AGL Continuous Integration tests

Documentation and methodology

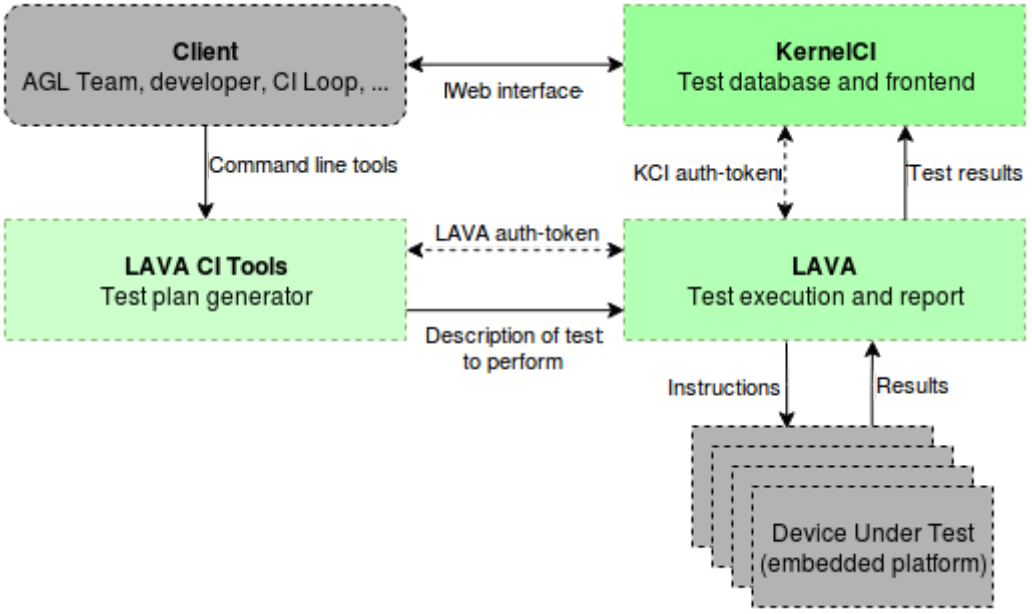
Purpose

The purpose of this paper is to provide documentation to the reader on how to write Automotive Grade Linux (AGL) Continuous Integration (CI) tests.

Introduction

The AGL Continuous Integration loop is split into three main components:

- LAVA CI Tools: a set of tools to generate test plans.
- LAVA: a solution that deploys operating systems onto hardware for running tests.
- KernelCI: a web service which records test results and presents them in a tailored way for the client (AGL teams, developer, tester, automation frameworks, …).



AGL CI Architecture overview

The purpose of this document being how to write tests for AGL. It will specify how to use the LAVA CI Tools and how to extend the set of available tests by writing new custom tests.

The document also explains how the tests are run by LAVA and parsed by Kernel CI to help the user write meaningful tests. However, it does not document LAVA and Kernel CI internal test handling.

Prerequisites

A general understanding of the AGL project helps but is not mandatory.
LAVA CI Tools are written with Python and uses templates formatted with jinja2.
To write new tests the user needs:

- A text editor

To generate new test plans:

- Python version >= 2.7.1
- Jinja2 version >= 2.9

To submit tests to a LAVA instance:

- The URL of a running LAVA instance
- An authentication token

To have the tests recorded in a KernelCI instance:

- The URL of a running KernelCI instance
- A “callback” authentication token and the corresponding LAVA lab name

Test jobs

This section documents how the tests are received, parsed and executed by LAVA on the specified hardware. The test jobs are sent to LAVA as YAML files. Each YAML file is called a job, because it is interpreted by LAVA as a job to execute. Hence, we will send files to lava such as: “m3ulcb_AGL_smoketests_job.yaml”.

Mandatory sections

There are several sections in the job files. Some are mandatory such as: device_type [1], protocols[2], actions[3] and timeouts[4]. These sections specify all the data that has to be known by LAVA for executing the job onto a specific device with the correct OS and boot protocol. Timeouts are specified as well in case the board goes into an unknown state.

Other mandatory fields are: job_name, priority, visibility and notify. These sections are used by LAVA to manage the pool of jobs received and to report each job.

