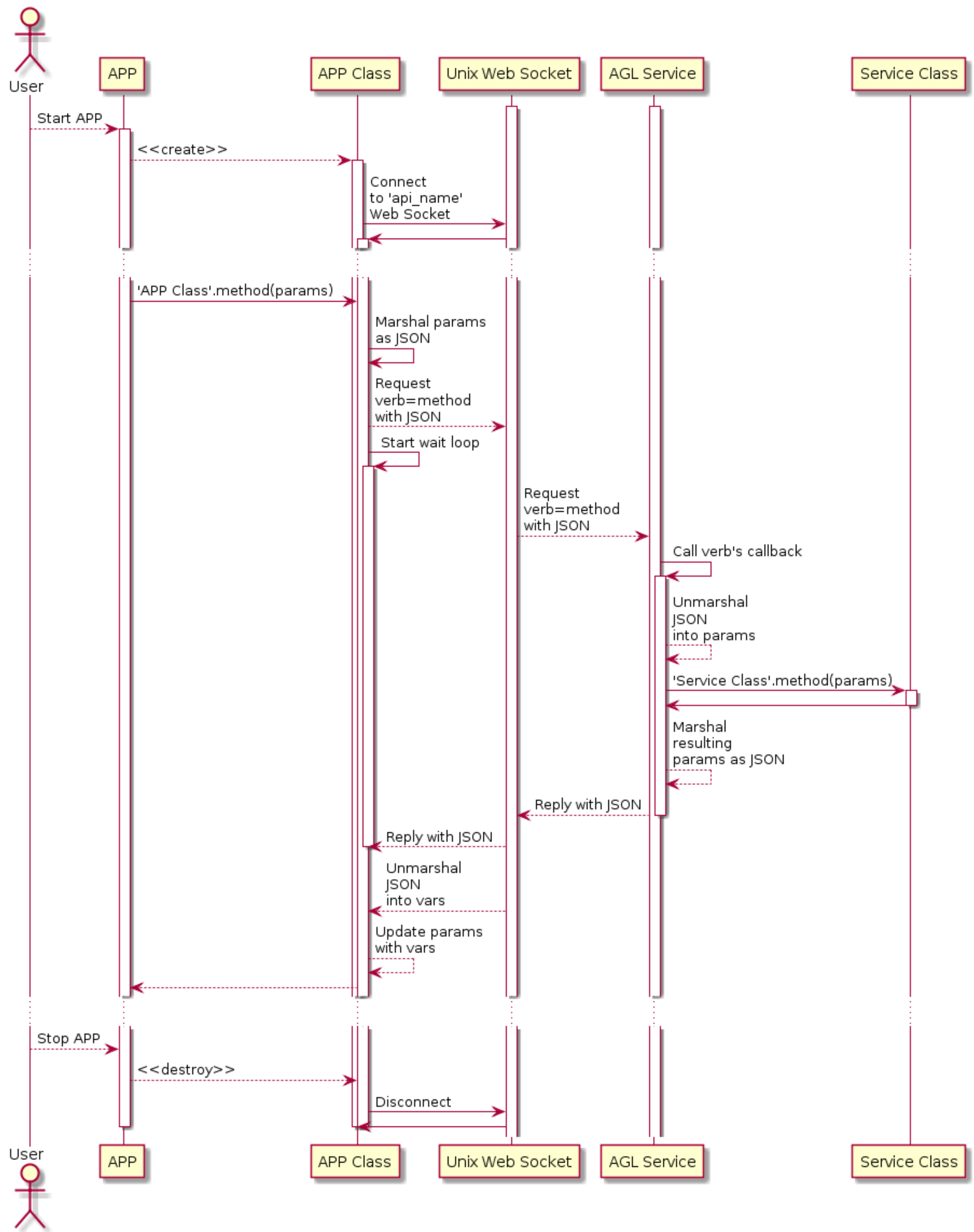

Fig. 6.3: Web Socket Communication

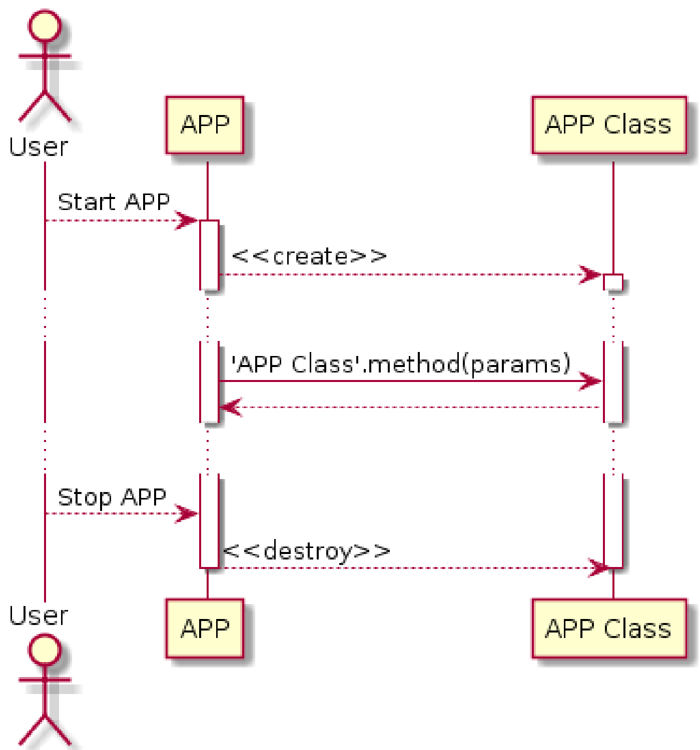

Fig. 6.4: Web Socket Communication

CHAPTER 7

Deployment View

Fig. 7.1 and Fig. 7.2 show the structure of the `raml2agl` repository. Note that `src/template/` directory holds all the templates that feed the *Jinja2 Environment* to generate the components also shown in the corresponding diagram.

