



Apertis integration testing with LAVA

# 1 Contents

2	Integration testing example . . . . .	2
3	Local testing . . . . .	2
4	Testing in LAVA . . . . .	3
5	Changes in testing script . . . . .	3
6	Create GIT repository for the test suite . . . . .	5
7	Add the test into Apertis LAVA CI . . . . .	5
8	Details on test job templates . . . . .	10
9	Using short-lived CI tokens . . . . .	11
10	Non-public jobs . . . . .	12

11 **LAVA**<sup>1</sup> is a testing system allowing the deployment of operating systems to  
12 physical and virtual devices, sharing access to devices between developers. As  
13 a rule tests are started in non-interactive unattended mode and LAVA provides  
14 logs and results in a human-readable form for analysis.

15 As a common part of the development cycle we need to do some integration  
16 testing of the application and validate it's behavior on different hardware and  
17 software platforms. LAVA provides the ability for Apertis to share a pool of  
18 test devices, ensuring good utilization of these resources in addition to providing  
19 automated testing.

## 20 Integration testing example

21 Let's take the `systemd` service and `systemctl` CLI tool as an example to illustrate  
22 how to test an application with a D-Bus interface.

23 The goal could be defined as follows:

24       As a developer of the `systemctl` CLI tool, I want to ensure that  
25       `systemctl` is able to provide correct information about the system  
26       state.

## 27 Local testing

28 To simplify the guide we are testing only the status of `systemd` with the command  
29 below:

```
30 $ systemctl is-system-running  
31 running
```

32 It doesn't matter if `systemctl` is reporting some other status, `degraded` for in-  
33 stance. The goal is to validate if `systemctl` is able to provide a proper status,  
34 rather than to check the `systemd` status itself.

35 To ensure that the `systemctl` tool is providing the correct information we may  
36 check the system state additionally via the `systemd` D-Bus interface:

---

<sup>1</sup><https://www.lavasoftware.org/>

```
37 $ gdbus call --system --dest=org.freedesktop.systemd1 --object-path "/org/freedesktop/systemd1" -
38 -method org.freedesktop.DBus.Properties.Get org.freedesktop.systemd1.Manager SystemState
39 (<'running'>,)

```

40 So, for local testing during development we are able to create a simple script  
41 validating that `systemctl` works well in our development environment:

```
42 #!/bin/sh
43
44 status=$(systemctl is-system-running)
45
46 gdbus call --system --dest=org.freedesktop.systemd1 \
47 --object-path "/org/freedesktop/systemd1" \
48 --method org.freedesktop.DBus.Properties.Get org.freedesktop.systemd1.Manager SystemState | \
49 grep "${status}"
50
51 if [ $? -eq 0 ]; then
52     echo "systemctl is working"
53 else
54     echo "systemctl is not working"
55 fi

```

## 56 **Testing in LAVA**

57 As soon as we are done with development, we push all changes to GitLab and CI  
58 will prepare a new version of the package and OS images. But we do not know  
59 if the updated version of `systemctl` is working well for all supported devices and  
60 OS variants, so we want to have the integration test to be run by LAVA.

61 Since the LAVA is a part of CI and works in non-interactive unattended mode  
62 we can't use the test script above as is.

63 To start the test with LAVA automation we need to:

- 64 1. Adopt the script for LAVA
- 65 2. Integrate the testing script into Apertis LAVA CI

## 66 **Changes in testing script**

67 The script above is not suitable for unattended testing in LAVA due some issues:

- 68 • LAVA relies on exit code to determine if test a passed or not. The exam-  
69 ple above always return the `success` code, only a human-readable string  
70 printed by the script provides an indication of the status of the test
- 71 • if `systemctl is-system-running` call fails for some other reason (with a  
72 segfault for instance), the script will proceed further without that error  
73 being detected and LAVA will set the test as passed, so we will have a  
74 false positive result

75 • LAVA is able to report separately for any part of the test suite –just need  
76 to use LAVA-friendly output pattern