Boot and test sections

With the mandatory fields defined, LAVA has all the data it needs to execute the job. However it does not contain the job’s instructions. For that, other sections are added to the job file. A “boot” section which is device specific. It tells LAVA how to boot the platform and how to get a prompt for executing further instructions. Finally, the “test” section describing the tests that we would like to perform on the platform.

Notify section

Looking at “*AGL CI Architecture overview*” page 3. Our job definition could now be sent from the bottom left “LAVA CI Tools” to the bottom right “LAVA” block. The LAVA instance would be able to perform the tests on the specified platform. However, the job description does not specify yet information to communicate with a KernelCI instance (top right). For this purpose a notify section has to be added. This section contains the URL of the KernelCI instance to submit job results to as well as an authentication token.

[1] Lava doc for device_type.
[2] Lava doc for protocols.
[3] Lava doc for actions.
[4] Lava doc for timeouts.

```yaml
job_name: AGL-short-smoke-wip
priority: medium
visibility: public

notify:
  criteria:
    status: finished
  callback:
    url: http://api.dev.baylibre.com/callback/lava/test?lab_name=lab-baylibre-dev&status={STATUS}
    method: POST
    dataset: all
    token: 12345678-XXXX-XXXX-XXXX-123456789012
    content-type: json

device_type: r8a7796-m3ulcb

timeouts:
  job:
    minutes: 30
  action:
    minutes: 15
  connection:
    minutes: 5

protocols:
  lava-xnbd:
    port: auto

actions:
- deploy:
    timeout:
      minutes: 15
    to: nbd
    os: oe
    failure_retry: 2
    kernel:
      url: http://www.baylibre.com/pub/agl/ci/m3ulcb-nogfx/Image
    initrd:
      url: http://www.baylibre.com/pub/agl/ci/m3ulcb-nogfx/initramfs-netboot-image-m3ulcb.ext4.gz
      allow_modify: false
    nbdroot:
      url: http://www.baylibre.com/pub/agl/ci/m3ulcb-nogfx/core-image-minimal-m3ulcb.ext4.xz
      compression: xz
    dtb:
      url: http://www.baylibre.com/pub/agl/ci/m3ulcb-nogfx/Image-r8a7796-m3ulcb.dtb

- boot:
    timeout:
      minutes: 10
    method: u-boot
    prompts: ["root@m3ulcb:~"]
    auto_login:
      login_prompt: "login:"
      username: root
    type: booti
    commands: nbd
    transfer_overlay:
      download_command: wget
      unpack_command: tar -C / -xvpf

- test:
    definitions:
    - repository: git://git.linaro.org/qa/test-definitions.git
      from: git
      path: automated/linux/smoke/smoke.yaml
      name: smoke-tests
```

Example of test plan as sent to LAVA: “m3ulcb_AGL_smoke.yaml”

Generate test jobs

Our interest being running tests; we will use test generation tools to automatically generate the mandatory, boot and callback sections with default and/or device specific, values. We will see however, how to write new test sections and how to use the tools to include them in a job file.

The tools for generating test jobs are part of `releng-scripts`. It is within this repo that the list of available tests is located.

Download the tools

releng-scripts is available via git. To get the code, just clone the repo locally:

```
$ git clone https://git.automotivelinux.org/AGL/releng-scripts/
```

The up to date documentation of the tools is available in the README.md. The tools are located in the ./utils/ folder and also provide "--help" documentation for usage.

Generate jobs

The generation tool is called: create-jobs.py

```
$ ./utils/create-jobs.py m3ulcb -o myjob.yaml
```

This command will generate the default job file for the specified board. In the examples, we specified to create a new job for the Renesas m3ulcb board and write the output to the file: "myjob.yaml"

To add test plans to the job just append the command with --test "test_plan_name" to execute only a specific test plan or "all" to run all available tests plans for the platform.

```
$ ./utils/create-jobs.py m3ulcb -o myjob.yaml --test all
```

To add callbacks to submit the job results to a KernelCI instance append the command with --callback "lava-lab-name". Within releng-scripts, the callback folder contains predefined LAVA labs and corresponding KernelCI instances to submit the results to. The lava-lab-name defines the lab name as registered with the kernelCI backend instance and the authentication token to use.

```
$ ./utils/create-jobs.py m3ulcb -o myjob.yaml --test all --callback lab-baylibre-lavabox
```

Test definitions

The test plans definitions are located in the ./templates/tests/ folder. Each file within this folder describes a test plan. Every test plan may contain several test definitions. Further in the doc is documented how to add test plan to the existing pool, and how to write test definitions.