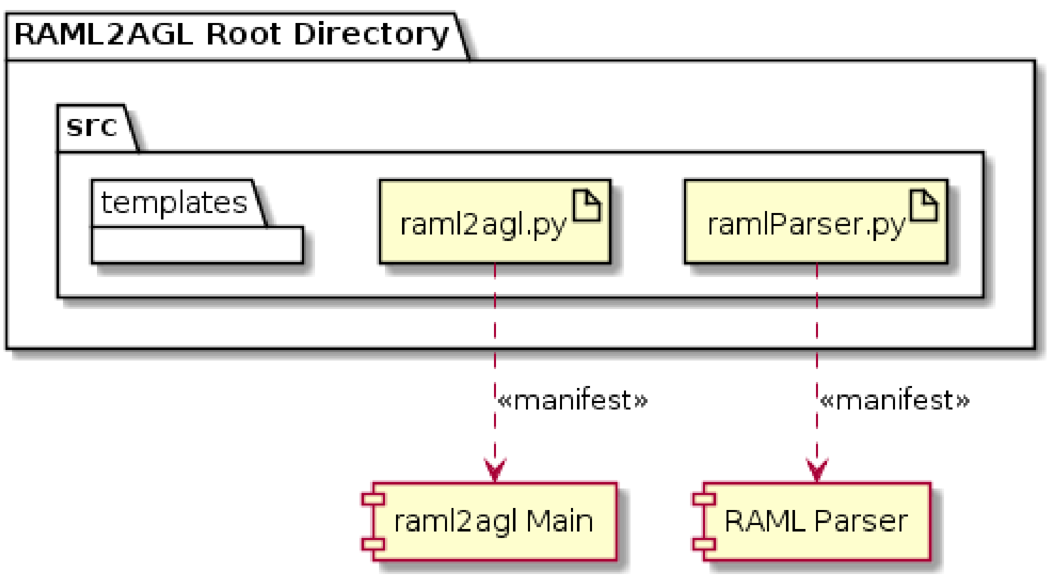

Fig. 7.1: Deployment Diagram of RAML2AGL Root

Moreover, the Application and Service source files are separately compiled and deployed at different abstraction layers within the AGL architecture. Fig. 7.3

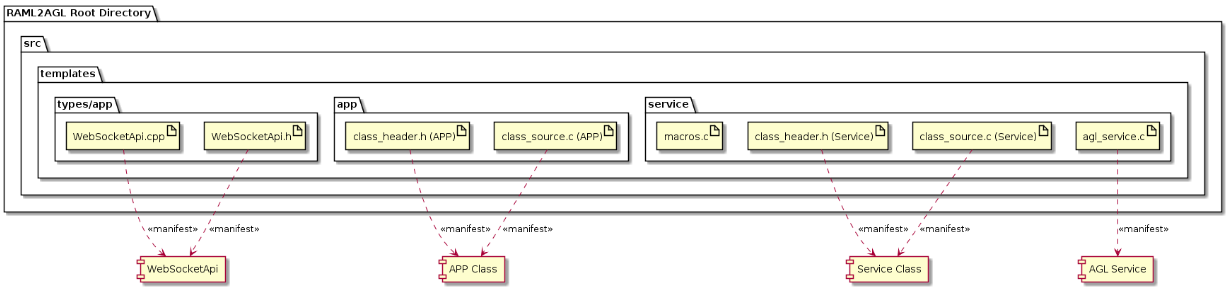

Fig. 7.2: Deployment Diagram of RAML2AGL Root (With Templates)