77 So, more sophisticated script suitable both for local and unattended testing in  
78 LAVA could be the following:

```
79 #!/bin/sh
80
81 # Test if systemctl is not crashed
82 testname="test-systemctl-crash"
83 status=$(systemctl is-system-running)
84 if [ $? -le 4 ]; then
85     echo "${testname}: pass"
86 else
87     echo "${testname}: fail"
88     exit 1
89 fi
90
91 # Test if systemctl return non-empty string
92 testname="test-systemctl-value"
93 if [ -n "$status" ]; then
94     echo "${testname}: pass"
95 else
96     echo "${testname}: fail"
97     exit 1
98 fi
99
100 # Test if systemctl is reporting the same status as
101 # systemd exposing via D-Bus
102 testname="test-systemctl-dbus-status"
103 gdbus call --system --dest=org.freedesktop.systemd1 \
104     --object-path "/org/freedesktop/systemd1" \
105     --method org.freedesktop.DBus.Properties.Get \
106     org.freedesktop.systemd1.Manager SystemState | \
107     grep "${status}"
108 if [ $? -eq 0 ]; then
109     echo "${testname}: pass"
110 else
111     echo "${testname}: fail"
112     exit 1
113 fi
```

114 Now the script is ready for adding into LAVA testing. Pay attention to output  
115 format which will be used by LAVA to detect separate tests from our single  
116 script. The exit code from the testing script must be non-zero to indicate the  
117 test suite failure.

## 118 **Create GIT repository for the test suite**

119 We assume the developer is already familiar with [GIT version control system](#)<sup>2</sup>  
120 and has an account for the [Apertis GitLab](#)<sup>3</sup> as described in the [Development](#)  
121 [Process guide](#)<sup>4</sup>

122 The test script must be accessible by LAVA for downloading. LAVA has support  
123 for several methods for downloading but for Apertis the GIT fetch is preferable  
124 since we are using separate versions of test scripts for each release.

125 It is strongly recommended to create a separate repository with test scripts and  
126 tools for each single test suite.

127 As a first step we need a fresh and empty GIT repository somewhere (for example  
128 in your personal space of the GitLab instance) which needs to be cloned locally:

```
129 git clone git@gitlab.apertis.org:d4s/test-systemctl.git  
130 cd test-systemctl
```

131 By default the branch name is set to `main` but Apertis automation require to use  
132 the branch name aimed at a selected release (for instance `apertis/v2022dev1`), so  
133 need to create it:

```
134 git checkout HEAD -b apertis/v2022dev1
```

135 Copy your script into GIT repository, commit and push it into GitLab:

```
136 chmod a+x test-systemctl.sh  
137 git add test-systemctl.sh  
138 git commit -s -m "Add test script" test-systemctl.sh  
139 git push -u origin apertis/v2022dev1
```

## 140 **Add the test into Apertis LAVA CI**

141 Apertis test automation could be found in the [GIT repository for Apertis test](#)  
142 [cases](#)<sup>5</sup>, so we need to fetch a local copy and create a work branch `wip/example`  
143 for our changes:

```
144 git clone git@gitlab.apertis.org:tests/apertis-test-cases.git  
145 cd apertis-test-cases  
146 git checkout HEAD -b wip/example
```

147 1. Create test case description.

148 First of all we need to create the instruction for LAVA with following  
149 information:

- 150 • where to get the test

---

<sup>2</sup>[https://www.apertis.org/guides/app\\_devel/version\\_control/](https://www.apertis.org/guides/app_devel/version_control/)

<sup>3</sup><https://gitlab.apertis.org/>

<sup>4</sup>[https://www.apertis.org/guides/app\\_devel/development\\_process/](https://www.apertis.org/guides/app_devel/development_process/)

<sup>5</sup><https://gitlab.apertis.org/tests/apertis-test-cases>

151  
152  
153

- how to run the test

Create the test case file `test-cases/test-systemctl.yaml` with your favorite editor:

---

```
1 metadata:
2   name: test-systemctl
3   format: "Apertis Test Definition 1.0"
4   image-types:
5     fixedfunction: [ armhf, arm64, amd64 ]
6   image-deployment:
7     - OSTree
8   group: systemctl
9   type: functional
10  exec-type: automated
11  priority: medium
12  maintainer: "Apertis Project"
13  description: "Test the systemctl."
14
15  expected:
16    - "The output should show pass."
17
18  install:
19    git-repos:
20      - url: https://gitlab.apertis.org/d4s/test-systemctl.git
21        branch: apertis/v2022dev1
22
23  run:
24    steps:
25      - "# Enter test directory:"
26      - cd test-systemctl
27      - "# Execute the following command:"
28      - lava-test-case test-systemctl --shell ./test-systemctl.sh
29
30  parse:
31    pattern: "(?P<test_case_id>.*):\\s+(?P<result>(pass|fail))"
```

---

154  
155  
156  
157  
158  
159  
160

This test is aimed to be run for an ostree-based fixedfunction Apertis image for all supported architectures. However the metadata is mostly needed for documentation purposes.

The `group` field is used to group test cases into the same LAVA job description. See the job templates below.

Action “install” points to the GIT repository as a source for the test, so LAVA will fetch and deploy this repository for us.