Submit test jobs

The tool for submitting test jobs is called lava-tool. The documentation for using the tool to submit jobs to a LAVA instance is available here.

Installing lava-tool

The latest version of lava-tool is available in the Debian jessie-backports: lava-tool.
Just install it like a classic deb file.

Adding authentication credentials to lava-tool

Before submitting jobs the user needs to add his authentication credentials to lava-tool.
Let's assume that the administrator created a new user credential for the lava instance:

- http://lavabox:10080

The created user credentials are:

- Username: "demo"
- Token: "tokendemo"

The user would just have to run this command to add his credentials:

```
$ lava-tool auth-add http://demo@lavabox:10080/RPC2/
Paste token for http://demo@lavabox:10080/RPC2/: <Paste token: tokendemo>
Token added successfully for user demo.
```

In case you have problems with lava-tool saving tokens such as having to type a password. You can edit the python keyring file to use plain text keyrings:

```
$ vi  ~/.local/share/python_keyring/keyringrc.cfg
[backend]
default-keyring=keyring.backends.file.PlaintextKeyring
```

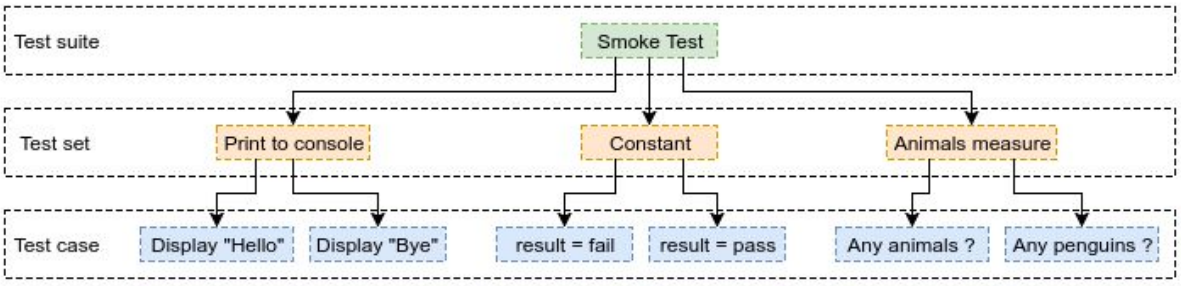
Submitting jobs to LAVA

The user can now use submit-jobs to submit any generated job to the lava instance. Supposing that the myjob.yaml file previously generated is in the current path:

```
$ lava-tool submit-job http://demo@lavabox:10080/RPC2/ myjob.yaml
submitted as job: http://lavabox:10080/scheduler/job/16
```

AGL test structure

The AGL tests are organised using a tree structure. The smallest block is called a test case and specifies a single operation to be performed with a single result. Test cases are grouped together in test sets. And tests sets are themselves grouped in test suites. The test plans will contain one or more test suite (themselves containing sets and cases) in their definition.



AGL Tests structure

Test plan

From the test job file described in [Test jobs] LAVA will parse and execute the test plan. The test plan corresponds to the list of test definitions section:

```
- test:
    definitions:
        # The test suite definition that will be parsed and executed goes here
- test:
    definitions:
        # A second test suite
- test:
    definitions:
        # A third test suite, all part of the same test plan
```

Test definition will be reported as test suites. The test suite will be executed in LAVA Test Shells. The LAVA Test Shell, is based on the available shell for each platform. The set of commands is extended with “lava-test-case” and “lava-test-set” to create test elements from the test execution.

Reporting tests elements

In this section we will document the recommended solutions to report tests elements.

From lava test shells

To report test results within a lava shell environment just use the “lava-test-set” and “lava-test-case” commands.

From other scripts

Any test starts with the execution of a shell script which will run in a LAVA test shell environment. The main test script can call other scripts. To report test results just print “lava-test-set” or “lava-test-case” to the standard output. The main script will parse the standard output looking for these markers and create elements for any marker found.