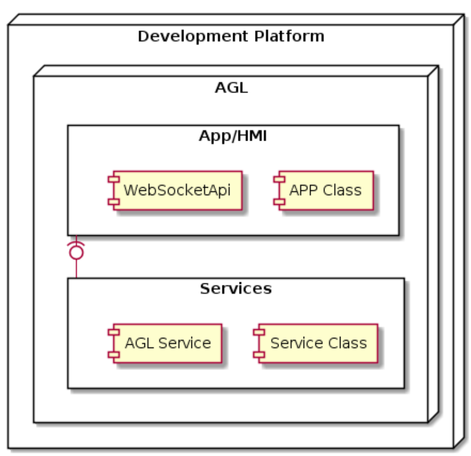

Fig. 7.3: Deployment Diagram of RAML2AGL Root (Runtime)

CHAPTER 8

Cross-cutting Concepts

8.1 RPC over Web Socket

Since the `raml2agl` implements an RPC over a Web Socket, Fig. 8.1 shows a generic RPC and Fig. 8.2 shows a generic Web Socket communication. Note that in order to communicate over Web Socket a connection between *Client* and *Server* has to be acknowledged. Similarly, the connection has to be closed once it's not going to be used anymore. This part is handled in the *WebSocketApi* constructor and destructor, respectively. Moreover, the *APP Class* and the *AGL Service* handle the messaging and thus simulating an RPC.

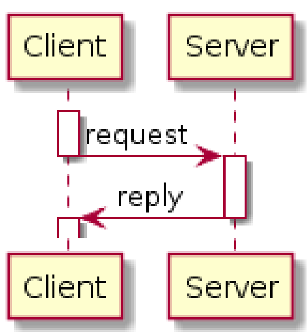

Fig. 8.1: RPC Model

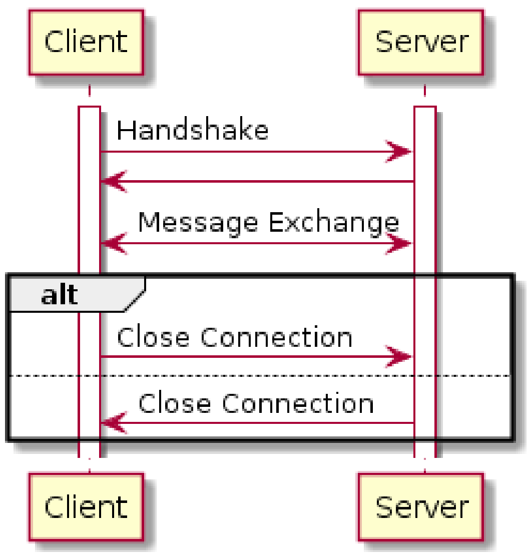

Fig. 8.2: Web Socket Model

CHAPTER 9

Design Decisions

9.1 RESTful Modeling Language Selection

There is a handful of Modeling Language that can be used for modeling RESTful APIs. The main criteria to select the modeling language to be used was that it has to be machine- and human-readable format, filtering the possibilities to those using JSON and YAML formats. Options like *API Blueprint* were filtered out because it's written using Markdown which is more human-readable but much less machine-readable. In contrast, XML-based modeling languages were also left out, because it is not a human-readable format.

The analysis was, therefore, focus on *OpenAPI* and *RAML*. Nevertheless, after analyzing their specifications *[19]* and *[18]*, RAML was considered to be equally descriptive and much less verbose.

9.2 Python for `raml2agl`

Python was selected to develop `raml2agl`, because of its simplicity. Also, there are many Python libraries that make the development process faster and easier. For instance, Jinja2 makes the entire automatic code generation with relatively less effort. Python YAML parsing library is also used for RAML parsing. Moreover, Python's dictionaries are a key language feature that proofs to be useful for parsing file's content. As shown in *[15]* doesn't perform the best compared to a comparable implementation in other languages. Nevertheless, a high performance isn't required from `raml2agl` since the code generation isn't being done online nor frequently.