161 Action “run” provides the step-by-step instructions on how to execute the  
162 test. Please note that it is recommended to use wrapper for the test for  
163 integration with LAVA.

164 Action “parse” provides its own detection for the status of test results  
165 printed by script.

166 2. Push the test case to the GIT repository.

167 This step is mandatory since the test case would be checked out by LAVA  
168 internally during the test preparation.

```
169 git add test-cases/test-systemctl.yaml  
170 git commit -s -m "add test case for systemctl" test-cases/test-  
171 systemctl.yaml  
172 git push --set-upstream origin wip/example
```

173 3. If needed, add a job template to be run in lava. Job templates contain  
174 all needed information for LAVA to boot the target device and deploy the  
175 OS image onto it.

176 Job template files must be named `lava/group-[GROUP]-tpl.yaml`.

177 e.g.: Create the simple template `lava/group-systemctl-tpl.yaml` with your  
178 lovely editor:

```
179 job_name: systemctl test on {{release_version}} {{pretty}} {{image_date}}  
180 {% if device_type == 'qemu' %}  
181 {% include 'common-qemu-boot-tpl.yaml' %}  
182 {% else %}  
183 {% include 'common-boot-tpl.yaml' %}  
184 {% endif %}  
185 - test:  
186     timeout:  
187         minutes: 15  
188     namespace: system  
189     name: {{group}}-tests  
190     definitions:  
191     {%- for test_name in tests %}  
192         - repository: https://gitlab.apertis.org/tests/apertis-test-  
193 cases.git  
194         branch: 'wip/example'  
195         from: git  
196         name: {{test_name}}  
197         path: test-cases/{{test_name}}.yaml  
198     {%- endfor -%}
```

199 If no template exists for a given group, the default template (`lava/group-  
200 default-tpl.yaml`) will be used, still creating a different job per group. It

201 looks a lot like the example template above. This is useful if you do not  
202 need any specific variables set or special boot steps.

203 Hopefully you don't need to deal with the HW-related part, boot and  
204 deploy since we already have those instructions for all supported boards  
205 and Apertis OS images. See [common boot template](#)<sup>6</sup> for instance.

206 Please pay attention to `branch` -it must point to your development branch  
207 while you are working on your test.

208 It is highly recommended to use a temporary group specific to the test  
209 you are working on to avoid unnecessary workload on LAVA while you're  
210 developing the test.

#### 211 4. Generate the job descriptions.

212 Since LAVA is a part of Apertis OS CI, it requires some variables to be  
213 provided for using Apertis templates. Let's define the board we will use  
214 for testing, as well as the image release and variant:

```
215 release=v2023dev1  
216 version=v2023dev1.0rc2  
217 variant=fixedfunction  
218 arch=armhf  
219 board=uboot  
220 baseurl="https://images.apertis.org"  
221 imgpath="release/${release}"  
222 image_name=apertis_ostree_${release}-${variant}-${arch}-${board}_${version}
```

223 To generate the test job description, `generate-jobs.py` is used:

```
224 ./generate-jobs.py -d lava/devices.yaml --config lava/config.yaml  
225     --release ${release} --arch ${arch} --board ${board} --osname apertis  
226     --deployment ostree --type ${variant} --date ${version}  
227     --name ${image_name}  
228     -t visibility:"{'group': ['Apertis']}" -t priority:"medium"
```

229 It will generate one job description file for each group that is found com-  
230 patible with those parameters.

231 `generate-jobs.py` can be found [here](#)<sup>7</sup>

232 There should not be any error or warning. You may want to add the `-v`  
233 argument to see the generated LAVA job.

234 If the test definition is on an external git repository, you can specify the  
235 folder to load the test cases from with `--tests-dir` or, for debugging one  
236 specific test case, specify it with `--test-case`.

---

<sup>6</sup><https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2022/lava/common-boot-tpl.yaml>

<sup>7</sup><https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2023dev1/generate-jobs.py>



237 It is recommended to set `visibility` variable to “Apertis”group during  
238 development to avoid any credentials/passwords leak by occasion. Setting  
239 the additional variable `priority` to `high` allows you to bypass the jobs  
240 common queue if you do not want to wait for your job results for ages.

241 The `generate-jobs.py` tool generates the test job from local files, so you  
242 don’t need to push your changes to GIT until your test job is working as  
243 designed.

## 244 5. Configure and test the `lqa` tool.

245 For interaction with LAVA you need to have the `lqa` tool installed and  
246 configured as described in [LQA](#)<sup>8</sup> tutorial.