Test case

Mandatory fields

The lava-test-case takes two parameters: a name and a result.
The result must be passed using --result <RESULT_STATE>.
Available result states are: pass, fail, skip, unknown

Examples:

```
$ lava-test-case find-something --result pass
```

Or from python:

```
#!/usr/bin/env python
import os
os.system("lava-test-case %s --result pass" % "find-something")
```

Extra fields

Measurements and units can also be passed to a lava-test-case as extra arguments. The test cases supports measurements and units per test at a precision of 10 digits. When recording a test-case with measurement the unit parameter becomes mandatory.

Examples:

```
$lava-test-case any-animals --result pass --measurement 3 --units animals
$lava-test-case fast-copy --result fail --measurement 10 --units seconds
```

Test set

The lava-test-set should be used around lava-test-case.
To create a new test set use: `$ lava-test-set start my-test-set`
To stop a test set use: `$ lava-test-set stop my-test-set`
Example:

```
$ lava-test-set start my-test-set
$ lava-test-case find-something --result pass
$ lava-test-set stop my-test-set
```

Test suite

The test suite name is determined by the name parameter within the repository section of the test definition.

Example:

```
- test:
   definitions:
    - repository: git://github.com/owner/agl-test-definitions.git
      from: git
      path: examples/custom-script.yaml
      name: test-example-custom-script
```

The test suite name would be “test-example-custom-script”.

Test definition

```
- test:
    timeout:
      minutes: 2
    definitions:
```

This section specifies the content of the test action section of a LAVA job. It documents the mandatory fields for this section and the different methods to write test definitions. The methods are documented in order from basic to advanced. Most advanced solutions will also require stronger coding skills and more complicated environment setup.

Examples: local, inline test

The following code sample shows a basic test definition shipped with the tool as an example.

```
- test:
    timeout:
      minutes: 2
    definitions:
    - repository:
        metadata:
          format: Lava-Test Test Definition 1.0
          name: inline-test
          description: "Inline test to validate test framewrok health"
          os:
          - debian
          scope:
          - functional
        run:
          steps:
          - lava-test-set start set-pass
          - lava-test-case always-pass --shell true
          - lava-test-set stop set-pass
          - lava-test-set start set-fail
          - lava-test-case always-fail --shell false
          - lava-test-set stop set-fail
      from: inline
      name: health-test
      path: inline/health-test.yaml
```

<releng-scripts-folder>/templates/tests/health-test.jinja2

The test definition is contained in a “**repository**” section. Each test definition has it’s own “- repository” section.

The “**metadata**” section contains metadata such as the test suite name, format and description.

The “**from**” parameter defines that this test is an inline test, the test code is inlined with the test definition. The “**name**” and “**path**” parameters are used to store the test definition to a specific location for its future execution.

The “**run**” section is a list of steps to execute. As this test code is inlined, each step will be executed sequentially within a lava test shell instance. More documentation about these steps is provided in section: [Reporting tests elements].

With this method, the test definition is stored within the test generation tool. It is useful for sharing tests definitions examples with the tool but it cannot be used for extensive testing. The developers would have to suggest modifications to the tool's repository for each change in a test. Which would not be maintainable.

Basic: remote, inline test

The solution for test maintainability is to keep the test definition in an external git repository. The test generation tool will only have the URL of the git repository to fetch the definition.
The following code sample shows how to link to a remote test definition.

```
- test:
    definitions:
    - repository: https://git.automotivelinux.org/src/qa-testdefinitions
      from: git
      path: test-suites/short-smoke/smoke-tests-basic.yaml
      name: smoke-tests-basic
```

<releng-scripts-folder>/templates/tests/smoke.jinja2

The test definition will be fetched at runtime from **`path`** within the given git repo (e.g. examples/basic-inline.yaml):