9.3 RAML Parser vs `pyraml-parser`/`ramlifications`

Even though there are reference implementations of a RAML parser called, `pyraml-parser` and `ramlifications`, it was decided to not use them for now since they only support up to RAML 0.8, whereas `raml2agl` plans to support RAML 1.0.

This fact adds a little overhead to the development and also includes some risks (discussed in *PyRAML/ramlifications Adoption*). Nevertheless, the RAML Parser didn't represent much effort to develop and generates the expected behavior efficiently.

Since `ramlifications` plans to support RAML 1.0 *[36]*, it might be a good idea to integrate it into the RAML Parser once it's supported.

9.4 RPC over Web Socket Communication

Web Socket communication is a powerful communication and design pattern. For instance, Web Socket Communication enables bi-directional and asynchronous communication. Whereas, RPC is a unidirectional and synchronous communication.

Therefore, implementing an RPC on top of Web Socket Communication means losing some communication capabilities. This design decision is probably the most important done regarding the MDD approach.

Even when the RPC communication model isn't desired, `raml2agl` can still be used. For instance, it can still be used to automatically generate the *AGL Service* and the *Service Class*, since the RPC model is only implemented in *APP Class* and *WebSocketApi*.

CHAPTER 10

Quality Requirements

In this chapter, the quality requirement presented in *Quality Requirements* are evaluated. Besides, other quality aspects are also introduced an evaluated.

`raml2agl` tool fulfills Transparency (REQ_001) by maintaining a clear mapping between the *Service Class*'s and the *APP Class*'s methods. Hence creating as well an Object Oriented interface that abstracts the Unix Web Socket communication and thus fulfilling Abstraction (REQ_002) as well.

The adoption of RAML as the interface modeling language speaks for the fulfillment Standardization (REQ_003). Moreover, `raml2agl` uses broadly adopted tools, such as Jinja2. Also, `raml2agl` follows standard coding styles such as the Kernel's coding style and PEP8. Both broadly adopted tools and the use of standard coding styles, also contribute towards Standardization (REQ_003) fulfillment.

`raml2agl` allows the user to set the output directories and decide what components to generate. Additionally, by supporting RAML `raml2agl` enables the user to generate a wide variety of interfaces. These are two already-implemented customization mechanisms for the proposed MDD approach. Therefore fulfilling Flexibility (REQ_004).

As for Testability and Debugability (REQ_005), generates intermediate probing points with well-defined interfaces which allows the user to develop unit testing for the system's main components. For instance, the AGL service developer can create unit testing for the *Service Class*, which would test the actual AGL Service's purpose. Similarly, the AGL Service developer could interact with the RESTful interface directly using tools like Postman *[12]*. This will test a different aspect of the components interaction, which is the marshaling and unmarshaling of the JSON in the AGL Service side, as well as the mapping with the *Service Class*'s methods. In the *APP Service* side, a similar testing can be done to verify the marshaling and unmarshaling of the methods' parameters into JSON.

By defining a standard interface also enables a decoupled development process, where AGL Service and AGL Application can be developed in parallel. Moreover, mocking *[10]* mechanisms can be easily implemented using the interface's definition. For instance, the *APP Service* interface could be mocked using *Google Test [24]*, thus enabling testing at AGL application level without the need of running in the actual system, which at the same time enables faster development.

Interestingly, the mocking and components unit tests can be also automatically generated out of the RAML model, also contributing towards Flexibility (REQ_004). Moreover, by having a deterministic mapping between the RAML model and the components' behavior correctness can be verified once and guaranteed for everyone *[33]*, thus minimizing the testing effort. Note that correct memory management is also considered part of the code's behavior correctness as it's one of the system's constraints (*TC3*) as specified in *Constraints*. For instance, all the developed unit testing could be tested under memory management checking tools such as `valgrind` to validate its correctness. By doing so, the memory management correctness is verified without any more testing effort since the same unit tests are run, but on top of `valgrind`. In fact, this was done while testing `raml2agl`'s behavior.

CHAPTER 11

Risks and Technical Debts

Each of the subsections discusses a risk and technical debt aspect.

11.1 PyRAML/ramlifications Adoption

As mentioned before, `pyraml` nor `ramlifications` weren't adopted to develop `raml2agl`. This leaves an important technical debt since the compliance with the RAML standard isn't verified in the implemented RAML parser. Meaning, that some modeling language syntax error in an input RAML model wouldn't be caught.