247 The tool is pretty easy to install in the Apertis SDK:

```
248 $ sudo apt-get update  
249 $ sudo apt-get install -y lqa
```

250 To configure the tool you need to create file `~/.config/lqa.yaml` with the  
251 following authentication information:

```
252 user: '<REPLACE_THIS_WITH_YOUR_LAVA_USERNAME>'  
253 auth-token: '<REPLACE_THIS_WITH_YOUR_AUTH_TOKEN>'  
254 server: 'https://lava.collabora.dev/'
```

255 where `user` is your login name for LAVA and `auth-token` must be obtained  
256 from LAVA API: <https://lava.collabora.dev/api/tokens/>

257 To test the setup just run command below, if the configuration is correct  
258 you should see your LAVA login name:

```
259 $ lqa whoami  
260 d4s
```

## 261 6. Submit your first job to LAVA.

262 Jobs can be submitted with `lava-submit.py`. It can be found [here](#)<sup>9</sup>.

263 You can select the job files you want to send, here it will be the one for  
264 our new test group `systemctl`:

```
265 job-apertis_ostree_v2023dev1-fixedfunction-armhf-uboot_v2023dev1.0rc2-  
266 systemctl.yaml
```

267 and can be sent with:

```
268 $ ./lava-submit.py -c ~/.config/lqa.yaml submit  
269     job-apertis_ostree_v2023dev1-fixedfunction-armhf-uboot_v2023dev1.0rc2-  
270     systemctl.yaml
```

---

<sup>8</sup><https://www.apertis.org/qa/lqa/>

<sup>9</sup><https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2023dev1/lava-submit.py>

271 Submitted job job-apertis\_ostree\_v2023dev1-fixedfunction-armhf-  
272 uboot\_v2023dev1.0rc2-systemctl.yaml with id 3463731

273 It is possible to check the job status by URL with the ID returned by the  
274 above command: <https://lava.collabora.dev/scheduler/job/3463731>

275 The `lava-submit.py` tool is currently only a wrapper around the `lqa` tool. It  
276 is also capable to communicate the tested image to the [QA Report App](#)<sup>10</sup>.

277 7. Push your template changes.

278 Once your test case works as expected you should make sure it is in the  
279 right group, change the `branch` key in file `lava/group-systemctl-tpl.yaml` to  
280 a suitable target branch and submit your changes:

```
281 git add lava/group-systemctl-tpl.yaml  
282 git commit -a -m "hello world template added"  
283 git push
```

284 As a last step you need to create a merge request in GitLab. As soon as it gets  
285 accepted your test becomes part of Apertis testing CI.

## 286 Details on test job templates

287 The boot process for non-emulated devices and for QEMU differs, and due to  
288 the amount of differences the definitions are split into two separate template  
289 files.

290 `common-boot-tpl.yaml` contains definition needed to boot Apertis images on real  
291 (non-emulated devices). Since they cannot boot images directly, the boot pro-  
292 cess is separated in two stages: flashing the image onto a device from which the  
293 board can boot, and booting into the image and running tests.

294 The first stage boots over NFS into a (currently) Debian stretch image with a  
295 few extra tools needed to flash the image, downloads the image using HTTP,  
296 flashes it and reboots. This stage is defined using `namespace: flash` in the job  
297 YAML file. In most cases you won't need to edit bits related to this stage. The  
298 second stage is common for both non-emulated devices and QEMU, despite  
299 them having certain differences. It is used to boot the image itself, prepare the  
300 LAVA test runner and run tests. This stage is defined using `namespace: system`.  
301 You *normally* don't need to edit this stage either. The exception to this is when  
302 you need to load an image from a different source than `images.apertis.org`.

303 Image URLs are defined in the `deploy` action. For `common-boot-tpl.yaml`, it is  
304 necessary to specify URLs to both image itself and its `bmap` file, which is used  
305 to speed up the flashing process and avoid unnecessary excessive device wear.  
306 For `common-qemu-boot-tpl.yaml`, only the URL to the image itself is needed, as  
307 QEMU doesn't support `bmap` files yet.

---

<sup>10</sup><https://qa.apertis.org/>

308 The second stage always performs two tests: `sanity-check`, which basically checks  
309 that the system actually works, and `add-repo`, which isn't actually a test, and is  
310 used to add repositories to `/etc/apt/sources.list` on certain devices.