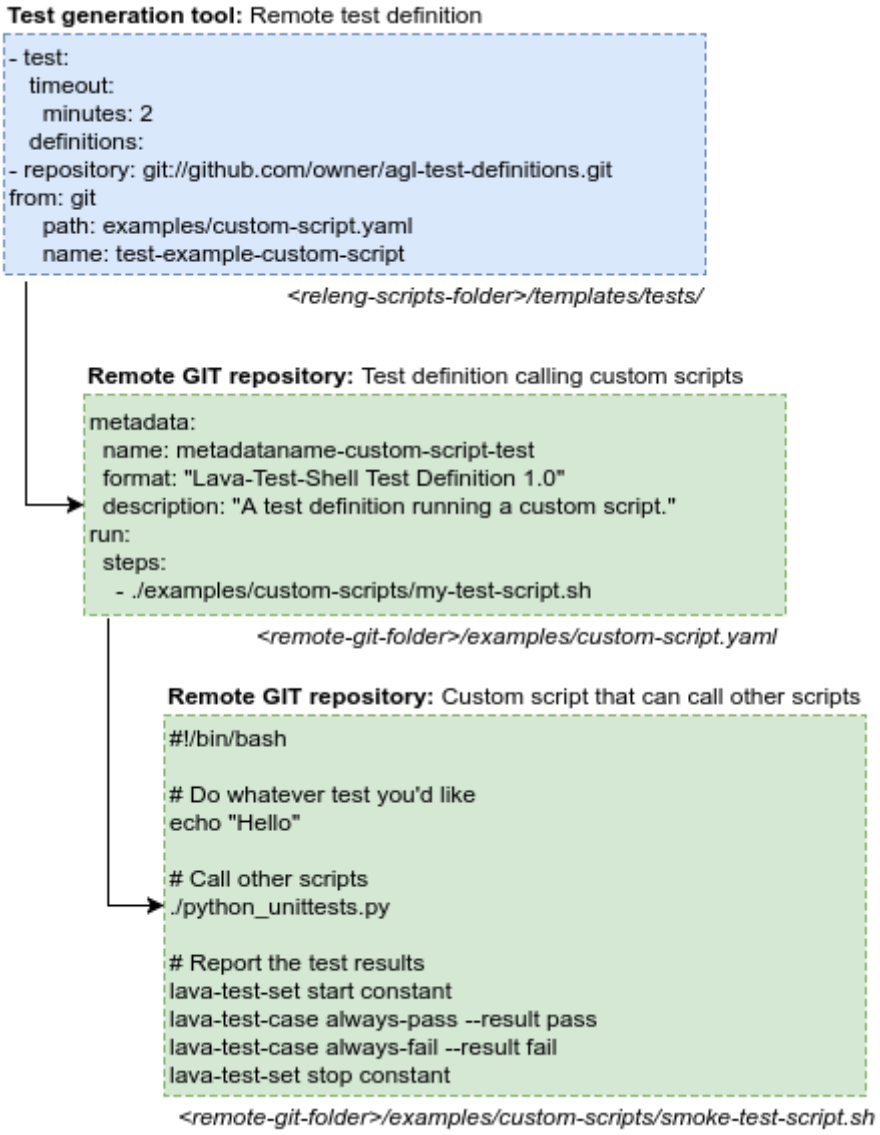
```
metadata:
    format: Lava-Test Test Definition 1.0
    name: smoke-tests-basic
    description: "Basic system test command for Linaro OpenEmbedded images"
    maintainer:
        - dave.pigott@linaro.org
    os:
        - openembedded
    scope:
        - functional

run:
    steps:
        - lava-test-case linux-linaro-openembedded-pwd --shell pwd
        - lava-test-case linux-linaro-openembedded-uname --shell uname -a
        - lava-test-case linux-linaro-openembedded-vmstat --shell vmstat
        - lava-test-case linux-linaro-openembedded-ifconfig --shell ifconfig -a
        - lava-test-case linux-linaro-openembedded-lsusb --shell lsusb
```

<qa-testdefinitions-folder>/test-suites/short-smoke/smoke-tests-basic.yaml

Using this method, the test is now remotely hosted allowing the owner to version and modify it as he likes. But the test code is still inlined within the "run" section. The user is limited in the range of tests he can write. And he must use shell commands only, each step is interpreted as a LAVA-Test-Shell command.

Extensive: remote, test scripts

The most extensive solution for writing tests is to use test scripts. The “run” section will list calls to test shell scripts. These scripts will be executed in their own LAVA Test Shells. From these scripts, the user has the freedom to use the coding language he likes, and use his own test structure. He could call other scripts written in Python for example.



The methodology to create these tests is documented in the following section: [Add new tests to AGL CI Loop].

Add new tests to the AGL CI Loop

[Prerequisites]

Clone the releng-scripts repo:

```
$ git clone https://git.automotivelinux.org/AGL/releng-scripts/
```

Fork or clone the example test definitions repository: agl-test-definitions.