Moreover, the by the time of writing this document, the RAML parser doesn't support all RAML 1.0 features but are being increasingly supported. Thus, creating a gap between the RAML 1.0 modeling features and the `raml2agl` features. Nevertheless, the most important RAML 1.0 modeling features are supported in `raml2agl`. Please review *[22]* for an updated list of RAML 1.0 supported features.

11.2 RPC Limitations

The use of an RPC communication model on top of Web Socket represents a risk and technical debt since some applications might work better on top of the raw Web Socket communication. Nevertheless, `raml2agl` can still be used for the client side automatically code generation as mentioned in *RPC over Web Socket Communication*.

CHAPTER 12

Glossary

RESTful API An API that uses GET, PUT, POST, and DELETE HTTP requests to expose the functionalities.

RAML RESTful API Modeling Language

Web Socket A networks communication protocol, over TCP, located at layer 7 of the OSI model.

RPC Remote Procedure Call is when a section of a program's code is executed in a different address space or system.

MDD Model Driven Development

Bibliography

[1] Api blueprint. URL: https://apiblueprint.org/.

[2] App template. URL: https://gerrit.automotivelinux.org/gerrit/p/apps/app-templates.git.

[3] Apache ant. URL: http://ant.apache.org/.

[4] Build, test and package your software with cmake. URL: https://cmake.org/.

[5] Gnu bash. URL: https://www.gnu.org/software/bash/.

[6] Gradle. URL: https://gradle.org/.

[7] Gradle - wikipedia. URL: https://en.wikipedia.org/wiki/Gradle.

[8] *Linux kernel coding style*. URL: https://www.kernel.org/doc/html/v4.10/process/coding-style.html.

[9] Mqtt. URL: http://mqtt.org/.

[10] Mock object. URL: https://en.wikipedia.org/wiki/Mock_object.

[11] *PEP 8 – Style Guide for Python Code*. URL: https://www.python.org/dev/peps/pep-0008/.

[12] Postman. URL: https://www.getpostman.com/.

[13] Pyraml. URL: https://github.com/an2deg/pyraml-parser.

[14] Python. URL: https://www.python.org/.

[15] Python 3 programs versus c++ g++. URL: https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gpp.

[16] Raml org. URL: https://raml.org/.

[17] Raml projects. URL: https://raml.org/projects.

[18] *RAML Version 1.0: RESTful API Modeling Language*. URL: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/.

[19] The openapi specification. URL: https://github.com/OAI/OpenAPI-Specification.

[20] Welcome to apache maven. URL: https://maven.apache.org/.

[21] Repo. URL: https://gerrit.googlesource.com/git-repo.

[22] Pedro Cuadra. Raml to agl. URL: https://github.com/pjcuadra/raml2agl.

[23] Pedro Cuadra. Websocketapi.cpp. URL: https://github.com/pjcuadra/raml2agl/blob/master/src/templates/types/app/WebSocketApi.cpp.

[24] Google. Googletest. URL: https://github.com/google/googletest.

[25] Google. Protobuffers. URL: https://developers.google.com/protocol-buffers/.

[26] Automotive Grade Linux. Apis & services. URL: http://docs.automotivelinux.org/docs/apis_services/en/dev/.

[27] Automotive Grade Linux. About. URL: https://www.automotivelinux.org/about.

[28] Automotive Grade Linux. Architecture guide. URL: http://docs.automotivelinux.org/docs/architecture/en/dev/.

[29] Automotive Grade Linux. Bindings reference. URL: http://docs.automotivelinux.org/docs/apis_services/en/dev/reference/af-binder/afb-binding-references.html.

[30] Automotive Grade Linux. Developer guides. URL: http://docs.automotivelinux.org/docs/devguides/en/dev/.

[31] Automotive Grade Linux. The application framework daemons. URL: http://docs.automotivelinux.org/docs/apis_services/en/dev/reference/af-main/1-afm-daemons.html.

[32] Automotive Grade Linux. Automotive grade linux requirements specifications. May 2015. URL: http://docs.automotivelinux.org/docs/architecture/en/dev/reference/AGL_Specifications/agl_spec_v1.0_final.pdf.

[33] Collin O'Halloran. Model base code verification. *Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings*, 2003.

[34] Raspberrypi.org. *RASPBERRY PI 3 MODEL B*. Raspberrypi.org. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[35] Armin Ronacher. *Welcome to Jinja2*. URL: http://jinja.pocoo.org/docs/2.10/.

[36] Spotify. Ramlfications. URL: https://github.com/spotify/ramlfications.

Index

M

R

W