### 311 Using short-lived CI tokens

312 Gitlab provides a short-lived token called `CI_JOB_TOKEN` which can be used to give  
313 access to the contents of internal and private repositories during CI runs. From  
314 `apertis/v2023dev3` we can make use of this token, using a different approach to  
315 job submission to the one described in the previous sections. That is, so far in  
316 this document, we've run `lava-submit.py` to batch upload the jobs generated by  
317 `generate-jobs.py` to LAVA. If we do the same thing in our CI pipeline, then the  
318 CI job will terminate shortly after the jobs are uploaded, invalidating our job  
319 token.

320 Do not expose `CI_JOB_TOKEN` to the wider public by submitting publicly visible  
321 jobs. You should submit jobs with tokens in them as `private`. You should also  
322 [reduce the privileges of job tokens](#)<sup>11</sup> when using `CI_JOB_TOKEN` in LAVA jobs.

323 For this reason, instead of using `lava-submit.py`, we use a different tool, `generate-`  
324 `test-pipeline.py`, from the same repository when running CI tests. This makes  
325 a dynamic Gitlab pipeline to run the generated jobs. Each LAVA job will have  
326 its own Gitlab job to track it, and that means there is a short-lived token  
327 available that will remain valid for as long as the LAVA job runs. `generate-`  
328 `test-pipeline.py` can be found [here](#)<sup>12</sup>.

329 There are two different places you might want to use such tokens with LAVA,  
330 and they require slightly different approaches. To use a short-lived token to  
331 gain access to a repository from a LAVA job description, for example to obtain  
332 test files from a private repository, the repository URL needs to be altered to  
333 show where to substitute the token. For example:

```
334 https://gitlab-ci-token:{{ '{{job.CI_JOB_TOKEN}}' }}@gitlab.apertis.org/tests/apertis-  
335 test-cases.git
```

336 The odd appearance is because two rounds of templating are occurring: we  
337 escape the template for the job token so that `generate-jobs.py` will preserve it.  
338 When our dynamic pipeline runs, the LAVA runner will substitute its own value  
339 for `CI_JOB_TOKEN`.

340 To use a short-lived token from within a test-case, we need to do two things.  
341 First, we need to add a parameter to the test's group template with the full  
342 URL for the repository we wish to include. The group templates form part of  
343 the job definition, and so we can modify the URL in exactly the same way as  
344 before.

---

<sup>11</sup>[https://docs.gitlab.com/ee/ci/jobs/ci\\_job\\_token.html#configure-the-job-token-scope-limit](https://docs.gitlab.com/ee/ci/jobs/ci_job_token.html#configure-the-job-token-scope-limit)

<sup>12</sup><https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2023dev3/generate-test-pipeline.py>

345 Secondly, we need to replace the repository URL in the test case with the new  
346 parameter. You cannot use templating within test cases themselves, you must  
347 setup a parameter or environment variable in the job definition that the test  
348 case can use. Parameters are preferable because they can be used in the `install`  
349 section of a test.

350 Putting things together, let's look at a section of a group template that:

- 351 • Pulls test case files from `apertis-test-cases` using a short-lived token.
- 352 • Sets up a parameter which contains the URL to clone `glib-gio-fs` using a  
353 short-lived token as authentication. We can use this parameter in a test  
354 case to obtain our test data.

---

```
1   - test:
2     timeout:
3       minutes: 180
4     namespace: system
5     name: {{group}}-tests
6     definitions:
7   {% for test_name in tests %}
8     - repository: https://gitlab-ci-token:{{ '{{job.CI_JOB_TOKEN}}' }}@gitlab.apertis.org/tests/ap
9       branch: 'apertis/v2023dev3'
10      history: False
11      from: git
12      name: {{test_name}}
13      path: test-cases/{{test_name}}.yaml
14      parameters:
15        EXAMPLE_REPO_URL: |-
16          https://gitlab-ci-token:{{ '{{job.CI_JOB_TOKEN}}' }}@gitlab.apertis.org/tests/glib-gio-f
17   {% endfor -%}
```

---

355 We could then amend our test-case in `apertis-test-cases` to use the parameter  
356 like this (note that there is no `§` when substituting the parameter in an `install`  
357 section):

---

```
1   install:
2     git-repos:
3       - url: EXAMPLE_REPO_URL
4         branch: 'apertis/v2023dev3'
```

---

### 358 **Non-public jobs**

359 These instructions are written to submit LAVA jobs for **ONLY PUBLIC** Aper-  
360 tis images. If you need to submit a LAVA job for a private image, there are  
361 few things that need to be taken into consideration and few changes need to be

<sup>362</sup> made to these instructions: `personal` or `group` visibility should be selected for  
<sup>363</sup> your jobs.

<sup>364</sup> If you really need to submit a private job, please contact the Apertis QA team.