[STEP 1] **Create a new test plan**

Create a new test plan file within the ./templates/tests/ folder or releng-scripts.
This folder is the pool of available test plans for AGL:

```
$ touch <releng-scripts>/templates/tests/my-test-plan-name.jinja2
```

[STEP 2] **Link to the test definitions within the releng-scripts tool**

Link to the the remote test definition:

```
$ vim <releng-scripts>/templates/tests/my-test-plan-name.jinja2
```

```
- test:
    timeout:
      minutes: 2
    definitions:
    - repository: git://github.com/OWNER/agl-test-definitions.git
      from: git
      path: examples/my-test-definition.yaml
      name: my-test-definition
```

[STEP 3] **Create the test definition on the remote repository**

Clone your fork of the AGL test definitions repository locally:

```
$ git clone https://github.com/OWNER/agl-test-definitions.git
```

Use the same architecture as defined in my-test-plan-name.jinja2 path to create the files.
Example:

```
$ cd <agl-test-definitions>
$ mkdir examples
$ touch ./examples/my-test-definition.yaml
$ vim ./examples/my-test-definition.yaml
```

```
metadata:
  name: metadataname-my-test-definition
  format: "Lava-Test-Shell Test Definition 1.0"
  description: "A test definition running a custom script."
run:
  steps:
    - ./examples/custom-scripts/my-test-script.sh
```

Other test definition available in agl-test-definition documentation: skeleton.yaml

[STEP 4] **Create the tests scripts and write the tests**

Use the same architecture as defined in my-test-definition.yaml path to create the files.

```
$ mkdir ./examples/custom-scripts
$ touch ./examples/custom-scripts/my-test-script.sh
$ vim ./examples/custom-scripts/my-test-script.sh
```

```
#!/bin/bash

# Do whatever test you'd like
echo "Hello"

# Call other scripts
python python_unittests.py

# Report the test results
lava-test-set start constant
lava-test-case always-pass --result pass
lava-test-case always-fail --result fail
lava-test-set stop constant
```

Other test script example available in agl-test-definition documentation: skeleton.sh

[STEP 4 bis] **Push the changes to the remote repo**

Use git to add, commit and push your changes to your fork of the test definitions repository.

```
$ git add ./examples/my-test-definition.yaml
$ git add ./examples/custom-scripts/my-test-script.sh
$ git commit && git push
```

[STEP 5] **Generate the test job**

Now that the test has been fully written from the test definition to the test cases. You can generate a new test job for LAVA using the generation tool:

```
$./utils/create-jobs.py m3ulcb -o myjob.yaml --test my-test-plan-name --callback lab-baylibre-lavabox
```

Well done, your test job is now written in the “myjob.yaml” file. You can have a look at it to make sure your test definition is correct.

[STEP 6] **Submitting the test job to LAVA**

To install and add authentication to LAVA please read this section: [Submit test jobs].
Once lava-tool is configured you can send your job definition to a LAVA instance using the command:

```
$ lava-tool submit-job http://demo@lavabox:10080/RPC2/ myjob.yaml
```

```
submitted as job: http://lavabox:10080/scheduler/job/XX
```

Further documentation on LAVA tests

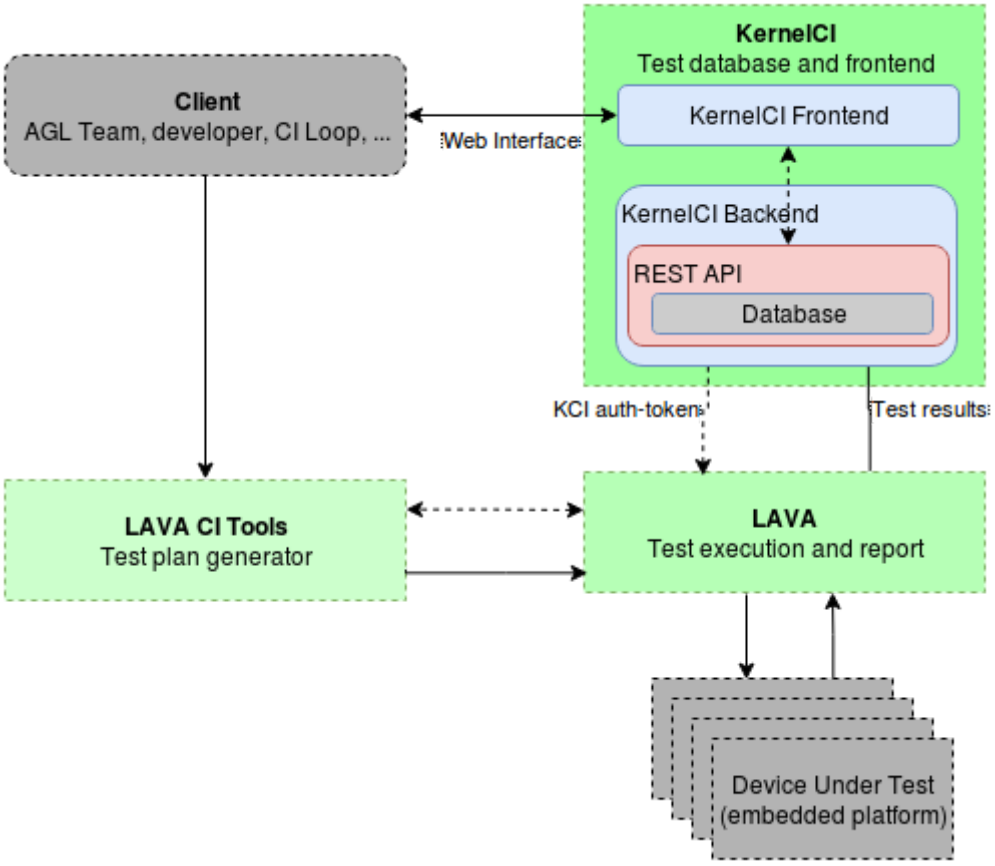
The following list is links to the LAVA documentation for further documentation on tests.

- Lava doc for writing tests; link
- Results in LAVA: test suite, set and case; link

- The LAVA test Shell and “lava-test-case”; link
- Best practices for writing LAVA test jobs; link
- Test Action references; link

From LAVA to KernelCI

This section will briefly describe how the test results are submitted from LAVA to KernelCI, parsed and saved. The client can then see these tests results using a web interface. The following diagram is a revision of the one available in the Introduction section.



AGL CI Architecture overview with Kernel CI internals

The Kernel CI block is expanded and we can find two internal blocks:

- KernelCI Frontend: a web server written in Python and Javascript; source code on GitHub.
- KernelCI Backend: a database containing tests data and an API to interact with the database implementing an authentication protocol; source code on GitHub.

LAVA callback to KernelCI

Thanks to the notify section of LAVA, when a job is finished, the LAVA instance will send the job results to a KernelCI instance.

```
notify:
  criteria:
       status: finished
  callback:
       url:
http://api.lavabox:8080/callback/lava/test?lab_name=lab-baylibre-lavabox&status={STATUS}
&status_string={STATUS_STRING}
       method: POST
       dataset: all
       token: c5ec62a9-98ea-4acf-9e45-bdaaec44dbe2
       content-type: json
```

Extract of a LAVA job definition: the notify section

Looking in detail at this section, it specifies all the needed information for LAVA to submit job results:

- url: the KernelCI backend URL to send jobs to.
- token: the authentication token to communicate through the REST API
- content-type: the result packaging type: json.

Using theses information, LAVA will wait for the job to finish and then send a POST request containing the job results to the specified KernelCI backend URL.

KernelCI callback event

The KernelCI backend instances are accessible through a REST API. This API is exposed on a specific URL and port number. On the same URL, a documentation on how to access the API is provided by default. The main KernelCI backend API documentation is available at this address: api.kernelci.org.

When Kernel CI backend receives a callback event on the API a serie of events happens. kCI first validates that the authentication credentials are correct. On success, it parses the received JSON job results. From this job results, database objects are created and stored on the server's database.

KernelCI frontend

The Kernel CI frontend is a web interface. It provides a way for any user to visualize test results stored in the KernelCI backend database. One of the instance of KernelCI frontend containing test results is available at this address: kernelci.dev.baylibre.com.

Going to the test section the user is able to browse through the test results sorting by board name, kernel version, ...